

Trusted Platform Module Library

Part 4: Supporting Routines

Family “2.0”

Level 00 Revision 01.81

November 29, 2023

Committee Draft

Contact: admin@trustedcomputinggroup.org

Work in Progress

This document is an intermediate draft for comment only and is subject to change without notice. Readers should not design products based on this document.

Licenses and Notices

Copyright Licenses:

- Trusted Computing Group (TCG) grants to the user of the source code in this specification (the "Source Code") a worldwide, irrevocable, nonexclusive, royalty free, copyright license to reproduce, create derivative works, distribute, display and perform the Source Code and derivative works thereof, and to grant others the rights granted herein.
- The TCG grants to the user of the other parts of the specification (other than the Source Code) the rights to reproduce, distribute, display, and perform the specification solely for the purpose of developing products based on such documents.

Source Code Distribution Conditions:

- Redistributions of Source Code must retain the above copyright licenses, this list of conditions and the following disclaimers.
- Redistributions in binary form must reproduce the above copyright licenses, this list of conditions and the following disclaimers in the documentation and/or other materials provided with the distribution.

Disclaimers:

- THE COPYRIGHT LICENSES SET FORTH ABOVE DO NOT REPRESENT ANY FORM OF LICENSE OR WAIVER, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, WITH RESPECT TO PATENT RIGHTS HELD BY TCG MEMBERS (OR OTHER THIRD PARTIES) THAT MAY BE NECESSARY TO IMPLEMENT THIS SPECIFICATION OR OTHERWISE. Contact TCG Administration (admin@trustedcomputinggroup.org) for information on specification licensing rights available through TCG membership agreements.
- THIS SPECIFICATION IS PROVIDED "AS IS" WITH NO EXPRESS OR IMPLIED WARRANTIES WHATSOEVER, INCLUDING ANY WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, ACCURACY, COMPLETENESS, OR NONINFRINGEMENT OF INTELLECTUAL PROPERTY RIGHTS, OR ANY WARRANTY OTHERWISE ARISING OUT OF ANY PROPOSAL, SPECIFICATION OR SAMPLE.
- Without limitation, TCG and its members and licensors disclaim all liability, including liability for infringement of any proprietary rights, relating to use of information in this specification and to the implementation of this specification, and TCG disclaims all liability for cost of procurement of substitute goods or services, lost profits, loss of use, loss of data or any incidental, consequential, direct, indirect, or special damages, whether under contract, tort, warranty or otherwise, arising in any way out of use or reliance upon this specification or any information herein.

Any marks and brands contained herein are the property of their respective owners.

CONTENTS

CONTENTS.....	3
1 Scope	1
2 Terms and definitions	1
3 Symbols and abbreviated terms	1
4 Automation	1
4.1 Configuration Parser.....	1
4.2 Structure Parser	1
4.2.1 Introduction	1
4.2.2 Unmarshaling Code Prototype.....	2
4.2.2.1 Simple Types and Structures.....	2
4.2.2.2 Union Types	2
4.2.2.3 Null Types	3
4.2.2.4 Arrays.....	3
4.2.3 Marshaling Code Function Prototypes	3
4.2.3.1 Simple Types and Structures.....	3
4.2.3.2 Union Types	4
4.2.3.3 Arrays.....	4
4.2.4 Table-driven Marshaling.....	4
4.3 Part 3 Parsing	4
4.4 Function Prototypes.....	5
4.4.1 /tpm/include/private/prototypes/Create_fp.h.....	5
4.5 Portability.....	6
5 Marshaling	7
5.1 Introduction	7
5.2 Unmarshal and Marshal a Value	7
5.3 Unmarshal and Marshal a Union	8
5.4 Unmarshal and Marshal a Structure	10
5.5 Unmarshal and Marshal an Array	11
5.6 TPM2B Handling	13
6 TPM Reference Implementation Include Files.....	14
6.1 /tpm/include/platform_interface/pcrstruct.h	14
6.2 /tpm/include/platform_interface/platform_to_tpm_interface.h	15
6.3 /tpm/include/platform_interface/tpm_to_platform_interface.h	15
6.4 /tpm/include/platform_interface/prototypes/ExecCommand_fp.h.....	21
6.5 /tpm/include/platform_interface/prototypes/Manufacture_fp.h.....	22
6.6 /tpm/include/platform_interface/prototypes/platform_pcr_fp.h	23
6.7 /tpm/include/platform_interface/prototypes/_TPM_Hash_Data_fp.h.....	24
6.8 /tpm/include/platform_interface/prototypes/_TPM_Hash_End_fp.h.....	24
6.9 /tpm/include/platform_interface/prototypes/_TPM_Hash_Start_fp.h.....	24
6.10 /tpm/include/platform_interface/prototypes/_TPM_Init_fp.h	24
6.11 /tpm/include/private/CommandAttributeData.h	25
6.12 /tpm/include/private/CommandAttributes.h	39
6.13 /tpm/include/private/CommandDispatchData.h.....	39
6.14 /tpm/include/private/CommandDispatcher.h.....	125
6.15 /tpm/include/private/Commands.h	166
6.16 /tpm/include/private/CryptEcc.h.....	172
6.17 /tpm/include/private/CryptHash.h	173
6.18 /tpm/include/private/CryptRand.h	177

6.19	/tpm/include/private/CryptRsa.h	180
6.20	/tpm/include/private/CryptSym.h	180
6.21	/tpm/include/private/CryptTest.h.....	182
6.22	/tpm/include/private/EccTestData.h	182
6.23	/tpm/include/private/Global.h	185
6.24	/tpm/include/private/HandleProcess.h	206
6.25	/tpm/include/private/HashTestData.h.....	228
6.26	/tpm/include/private/InternalRoutines.h	229
6.27	/tpm/include/private/KdfTestData.h	231
6.28	/tpm/include/private/LibSupport.h.....	232
6.29	/tpm/include/private/Marshal.h	232
6.30	/tpm/include/private/NV.h.....	232
6.31	/tpm/include/private/OIDs.h.....	234
6.32	/tpm/include/private/PRNG_TestVectors.h.....	238
6.33	/tpm/include/private/RsaTestData.h.....	240
6.34	/tpm/include/private/SelfTest.h	247
6.35	/tpm/include/private/SymmetricTest.h.....	248
6.36	/tpm/include/private/SymmetricTestData.h.....	249
6.37	/tpm/include/private/TableMarshal.h	251
6.38	/tpm/include/private/TableMarshalDefines.h	255
6.39	/tpm/include/private/TableMarshalMainTable.h	255
6.40	/tpm/include/private/TableMarshalPrototypes.h	255
6.41	/tpm/include/private/TableMarshalRedefines.h	255
6.42	/tpm/include/private/TableMarshalTypes.h.....	255
6.43	/tpm/include/private/Tpm.h.....	278
6.44	/tpm/include/private/TpmASN1.h	279
6.45	/tpm/include/private/X509.h	280
6.46	/tpm/include/private/prototypes/ActivateCredential_fp.h.....	282
6.47	/tpm/include/private/prototypes/ACT_SetTimeout_fp.h.....	282
6.48	/tpm/include/private/prototypes/ACT_spt_fp.h	283
6.49	/tpm/include/private/prototypes/AC_GetCapability_fp.h.....	284
6.50	/tpm/include/private/prototypes/AC_Send_fp.h	284
6.51	/tpm/include/private/prototypes/AC_spt_fp.h.....	285
6.52	/tpm/include/private/prototypes/AlgorithmCap_fp.h	285
6.53	/tpm/include/private/prototypes/AlgorithmTests_fp.h	286
6.54	/tpm/include/private/prototypes/Attest_spt_fp.h	287
6.55	/tpm/include/private/prototypes/Bits_fp.h.....	288
6.56	/tpm/include/private/prototypes/CertifyCreation_fp.h.....	288
6.57	/tpm/include/private/prototypes/CertifyX509_fp.h.....	289
6.58	/tpm/include/private/prototypes/Certify_fp.h.....	289
6.59	/tpm/include/private/prototypes/ChangeEPS_fp.h	290
6.60	/tpm/include/private/prototypes/ChangePPS_fp.h	290
6.61	/tpm/include/private/prototypes/ClearControl_fp.h	291
6.62	/tpm/include/private/prototypes/Clear_fp.h	291
6.63	/tpm/include/private/prototypes/ClockRateAdjust_fp.h	292
6.64	/tpm/include/private/prototypes/ClockSet_fp.h	292
6.65	/tpm/include/private/prototypes/CommandAudit_fp.h	292
6.66	/tpm/include/private/prototypes/CommandCodeAttributes_fp.h.....	294
6.67	/tpm/include/private/prototypes/CommandDispatcher_fp.h.....	296
6.68	/tpm/include/private/prototypes/Commit_fp.h	296
6.69	/tpm/include/private/prototypes/ContextLoad_fp.h	297
6.70	/tpm/include/private/prototypes/ContextSave_fp.h	297
6.71	/tpm/include/private/prototypes/Context_spt_fp.h	298
6.72	/tpm/include/private/prototypes/CreateLoaded_fp.h	299
6.73	/tpm/include/private/prototypes/CreatePrimary_fp.h.....	300
6.74	/tpm/include/private/prototypes/Create_fp.h	300
6.75	/tpm/include/private/prototypes/CryptCmac_fp.h.....	301
6.76	/tpm/include/private/prototypes/CryptEccCrypt_fp.h.....	302

6.77	/tpm/include/private/prototypes/CryptEccKeyExchange_fp.h	303
6.78	/tpm/include/private/prototypes/CryptEccMain_fp.h	303
6.79	/tpm/include/private/prototypes/CryptEccSignature_fp.h	307
6.80	/tpm/include/private/prototypes/CryptHash_fp.h	309
6.81	/tpm/include/private/prototypes/CryptPrimeSieve_fp.h	314
6.82	/tpm/include/private/prototypes/CryptPrime_fp.h	315
6.83	/tpm/include/private/prototypes/CryptRand_fp.h	316
6.84	/tpm/include/private/prototypes/CryptRsa_fp.h	319
6.85	/tpm/include/private/prototypes/CryptSelfTest_fp.h	321
6.86	/tpm/include/private/prototypes/CryptSmac_fp.h	322
6.87	/tpm/include/private/prototypes/CryptSym_fp.h	323
6.88	/tpm/include/private/prototypes/CryptUtil_fp.h	324
6.89	/tpm/include/private/prototypes/DA_fp.h	330
6.90	/tpm/include/private/prototypes/DictionaryAttackLockReset_fp.h	331
6.91	/tpm/include/private/prototypes/DictionaryAttackParameters_fp.h	331
6.92	/tpm/include/private/prototypes/Duplicate_fp.h	331
6.93	/tpm/include/private/prototypes/ECC_Decrypt_fp.h	332
6.94	/tpm/include/private/prototypes/ECC_Encrypt_fp.h	333
6.95	/tpm/include/private/prototypes/ECC_Parameters_fp.h	333
6.96	/tpm/include/private/prototypes/ECDH_KeyGen_fp.h	334
6.97	/tpm/include/private/prototypes/ECDH_ZGen_fp.h	334
6.98	/tpm/include/private/prototypes/EC_Ephemeral_fp.h	335
6.99	/tpm/include/private/prototypes/EncryptDecrypt2_fp.h	335
6.100	/tpm/include/private/prototypes/EncryptDecrypt_fp.h	336
6.101	/tpm/include/private/prototypes/EncryptDecrypt_spt_fp.h	336
6.102	/tpm/include/private/prototypes/Entity_fp.h	337
6.103	/tpm/include/private/prototypes/EventSequenceComplete_fp.h	338
6.104	/tpm/include/private/prototypes/EvictControl_fp.h	339
6.105	/tpm/include/private/prototypes/FieldUpgradeData_fp.h	339
6.106	/tpm/include/private/prototypes/FieldUpgradeStart_fp.h	340
6.107	/tpm/include/private/prototypes/FirmwareRead_fp.h	340
6.108	/tpm/include/private/prototypes/FlushContext_fp.h	341
6.109	/tpm/include/private/prototypes/GetCapability_fp.h	341
6.110	/tpm/include/private/prototypes/GetCommandAuditDigest_fp.h	342
6.111	/tpm/include/private/prototypes/GetRandom_fp.h	342
6.112	/tpm/include/private/prototypes/GetSessionAuditDigest_fp.h	343
6.113	/tpm/include/private/prototypes/GetTestResult_fp.h	343
6.114	/tpm/include/private/prototypes/GetTime_fp.h	344
6.115	/tpm/include/private/prototypes/Handle_fp.h	344
6.116	/tpm/include/private/prototypes/HashSequenceStart_fp.h	345
6.117	/tpm/include/private/prototypes/Hash_fp.h	346
6.118	/tpm/include/private/prototypes/HierarchyChangeAuth_fp.h	346
6.119	/tpm/include/private/prototypes/HierarchyControl_fp.h	347
6.120	/tpm/include/private/prototypes/Hierarchy_fp.h	347
6.121	/tpm/include/private/prototypes/HMAC_fp.h	349
6.122	/tpm/include/private/prototypes/HMAC_Start_fp.h	349
6.123	/tpm/include/private/prototypes/Import_fp.h	350
6.124	/tpm/include/private/prototypes/IncrementalSelfTest_fp.h	350
6.125	/tpm/include/private/prototypes/IOBuffers_fp.h	351
6.126	/tpm/include/private/prototypes/LoadExternal_fp.h	351
6.127	/tpm/include/private/prototypes/Load_fp.h	352
6.128	/tpm/include/private/prototypes/Locality_fp.h	353
6.129	/tpm/include/private/prototypes/MAC_fp.h	353
6.130	/tpm/include/private/prototypes/MAC_Start_fp.h	353
6.131	/tpm/include/private/prototypes/MakeCredential_fp.h	354
6.132	/tpm/include/private/prototypes/Marshal_fp.h	355
6.133	/tpm/include/private/prototypes/MathOnByteBuffers_fp.h	391
6.134	/tpm/include/private/prototypes/Memory_fp.h	393

6.135/tpm/include/private/prototypes/NvDynamic_fp.h	394
6.136/tpm/include/private/prototypes/NvReserved_fp.h	400
6.137/tpm/include/private/prototypes/NV_Certify_fp.h	401
6.138/tpm/include/private/prototypes/NV_ChangeAuth_fp.h	402
6.139/tpm/include/private/prototypes/NV_DefineSpace2_fp.h	402
6.140/tpm/include/private/prototypes/NV_DefineSpace_fp.h	403
6.141/tpm/include/private/prototypes/NV_Extend_fp.h	403
6.142/tpm/include/private/prototypes/NV_GlobalWriteLock_fp.h	404
6.143/tpm/include/private/prototypes/NV_Increment_fp.h	404
6.144/tpm/include/private/prototypes/NV_ReadLock_fp.h	404
6.145/tpm/include/private/prototypes/NV_ReadPublic2_fp.h	405
6.146/tpm/include/private/prototypes/NV_ReadPublic_fp.h	405
6.147/tpm/include/private/prototypes/NV_Read_fp.h	406
6.148/tpm/include/private/prototypes/NV_SetBits_fp.h	406
6.149/tpm/include/private/prototypes/NV_spt_fp.h	407
6.150/tpm/include/private/prototypes/NV_UndefineSpaceSpecial_fp.h	408
6.151/tpm/include/private/prototypes/NV_UndefineSpace_fp.h	409
6.152/tpm/include/private/prototypes/NV_WriteLock_fp.h	409
6.153/tpm/include/private/prototypes/NV_Write_fp.h	410
6.154/tpm/include/private/prototypes/ObjectChangeAuth_fp.h	410
6.155/tpm/include/private/prototypes/Object_fp.h	411
6.156/tpm/include/private/prototypes/Object_spt_fp.h	415
6.157/tpm/include/private/prototypes/PCR_Allocate_fp.h	420
6.158/tpm/include/private/prototypes/PCR_Event_fp.h	421
6.159/tpm/include/private/prototypes/PCR_Extend_fp.h	421
6.160/tpm/include/private/prototypes/PCR_fp.h	422
6.161/tpm/include/private/prototypes/PCR_Read_fp.h	425
6.162/tpm/include/private/prototypes/PCR_Reset_fp.h	426
6.163/tpm/include/private/prototypes/PCR_SetAuthPolicy_fp.h	426
6.164/tpm/include/private/prototypes/PCR_SetAuthValue_fp.h	427
6.165/tpm/include/private/prototypes/PolicyAuthorizeNV_fp.h	427
6.166/tpm/include/private/prototypes/PolicyAuthorize_fp.h	428
6.167/tpm/include/private/prototypes/PolicyAuthValue_fp.h	428
6.168/tpm/include/private/prototypes/PolicyCapability_fp.h	428
6.169/tpm/include/private/prototypes/PolicyCommandCode_fp.h	429
6.170/tpm/include/private/prototypes/PolicyCounterTimer_fp.h	429
6.171/tpm/include/private/prototypes/PolicyCpHash_fp.h	430
6.172/tpm/include/private/prototypes/PolicyDuplicationSelect_fp.h	430
6.173/tpm/include/private/prototypes/PolicyGetDigest_fp.h	431
6.174/tpm/include/private/prototypes/PolicyLocality_fp.h	431
6.175/tpm/include/private/prototypes/PolicyNameHash_fp.h	432
6.176/tpm/include/private/prototypes/PolicyNvWritten_fp.h	432
6.177/tpm/include/private/prototypes/PolicyNV_fp.h	433
6.178/tpm/include/private/prototypes/PolicyOR_fp.h	433
6.179/tpm/include/private/prototypes/PolicyParameters_fp.h	434
6.180/tpm/include/private/prototypes/PolicyPassword_fp.h	434
6.181/tpm/include/private/prototypes/PolicyPCR_fp.h	434
6.182/tpm/include/private/prototypes/PolicyPhysicalPresence_fp.h	435
6.183/tpm/include/private/prototypes/PolicyRestart_fp.h	435
6.184/tpm/include/private/prototypes/PolicySecret_fp.h	436
6.185/tpm/include/private/prototypes/PolicySigned_fp.h	436
6.186/tpm/include/private/prototypes/PolicyTemplate_fp.h	437
6.187/tpm/include/private/prototypes/PolicyTicket_fp.h	437
6.188/tpm/include/private/prototypes/Policy_AC_SendSelect_fp.h	438
6.189/tpm/include/private/prototypes/Policy_spt_fp.h	438
6.190/tpm/include/private/prototypes/Power_fp.h	439
6.191/tpm/include/private/prototypes/PP_Commands_fp.h	440
6.192/tpm/include/private/prototypes/PP_fp.h	440

6.193/tpm/include/private/prototypes/PropertyCap_fp.h	441
6.194/tpm/include/private/prototypes/Quote_fp.h	442
6.195/tpm/include/private/prototypes/ReadClock_fp.h	442
6.196/tpm/include/private/prototypes/ReadPublic_fp.h	443
6.197/tpm/include/private/prototypes/ResponseCodeProcessing_fp.h	443
6.198/tpm/include/private/prototypes/Response_fp.h	444
6.199/tpm/include/private/prototypes/Rewrap_fp.h	444
6.200/tpm/include/private/prototypes/RsaKeyCache_fp.h	444
6.201/tpm/include/private/prototypes/RSA_Decrypt_fp.h	445
6.202/tpm/include/private/prototypes/RSA_Encrypt_fp.h	446
6.203/tpm/include/private/prototypes/SelfTest_fp.h	446
6.204/tpm/include/private/prototypes/SequenceComplete_fp.h	447
6.205/tpm/include/private/prototypes/SequenceUpdate_fp.h	447
6.206/tpm/include/private/prototypes/SessionProcess_fp.h	448
6.207/tpm/include/private/prototypes/Session_fp.h	449
6.208/tpm/include/private/prototypes/SetAlgorithmSet_fp.h	452
6.209/tpm/include/private/prototypes/SetCapability_fp.h	453
6.210/tpm/include/private/prototypes/SetCommandCodeAuditStatus_fp.h	453
6.211/tpm/include/private/prototypes/SetPrimaryPolicy_fp.h	454
6.212/tpm/include/private/prototypes/Shutdown_fp.h	454
6.213/tpm/include/private/prototypes/Sign_fp.h	455
6.214/tpm/include/private/prototypes/StartAuthSession_fp.h	455
6.215/tpm/include/private/prototypes/Startup_fp.h	456
6.216/tpm/include/private/prototypes/StirRandom_fp.h	456
6.217/tpm/include/private/prototypes/TableDrivenMarshal_fp.h	457
6.218/tpm/include/private/prototypes/TestParms_fp.h	458
6.219/tpm/include/private/prototypes/Ticket_fp.h	458
6.220/tpm/include/private/prototypes/Time_fp.h	459
6.221/tpm/include/private/prototypes/TpmASN1_fp.h	460
6.222/tpm/include/private/prototypes/TpmEcc_Signature_ECDSA_fp.h	463
6.223/tpm/include/private/prototypes/TpmEcc_Signature_ECDSA_fp.h	463
6.224/tpm/include/private/prototypes/TpmEcc_Signature_Schnorr_fp.h	464
6.225/tpm/include/private/prototypes/TpmEcc_Signature_SM2_fp.h	464
6.226/tpm/include/private/prototypes/TpmEcc_Signature_Util_fp.h	465
6.227/tpm/include/private/prototypes/TpmEcc_Util_fp.h	465
6.228/tpm/include/private/prototypes/TpmMath_Debug_fp.h	466
6.229/tpm/include/private/prototypes/TpmMath_Util_fp.h	466
6.230/tpm/include/private/prototypes/TpmSizeChecks_fp.h	467
6.231/tpm/include/private/prototypes/Unseal_fp.h	468
6.232/tpm/include/private/prototypes/Vendor_TCG_Test_fp.h	468
6.233/tpm/include/private/prototypes/VerifySignature_fp.h	469
6.234/tpm/include/private/prototypes/X509_ECC_fp.h	469
6.235/tpm/include/private/prototypes/X509_RSA_fp.h	470
6.236/tpm/include/private/prototypes/X509_spt_fp.h	470
6.237/tpm/include/private/prototypes/ZGen_2Phase_fp.h	471
6.238/tpm/include/public/ACT.h	472
6.239/tpm/include/public/BaseTypes.h	475
6.240/tpm/include/public/Capabilities.h	476
6.241/tpm/include/public/CompilerDependencies.h	476
6.242/tpm/include/public/CompilerDependencies_gcc.h	477
6.243/tpm/include/public/CompilerDependencies_msvc.h	477
6.244/tpm/include/public/endian_swap.h	479
6.245/tpm/include/public/GpMacros.h	480
6.246/tpm/include/public/MinMax.h	487
6.247/tpm/include/public/TpmAlgorithmDefines.h	487
6.248/tpm/include/public/TPMB.h	490
6.249/tpm/include/public/TpmCalculatedAttributes.h	491
6.250/tpm/include/public/TpmTypes.h	493

6.251	/tpm/include/public/tpm_public.h	542
6.252	/tpm/include/public/tpm_radix.h	542
6.253	/tpm/include/public/VerifyConfiguration.h	544
6.254	/tpm/include/public/prototypes/TpmFail_fp.h	545
7	TPM Reference Implementation Source Files	547
7.1	/tpm/src/command/Asymmetric/ECC_Decrypt.c	547
7.2	/tpm/src/command/Asymmetric/ECC_Encrypt.c	547
7.3	/tpm/src/command/Asymmetric/ECC_Parameters.c	548
7.4	/tpm/src/command/Asymmetric/ECDH_KeyGen.c	548
7.5	/tpm/src/command/Asymmetric/ECDH_ZGen.c	549
7.6	/tpm/src/command/Asymmetric/EC_Ephemeral.c	550
7.7	/tpm/src/command/Asymmetric/RSA_Decrypt.c	551
7.8	/tpm/src/command/Asymmetric/RSA_Encrypt.c	552
7.9	/tpm/src/command/Asymmetric/ZGen_2Phase.c	553
7.10	/tpm/src/command/AttachedComponent/AC_GetCapability.c	554
7.11	/tpm/src/command/AttachedComponent/AC_Send.c	554
7.12	/tpm/src/command/AttachedComponent/AC_spt.c	555
7.13	/tpm/src/command/AttachedComponent/Policy_AC_SendSelect.c	557
7.14	/tpm/src/command/Attestation/Attest_spt.c	558
7.15	/tpm/src/command/Attestation/Certify.c	561
7.16	/tpm/src/command/Attestation/CertifyCreation.c	562
7.17	/tpm/src/command/Attestation/CertifyX509.c	563
7.18	/tpm/src/command/Attestation/GetCommandAuditDigest.c	567
7.19	/tpm/src/command/Attestation/GetSessionAuditDigest.c	568
7.20	/tpm/src/command/Attestation/GetTime.c	569
7.21	/tpm/src/command/Attestation/Quote.c	570
7.22	/tpm/src/command/Capability/GetCapability.c	571
7.23	/tpm/src/command/Capability/SetCapability.c	573
7.24	/tpm/src/command/Capability/TestParms.c	574
7.25	/tpm/src/command/ClockTimer/ACT_SetTimeout.c	574
7.26	/tpm/src/command/ClockTimer/ACT_spt.c	575
7.27	/tpm/src/command/ClockTimer/ClockRateAdjust.c	579
7.28	/tpm/src/command/ClockTimer/ClockSet.c	579
7.29	/tpm/src/command/ClockTimer/ReadClock.c	580
7.30	/tpm/src/command/CommandAudit/SetCommandCodeAuditStatus.c	580
7.31	/tpm/src/command/Context/ContextLoad.c	581
7.32	/tpm/src/command/Context/ContextSave.c	584
7.33	/tpm/src/command/Context/Context_spt.c	587
7.34	/tpm/src/command/Context/EvictControl.c	590
7.35	/tpm/src/command/Context/FlushContext.c	592
7.36	/tpm/src/command/DA/DictionaryAttackLockReset.c	593
7.37	/tpm/src/command/DA/DictionaryAttackParameters.c	593
7.38	/tpm/src/command/Duplication/Duplicate.c	594
7.39	/tpm/src/command/Duplication/Import.c	596
7.40	/tpm/src/command/Duplication/Rewrap.c	599
7.41	/tpm/src/command/EA/PolicyAuthorize.c	601
7.42	/tpm/src/command/EA/PolicyAuthorizeNV.c	603
7.43	/tpm/src/command/EA/PolicyAuthValue.c	604
7.44	/tpm/src/command/EA/PolicyCapability.c	605
7.45	/tpm/src/command/EA/PolicyCommandCode.c	609
7.46	/tpm/src/command/EA/PolicyCounterTimer.c	610
7.47	/tpm/src/command/EA/PolicyCpHash.c	611
7.48	/tpm/src/command/EA/PolicyDuplicationSelect.c	612
7.49	/tpm/src/command/EA/PolicyGetDigest.c	614
7.50	/tpm/src/command/EA/PolicyLocality.c	614
7.51	/tpm/src/command/EA/PolicyNameHash.c	616
7.52	/tpm/src/command/EA/PolicyNV.c	617
7.53	/tpm/src/command/EA/PolicyNvWritten.c	618

7.54	/tpm/src/command/EA/PolicyOR.c	619
7.55	/tpm/src/command/EA/PolicyParameters.c	620
7.56	/tpm/src/command/EA/PolicyPassword.c	621
7.57	/tpm/src/command/EA/PolicyPCR.c	622
7.58	/tpm/src/command/EA/PolicyPhysicalPresence.c	623
7.59	/tpm/src/command/EA/PolicySecret.c	624
7.60	/tpm/src/command/EA/PolicySigned.c	626
7.61	/tpm/src/command/EA/PolicyTemplate.c	628
7.62	/tpm/src/command/EA/PolicyTicket.c	629
7.63	/tpm/src/command/EA/Policy_spt.c	631
7.64	/tpm/src/command/Ecdsa/Commit.c	635
7.65	/tpm/src/command/FieldUpgrade/FieldUpgradeData.c	637
7.66	/tpm/src/command/FieldUpgrade/FieldUpgradeStart.c	637
7.67	/tpm/src/command/FieldUpgrade/FirmwareRead.c	637
7.68	/tpm/src/command/HashHMAC/EventSequenceComplete.c	638
7.69	/tpm/src/command/HashHMAC/HashSequenceStart.c	639
7.70	/tpm/src/command/HashHMAC/HMAC_Start.c	640
7.71	/tpm/src/command/HashHMAC/MAC_Start.c	641
7.72	/tpm/src/command/HashHMAC/SequenceComplete.c	642
7.73	/tpm/src/command/HashHMAC/SequenceUpdate.c	643
7.74	/tpm/src/command/Hierarchy/ChangeEPS.c	644
7.75	/tpm/src/command/Hierarchy/ChangePPS.c	645
7.76	/tpm/src/command/Hierarchy/Clear.c	646
7.77	/tpm/src/command/Hierarchy/ClearControl.c	648
7.78	/tpm/src/command/Hierarchy/CreatePrimary.c	648
7.79	/tpm/src/command/Hierarchy/HierarchyChangeAuth.c	650
7.80	/tpm/src/command/Hierarchy/HierarchyControl.c	651
7.81	/tpm/src/command/Hierarchy/SetPrimaryPolicy.c	653
7.82	/tpm/src/command/Misc/PP_Commands.c	654
7.83	/tpm/src/command/Misc/SetAlgorithmSet.c	655
7.84	/tpm/src/command/NVStorage/NV_Certify.c	655
7.85	/tpm/src/command/NVStorage/NV_ChangeAuth.c	657
7.86	/tpm/src/command/NVStorage/NV_DefineSpace.c	658
7.87	/tpm/src/command/NVStorage/NV_DefineSpace2.c	658
7.88	/tpm/src/command/NVStorage/NV_Extend.c	659
7.89	/tpm/src/command/NVStorage/NV_GlobalWriteLock.c	661
7.90	/tpm/src/command/NVStorage/NV_Increment.c	661
7.91	/tpm/src/command/NVStorage/NV_Read.c	662
7.92	/tpm/src/command/NVStorage/NV_ReadLock.c	663
7.93	/tpm/src/command/NVStorage/NV_ReadPublic.c	664
7.94	/tpm/src/command/NVStorage/NV_ReadPublic2.c	665
7.95	/tpm/src/command/NVStorage/NV_SetBits.c	665
7.96	/tpm/src/command/NVStorage/NV_spt.c	666
7.97	/tpm/src/command/NVStorage/NV_UndefineSpace.c	673
7.98	/tpm/src/command/NVStorage/NV_UndefineSpaceSpecial.c	674
7.99	/tpm/src/command/NVStorage/NV_Write.c	674
7.100	/tpm/src/command/NVStorage/NV_WriteLock.c	675
7.101	/tpm/src/command/Object/ActivateCredential.c	676
7.102	/tpm/src/command/Object/Create.c	677
7.103	/tpm/src/command/Object/CreateLoaded.c	679
7.104	/tpm/src/command/Object/Load.c	683
7.105	/tpm/src/command/Object/LoadExternal.c	684
7.106	/tpm/src/command/Object/MakeCredential.c	686
7.107	/tpm/src/command/Object/ObjectChangeAuth.c	687
7.108	/tpm/src/command/Object/Object_spt.c	687
7.109	/tpm/src/command/Object/ReadPublic.c	712
7.110	/tpm/src/command/Object/Unseal.c	713
7.111	/tpm/src/command/PCR/PCR_Allocate.c	713

7.112/tpm/src/command/PCR/PCR_Event.c	714
7.113/tpm/src/command/PCR/PCR_Extend.c	715
7.114/tpm/src/command/PCR/PCR_Read.c	716
7.115/tpm/src/command/PCR/PCR_Reset.c	716
7.116/tpm/src/command/PCR/PCR_SetAuthPolicy.c	717
7.117/tpm/src/command/PCR/PCR_SetAuthValue.c	718
7.118/tpm/src/command/Random/GetRandom.c	718
7.119/tpm/src/command/Random/StirRandom.c	719
7.120/tpm/src/command/Session/PolicyRestart.c	719
7.121/tpm/src/command/Session/StartAuthSession.c	720
7.122/tpm/src/command/Signature/Sign.c	722
7.123/tpm/src/command/Signature/VerifySignature.c	723
7.124/tpm/src/command/Startup/Shutdown.c	724
7.125/tpm/src/command/Startup/Startup.c	725
7.126/tpm/src/command/Symmetric/EncryptDecrypt.c	728
7.127/tpm/src/command/Symmetric/EncryptDecrypt2.c	731
7.128/tpm/src/command/Symmetric/EncryptDecrypt_spt.c	731
7.129/tpm/src/command/Symmetric/Hash.c	733
7.130/tpm/src/command/Symmetric/HMAC.c	734
7.131/tpm/src/command/Symmetric/MAC.c	736
7.132/tpm/src/command/Testing/GetTestResult.c	737
7.133/tpm/src/command/Testing/IncrementalSelfTest.c	737
7.134/tpm/src/command/Testing/SelfTest.c	738
7.135/tpm/src/command/Vendor/Vendor_TCG_Test.c	738
7.136/tpm/src/crypt/AlgorithmTests.c	738
7.137/tpm/src/crypt/CryptCmac.c	752
7.138/tpm/src/crypt/CryptEccCrypt.c	754
7.139/tpm/src/crypt/CryptEccData.c	757
7.140/tpm/src/crypt/CryptEccKeyExchange.c	758
7.141/tpm/src/crypt/CryptEccMain.c	763
7.142/tpm/src/crypt/CryptEccSignature.c	774
7.143/tpm/src/crypt/CryptHash.c	778
7.144/tpm/src/crypt/CryptPrime.c	790
7.145/tpm/src/crypt/CryptPrimeSieve.c	796
7.146/tpm/src/crypt/CryptRand.c	804
7.147/tpm/src/crypt/CryptRsa.c	818
7.148/tpm/src/crypt/CryptSelfTest.c	839
7.149/tpm/src/crypt/CryptSmac.c	842
7.150/tpm/src/crypt/CryptSym.c	843
7.151/tpm/src/crypt/CryptUtil.c	850
7.152/tpm/src/crypt/PrimeData.c	879
7.153/tpm/src/crypt/RsaKeyCache.c	884
7.154/tpm/src/crypt/Ticket.c	887
7.155/tpm/src/crypt/ecc/TpmEcc_Signature_ECDSA.c	891
7.156/tpm/src/crypt/ecc/TpmEcc_Signature_ECDSA.c	893
7.157/tpm/src/crypt/ecc/TpmEcc_Signature_Schnorr.c	896
7.158/tpm/src/crypt/ecc/TpmEcc_Signature_SM2.c	898
7.159/tpm/src/crypt/ecc/TpmEcc_Signature_Util.c	901
7.160/tpm/src/crypt/ecc/TpmEcc_Util.c	902
7.161/tpm/src/crypt/math/TpmMath_Debug.c	903
7.162/tpm/src/crypt/math/TpmMath_Util.c	904
7.163/tpm/src/events/_TPM_Hash_Data.c	907
7.164/tpm/src/events/_TPM_Hash_End.c	908
7.165/tpm/src/events/_TPM_Hash_Start.c	909
7.166/tpm/src/events/_TPM_Init.c	909
7.167/tpm/src/main/CommandDispatcher.c	910
7.168/tpm/src/main/ExecCommand.c	917
7.169/tpm/src/main/SessionProcess.c	921

7.170/tpm/src/subsystem/CommandAudit.c	954
7.171/tpm/src/subsystem/DA.c	957
7.172/tpm/src/subsystem/Hierarchy.c	960
7.173/tpm/src/subsystem/NvDynamic.c	968
7.174/tpm/src/subsystem/NvReserved.c	995
7.175/tpm/src/subsystem/Object.c	999
7.176/tpm/src/subsystem/PCR.c	1013
7.177/tpm/src/subsystem/PP.c	1032
7.178/tpm/src/subsystem/Session.c	1034
7.179/tpm/src/subsystem/Time.c	1050
7.180/tpm/src/support/AlgorithmCap.c	1053
7.181/tpm/src/support/Bits.c	1057
7.182/tpm/src/support/CommandCodeAttributes.c	1058
7.183/tpm/src/support/Entity.c	1066
7.184/tpm/src/support/Global.c	1073
7.185/tpm/src/support/Handle.c	1074
7.186/tpm/src/support/IOBuffers.c	1078
7.187/tpm/src/support/Locality.c	1079
7.188/tpm/src/support/Manufacture.c	1079
7.189/tpm/src/support/Marshal.c	1082
7.190/tpm/src/support/MathOnByteBuffers.c	1192
7.191/tpm/src/support/Memory.c	1195
7.192/tpm/src/support/Power.c	1198
7.193/tpm/src/support/PropertyCap.c	1199
7.194/tpm/src/support/Response.c	1208
7.195/tpm/src/support/ResponseCodeProcessing.c	1209
7.196/tpm/src/support/TableDrivenMarshal.c	1209
7.197/tpm/src/support/TableMarshalData.c	1222
7.198/tpm/src/support/TpmFail.c	1243
7.199/tpm/src/support/TpmSizeChecks.c	1248
7.200/tpm/src/X509/TpmASN1.c	1251
7.201/tpm/src/X509/X509_ECC.c	1258
7.202/tpm/src/X509/X509_RSA.c	1259
7.203/tpm/src/X509/X509_spt.c	1262
Annex A (informative) Implementation Dependent	1267
A.1 Introduction	1267
A.1.1 /TpmConfiguration/TpmConfiguration/TpmBuildSwitches.h	1267
A.1.2 /TpmConfiguration/TpmConfiguration/TpmProfile.h	1270
A.1.3 /TpmConfiguration/TpmConfiguration/TpmProfile_CommandList.h	1270
A.1.4 /TpmConfiguration/TpmConfiguration/TpmProfile_Common.h	1273
A.1.5 /TpmConfiguration/TpmConfiguration/TpmProfile_ErrorCodes.h	1276
A.1.6 /TpmConfiguration/TpmConfiguration/TpmProfile_Misc.h	1277
A.1.7 /TpmConfiguration/TpmConfiguration/VendorInfo.h	1278
Annex B (informative) Library-Specific	1280
B.1 Introduction	1280
B.2 Common Cryptographic Functionality	1280
B.2.1 /tpm/cryptolib/common/include/MathLibraryInterface.h	1280
B.2.2 /tpm/cryptolib/common/include/MathLibraryInterfaceTypes.h	1286
B.3 TpmBigNum	1287
B.3.1 /tpm/cryptolib/TpmBigNum/BnConvert.c	1287
B.3.2 /tpm/cryptolib/TpmBigNum/BnEccConstants.c	1290
B.3.3 /tpm/cryptolib/TpmBigNum/BnMath.c	1293
B.3.4 /tpm/cryptolib/TpmBigNum/BnMemory.c	1301
B.3.5 /tpm/cryptolib/TpmBigNum/BnUtil.c	1303
B.3.6 /tpm/cryptolib/TpmBigNum/TpmBigNum.h	1303

B.3.7.	/tpm/cryptolib/TpmBigNum/TpmBigNumThunks.c	1303
B.3.8.	/tpm/cryptolib/TpmBigNum/include/BnConvert_fp.h	1310
B.3.9.	/tpm/cryptolib/TpmBigNum/include/BnMath_fp.h	1312
B.3.10.	/tpm/cryptolib/TpmBigNum/include/BnMemory_fp.h	1313
B.3.11.	/tpm/cryptolib/TpmBigNum/include/BnSupport_Interface.h	1314
B.3.12.	/tpm/cryptolib/TpmBigNum/include/BnUtil_fp.h	1316
B.3.13.	/tpm/cryptolib/TpmBigNum/include/BnValues.h	1316
B.3.14.	/tpm/cryptolib/TpmBigNum/include/TpmBigNum/TpmToTpmBigNumMath.h ..	1321
B.4	OpenSSL-Specific Files	1322
B.4.1.	Introduction	1322
B.4.1.1.	/tpm/cryptolib/Ossl/BnToOsslMath.c	1322
B.4.1.2.	/tpm/cryptolib/Ossl/TpmToOsslSupport.c	1331
B.4.1.3.	/tpm/cryptolib/Ossl/include/BnOssl.h	1332
B.4.1.4.	/tpm/cryptolib/Ossl/include/Ossl/BnToOsslMath.h	1332
B.4.1.5.	/tpm/cryptolib/Ossl/include/Ossl/BnToOsslMath_fp.h	1333
B.4.1.6.	/tpm/cryptolib/Ossl/include/Ossl/TpmToOsslHash.h	1334
B.4.1.7.	/tpm/cryptolib/Ossl/include/Ossl/TpmToOsslSupport_fp.h	1336
B.4.1.8.	/tpm/cryptolib/Ossl/include/Ossl/TpmToOsslSym.h	1337
Annex C (informative)	Simulation Environment	1340
C.1	Introduction	1340
C.1.1.	/Platform/include/Platform.h	1340
C.1.2.	/Platform/include/PlatformACT.h	1340
C.1.3.	/Platform/include/PlatformClock.h	1342
C.1.4.	/Platform/include/PlatformData.h	1343
C.1.5.	/Platform/include/prototypes/platform_public_interface.h	1344
C.1.6.	/Platform/src/Cancel.c	1346
C.1.7.	/Platform/src/Clock.c	1347
C.1.8.	/Platform/src/DebugHelpers.c	1351
C.1.9.	/Platform/src/Entropy.c	1352
C.1.10.	/Platform/src/ExtraData.c	1354
C.1.11.	/Platform/src/LocalityPlat.c	1354
C.1.12.	/Platform/src/NVMem.c	1355
C.1.13.	/Platform/src/PlatformACT.c	1361
C.1.14.	/Platform/src/PlatformData.c	1365
C.1.15.	/Platform/src/PlatformPcr.c	1365
C.1.16.	/Platform/src/PowerPlat.c	1367
C.1.17.	/Platform/src/PPPlat.c	1369
C.1.18.	/Platform/src/RunCommand.c	1369
C.1.19.	/Platform/src/Unique.c	1370
C.1.20.	/Platform/src/VendorInfo.c	1371
Annex D (informative)	Remote Procedure Interface	1374
D.1	Introduction	1374
D.1.1.	/Simulator/include/TpmTcpProtocol.h	1374
D.1.2.	/Simulator/include/prototypes/Simulator_fp.h	1375
D.1.3.	/Simulator/src/simulatorPrivate.h	1378
D.1.4.	/Simulator/src/simulator_sysheaders.h	1378
D.1.5.	/Simulator/src/TcpServer.c	1379
D.1.6.	/Simulator/src/	1391
D.1.7.	/Simulator/src/	1394

Trusted Platform Module Library

Part 4: Supporting Routines

1 Scope

Part 4 is provided in order to enable the explanation and demonstration, via the TCG's reference code (reproduced in Parts 3 and 4), of the functionality described in Parts 1, 2, and 3. The code in Parts 3 and 4 is not written or guaranteed to meet any level of conformance, nor does this specification require that a TPM meet any particular level of conformance. In some instances (e.g., firmware update), Part 4 cannot describe a compliant implementation. Therefore, an implementor of TPM 2.0 may decide to replace Part 4 code with vendor-specific code that enables compliance.

2 Terms and definitions

For the purposes of this document, the terms and definitions given in TPM 2.0 Part 1 apply.

3 Symbols and abbreviated terms

For the purposes of this document, the symbols and abbreviated terms given in TPM 2.0 Part 1 apply.

4 Automation

TPM 2.0 Part 2 and 3 are constructed so that they can be processed by an automated parser. For example, TPM 2.0 Part 2 can be processed to generate header file contents such as structures, typedefs, and enums. TPM 2.0 Part 3 can be processed to generate command and response marshaling and unmarshaling code.

The automated processor is not provided by the TCG. It was used to generate the Microsoft Visual Studio TPM simulator files. These files are not specification reference code, but rather design examples.

The automation produces `TPM_Types.h`, a header representing TPM 2.0 Part 2. It also produces, for each major clause of Part 4, a header of the form `_fp.h` with the function prototypes.

EXAMPLE The header file for `SessionProcess.c` is `SessionProcess_fp.h`.

4.1 Configuration Parser

The TPM configuration is largely defined by `TpmProfiles.h`. This file may be edited in order to change the algorithms and commands supported by a TPM implementation.

A parser exists to process a Word document that defines the TPM configuration. This parser is used to create `TpmProfiles.h`.

4.2 Structure Parser

4.2.1 Introduction

The program that processes the tables in TPM 2.0 Part 2 is called "The TPM 2.0 Part 2 Structure Parser."

NOTE A Perl script was used to parse the tables in TPM 2.0 Part 2 to produce the header files and unmarshaling code in for the reference implementation.

The TPM 2.0 Part 2 Structure Parser takes as input the files produced by the TPM 2.0 Part 2 Configuration Parser and the same TPM 2.0 Part 2 specification that was used as input to the TPM 2.0 Part 2 Configuration Parser. The TPM 2.0 Part 2 Structure Parser will generate all of the C structure

constant definitions that are required by the TPM interface. Additionally, the parser will generate unmarshaling code for all structures passed to the TPM and marshaling code for structures passed from the TPM.

The unmarshaling code produced by the parser uses the prototypes defined below. The unmarshaling code will perform validations of the data to ensure that it is compliant with the limitations on the data imposed by the structure definition and use the response code provided in the table if not.

EXAMPLE: The definition for a `TPMI_RH_PROVISION` indicates that the primitive data type is a `TPM_HANDLE` and the only allowed values are `TPM_RH_OWNER` and `TPM_RH_PLATFORM`. The definition also indicates that the TPM shall indicate `TPM_RC_HANDLE` if the input value is not none of these values. The unmarshaling code will validate that the input value has one of those allowed values and return `TPM_RC_HANDLE` if not.

The clauses below describe the function prototypes for the marshaling and unmarshaling code that is automatically generated by the TPM 2.0 Part 2 Structure Parser. These prototypes are described here as the unmarshaling and marshaling of various types occurs in places other than when the command is being parsed or the response is being built. The prototypes and the description of the interface are intended to aid in the comprehension of the code that uses these auto-generated routines.

4.2.2 Unmarshaling Code Prototype

4.2.2.1 Simple Types and Structures

The general form for the unmarshaling code for a simple type or a structure is:

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size);
```

Where:

TYPE	name of the data type or structure
*target	location in the TPM memory into which the data from **buffer is placed
**buffer	location in input buffer containing the most significant octet (MSO) of *target
*size	number of octets remaining in **buffer

When the data is successfully unmarshaled, the called routine will return `TPM_RC_SUCCESS`. Otherwise, it will return a Format-One response code (see TPM 2.0 Part 2).

If the data is successfully unmarshaled, ***buffer** is advanced point to the first octet of the next parameter in the input buffer and **size** is reduced by the number of octets removed from the buffer.

When the data type is a simple type, the parser will generate code that will unmarshal the underlying type and then perform checks on the type as indicated by the type definition.

When the data type is a structure, the parser will generate code that unmarshals each of the structure elements in turn and performs any additional parameter checks as indicated by the data type.

4.2.2.2 Union Types

When a union is defined, an extra parameter is defined for the unmarshaling code. This parameter is the selector for the type. The unmarshaling code for the union will unmarshal the type indicated by the selector.

The function prototype for a union has the form:

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size, UINT32 selector);
```

where:

TYPE	name of the union type or structure
*target	location in the TPM memory into which the data from **buffer is placed
**buffer	location in input buffer containing the most significant octet (MSO) of *target
*size	number of octets remaining in **buffer
selector	union selector that determines what will be unmarshaled into *target

4.2.2.3 Null Types

In some cases, the structure definition allows an optional “null” value. The “null” value allows the use of the same C type for the entity even though it does not always have the same members.

For example, the `TPM1_ALG_HASH` data type is used in many places. In some cases, `TPM1_ALG_NULL` is permitted and in some cases it is not. If two different data types had to be defined, the interfaces and code would become more complex because of the number of cast operations that would be necessary. Rather than encumber the code, the “null” value is defined and the unmarshaling code is given a flag to indicate if this instance of the type accepts the “null” parameter or not. When the data type has a “null” value, the function prototype is

```
TPM_RC TYPE_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size, BOOL flag);
```

The parser detects when the type allows a “null” value and will always include `flag` in any call to `unmarshal` that type. `flag TRUE` indicates that null is accepted.

4.2.2.4 Arrays

Any data type may be included in an array. The function prototype use to unmarshal an array for a `TYPE` is

```
TPM_RC TYPE_Array_Unmarshal(TYPE *target, BYTE **buffer, INT32 *size, INT32 count);
```

The generated code for an array uses a `count`-limited loop within which it calls the unmarshaling code for `TYPE`.

4.2.3 Marshaling Code Function Prototypes

4.2.3.1 Simple Types and Structures

The general form for the marshaling code for a simple type or a structure is:

```
UINT16 TYPE_Marshal(TYPE *source, BYTE **buffer, INT32 *size);
```

Where:

TYPE	name of the data type or structure
*source	location in the TPM memory containing the value that is to be marshaled in to the designated buffer
**buffer	location in the output buffer where the first octet of the <code>TYPE</code> is to be placed
*size	number of octets remaining in **buffer .

If `buffer` is a NULL pointer, then no data is marshaled, but the routine will compute and return the size of the memory required to marshal the indicated type. `*size` is not changed.

If **buffer** is not a NULL pointer, data is marshaled, ***buffer** is advanced to point to the first octet of the next location in the output buffer, and the called routine will return the number of octets marshaled into ****buffer**. This occurs even if **size** is a NULL pointer. If **size** is a not NULL pointer ***size** is reduced by the number of octets placed in the buffer.

When the data type is a simple type, the parser will generate code that will marshal the underlying type. The presumption is that the TPM internal structures are consistent and correct so the marshaling code does not validate that the data placed in the buffer has a permissible value. The presumption is also that the **size** is sufficient for the source being marshaled.

When the data type is a structure, the parser will generate code that marshals each of the structure elements in turn.

4.2.3.2 Union Types

An extra parameter is defined for the marshaling function of a union. This parameter is the selector for the type. The marshaling code for the union will marshal the type indicated by the selector.

The function prototype for a union has the form:

```
UINT16 TYPE_Marshal(TYPE *source, BYTE **buffer, INT32 *size, UINT32 selector);
```

The parameters have a similar meaning as those in 4.2.2.2 but the data movement is from **source** to **buffer**.

4.2.3.3 Arrays

Any type may be included in an array. The function prototype use to unmarshal an array is:

```
UINT16 TYPE_Array_Marshal(TYPE *source, BYTE **buffer, INT32 *size, INT32 count);
```

The generated code for an array uses a **count**-limited loop within which it calls the marshaling code for **TYPE**.

4.2.4 Table-driven Marshaling

The most recent versions of the TPM code includes the option to use table-driven marshaling rather than the procedural marshaling described in previous clauses in 4.2.2. The structure and processing of this code is complex and is provided in the code.

4.3 Part 3 Parsing

The Command / Response tables in Part 3 of this specification are processed by scripts to produce the command-specific data structures used by functions in this TPM 2.0 Part 4. They are:

- **CommandAttributeData.h** -- This file contains the command attributes reported by TPM2_GetCapability.
- **CommandAttributes.h** -- This file contains the definition of command attributes that are extracted by the parsing code. The file mainly exists to ensure that the parsing code and the function code are using the same attributes.
- **CommandDispatchData.h** -- This file contains the data definitions for the table driven version of the command dispatcher.

Part 3 parsing also produces special function prototype files as described in 4.4.

4.4 Function Prototypes

For functions that have entry definitions not defined by Part 3 tables, a script is used to extract function prototypes from the code. For each .c file that is not in Part 3, a file with the same name is created with a suffix of _fp.h. For example, the function prototypes for Create.c will be placed in a file called Create_fp.h. The _fp.h is added because some files have two types of associated headers: the one containing the function prototypes for the file and another containing definitions that are specific to that file.

In some cases, a function will be replaced by a macro. The macro is defined in the .c file and extracted by the function prototype processor. A special comment tag ("//%") is used to indicate that the line is to be included in the function prototype file. If the "//%" tag occurs at the start of the line, it is deleted. If it occurs later in the line, it is preserved. Removing the "//%" at the start of the line allows the macro to be placed in the .c file with the tag as a prefix, and then show up in the _fp.h file as the actual macro. This allows the code that includes that function prototype code to use the appropriate macro.

For files that contain the command actions, a special _fp.h file is created from the tables in Part 3. These files contain:

- the definition of the input and output structure of the function;
- definition of command-specific return code modifiers (parameter identifiers); and
- the function prototype for the command action function.

Create_fp.h (shown below) is prototypical of the command _fp.h files.

4.4.1 /tpm/include/private/prototypes/Create_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_Create  // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATE_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATE_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT      parentHandle;
12     TPM2B_SENSITIVE_CREATE inSensitive;
13     TPM2B_PUBLIC         inPublic;
14     TPM2B_DATA           outsideInfo;
15     TPML_PCR_SELECTION   creationPCR;
16 } Create_In;
17
18 // Output structure definition
19 typedef struct
20 {
21     TPM2B_PRIVATE        outPrivate;
22     TPM2B_PUBLIC          outPublic;
23     TPM2B_CREATION_DATA   creationData;
24     TPM2B_DIGEST          creationHash;
25     TPMT_TK_CREATION      creationTicket;
26 } Create_Out;
27
28 // Response code modifiers
29 #   define RC_Create_parentHandle (TPM_RC_H + TPM_RC_1)
30 #   define RC_Create_inSensitive (TPM_RC_P + TPM_RC_1)
31 #   define RC_Create_inPublic (TPM_RC_P + TPM_RC_2)
32 #   define RC_Create_outsideInfo (TPM_RC_P + TPM_RC_3)
33 #   define RC_Create_creationPCR (TPM_RC_P + TPM_RC_4)

```

```
34
35 // Function prototype
36 TPM_RC
37 TPM2_Create(Create_In* in, Create_Out* out);
38
39 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATE_FP_H_
40 #endif // CC_Create
41
```

4.5 Portability

Where reasonable, the code is written to be portable. There are a few known cases where the code is not portable. Specifically, the handling of bit fields will not always be portable. The bit fields are marshaled and unmarshaled as a simple element of the underlying type. For example, a `TPMA_SESSION` is defined as a bit field in an octet (`BYTE`). When sent on the interface a `TPMA_SESSION` will occupy one octet. When unmarshaled, it is unmarshaled as a `UINT8`. The ramifications of this are that a `TPMA_SESSION` will occupy the 0th octet of the structure in which it is placed regardless of the size of the structure.

Many compilers will pad a bit field to some "natural" size for the processor, often 4 octets, meaning that `sizeof(TPMA_SESSION)` would return 4 rather than 1 (the canonical size of a `TPMA_SESSION`).

For a little endian machine, padding of bit fields should have little consequence since the 0th octet always contains the 0th bit of the structure no matter how large the structure. However, for a big endian machine, the 0th bit will be in the highest numbered octet. When unmarshaling a `TPMA_SESSION`, the current unmarshaling code will place the input octet at the 0th octet of the `TPMA_SESSION`. Since the 0th octet is most significant octet, this has the effect of shifting all the session attribute bits left by 24 places.

As a consequence, someone implementing on a big endian machine should do one of two things:

- a) allocate all structures as packed to a byte boundary (this may not be possible if the processor does not handle unaligned accesses); or
- b) modify the code that manipulates bit fields that are not defined as being the alignment size of the system.

For many RISC processors, option #2 would be the only choice. This is may not be a terribly daunting task since only two attribute structures are not 32-bits (`TPMA_SESSION` and `TPMA_LOCALITY`).

5 Marshaling

5.1 Introduction

The marshaling and unmarshaling code and function prototypes are not listed, as the code is repetitive, long, and not very useful to read. Examples of a few unmarshaling routines are provided. Most of the others are similar.

Depending on the table header flags, a type will have an unmarshaling routine and a marshaling routine. The table header flags that control the generation of the unmarshaling and marshaling code are delimited by angle brackets (" $<>$ ") in the table header. If no brackets are present, then both unmarshaling and marshaling code is generated (i.e., generation of both marshaling and unmarshaling code is the default).

5.2 Unmarshal and Marshal a Value

In TPM 2.0 Part 2, a TPMI_DH_OBJECT is defined by this table:

Table xxx — Definition of (TPM_HANDLE) TPMI_DH_OBJECT Type

Values	Comments
{TRANSIENT_FIRST:TRANSIENT_LAST}	allowed range for transient objects
{PERSISTENT_FIRST:PERSISTENT_LAST}	allowed range for persistent objects
+TPM_RH_NULL	the null handle
#TPM_RC_VALUE	

This generates the following unmarshaling code:

```

TPM_RC
TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
                          BOOL flag)
{
    TPM_RC    result;
    result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    if(*target == TPM_RH_NULL)
    {
        if(flag)
            return TPM_RC_SUCCESS;
        else
            return TPM_RC_VALUE;
    }
    if((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
        &&((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST))
        return TPM_RC_VALUE;
    return TPM_RC_SUCCESS;
}

```

and the following marshaling code:

NOTE The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data .

```

UINT16
TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
{
    return UINT32_Marshal((UINT32 *)source, buffer, size);
}

```

An additional script is used to do the work that might be done by a linker or globally optimizing compiler. It searches for functions like `TPMI_DH_OBJECT_Marshal()` that do nothing but call another function and replaces the function with a `#define`.

```
#define TPMI_DH_OBJECT_Marshal(source, buffer, size) \
    UINT32_Marshal((UINT32 *)source, buffer, size)
```

When replacing the function with a `#define`, the `#define` is placed in `marshal_fp.h` and the function body is removed from `marshal.c`.

5.3 Unmarshal and Marshal a Union

In TPM 2.0 Part 2, a `TPMU_PUBLIC_PARMS` union is defined by:

Table xxx — Definition of TPMU_PUBLIC_PARMS Union <IN/OUT, S>

Parameter	Type	Selector	Description
keyedHash	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH	sign encrypt neither
symDetail	TPMT_SYM_DEF_OBJECT	TPM_ALG_SYMCIPHER	a symmetric block cipher
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA	decrypt + sign
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC	decrypt + sign
asymDetail	TPMS_ASYM_PARMS		common scheme structure for RSA and ECC keys

NOTE The Description column indicates which of `TPMA_OBJECT.decrypt` or `TPMA_OBJECT.sign` may be set. "+" indicates that both may be set but one shall be set. "|" indicates the optional settings.

From this table, the following unmarshaling code is generated.

```
TPM_RC
TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size,
    UINT32 selector)
{
    switch(selector) {
    #if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Unmarshal(
                (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
    #endif
    #if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Unmarshal(
                (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
    #endif
    #if ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Unmarshal(
                (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
    #endif
    #if ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Unmarshal(
                (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
    #endif
    }
    return TPM_RC_SELECTOR;
}
```


NOTE The `#if/#endif` directives are added whenever a value is dependent on an algorithm ID so that removing the algorithm definition will remove the related code.

The marshaling code for the union is:

```
UINT16
TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
                          UINT32 selector)
{
    switch(selector) {
    #if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Marshal(
                (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
    #endif
    #if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Marshal(
                (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
    #endif
    #if ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Marshal(
                (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
    #endif
    #if ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Marshal(
                (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
    #endif
    }
    assert(1);
    return 0;
}
```

For the marshaling and unmarshaling code, a value in the structure containing the union provides the value used for *selector*. The example in the next section illustrates this.

5.4 Unmarshal and Marshal a Structure

In TPM 2.0 Part 2, the TPMT_PUBLIC structure is defined by:

Table xxx — Definition of TPMT_PUBLIC Structure

Parameter	Type	Description
type	TPMI_ALG_PUBLIC	“algorithm” associated with this object
nameAlg	+TPMI_ALG_HASH	algorithm used for computing the Name of the object NOTE The "+" indicates that the instance of a TPMT_PUBLIC may have a "+" to indicate that the nameAlg may be TPM_ALG_NULL.
objectAttributes	TPMA_OBJECT	attributes that, along with <i>type</i> , determine the manipulations of this object
authPolicy	TPM2B_DIGEST	optional policy for using this key The policy is computed using the <i>nameAlg</i> of the object. NOTE shall be the Empty Buffer if no authorization policy is present
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

This structure is tagged (the first value indicates the structure type), and that tag is used to determine how the parameters and unique fields are unmarshaled and marshaled. The use of the type for specifying the union selector is emphasized below.

The unmarshaling code for the structure in the table above is:

```

TPM_RC
TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
{
    TPM_RC    result;
    result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
                                         buffer, size);

    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
                                       buffer, size, flag);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
                                    buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
                                     buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
                                          buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;

    result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
                                       buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}

```

The marshaling code for the TPMT_PUBLIC structure is:

```

UINT16
TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16) (result + TPMI_ALG_PUBLIC_Marshal(
        (TPMI_ALG_PUBLIC *) &(source->type), buffer, size));
    result = (UINT16) (result + TPMI_ALG_HASH_Marshal(
        (TPMI_ALG_HASH *) &(source->nameAlg), buffer, size));
    result = (UINT16) (result + TPMA_OBJECT_Marshal(
        (TPMA_OBJECT *) &(source->objectAttributes), buffer, size));

    result = (UINT16) (result + TPM2B_DIGEST_Marshal(
        (TPM2B_DIGEST *) &(source->authPolicy), buffer, size));

    result = (UINT16) (result + TPMU_PUBLIC_PARMS_Marshal(
        (TPMU_PUBLIC_PARMS *) &(source->parameters), buffer, size,
        (UINT32) source->type));

    result = (UINT16) (result + TPMU_PUBLIC_ID_Marshal(
        (TPMU_PUBLIC_ID *) &(source->unique), buffer, size,
        (UINT32) source->type));

    return result;
}

```

5.5 Unmarshal and Marshal an Array

In TPM 2.0 Part 2, the TPML_DIGEST is defined by:

Table xxx — Definition of TPML_DIGEST Structure

Parameter	Type	Description
count {2:}	UINT32	number of digests in the list, minimum is two
digests[count]{:8}	TPM2B_DIGEST	a list of digests For TPM2_PolicyOR(), all digests will have been computed using the digest of the policy session. For TPM2_PCR_Read(), each digest will be the size of the digest for the bank containing the PCR.
#TPM_RC_SIZE		response code when count is not at least two or is greater than 8

The *digests* parameter is an array of up to *count* structures (TPM2B_DIGESTS). The auto-generated code to Unmarshal this structure is:

```

TPM_RC
TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT32_Unmarshal((UINT32 *) &(target->count), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;

    if( (target->count < 2))           // This check is triggered by the {2:} notation
                                     // on 'count'
        return TPM_RC_SIZE;

    if((target->count) > 8)           // This check is triggered by the {:8} notation
                                     // on 'digests'.

```

```

        return TPM_RC_SIZE;

    result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *) (target->digests),
                                          buffer, size, (INT32) (target->count));
    if(result != TPM_RC_SUCCESS)
        return result;

    return TPM_RC_SUCCESS;
}

```

The routine unmarshals a *count* value and passes that value to a routine that unmarshals an array of TPM2B_DIGEST values. The unmarshaling code for the array is:

```

TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}

```

Marshaling of the TPML_DIGEST uses a similar scheme with a structure specifying the number of elements in an array and a subsequent call to a routine to marshal an array of that type.

```

UINT16
TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
{
    UINT16    result = 0;
    result = (UINT16) (result + UINT32_Marshal((UINT32 *) &(source->count), buffer,
                                              size));
    result = (UINT16) (result + TPM2B_DIGEST_Array_Marshal(
        (TPM2B_DIGEST *) (source->digests), buffer, size,
        (INT32) (source->count)));

    return result;
}

```

The marshaling code for the array is:

```

TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
                             INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}

```

5.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 5.5 but the unmarshaling/marshaling is to a union element. Each TPM2B is a union of two sized buffers, one of which is type specific (the 't' element) and the other is a generic value (the 'b' element). This allows each of the TPM2B structures to have some inheritance property with all other TPM2B. The purpose is to allow functions that have parameters that can be any TPM2B structure while allowing other functions to be specific about the type of the TPM2B that is used. When the generic structure is allowed, the input parameter would use the 'b' element and when the type-specific structure is required, the 't' element is used.

When marshaling a TPM2B where the second member is a BYTE array, the size parameter indicates the size of the array. The second member can also be a structure. In this case, the caller does not prefill the size member. The marshaling code must marshal the structure and then back fill the calculated size.

Table xxx — Definition of TPM2B_EVENT Structure

Parameter	Type	Description
size	UINT16	Size of the operand
buffer [size] {:1024}	BYTE	The operand

TPM_RC

```

TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
{
    TPM_RC    result;
    result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    // if size equal to 0, the rest of the structure is a zero buffer
    // so stop processing
    if(target->t.size == 0)
        return TPM_RC_SUCCESS;
    if((target->t.size) > 1024)    // This check is triggered by the {:1024}
                                // notation on 'buffer'
        return TPM_RC_SIZE;
    result = BYTE_Array_Unmarshal((BYTE *) (target->t.buffer), buffer, size,
                                (INT32) (target->t.size));
    if(result != TPM_RC_SUCCESS)
        return result;
    return TPM_RC_SUCCESS;
}

```

using these structure definitions:

```

typedef union {
    struct {
        UINT16    size;
        BYTE      buffer[1024];
    }
    TPM2B        b;
} TPM2B_EVENT;

```

6 TPM Reference Implementation Include Files

6.1 /tpm/include/platform_interface/pcrstruct.h

```

1  //
2  // This file defines the PCR and PCR_Attributes structures and
3  // related interface functions
4  //
5
6  #ifndef _PCRSTRUCT_H_
7  #define _PCRSTRUCT_H_
8
9  #include <public/BaseTypes.h>
10 #include <public/TpmAlgorithmDefines.h>
11 #include <public/TpmTypes.h>
12
13 // a single PCR
14 typedef struct
15 {
16     #if ALG_SHA1
17         BYTE Sha1Pcr[SHA1_DIGEST_SIZE];
18     #endif
19     #if ALG_SHA256
20         BYTE Sha256Pcr[SHA256_DIGEST_SIZE];
21     #endif
22     #if ALG_SHA384
23         BYTE Sha384[SHA384_DIGEST_SIZE];
24     #endif
25     #if ALG_SHA512
26         BYTE Sha512[SHA512_DIGEST_SIZE];
27     #endif
28     #if ALG_SM3_256
29         BYTE Sm3_256[SM3_256_DIGEST_SIZE];
30     #endif
31     #if ALG_SHA3_256
32         BYTE Sha3_256[SHA3_256_DIGEST_SIZE];
33     #endif
34     #if ALG_SHA3_384
35         BYTE Sha3_384[SHA3_384_DIGEST_SIZE];
36     #endif
37     #if ALG_SHA3_512
38         BYTE Sha3_512[SHA3_512_DIGEST_SIZE];
39     #endif
40 } PCR;
41
42 // see the comments below for supportsPolicyAuth to explain this
43 #define MAX_PCR_GROUP_BITS 3
44
45 typedef struct
46 {
47     // SET if the PCR value should be saved in state save
48     unsigned int stateSave : 1;
49
50     // SET if the PCR is part of the "TCB group", causes the PCR counter not to
increment
51     unsigned int doNotIncrementPcrCounter : 1;
52
53     // PCRs may support policy or auth-value authorization.
54     //
55     // Such authorization values, if supported, are set by
56     // TPM2_PCR_SetAuthPolicy and/or TPM2_PCR_SetAuthValue.
57     //
58     // PCRs that share the same policy/auth value are said to be in a "group".
59     // PCRs that don't support authorization are said to be in group Zero.

```

```

60 //
61 // Group numbers are only used internally to indicate which PCRs share an
62 // authorization value. IOW the TPM client cannot refer to PCRs by group
63 // number; the range of group numbers is implementation defined. zero
64 // indicates the PCR doesn't support policy or auth verification.
65 //
66 // The size of this field must be large enough to support
67 // NUM_POLICY_PCR_GROUP & NUM_AUTHVALUE_PCR_GROUP; the maximum number of groups
68 // actually supported by this build of the core library.
69 //
70 // The number of bits allocated here does not control the number of groups,
71 // but there is a static assert that the number of bits here is large
72 // enough.
73 unsigned int policyAuthGroup : MAX_PCR_GROUP_BITS;
74 unsigned int authValuesGroup : MAX_PCR_GROUP_BITS;
75
76 // these bitfields indicating the localities that can
77 // reset or extend this PCR. A SET bit indicates the PCR can
78 // be extended or reset from that locality. The low-order bit in
79 // each field is locality zero, and the high-order bit is locality 4.
80 unsigned int resetLocality : 5;
81 unsigned int extendLocality : 5;
82 } PCR_Attributes;
83
84 // Get pointer to particular PCR from array if that PCR is allocated.
85 // otherwise returns NULL
86 BYTE* GetPcrPointerIfAllocated(PCR* pPcrArray,
87                                TPM_ALG_ID alg, // IN: algorithm for bank
88                                UINT32 pcrNumber // IN: PCR number
89 );
90
91 // get a PCR pointer from the TPM's internal list, if it's allocated
92 // otherwise NULL
93 BYTE* GetPcrPointer(TPM_ALG_ID alg, // IN: algorithm for bank
94                    UINT32 pcrNumber // IN: PCR number
95 );
96
97 #endif

```

6.2 /tpm/include/platform_interface/platform_to_tpm_interface.h

```

1 #include "prototypes/_TPM_Hash_Data_fp.h"
2 #include "prototypes/_TPM_Hash_End_fp.h"
3 #include "prototypes/_TPM_Hash_Start_fp.h"
4 #include "prototypes/_TPM_Init_fp.h"
5 #include "prototypes/ExecCommand_fp.h"
6 #include "prototypes/Manufacture_fp.h"
7 // TODO_RENAME_INC_FOLDER: public refers to the TPM_CoreLib public headers
8 #include <public/prototypes/TpmFail_fp.h>

```

6.3 /tpm/include/platform_interface/tpm_to_platform_interface.h

```

1 // This file represents the functional interface that all platform libraries must
2 // provide because they are called by the Core TPM library.
3 #ifndef _TPM_TO_PLATFORM_INTERFACE_H_
4 #define _TPM_TO_PLATFORM_INTERFACE_H_
5
6 // need to read configuration for ACT_SUPPORT flag check below
7 #include <TpmConfiguration/TpmBuildSwitches.h>
8 #include <TpmConfiguration/TpmProfile.h>
9 #include <stddef.h>
10
11 /** From Cancel.c
12

```



```

13  /***_plat_IsCanceled()
14  // Check if the cancel flag is set
15  // Return Type: int
16  //     TRUE(1)         if cancel flag is set
17  //     FALSE(0)        if cancel flag is not set
18  LIB_EXPORT int _plat_IsCanceled(void);
19
20  /***_plat_TimerRead()
21  // This function provides access to the tick timer of the platform. The TPM code
22  // uses this value to drive the TPM Clock.
23  //
24  // The tick timer is supposed to run when power is applied to the device. This timer
25  // should not be reset by time events including _TPM_Init. It should only be reset
26  // when TPM power is re-applied.
27  //
28  // If the TPM is run in a protected environment, that environment may provide the
29  // tick time to the TPM as long as the time provided by the environment is not
30  // allowed to go backwards. If the time provided by the system can go backwards
31  // during a power discontinuity, then the _plat_Signal_PowerOn should call
32  // _plat_TimerReset().
33  LIB_EXPORT uint64_t _plat_TimerRead(void);
34
35  /***_plat_TimerWasReset()
36  // This function is used to interrogate the flag indicating if the tick timer has
37  // been reset.
38  //
39  // If the resetFlag parameter is SET, then the flag will be CLEAR before the
40  // function returns.
41  LIB_EXPORT int _plat_TimerWasReset(void);
42
43  /***_plat_TimerWasStopped()
44  // This function is used to interrogate the flag indicating if the tick timer has
45  // been stopped. If so, this is typically a reason to roll the nonce.
46  //
47  // This function will CLEAR the s_timerStopped flag before returning. This provides
48  // functionality that is similar to status register that is cleared when read. This
49  // is the model used here because it is the one that has the most impact on the TPM
50  // code as the flag can only be accessed by one entity in the TPM. Any other
51  // implementation of the hardware can be made to look like a read-once register.
52  LIB_EXPORT int _plat_TimerWasStopped(void);
53
54  /***_plat_ClockRateAdjust()
55  // Adjust the clock rate
56  // the old function name is ClockAdjustRate, and took a value which was an absolute
57  // number of ticks.
58  //
59  // ClockRateAdjust uses predefined signal values and encapsulates the platform
60  // specifics regarding the number of ticks the underlying clock is running at.
61  //
62  // The adjustment must be one of these values. A COARSE adjustment is 1%, MEDIUM
63  // is 0.1%, and FINE is the smallest amount supported by the platform. The
64  // total (cumulative) adjustment is limited to ~15% total. Attempts to adjust
65  // the clock further are silently ignored as are any invalid values. These
66  // values are defined here to insulate them from spec changes and to avoid
67  // needing visibility to the doc-generated structure headers.
68  typedef enum _plat_ClockAdjustStep
69  {
70      PLAT_TPM_CLOCK_ADJUST_COARSE_SLOWER = -3,
71      PLAT_TPM_CLOCK_ADJUST_MEDIUM_SLOWER = -2,
72      PLAT_TPM_CLOCK_ADJUST_FINE_SLOWER   = -1,
73      PLAT_TPM_CLOCK_ADJUST_FINE_FASTER   = 1,
74      PLAT_TPM_CLOCK_ADJUST_MEDIUM_FASTER = 2,
75      PLAT_TPM_CLOCK_ADJUST_COARSE_FASTER = 3
76  } _plat_ClockAdjustStep;
77  LIB_EXPORT void _plat_ClockRateAdjust(_plat_ClockAdjustStep adjustment);
78

```



```

79  /** From DebugHelpers.c
80
81  #if CERTIFYX509_DEBUG
82
83  /** DebugFileInit()
84  // This function opens the file used to hold the debug data.
85  // Return Type: int
86  // 0 success
87  // != 0 error
88  int DebugFileInit(void);
89
90  /** DebugDumpBuffer()
91  void DebugDumpBuffer(int size, unsigned char* buf, const char* identifier);
92  #endif // CERTIFYX509_DEBUG
93
94  /** From Entropy.c
95
96  /** _plat_GetEntropy()
97  // This function is used to get available hardware entropy. In a hardware
98  // implementation of this function, there would be no call to the system
99  // to get entropy.
100  // Return Type: int32_t
101  // < 0 hardware failure of the entropy generator, this is sticky
102  // >= 0 the returned amount of entropy (bytes)
103  //
104  LIB_EXPORT int32_t _plat_GetEntropy(unsigned char* entropy, // output buffer
105                                     uint32_t amount // amount requested
106 );
107
108  /** From LocalityPlat.c
109
110  /** _plat_LocalityGet()
111  // Get the most recent command locality in locality value form.
112  // This is an integer value for locality and not a locality structure
113  // The locality can be 0-4 or 32-255. 5-31 is not allowed.
114  LIB_EXPORT unsigned char _plat_LocalityGet(void);
115
116  /** _plat_NVEnable()
117  // Enable NV memory.
118  //
119  // This version just pulls in data from a file. In a real TPM, with NV on chip,
120  // this function would verify the integrity of the saved context. If the NV
121  // memory was not on chip but was in something like RPMB, the NV state would be
122  // read in, decrypted and integrity checked.
123  //
124  // The recovery from an integrity failure depends on where the error occurred. It
125  // it was in the state that is discarded by TPM Reset, then the error is
126  // recoverable if the TPM is reset. Otherwise, the TPM must go into failure mode.
127  //
128  // Return Type: int
129  // 0 if success
130  // >0 if recoverable error
131  // <0 if unrecoverable error
132  LIB_EXPORT int _plat_NVEnable(
133      void* platParameter, // platform specific parameter
134      size_t paramSize // size of parameter. If size == 0, then
135                      // parameter is a sizeof(void*) scalar and should
136                      // be cast to an integer (intptr_t), not dereferenced.
137 );
138
139  /** _plat_GetNvReadyState()
140  // Check if NV is available
141  // Return Type: int
142  // 0 NV is available
143  // 1 NV is not available due to write failure
144  // 2 NV is not available due to rate limit

```

```

145 #define NV_READY 0
146 #define NV_WRITEFAILURE 1
147 #define NV_RATE_LIMIT 2
148 LIB_EXPORT int _plat_GetNvReadyState(void);
149
150 /***_plat_NvMemoryRead()
151 // Function: Read a chunk of NV memory
152 // Return Type: int
153 // TRUE(1) offset and size is within available NV size
154 // FALSE(0) otherwise; also trigger failure mode
155 LIB_EXPORT int _plat_NvMemoryRead(unsigned int startOffset, // IN: read start
156 unsigned int size, // IN: size of bytes to read
157 void* data // OUT: data buffer
158 );
159
160 /***_plat_NvGetChangedStatus()
161 // This function checks to see if the NV is different from the test value. This is
162 // so that NV will not be written if it has not changed.
163 // Return Type: int
164 // NV_HAS_CHANGED(1) the NV location is different from the test value
165 // NV_IS_SAME(0) the NV location is the same as the test value
166 // NV_INVALID_LOCATION(-1) the NV location is invalid; also triggers failure mode
167 #define NV_HAS_CHANGED (1)
168 #define NV_IS_SAME (0)
169 #define NV_INVALID_LOCATION (-1)
170 LIB_EXPORT int _plat_NvGetChangedStatus(
171 unsigned int startOffset, // IN: read start
172 unsigned int size, // IN: size of bytes to read
173 void* data // IN: data buffer
174 );
175
176 /***_plat_NvMemoryWrite()
177 // This function is used to update NV memory. The "write" is to a memory copy of
178 // NV. At the end of the current command, any changes are written to
179 // the actual NV memory.
180 // NOTE: A useful optimization would be for this code to compare the current
181 // contents of NV with the local copy and note the blocks that have changed. Then
182 // only write those blocks when _plat_NvCommit() is called.
183 // Return Type: int
184 // TRUE(1) offset and size is within available NV size
185 // FALSE(0) otherwise; also trigger failure mode
186 LIB_EXPORT int _plat_NvMemoryWrite(unsigned int startOffset, // IN: write start
187 unsigned int size, // IN: size of bytes to write
188 void* data // OUT: data buffer
189 );
190
191 /***_plat_NvMemoryClear()
192 // Function is used to set a range of NV memory bytes to an implementation-dependent
193 // value. The value represents the erase state of the memory.
194 LIB_EXPORT int _plat_NvMemoryClear(unsigned int startOffset, // IN: clear start
195 unsigned int size // IN: number of bytes to clear
196 );
197
198 /***_plat_NvMemoryMove()
199 // Function: Move a chunk of NV memory from source to destination
200 // This function should ensure that if there overlap, the original data is
201 // copied before it is written
202 LIB_EXPORT int _plat_NvMemoryMove(unsigned int sourceOffset, // IN: source offset
203 unsigned int destOffset, // IN: destination offset
204 unsigned int size // IN: size of data being moved
205 );
206
207 /***_plat_NvCommit()
208 // This function writes the local copy of NV to NV for permanent store. It will write
209 // NV_MEMORY_SIZE bytes to NV. If a file is use, the entire file is written.
210 // Return Type: int

```

```

211 // 0      NV write success
212 // non-0  NV write fail
213 LIB_EXPORT int _plat__NvCommit(void);
214
215 /***_plat__TearDown
216 // notify platform that TPM_TearDown was called so platform can cleanup or
217 // zeroize anything in the Platform. This should zeroize NV as well.
218 LIB_EXPORT void _plat__TearDown();
219
220 /*** From PlatformACT.c
221
222 #if ACT_SUPPORT
223 /***_plat__ACT_GetImplemented()
224 // This function tests to see if an ACT is implemented. It is a belt and suspenders
225 // function because the TPM should not be calling to manipulate an ACT that is not
226 // implemented. However, this could help the simulator code which doesn't necessarily
227 // know if an ACT is implemented or not.
228 LIB_EXPORT int _plat__ACT_GetImplemented(uint32_t act);
229
230 /***_plat__ACT_GetRemaining()
231 // This function returns the remaining time. If an update is pending, 'newValue' is
232 // returned. Otherwise, the current counter value is returned. Note that since the
233 // timers keep running, the returned value can get stale immediately. The actual count
234 // value will be no greater than the returned value.
235 LIB_EXPORT uint32_t _plat__ACT_GetRemaining(uint32_t act //IN: the ACT selector
236 );
237
238 /***_plat__ACT_GetSignaled()
239 LIB_EXPORT int _plat__ACT_GetSignaled(uint32_t act //IN: number of ACT to check
240 );
241
242 /***_plat__ACT_SetSignaled()
243 LIB_EXPORT void _plat__ACT_SetSignaled(uint32_t act, int on);
244
245 /***_plat__ACT_UpdateCounter()
246 // This function is used to write the newValue for the counter. If an update is
247 // pending, then no update occurs and the function returns FALSE. If 'setSignaled'
248 // is TRUE, then the ACT signaled state is SET and if 'newValue' is 0, nothing
249 // is posted.
250 LIB_EXPORT int _plat__ACT_UpdateCounter(uint32_t act, // IN: ACT to update
251                                         uint32_t newValue // IN: the value to post
252 );
253
254 /***_plat__ACT_EnableTicks()
255 // This enables and disables the processing of the once-per-second ticks. This should
256 // be turned off ('enable' = FALSE) by _TPM_Init and turned on ('enable' = TRUE) by
257 // TPM2_Startup() after all the initializations have completed.
258 LIB_EXPORT void _plat__ACT_EnableTicks(int enable);
259
260 /***_plat__ACT_Initialize()
261 // This function initializes the ACT hardware and data structures
262 LIB_EXPORT int _plat__ACT_Initialize(void);
263
264 #endif // ACT_SUPPORT
265
266 /*** From PowerPlat.c
267
268 /***_plat__WasPowerLost()
269 // Test whether power was lost before a _TPM_Init.
270 //
271 // This function will clear the "hardware" indication of power loss before return.
272 // This means that there can only be one spot in the TPM code where this value
273 // gets read. This method is used here as it is the most difficult to manage in the
274 // TPM code and, if the hardware actually works this way, it is hard to make it
275 // look like anything else. So, the burden is placed on the TPM code rather than the
276 // platform code

```

```

277 // Return Type: int
278 //     TRUE(1)           power was lost
279 //     FALSE(0)          power was not lost
280 LIB_EXPORT int _plat__WasPowerLost(void);
281
282 /** From PPPlat.c
283
284 /***_ plat__PhysicalPresenceAsserted()
285 // Check if physical presence is signaled
286 // Return Type: int
287 //     TRUE(1)           if physical presence is signaled
288 //     FALSE(0)          if physical presence is not signaled
289 LIB_EXPORT int _plat__PhysicalPresenceAsserted(void);
290
291 /***_ plat__Fail()
292 // This is the platform depended failure exit for the TPM.
293 LIB_EXPORT NORETURN void _plat__Fail(void);
294
295 /** From Unique.c
296
297 #if VENDOR_PERMANENT_AUTH_ENABLED == YES
298 /***_ plat__GetUnique()
299 // This function is used to access the platform-specific unique values.
300 // This function places the unique value in the provided buffer ('b')
301 // and returns the number of bytes transferred. The function will not
302 // copy more data than 'bSize'.
303 // zero indicates value does not exist or an error occurred.
304 //
305 // 'which' indicates the unique value to return:
306 // 0 = RESERVED, do not use
307 // 1 = the VENDOR_PERMANENT_AUTH_HANDLE authorization value for this device
308 LIB_EXPORT uint32_t _plat__GetUnique(uint32_t which,
309                                     uint32_t bSize, // size of the buffer
310                                     unsigned char* b // output buffer
311 );
312 #endif
313
314 /***_ plat__GetPlatformManufactureData
315 // This function allows the platform to provide a small amount of data to be
316 // stored as part of the TPM's PERSISTENT_DATA structure during manufacture. Of
317 // course the platform can store data separately as well, but this allows a
318 // simple platform implementation to store a few bytes of data without
319 // implementing a multi-layer storage system. This function is called on
320 // manufacture and CLEAR. The buffer will contain the last value provided
321 // to the Core library.
322 LIB_EXPORT void _plat__GetPlatformManufactureData(uint8_t* pPlatformPersistentData,
323                                                    uint32_t bufferSize);
324
325 // return the 4 character Manufacturer Capability code. This
326 // should come from the platform library since that is provided by the manufacturer
327 LIB_EXPORT uint32_t _plat__GetManufacturerCapabilityCode();
328
329 // return the 4 character VendorStrings for Capabilities.
330 // Index is ONE-BASED, and may be in the range [1,4] inclusive.
331 // Any other index returns all zeros. The return value will be interpreted
332 // as an array of 4 ASCII characters (with no null terminator)
333 LIB_EXPORT uint32_t _plat__GetVendorCapabilityCode(int index);
334
335 // return the most-significant 32-bits of the TPM Firmware Version reported by
336 // getCapability.
337 LIB_EXPORT uint32_t _plat__GetTpmFirmwareVersionHigh();
338
339 // return the least-significant 32-bits of the TPM Firmware Version reported by
340 // getCapability.
341 LIB_EXPORT uint32_t _plat__GetTpmFirmwareVersionLow();
342

```

```

343 // return the TPM Firmware's current SVN.
344 LIB_EXPORT uint16_t _plat__GetTpmFirmwareSvn(void);
345
346 // return the maximum value that the TPM Firmware SVN may take.
347 LIB_EXPORT uint16_t _plat__GetTpmFirmwareMaxSvn(void);
348
349 #if SVN_LIMITED_SUPPORT
350 /***_plat__GetTpmFirmwareSvnSecret()
351 // Function: Obtain a Firmware SVN Secret bound to the given SVN. Fails if the
352 // given SVN is greater than the firmware's current SVN.
353 // size must equal PRIMARY_SEED_SIZE.
354 // Return Type: int
355 // 0 success
356 // != 0 error
357 LIB_EXPORT int _plat__GetTpmFirmwareSvnSecret(
358     uint16_t svn, // IN: specified SVN
359     uint16_t secret_buf_size, // IN: size of secret buffer
360     uint8_t* secret_buf, // OUT: secret buffer
361     uint16_t* secret_size // OUT: secret buffer
362 );
363 #endif // SVN_LIMITED_SUPPORT
364
365 #if FW_LIMITED_SUPPORT
366 /***_plat__GetTpmFirmwareSecret()
367 // Function: Obtain a Firmware Secret bound to the current firmware image.
368 // Return Type: int
369 // 0 success
370 // != 0 error
371 LIB_EXPORT int _plat__GetTpmFirmwareSecret(
372     uint16_t secret_buf_size, // IN: size of secret buffer
373     uint8_t* secret_buf, // OUT: secret buffer
374     uint16_t* secret_size // OUT: secret buffer
375 );
376 #endif // FW_LIMITED_SUPPORT
377
378 // return the TPM Type returned by TPM_PT_VENDOR_TPM_TYPE
379 LIB_EXPORT uint32_t _plat__GetTpmType();
380
381 // platform PCR initialization functions
382 #include <platform_interface/prototypes/platform_pcr_fp.h>
383
384 #endif // _TPM_TO_PLATFORM_INTERFACE_H_

```

6.4 /tpm/include/platform_interface/prototypes/ExecCommand_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 28, 2019 Time: 08:25:19PM
4 */
5
6 #ifndef EXEC_COMMAND_FP_H_
7 #define EXEC_COMMAND_FP_H_
8
9 /***_ExecuteCommand()
10 //
11 // The function performs the following steps.
12 //
13 // a) Parses the command header from input buffer.
14 // b) Calls ParseHandleBuffer() to parse the handle area of the command.
15 // c) Validates that each of the handles references a loaded entity.
16 // d) Calls ParseSessionBuffer() to:
17 // 1) unmarshal and parse the session area;
18 // 2) check the authorizations; and
19 // 3) when necessary, decrypt a parameter.
20 // e) Calls CommandDispatcher() to:

```



```

21 //      1) unmarshal the command parameters from the command buffer;
22 //      2) call the routine that performs the command actions; and
23 //      3) marshal the responses into the response buffer.
24 // f) If any error occurs in any of the steps above create the error response
25 // and return.
26 // g) Calls BuildResponseSession() to:
27 //      1) when necessary, encrypt a parameter
28 //      2) build the response authorization sessions
29 //      3) update the audit sessions and nonces
30 // h) Calls BuildResponseHeader() to complete the construction of the response.
31 //
32 // 'responseSize' is set by the caller to the maximum number of bytes available in
33 // the output buffer. ExecuteCommand will adjust the value and return the number
34 // of bytes placed in the buffer.
35 //
36 // 'response' is also set by the caller to indicate the buffer into which
37 // ExecuteCommand is to place the response.
38 //
39 // 'request' and 'response' may point to the same buffer
40 //
41 // Note: As of February, 2016, the failure processing has been moved to the
42 // platform-specific code. When the TPM code encounters an unrecoverable failure, it
43 // will SET g_inFailureMode and call _plat_Fail(). That function should not return
44 // but may call ExecuteCommand().
45 //
46 LIB_EXPORT void ExecuteCommand(
47     uint32_t      requestSize, // IN: command buffer size
48     unsigned char* request,    // IN: command buffer
49     uint32_t*     responseSize, // IN/OUT: response buffer size
50     unsigned char** response   // IN/OUT: response buffer
51 );
52
53 #endif // _EXEC_COMMAND_FP_H_

```

6.5 /tpm/include/platform_interface/prototypes/Manufacture_fp.h

```

1  #ifndef MANUFACTURE_FP_H_
2  #define MANUFACTURE_FP_H_
3
4  /*** TPM_Manufacture()
5  // This function initializes the TPM values in preparation for the TPM's first
6  // use. This function will fail if previously called. The TPM can be re-manufactured
7  // by calling TPM_TearDown() first and then calling this function again.
8  // NV must be enabled first (typically with NvPowerOn() via _TPM_Init)
9  //
10 // return type: int
11 //      -2      NV System not available
12 //      -1      FAILURE - System is incorrectly compiled.
13 //      0       success
14 //      1       manufacturing process previously performed
15 // returns
16 #define MANUF_NV_NOT_READY      (-2)
17 #define MANUF_INVALID_CONFIG    (-1)
18 #define MANUF_OK                0
19 #define MANUF_ALREADY_DONE      1
20 // params
21 #define MANUF_FIRST_TIME        1
22 #define MANUF_REMANUFACTURE     0
23 LIB_EXPORT int TPM_Manufacture(
24     int firstTime // IN: indicates if this is the first call from
25                 //      main()
26 );
27
28 /*** TPM_TearDown()
29 // This function prepares the TPM for re-manufacture. It should not be implemented

```

```

30 // in anything other than a simulated TPM.
31 //
32 // In this implementation, all that is needs is to stop the cryptographic units
33 // and set a flag to indicate that the TPM can be re-manufactured. This should
34 // be all that is necessary to start the manufacturing process again.
35 // Return Type: int
36 //     0      success
37 //     1      TPM not previously manufactured
38 #define TEARDOWN_OK 0
39 #define TEARDOWN_NOTHINGDONE 1
40 LIB_EXPORT int TPM_TearDown(void);
41
42 /*** TpmEndSimulation()
43 // This function is called at the end of the simulation run. It is used to provoke
44 // printing of any statistics that might be needed.
45 LIB_EXPORT void TpmEndSimulation(void);
46
47 #endif // _MANUFACTURE_FP_H_

```

6.6 /tpm/include/platform_interface/prototypes/platform_pcr_fp.h

```

1 // platform PCR functions called by the TPM library
2
3 #ifndef _PLATFORM_PCR_FP_H_
4 #define _PLATFORM_PCR_FP_H_
5
6 #include <public/BaseTypes.h>
7 #include <public/TpmTypes.h>
8 #include <platform_interface/pcrstruct.h>
9
10 // return the number of PCRs the platform recognizes for
11 // GetPcrInitializationAttributes.
12 // PCRs are numbered starting at zero.
13 // Note: The TPM Library will enter failure mode if this number doesn't match
14 // IMPLEMENTATION_PCR.
15 UINT32 _platPcr__NumberOfPcrs(void);
16
17 // return the initialization attributes of a given PCR.
18 // pcrNumber expected to be in [0, _platPcr__NumberOfPcrs)
19 // returns the attributes for PCR[0] if the requested pcrNumber is out of range.
20 // Note this returns a structure by-value, which is fast because the structure is
21 // a bitfield.
22 PCR_Attributes _platPcr__GetPcrInitializationAttributes(UINT32 pcrNumber);
23
24 // Fill a given buffer with the PCR initialization value for a particular PCR and hash
25 // combination, and return its length. If the platform doesn't have a value, then
26 // the result size is expected to be zero, and the rfunction will return TPM_RC_PCR.
27 // If a valid is not available, then the core TPM library will ignore the value and
28 // treat it as non-existent and provide a default.
29 // If the buffer is not large enough for a pcr consistent with pcrAlg, then the
30 // platform will return TPM_RC_FAILURE.
31 TPM_RC _platPcr__GetInitialValueForPcr(
32     UINT32 pcrNumber, // IN: PCR to be initialized
33     TPM_ALG_ID pcrAlg, // IN: Algorithm of the PCR Bank being initialized
34     BYTE startupLocality, // IN: locality where startup is being called from
35     BYTE* pcrBuffer, // OUT: buffer to put PCR initialization value into
36     uint16_t bufferSize, // IN: maximum size of value buffer can hold
37     uint16_t* pcrLength); // OUT: size of initialization value returned in pcrBuffer
38
39 // should the given PCR algorithm default to active in a new TPM?
40 BOOL _platPcr__IsPcrBankDefaultActive(TPM_ALG_ID pcrAlg);
41
42 #endif // _PLATFORM_PCR_FP_H_

```

6.7 /tpm/include/platform_interface/prototypes/_TPM_Hash_Data_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 28, 2019   Time: 08:25:19PM
4  */
5
6  #ifndef __TPM_HASH_DATA_FP_H_
7  #define __TPM_HASH_DATA_FP_H_
8
9  // This function is called to process a _TPM_Hash_Data indication.
10 LIB_EXPORT void _TPM_Hash_Data(uint32_t dataSize, // IN: size of data to be extend
11                                unsigned char* data // IN: data buffer
12 );
13
14 #endif // __TPM_HASH_DATA_FP_H_

```

6.8 /tpm/include/platform_interface/prototypes/_TPM_Hash_End_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 28, 2019   Time: 08:25:19PM
4  */
5
6  #ifndef __TPM_HASH_END_FP_H_
7  #define __TPM_HASH_END_FP_H_
8
9  // This function is called to process a _TPM_Hash_End indication.
10 LIB_EXPORT void _TPM_Hash_End(void);
11
12 #endif // __TPM_HASH_END_FP_H_

```

6.9 /tpm/include/platform_interface/prototypes/_TPM_Hash_Start_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 28, 2019   Time: 08:25:19PM
4  */
5
6  #ifndef __TPM_HASH_START_FP_H_
7  #define __TPM_HASH_START_FP_H_
8
9  // This function is called to process a _TPM_Hash_Start indication.
10 LIB_EXPORT void _TPM_Hash_Start(void);
11
12 #endif // __TPM_HASH_START_FP_H_

```

6.10 /tpm/include/platform_interface/prototypes/_TPM_Init_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 28, 2019   Time: 08:25:19PM
4  */
5
6  #ifndef __TPM_INIT_FP_H_
7  #define __TPM_INIT_FP_H_
8
9  // This function is used to process a _TPM_Init indication.
10 LIB_EXPORT void _TPM_Init(void);
11
12 #endif // __TPM_INIT_FP_H_

```


6.11 /tpm/include/private/CommandAttributeData.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2  // clang-format off
3
4  // This file should only be included by CommandCodeAttributes.c
5  #ifndef _COMMAND_CODE_ATTRIBUTES_
6
7  #include "CommandAttributes.h"
8
9  #if COMPRESSED_LISTS
10 #   define    PAD_LIST    0
11 #else
12 #   define    PAD_LIST    1
13 #endif
14
15 // This is the command code attribute array for GetCapability.
16 // Both this array and s_commandAttributes provides command code attributes,
17 // but tuned for different purpose
18 const TPMA_CC    s_ccAttr [] = {
19 #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
20     TPMA_CC_INITIALIZER(0x011F, 0, 1, 0, 0, 2, 0, 0, 0),
21 #endif
22 #if (PAD_LIST || CC_EvictControl)
23     TPMA_CC_INITIALIZER(0x0120, 0, 1, 0, 0, 2, 0, 0, 0),
24 #endif
25 #if (PAD_LIST || CC_HierarchyControl)
26     TPMA_CC_INITIALIZER(0x0121, 0, 1, 1, 0, 1, 0, 0, 0),
27 #endif
28 #if (PAD_LIST || CC_NV_UndefineSpace)
29     TPMA_CC_INITIALIZER(0x0122, 0, 1, 0, 0, 2, 0, 0, 0),
30 #endif
31 #if (PAD_LIST)
32     TPMA_CC_INITIALIZER(0x0123, 0, 0, 0, 0, 0, 0, 0, 0),
33 #endif
34 #if (PAD_LIST || CC_ChangeEPS)
35     TPMA_CC_INITIALIZER(0x0124, 0, 1, 1, 0, 1, 0, 0, 0),
36 #endif
37 #if (PAD_LIST || CC_ChangePPS)
38     TPMA_CC_INITIALIZER(0x0125, 0, 1, 1, 0, 1, 0, 0, 0),
39 #endif
40 #if (PAD_LIST || CC_Clear)
41     TPMA_CC_INITIALIZER(0x0126, 0, 1, 1, 0, 1, 0, 0, 0),
42 #endif
43 #if (PAD_LIST || CC_ClearControl)
44     TPMA_CC_INITIALIZER(0x0127, 0, 1, 0, 0, 1, 0, 0, 0),
45 #endif
46 #if (PAD_LIST || CC_ClockSet)
47     TPMA_CC_INITIALIZER(0x0128, 0, 1, 0, 0, 1, 0, 0, 0),
48 #endif
49 #if (PAD_LIST || CC_HierarchyChangeAuth)
50     TPMA_CC_INITIALIZER(0x0129, 0, 1, 0, 0, 1, 0, 0, 0),
51 #endif
52 #if (PAD_LIST || CC_NV_DefineSpace)
53     TPMA_CC_INITIALIZER(0x012A, 0, 1, 0, 0, 1, 0, 0, 0),
54 #endif
55 #if (PAD_LIST || CC_PCR_Allocate)
56     TPMA_CC_INITIALIZER(0x012B, 0, 1, 0, 0, 1, 0, 0, 0),
57 #endif
58 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
59     TPMA_CC_INITIALIZER(0x012C, 0, 1, 0, 0, 1, 0, 0, 0),
60 #endif
61 #if (PAD_LIST || CC_PP_Commands)
62     TPMA_CC_INITIALIZER(0x012D, 0, 1, 0, 0, 1, 0, 0, 0),
63 #endif
64 #if (PAD_LIST || CC_SetPrimaryPolicy)

```

```
65     TPMA_CC_INITIALIZER(0x012E, 0, 1, 0, 0, 1, 0, 0, 0),
66 #endif
67 #if (PAD_LIST || CC_FieldUpgradeStart)
68     TPMA_CC_INITIALIZER(0x012F, 0, 0, 0, 0, 2, 0, 0, 0),
69 #endif
70 #if (PAD_LIST || CC_ClockRateAdjust)
71     TPMA_CC_INITIALIZER(0x0130, 0, 0, 0, 0, 1, 0, 0, 0),
72 #endif
73 #if (PAD_LIST || CC_CreatePrimary)
74     TPMA_CC_INITIALIZER(0x0131, 0, 0, 0, 0, 1, 1, 0, 0),
75 #endif
76 #if (PAD_LIST || CC_NV_GlobalWriteLock)
77     TPMA_CC_INITIALIZER(0x0132, 0, 1, 0, 0, 1, 0, 0, 0),
78 #endif
79 #if (PAD_LIST || CC_GetCommandAuditDigest)
80     TPMA_CC_INITIALIZER(0x0133, 0, 1, 0, 0, 2, 0, 0, 0),
81 #endif
82 #if (PAD_LIST || CC_NV_Increment)
83     TPMA_CC_INITIALIZER(0x0134, 0, 1, 0, 0, 2, 0, 0, 0),
84 #endif
85 #if (PAD_LIST || CC_NV_SetBits)
86     TPMA_CC_INITIALIZER(0x0135, 0, 1, 0, 0, 2, 0, 0, 0),
87 #endif
88 #if (PAD_LIST || CC_NV_Extend)
89     TPMA_CC_INITIALIZER(0x0136, 0, 1, 0, 0, 2, 0, 0, 0),
90 #endif
91 #if (PAD_LIST || CC_NV_Write)
92     TPMA_CC_INITIALIZER(0x0137, 0, 1, 0, 0, 2, 0, 0, 0),
93 #endif
94 #if (PAD_LIST || CC_NV_WriteLock)
95     TPMA_CC_INITIALIZER(0x0138, 0, 1, 0, 0, 2, 0, 0, 0),
96 #endif
97 #if (PAD_LIST || CC_DictionaryAttackLockReset)
98     TPMA_CC_INITIALIZER(0x0139, 0, 1, 0, 0, 1, 0, 0, 0),
99 #endif
100 #if (PAD_LIST || CC_DictionaryAttackParameters)
101     TPMA_CC_INITIALIZER(0x013A, 0, 1, 0, 0, 1, 0, 0, 0),
102 #endif
103 #if (PAD_LIST || CC_NV_ChangeAuth)
104     TPMA_CC_INITIALIZER(0x013B, 0, 1, 0, 0, 1, 0, 0, 0),
105 #endif
106 #if (PAD_LIST || CC_PCR_Event)
107     TPMA_CC_INITIALIZER(0x013C, 0, 1, 0, 0, 1, 0, 0, 0),
108 #endif
109 #if (PAD_LIST || CC_PCR_Reset)
110     TPMA_CC_INITIALIZER(0x013D, 0, 1, 0, 0, 1, 0, 0, 0),
111 #endif
112 #if (PAD_LIST || CC_SequenceComplete)
113     TPMA_CC_INITIALIZER(0x013E, 0, 0, 0, 1, 1, 0, 0, 0),
114 #endif
115 #if (PAD_LIST || CC_SetAlgorithmSet)
116     TPMA_CC_INITIALIZER(0x013F, 0, 1, 0, 0, 1, 0, 0, 0),
117 #endif
118 #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
119     TPMA_CC_INITIALIZER(0x0140, 0, 1, 0, 0, 1, 0, 0, 0),
120 #endif
121 #if (PAD_LIST || CC_FieldUpgradeData)
122     TPMA_CC_INITIALIZER(0x0141, 0, 1, 0, 0, 0, 0, 0, 0),
123 #endif
124 #if (PAD_LIST || CC_IncrementalSelfTest)
125     TPMA_CC_INITIALIZER(0x0142, 0, 1, 0, 0, 0, 0, 0, 0),
126 #endif
127 #if (PAD_LIST || CC_SelfTest)
128     TPMA_CC_INITIALIZER(0x0143, 0, 1, 0, 0, 0, 0, 0, 0),
129 #endif
130 #if (PAD_LIST || CC_Startup)
```

```
131     TPMA_CC_INITIALIZER(0x0144, 0, 1, 0, 0, 0, 0, 0, 0),
132 #endif
133 #if (PAD_LIST || CC_Shutdown)
134     TPMA_CC_INITIALIZER(0x0145, 0, 1, 0, 0, 0, 0, 0, 0),
135 #endif
136 #if (PAD_LIST || CC_StirRandom)
137     TPMA_CC_INITIALIZER(0x0146, 0, 1, 0, 0, 0, 0, 0, 0),
138 #endif
139 #if (PAD_LIST || CC_ActivateCredential)
140     TPMA_CC_INITIALIZER(0x0147, 0, 0, 0, 0, 0, 2, 0, 0),
141 #endif
142 #if (PAD_LIST || CC_Certify)
143     TPMA_CC_INITIALIZER(0x0148, 0, 0, 0, 0, 0, 2, 0, 0),
144 #endif
145 #if (PAD_LIST || CC_PolicyNV)
146     TPMA_CC_INITIALIZER(0x0149, 0, 0, 0, 0, 0, 3, 0, 0),
147 #endif
148 #if (PAD_LIST || CC_CertifyCreation)
149     TPMA_CC_INITIALIZER(0x014A, 0, 0, 0, 0, 0, 2, 0, 0),
150 #endif
151 #if (PAD_LIST || CC_Duplicate)
152     TPMA_CC_INITIALIZER(0x014B, 0, 0, 0, 0, 0, 2, 0, 0),
153 #endif
154 #if (PAD_LIST || CC_GetTime)
155     TPMA_CC_INITIALIZER(0x014C, 0, 0, 0, 0, 0, 2, 0, 0),
156 #endif
157 #if (PAD_LIST || CC_GetSessionAuditDigest)
158     TPMA_CC_INITIALIZER(0x014D, 0, 0, 0, 0, 0, 3, 0, 0),
159 #endif
160 #if (PAD_LIST || CC_NV_Read)
161     TPMA_CC_INITIALIZER(0x014E, 0, 0, 0, 0, 0, 2, 0, 0),
162 #endif
163 #if (PAD_LIST || CC_NV_ReadLock)
164     TPMA_CC_INITIALIZER(0x014F, 0, 1, 0, 0, 0, 2, 0, 0),
165 #endif
166 #if (PAD_LIST || CC_ObjectChangeAuth)
167     TPMA_CC_INITIALIZER(0x0150, 0, 0, 0, 0, 0, 2, 0, 0),
168 #endif
169 #if (PAD_LIST || CC_PolicySecret)
170     TPMA_CC_INITIALIZER(0x0151, 0, 0, 0, 0, 0, 2, 0, 0),
171 #endif
172 #if (PAD_LIST || CC_Rewrap)
173     TPMA_CC_INITIALIZER(0x0152, 0, 0, 0, 0, 0, 2, 0, 0),
174 #endif
175 #if (PAD_LIST || CC_Create)
176     TPMA_CC_INITIALIZER(0x0153, 0, 0, 0, 0, 0, 1, 0, 0),
177 #endif
178 #if (PAD_LIST || CC_ECDH_ZGen)
179     TPMA_CC_INITIALIZER(0x0154, 0, 0, 0, 0, 0, 1, 0, 0),
180 #endif
181 #if (PAD_LIST || (CC_HMAC || CC_MAC))
182     TPMA_CC_INITIALIZER(0x0155, 0, 0, 0, 0, 0, 1, 0, 0),
183 #endif
184 #if (PAD_LIST || CC_Import)
185     TPMA_CC_INITIALIZER(0x0156, 0, 0, 0, 0, 0, 1, 0, 0),
186 #endif
187 #if (PAD_LIST || CC_Load)
188     TPMA_CC_INITIALIZER(0x0157, 0, 0, 0, 0, 0, 1, 1, 0),
189 #endif
190 #if (PAD_LIST || CC_Quote)
191     TPMA_CC_INITIALIZER(0x0158, 0, 0, 0, 0, 0, 1, 0, 0),
192 #endif
193 #if (PAD_LIST || CC_RSA_Decrypt)
194     TPMA_CC_INITIALIZER(0x0159, 0, 0, 0, 0, 0, 1, 0, 0),
195 #endif
196 #if (PAD_LIST)
```

```
197     TPMA_CC_INITIALIZER(0x015A, 0, 0, 0, 0, 0, 0, 0, 0),
198 #endif
199 #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
200     TPMA_CC_INITIALIZER(0x015B, 0, 0, 0, 0, 1, 1, 0, 0),
201 #endif
202 #if (PAD_LIST || CC_SequenceUpdate)
203     TPMA_CC_INITIALIZER(0x015C, 0, 0, 0, 0, 1, 0, 0, 0),
204 #endif
205 #if (PAD_LIST || CC_Sign)
206     TPMA_CC_INITIALIZER(0x015D, 0, 0, 0, 0, 1, 0, 0, 0),
207 #endif
208 #if (PAD_LIST || CC_Unseal)
209     TPMA_CC_INITIALIZER(0x015E, 0, 0, 0, 0, 1, 0, 0, 0),
210 #endif
211 #if (PAD_LIST)
212     TPMA_CC_INITIALIZER(0x015F, 0, 0, 0, 0, 0, 0, 0, 0),
213 #endif
214 #if (PAD_LIST || CC_PolicySigned)
215     TPMA_CC_INITIALIZER(0x0160, 0, 0, 0, 0, 2, 0, 0, 0),
216 #endif
217 #if (PAD_LIST || CC_ContextLoad)
218     TPMA_CC_INITIALIZER(0x0161, 0, 0, 0, 0, 0, 1, 0, 0),
219 #endif
220 #if (PAD_LIST || CC_ContextSave)
221     TPMA_CC_INITIALIZER(0x0162, 0, 0, 0, 0, 1, 0, 0, 0),
222 #endif
223 #if (PAD_LIST || CC_ECDH_KeyGen)
224     TPMA_CC_INITIALIZER(0x0163, 0, 0, 0, 0, 1, 0, 0, 0),
225 #endif
226 #if (PAD_LIST || CC_EncryptDecrypt)
227     TPMA_CC_INITIALIZER(0x0164, 0, 0, 0, 0, 1, 0, 0, 0),
228 #endif
229 #if (PAD_LIST || CC_FlushContext)
230     TPMA_CC_INITIALIZER(0x0165, 0, 0, 0, 0, 0, 0, 0, 0),
231 #endif
232 #if (PAD_LIST)
233     TPMA_CC_INITIALIZER(0x0166, 0, 0, 0, 0, 0, 0, 0, 0),
234 #endif
235 #if (PAD_LIST || CC_LoadExternal)
236     TPMA_CC_INITIALIZER(0x0167, 0, 0, 0, 0, 0, 1, 0, 0),
237 #endif
238 #if (PAD_LIST || CC_MakeCredential)
239     TPMA_CC_INITIALIZER(0x0168, 0, 0, 0, 0, 1, 0, 0, 0),
240 #endif
241 #if (PAD_LIST || CC_NV_ReadPublic)
242     TPMA_CC_INITIALIZER(0x0169, 0, 0, 0, 0, 1, 0, 0, 0),
243 #endif
244 #if (PAD_LIST || CC_PolicyAuthorize)
245     TPMA_CC_INITIALIZER(0x016A, 0, 0, 0, 0, 1, 0, 0, 0),
246 #endif
247 #if (PAD_LIST || CC_PolicyAuthValue)
248     TPMA_CC_INITIALIZER(0x016B, 0, 0, 0, 0, 1, 0, 0, 0),
249 #endif
250 #if (PAD_LIST || CC_PolicyCommandCode)
251     TPMA_CC_INITIALIZER(0x016C, 0, 0, 0, 0, 1, 0, 0, 0),
252 #endif
253 #if (PAD_LIST || CC_PolicyCounterTimer)
254     TPMA_CC_INITIALIZER(0x016D, 0, 0, 0, 0, 1, 0, 0, 0),
255 #endif
256 #if (PAD_LIST || CC_PolicyCpHash)
257     TPMA_CC_INITIALIZER(0x016E, 0, 0, 0, 0, 1, 0, 0, 0),
258 #endif
259 #if (PAD_LIST || CC_PolicyLocality)
260     TPMA_CC_INITIALIZER(0x016F, 0, 0, 0, 0, 1, 0, 0, 0),
261 #endif
262 #if (PAD_LIST || CC_PolicyNameHash)
```

```
263     TPMA_CC_INITIALIZER(0x0170, 0, 0, 0, 0, 1, 0, 0, 0),
264 #endif
265 #if (PAD_LIST || CC_PolicyOR)
266     TPMA_CC_INITIALIZER(0x0171, 0, 0, 0, 0, 1, 0, 0, 0),
267 #endif
268 #if (PAD_LIST || CC_PolicyTicket)
269     TPMA_CC_INITIALIZER(0x0172, 0, 0, 0, 0, 1, 0, 0, 0),
270 #endif
271 #if (PAD_LIST || CC_ReadPublic)
272     TPMA_CC_INITIALIZER(0x0173, 0, 0, 0, 0, 1, 0, 0, 0),
273 #endif
274 #if (PAD_LIST || CC_RSA_Encrypt)
275     TPMA_CC_INITIALIZER(0x0174, 0, 0, 0, 0, 1, 0, 0, 0),
276 #endif
277 #if (PAD_LIST)
278     TPMA_CC_INITIALIZER(0x0175, 0, 0, 0, 0, 0, 0, 0, 0),
279 #endif
280 #if (PAD_LIST || CC_StartAuthSession)
281     TPMA_CC_INITIALIZER(0x0176, 0, 0, 0, 0, 2, 1, 0, 0),
282 #endif
283 #if (PAD_LIST || CC_VerifySignature)
284     TPMA_CC_INITIALIZER(0x0177, 0, 0, 0, 0, 1, 0, 0, 0),
285 #endif
286 #if (PAD_LIST || CC_ECC_Parameters)
287     TPMA_CC_INITIALIZER(0x0178, 0, 0, 0, 0, 0, 0, 0, 0),
288 #endif
289 #if (PAD_LIST || CC_FirmwareRead)
290     TPMA_CC_INITIALIZER(0x0179, 0, 0, 0, 0, 0, 0, 0, 0),
291 #endif
292 #if (PAD_LIST || CC_GetCapability)
293     TPMA_CC_INITIALIZER(0x017A, 0, 0, 0, 0, 0, 0, 0, 0),
294 #endif
295 #if (PAD_LIST || CC_GetRandom)
296     TPMA_CC_INITIALIZER(0x017B, 0, 0, 0, 0, 0, 0, 0, 0),
297 #endif
298 #if (PAD_LIST || CC_GetTestResult)
299     TPMA_CC_INITIALIZER(0x017C, 0, 0, 0, 0, 0, 0, 0, 0),
300 #endif
301 #if (PAD_LIST || CC_Hash)
302     TPMA_CC_INITIALIZER(0x017D, 0, 0, 0, 0, 0, 0, 0, 0),
303 #endif
304 #if (PAD_LIST || CC_PCR_Read)
305     TPMA_CC_INITIALIZER(0x017E, 0, 0, 0, 0, 0, 0, 0, 0),
306 #endif
307 #if (PAD_LIST || CC_PolicyPCR)
308     TPMA_CC_INITIALIZER(0x017F, 0, 0, 0, 0, 1, 0, 0, 0),
309 #endif
310 #if (PAD_LIST || CC_PolicyRestart)
311     TPMA_CC_INITIALIZER(0x0180, 0, 0, 0, 0, 1, 0, 0, 0),
312 #endif
313 #if (PAD_LIST || CC_ReadClock)
314     TPMA_CC_INITIALIZER(0x0181, 0, 0, 0, 0, 0, 0, 0, 0),
315 #endif
316 #if (PAD_LIST || CC_PCR_Extend)
317     TPMA_CC_INITIALIZER(0x0182, 0, 1, 0, 0, 1, 0, 0, 0),
318 #endif
319 #if (PAD_LIST || CC_PCR_SetAuthValue)
320     TPMA_CC_INITIALIZER(0x0183, 0, 0, 0, 0, 1, 0, 0, 0),
321 #endif
322 #if (PAD_LIST || CC_NV_Certify)
323     TPMA_CC_INITIALIZER(0x0184, 0, 0, 0, 0, 3, 0, 0, 0),
324 #endif
325 #if (PAD_LIST || CC_EventSequenceComplete)
326     TPMA_CC_INITIALIZER(0x0185, 0, 1, 0, 1, 2, 0, 0, 0),
327 #endif
328 #if (PAD_LIST || CC_HashSequenceStart)
```

```
329     TPMA_CC_INITIALIZER(0x0186, 0, 0, 0, 0, 0, 1, 0, 0),
330 #endif
331 #if (PAD_LIST || CC_PolicyPhysicalPresence)
332     TPMA_CC_INITIALIZER(0x0187, 0, 0, 0, 0, 0, 1, 0, 0, 0),
333 #endif
334 #if (PAD_LIST || CC_PolicyDuplicationSelect)
335     TPMA_CC_INITIALIZER(0x0188, 0, 0, 0, 0, 0, 1, 0, 0, 0),
336 #endif
337 #if (PAD_LIST || CC_PolicyGetDigest)
338     TPMA_CC_INITIALIZER(0x0189, 0, 0, 0, 0, 0, 1, 0, 0, 0),
339 #endif
340 #if (PAD_LIST || CC_TestParms)
341     TPMA_CC_INITIALIZER(0x018A, 0, 0, 0, 0, 0, 0, 0, 0, 0),
342 #endif
343 #if (PAD_LIST || CC_Commit)
344     TPMA_CC_INITIALIZER(0x018B, 0, 0, 0, 0, 0, 1, 0, 0, 0),
345 #endif
346 #if (PAD_LIST || CC_PolicyPassword)
347     TPMA_CC_INITIALIZER(0x018C, 0, 0, 0, 0, 0, 1, 0, 0, 0),
348 #endif
349 #if (PAD_LIST || CC_ZGen_2Phase)
350     TPMA_CC_INITIALIZER(0x018D, 0, 0, 0, 0, 0, 1, 0, 0, 0),
351 #endif
352 #if (PAD_LIST || CC_EC_Ephemeral)
353     TPMA_CC_INITIALIZER(0x018E, 0, 0, 0, 0, 0, 0, 0, 0, 0),
354 #endif
355 #if (PAD_LIST || CC_PolicyNvWritten)
356     TPMA_CC_INITIALIZER(0x018F, 0, 0, 0, 0, 0, 1, 0, 0, 0),
357 #endif
358 #if (PAD_LIST || CC_PolicyTemplate)
359     TPMA_CC_INITIALIZER(0x0190, 0, 0, 0, 0, 0, 1, 0, 0, 0),
360 #endif
361 #if (PAD_LIST || CC_CreateLoaded)
362     TPMA_CC_INITIALIZER(0x0191, 0, 0, 0, 0, 0, 1, 1, 0, 0),
363 #endif
364 #if (PAD_LIST || CC_PolicyAuthorizeNV)
365     TPMA_CC_INITIALIZER(0x0192, 0, 0, 0, 0, 0, 3, 0, 0, 0),
366 #endif
367 #if (PAD_LIST || CC_EncryptDecrypt2)
368     TPMA_CC_INITIALIZER(0x0193, 0, 0, 0, 0, 0, 1, 0, 0, 0),
369 #endif
370 #if (PAD_LIST || CC_AC_GetCapability)
371     TPMA_CC_INITIALIZER(0x0194, 0, 0, 0, 0, 0, 1, 0, 0, 0),
372 #endif
373 #if (PAD_LIST || CC_AC_Send)
374     TPMA_CC_INITIALIZER(0x0195, 0, 0, 0, 0, 0, 3, 0, 0, 0),
375 #endif
376 #if (PAD_LIST || CC_Policy_AC_SendSelect)
377     TPMA_CC_INITIALIZER(0x0196, 0, 0, 0, 0, 0, 1, 0, 0, 0),
378 #endif
379 #if (PAD_LIST || CC_CertifyX509)
380     TPMA_CC_INITIALIZER(0x0197, 0, 0, 0, 0, 0, 2, 0, 0, 0),
381 #endif
382 #if (PAD_LIST || CC_ACT_SetTimeout)
383     TPMA_CC_INITIALIZER(0x0198, 0, 0, 0, 0, 0, 1, 0, 0, 0),
384 #endif
385 #if (PAD_LIST || CC_ECC_Encrypt)
386     TPMA_CC_INITIALIZER(0x0199, 0, 0, 0, 0, 0, 1, 0, 0, 0),
387 #endif
388 #if (PAD_LIST || CC_ECC_Decrypt)
389     TPMA_CC_INITIALIZER(0x019A, 0, 0, 0, 0, 0, 1, 0, 0, 0),
390 #endif
391 #if (PAD_LIST || CC_PolicyCapability)
392     TPMA_CC_INITIALIZER(0x019B, 0, 0, 0, 0, 0, 1, 0, 0, 0),
393 #endif
394 #if (PAD_LIST || CC_PolicyParameters)
```



```

395     TPMA_CC_INITIALIZER(0x019C, 0, 0, 0, 0, 1, 0, 0, 0),
396 #endif
397 #if (PAD_LIST || CC_NV_DefineSpace2)
398     TPMA_CC_INITIALIZER(0x019D, 0, 1, 0, 0, 1, 0, 0, 0),
399 #endif
400 #if (PAD_LIST || CC_NV_ReadPublic2)
401     TPMA_CC_INITIALIZER(0x019E, 0, 0, 0, 0, 1, 0, 0, 0),
402 #endif
403 #if (PAD_LIST || CC_SetCapability)
404     TPMA_CC_INITIALIZER(0x019F, 0, 1, 0, 0, 1, 0, 0, 0),
405 #endif
406 #if (PAD_LIST || CC_Vendor_TCG_Test)
407     TPMA_CC_INITIALIZER(0x0000, 0, 0, 0, 0, 0, 0, 1, 0),
408 #endif
409     TPMA_ZERO_INITIALIZER()
410 };
411
412
413 // This is the command code attribute structure.
414 const COMMAND_ATTRIBUTES s_commandAttributes [] = {
415 #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
416     (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpaceSpecial * // 0x011F
417         (IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),
418 #endif
419 #if (PAD_LIST || CC_EvictControl)
420     (COMMAND_ATTRIBUTES)(CC_EvictControl * // 0x0120
421         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
422 #endif
423 #if (PAD_LIST || CC_HierarchyControl)
424     (COMMAND_ATTRIBUTES)(CC_HierarchyControl * // 0x0121
425         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
426 #endif
427 #if (PAD_LIST || CC_NV_UndefineSpace)
428     (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpace * // 0x0122
429         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
430 #endif
431 #if (PAD_LIST)
432     (COMMAND_ATTRIBUTES)(0), // 0x0123
433 #endif
434 #if (PAD_LIST || CC_ChangeEPS)
435     (COMMAND_ATTRIBUTES)(CC_ChangeEPS * // 0x0124
436         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
437 #endif
438 #if (PAD_LIST || CC_ChangePPS)
439     (COMMAND_ATTRIBUTES)(CC_ChangePPS * // 0x0125
440         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
441 #endif
442 #if (PAD_LIST || CC_Clear)
443     (COMMAND_ATTRIBUTES)(CC_Clear * // 0x0126
444         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
445 #endif
446 #if (PAD_LIST || CC_ClearControl)
447     (COMMAND_ATTRIBUTES)(CC_ClearControl * // 0x0127
448         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
449 #endif
450 #if (PAD_LIST || CC_ClockSet)
451     (COMMAND_ATTRIBUTES)(CC_ClockSet * // 0x0128
452         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
453 #endif
454 #if (PAD_LIST || CC_HierarchyChangeAuth)
455     (COMMAND_ATTRIBUTES)(CC_HierarchyChangeAuth * // 0x0129
456         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
457 #endif
458 #if (PAD_LIST || CC_NV_DefineSpace)
459     (COMMAND_ATTRIBUTES)(CC_NV_DefineSpace * // 0x012A
460         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),

```

```

461 #endif
462 #if (PAD_LIST || CC_PCR_Allocate)
463     (COMMAND_ATTRIBUTES) (CC_PCR_Allocate * // 0x012B
464         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
465 #endif
466 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
467     (COMMAND_ATTRIBUTES) (CC_PCR_SetAuthPolicy * // 0x012C
468         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
469 #endif
470 #if (PAD_LIST || CC_PP_Commands)
471     (COMMAND_ATTRIBUTES) (CC_PP_Commands * // 0x012D
472         (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),
473 #endif
474 #if (PAD_LIST || CC_SetPrimaryPolicy)
475     (COMMAND_ATTRIBUTES) (CC_SetPrimaryPolicy * // 0x012E
476         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
477 #endif
478 #if (PAD_LIST || CC_FieldUpgradeStart)
479     (COMMAND_ATTRIBUTES) (CC_FieldUpgradeStart * // 0x012F
480         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),
481 #endif
482 #if (PAD_LIST || CC_ClockRateAdjust)
483     (COMMAND_ATTRIBUTES) (CC_ClockRateAdjust * // 0x0130
484         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
485 #endif
486 #if (PAD_LIST || CC_CreatePrimary)
487     (COMMAND_ATTRIBUTES) (CC_CreatePrimary * // 0x0131
488         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
489 #endif
490 #if (PAD_LIST || CC_NV_GlobalWriteLock)
491     (COMMAND_ATTRIBUTES) (CC_NV_GlobalWriteLock * // 0x0132
492         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
493 #endif
494 #if (PAD_LIST || CC_GetCommandAuditDigest)
495     (COMMAND_ATTRIBUTES) (CC_GetCommandAuditDigest * // 0x0133
496         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
497 #endif
498 #if (PAD_LIST || CC_NV_Increment)
499     (COMMAND_ATTRIBUTES) (CC_NV_Increment * // 0x0134
500         (IS_IMPLEMENTED+HANDLE_1_USER)),
501 #endif
502 #if (PAD_LIST || CC_NV_SetBits)
503     (COMMAND_ATTRIBUTES) (CC_NV_SetBits * // 0x0135
504         (IS_IMPLEMENTED+HANDLE_1_USER)),
505 #endif
506 #if (PAD_LIST || CC_NV_Extend)
507     (COMMAND_ATTRIBUTES) (CC_NV_Extend * // 0x0136
508         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
509 #endif
510 #if (PAD_LIST || CC_NV_Write)
511     (COMMAND_ATTRIBUTES) (CC_NV_Write * // 0x0137
512         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
513 #endif
514 #if (PAD_LIST || CC_NV_WriteLock)
515     (COMMAND_ATTRIBUTES) (CC_NV_WriteLock * // 0x0138
516         (IS_IMPLEMENTED+HANDLE_1_USER)),
517 #endif
518 #if (PAD_LIST || CC_DictionaryAttackLockReset)
519     (COMMAND_ATTRIBUTES) (CC_DictionaryAttackLockReset * // 0x0139
520         (IS_IMPLEMENTED+HANDLE_1_USER)),
521 #endif
522 #if (PAD_LIST || CC_DictionaryAttackParameters)
523     (COMMAND_ATTRIBUTES) (CC_DictionaryAttackParameters * // 0x013A
524         (IS_IMPLEMENTED+HANDLE_1_USER)),
525 #endif
526 #if (PAD_LIST || CC_NV_ChangeAuth)

```



```
527         (COMMAND_ATTRIBUTES) (CC_NV_ChangeAuth * // 0x013B
528         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),
529 #endif
530 #if (PAD_LIST || CC_PCR_Event)
531     (COMMAND_ATTRIBUTES) (CC_PCR_Event * // 0x013C
532     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
533 #endif
534 #if (PAD_LIST || CC_PCR_Reset)
535     (COMMAND_ATTRIBUTES) (CC_PCR_Reset * // 0x013D
536     (IS_IMPLEMENTED+HANDLE_1_USER)),
537 #endif
538 #if (PAD_LIST || CC_SequenceComplete)
539     (COMMAND_ATTRIBUTES) (CC_SequenceComplete * // 0x013E
540     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
541 #endif
542 #if (PAD_LIST || CC_SetAlgorithmSet)
543     (COMMAND_ATTRIBUTES) (CC_SetAlgorithmSet * // 0x013F
544     (IS_IMPLEMENTED+HANDLE_1_USER)),
545 #endif
546 #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
547     (COMMAND_ATTRIBUTES) (CC_SetCommandCodeAuditStatus * // 0x0140
548     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
549 #endif
550 #if (PAD_LIST || CC_FieldUpgradeData)
551     (COMMAND_ATTRIBUTES) (CC_FieldUpgradeData * // 0x0141
552     (IS_IMPLEMENTED+DECRYPT_2)),
553 #endif
554 #if (PAD_LIST || CC_IncrementalSelfTest)
555     (COMMAND_ATTRIBUTES) (CC_IncrementalSelfTest * // 0x0142
556     (IS_IMPLEMENTED)),
557 #endif
558 #if (PAD_LIST || CC_SelfTest)
559     (COMMAND_ATTRIBUTES) (CC_SelfTest * // 0x0143
560     (IS_IMPLEMENTED)),
561 #endif
562 #if (PAD_LIST || CC_Startup)
563     (COMMAND_ATTRIBUTES) (CC_Startup * // 0x0144
564     (IS_IMPLEMENTED+NO_SESSIONS)),
565 #endif
566 #if (PAD_LIST || CC_Shutdown)
567     (COMMAND_ATTRIBUTES) (CC_Shutdown * // 0x0145
568     (IS_IMPLEMENTED)),
569 #endif
570 #if (PAD_LIST || CC_StirRandom)
571     (COMMAND_ATTRIBUTES) (CC_StirRandom * // 0x0146
572     (IS_IMPLEMENTED+DECRYPT_2)),
573 #endif
574 #if (PAD_LIST || CC_ActivateCredential)
575     (COMMAND_ATTRIBUTES) (CC_ActivateCredential * // 0x0147
576     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
577 #endif
578 #if (PAD_LIST || CC_Certify)
579     (COMMAND_ATTRIBUTES) (CC_Certify * // 0x0148
580     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
581 #endif
582 #if (PAD_LIST || CC_PolicyNV)
583     (COMMAND_ATTRIBUTES) (CC_PolicyNV * // 0x0149
584     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL)),
585 #endif
586 #if (PAD_LIST || CC_CertifyCreation)
587     (COMMAND_ATTRIBUTES) (CC_CertifyCreation * // 0x014A
588     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
589 #endif
590 #if (PAD_LIST || CC_Duplicate)
591     (COMMAND_ATTRIBUTES) (CC_Duplicate * // 0x014B
592     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),
```

```

593 #endif
594 #if (PAD_LIST || CC_GetTime)
595     (COMMAND_ATTRIBUTES) (CC_GetTime * // 0x014C
596         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
597 #endif
598 #if (PAD_LIST || CC_GetSessionAuditDigest)
599     (COMMAND_ATTRIBUTES) (CC_GetSessionAuditDigest * // 0x014D
600         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
601 #endif
602 #if (PAD_LIST || CC_NV_Read)
603     (COMMAND_ATTRIBUTES) (CC_NV_Read * // 0x014E
604         (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
605 #endif
606 #if (PAD_LIST || CC_NV_ReadLock)
607     (COMMAND_ATTRIBUTES) (CC_NV_ReadLock * // 0x014F
608         (IS_IMPLEMENTED+HANDLE_1_USER)),
609 #endif
610 #if (PAD_LIST || CC_ObjectChangeAuth)
611     (COMMAND_ATTRIBUTES) (CC_ObjectChangeAuth * // 0x0150
612         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),
613 #endif
614 #if (PAD_LIST || CC_PolicySecret)
615     (COMMAND_ATTRIBUTES) (CC_PolicySecret * // 0x0151
616         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL+ENCRYPT_2)),
617 #endif
618 #if (PAD_LIST || CC_Rewrap)
619     (COMMAND_ATTRIBUTES) (CC_Rewrap * // 0x0152
620         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
621 #endif
622 #if (PAD_LIST || CC_Create)
623     (COMMAND_ATTRIBUTES) (CC_Create * // 0x0153
624         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
625 #endif
626 #if (PAD_LIST || CC_ECDH_ZGen)
627     (COMMAND_ATTRIBUTES) (CC_ECDH_ZGen * // 0x0154
628         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
629 #endif
630 #if (PAD_LIST || (CC_HMAC || CC_MAC))
631     (COMMAND_ATTRIBUTES) ((CC_HMAC || CC_MAC) * // 0x0155
632         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
633 #endif
634 #if (PAD_LIST || CC_Import)
635     (COMMAND_ATTRIBUTES) (CC_Import * // 0x0156
636         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
637 #endif
638 #if (PAD_LIST || CC_Load)
639     (COMMAND_ATTRIBUTES) (CC_Load * // 0x0157
640         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),
641 #endif
642 #if (PAD_LIST || CC_Quote)
643     (COMMAND_ATTRIBUTES) (CC_Quote * // 0x0158
644         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
645 #endif
646 #if (PAD_LIST || CC_RSA_Decrypt)
647     (COMMAND_ATTRIBUTES) (CC_RSA_Decrypt * // 0x0159
648         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
649 #endif
650 #if (PAD_LIST)
651     (COMMAND_ATTRIBUTES) (0), // 0x015A
652 #endif
653 #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
654     (COMMAND_ATTRIBUTES) ((CC_HMAC_Start || CC_MAC_Start) * // 0x015B
655         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),
656 #endif
657 #if (PAD_LIST || CC_SequenceUpdate)
658     (COMMAND_ATTRIBUTES) (CC_SequenceUpdate * // 0x015C

```

```

659         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
660 #endif
661 #if (PAD_LIST || CC_Sign)
662     (COMMAND_ATTRIBUTES)(CC_Sign * // 0x015D
663         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
664 #endif
665 #if (PAD_LIST || CC_Unseal)
666     (COMMAND_ATTRIBUTES)(CC_Unseal * // 0x015E
667         (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
668 #endif
669 #if (PAD_LIST)
670     (COMMAND_ATTRIBUTES)(0), // 0x015F
671 #endif
672 #if (PAD_LIST || CC_PolicySigned)
673     (COMMAND_ATTRIBUTES)(CC_PolicySigned * // 0x0160
674         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL+ENCRYPT_2)),
675 #endif
676 #if (PAD_LIST || CC_ContextLoad)
677     (COMMAND_ATTRIBUTES)(CC_ContextLoad * // 0x0161
678         (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
679 #endif
680 #if (PAD_LIST || CC_ContextSave)
681     (COMMAND_ATTRIBUTES)(CC_ContextSave * // 0x0162
682         (IS_IMPLEMENTED+NO_SESSIONS)),
683 #endif
684 #if (PAD_LIST || CC_ECDH_KeyGen)
685     (COMMAND_ATTRIBUTES)(CC_ECDH_KeyGen * // 0x0163
686         (IS_IMPLEMENTED+ENCRYPT_2)),
687 #endif
688 #if (PAD_LIST || CC_EncryptDecrypt)
689     (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt * // 0x0164
690         (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
691 #endif
692 #if (PAD_LIST || CC_FlushContext)
693     (COMMAND_ATTRIBUTES)(CC_FlushContext * // 0x0165
694         (IS_IMPLEMENTED+NO_SESSIONS)),
695 #endif
696 #if (PAD_LIST)
697     (COMMAND_ATTRIBUTES)(0), // 0x0166
698 #endif
699 #if (PAD_LIST || CC_LoadExternal)
700     (COMMAND_ATTRIBUTES)(CC_LoadExternal * // 0x0167
701         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
702 #endif
703 #if (PAD_LIST || CC_MakeCredential)
704     (COMMAND_ATTRIBUTES)(CC_MakeCredential * // 0x0168
705         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
706 #endif
707 #if (PAD_LIST || CC_NV_ReadPublic)
708     (COMMAND_ATTRIBUTES)(CC_NV_ReadPublic * // 0x0169
709         (IS_IMPLEMENTED+ENCRYPT_2)),
710 #endif
711 #if (PAD_LIST || CC_PolicyAuthorize)
712     (COMMAND_ATTRIBUTES)(CC_PolicyAuthorize * // 0x016A
713         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
714 #endif
715 #if (PAD_LIST || CC_PolicyAuthValue)
716     (COMMAND_ATTRIBUTES)(CC_PolicyAuthValue * // 0x016B
717         (IS_IMPLEMENTED+ALLOW_TRIAL)),
718 #endif
719 #if (PAD_LIST || CC_PolicyCommandCode)
720     (COMMAND_ATTRIBUTES)(CC_PolicyCommandCode * // 0x016C
721         (IS_IMPLEMENTED+ALLOW_TRIAL)),
722 #endif
723 #if (PAD_LIST || CC_PolicyCounterTimer)
724     (COMMAND_ATTRIBUTES)(CC_PolicyCounterTimer * // 0x016D

```

```

725         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
726 #endif
727 #if (PAD_LIST || CC_PolicyCpHash)
728     (COMMAND_ATTRIBUTES)(CC_PolicyCpHash * // 0x016E
729         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
730 #endif
731 #if (PAD_LIST || CC_PolicyLocality)
732     (COMMAND_ATTRIBUTES)(CC_PolicyLocality * // 0x016F
733         (IS_IMPLEMENTED+ALLOW_TRIAL)),
734 #endif
735 #if (PAD_LIST || CC_PolicyNameHash)
736     (COMMAND_ATTRIBUTES)(CC_PolicyNameHash * // 0x0170
737         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
738 #endif
739 #if (PAD_LIST || CC_PolicyOR)
740     (COMMAND_ATTRIBUTES)(CC_PolicyOR * // 0x0171
741         (IS_IMPLEMENTED+ALLOW_TRIAL)),
742 #endif
743 #if (PAD_LIST || CC_PolicyTicket)
744     (COMMAND_ATTRIBUTES)(CC_PolicyTicket * // 0x0172
745         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
746 #endif
747 #if (PAD_LIST || CC_ReadPublic)
748     (COMMAND_ATTRIBUTES)(CC_ReadPublic * // 0x0173
749         (IS_IMPLEMENTED+ENCRYPT_2)),
750 #endif
751 #if (PAD_LIST || CC_RSA_Encrypt)
752     (COMMAND_ATTRIBUTES)(CC_RSA_Encrypt * // 0x0174
753         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
754 #endif
755 #if (PAD_LIST)
756     (COMMAND_ATTRIBUTES)(0), // 0x0175
757 #endif
758 #if (PAD_LIST || CC_StartAuthSession)
759     (COMMAND_ATTRIBUTES)(CC_StartAuthSession * // 0x0176
760         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
761 #endif
762 #if (PAD_LIST || CC_VerifySignature)
763     (COMMAND_ATTRIBUTES)(CC_VerifySignature * // 0x0177
764         (IS_IMPLEMENTED+DECRYPT_2)),
765 #endif
766 #if (PAD_LIST || CC_ECC_Parameters)
767     (COMMAND_ATTRIBUTES)(CC_ECC_Parameters * // 0x0178
768         (IS_IMPLEMENTED)),
769 #endif
770 #if (PAD_LIST || CC_FirmwareRead)
771     (COMMAND_ATTRIBUTES)(CC_FirmwareRead * // 0x0179
772         (IS_IMPLEMENTED+ENCRYPT_2)),
773 #endif
774 #if (PAD_LIST || CC_GetCapability)
775     (COMMAND_ATTRIBUTES)(CC_GetCapability * // 0x017A
776         (IS_IMPLEMENTED)),
777 #endif
778 #if (PAD_LIST || CC_GetRandom)
779     (COMMAND_ATTRIBUTES)(CC_GetRandom * // 0x017B
780         (IS_IMPLEMENTED+ENCRYPT_2)),
781 #endif
782 #if (PAD_LIST || CC_GetTestResult)
783     (COMMAND_ATTRIBUTES)(CC_GetTestResult * // 0x017C
784         (IS_IMPLEMENTED+ENCRYPT_2)),
785 #endif
786 #if (PAD_LIST || CC_Hash)
787     (COMMAND_ATTRIBUTES)(CC_Hash * // 0x017D
788         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
789 #endif
790 #if (PAD_LIST || CC_PCR_Read)

```

```

791         (COMMAND_ATTRIBUTES) (CC_PCR_Read * // 0x017E
792         (IS_IMPLEMENTED)),
793     #endif
794     #if (PAD_LIST || CC_PolicyPCR)
795         (COMMAND_ATTRIBUTES) (CC_PolicyPCR * // 0x017F
796         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
797     #endif
798     #if (PAD_LIST || CC_PolicyRestart)
799         (COMMAND_ATTRIBUTES) (CC_PolicyRestart * // 0x0180
800         (IS_IMPLEMENTED+ALLOW_TRIAL)),
801     #endif
802     #if (PAD_LIST || CC_ReadClock)
803         (COMMAND_ATTRIBUTES) (CC_ReadClock * // 0x0181
804         (IS_IMPLEMENTED)),
805     #endif
806     #if (PAD_LIST || CC_PCR_Extend)
807         (COMMAND_ATTRIBUTES) (CC_PCR_Extend * // 0x0182
808         (IS_IMPLEMENTED+HANDLE_1_USER)),
809     #endif
810     #if (PAD_LIST || CC_PCR_SetAuthValue)
811         (COMMAND_ATTRIBUTES) (CC_PCR_SetAuthValue * // 0x0183
812         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
813     #endif
814     #if (PAD_LIST || CC_NV_Certify)
815         (COMMAND_ATTRIBUTES) (CC_NV_Certify * // 0x0184
816         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
817     #endif
818     #if (PAD_LIST || CC_EventSequenceComplete)
819         (COMMAND_ATTRIBUTES) (CC_EventSequenceComplete * // 0x0185
820         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),
821     #endif
822     #if (PAD_LIST || CC_HashSequenceStart)
823         (COMMAND_ATTRIBUTES) (CC_HashSequenceStart * // 0x0186
824         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
825     #endif
826     #if (PAD_LIST || CC_PolicyPhysicalPresence)
827         (COMMAND_ATTRIBUTES) (CC_PolicyPhysicalPresence * // 0x0187
828         (IS_IMPLEMENTED+ALLOW_TRIAL)),
829     #endif
830     #if (PAD_LIST || CC_PolicyDuplicationSelect)
831         (COMMAND_ATTRIBUTES) (CC_PolicyDuplicationSelect * // 0x0188
832         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
833     #endif
834     #if (PAD_LIST || CC_PolicyGetDigest)
835         (COMMAND_ATTRIBUTES) (CC_PolicyGetDigest * // 0x0189
836         (IS_IMPLEMENTED+ALLOW_TRIAL+ENCRYPT_2)),
837     #endif
838     #if (PAD_LIST || CC_TestParms)
839         (COMMAND_ATTRIBUTES) (CC_TestParms * // 0x018A
840         (IS_IMPLEMENTED)),
841     #endif
842     #if (PAD_LIST || CC_Commit)
843         (COMMAND_ATTRIBUTES) (CC_Commit * // 0x018B
844         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
845     #endif
846     #if (PAD_LIST || CC_PolicyPassword)
847         (COMMAND_ATTRIBUTES) (CC_PolicyPassword * // 0x018C
848         (IS_IMPLEMENTED+ALLOW_TRIAL)),
849     #endif
850     #if (PAD_LIST || CC_ZGen_2Phase)
851         (COMMAND_ATTRIBUTES) (CC_ZGen_2Phase * // 0x018D
852         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
853     #endif
854     #if (PAD_LIST || CC_EC_Ephemeral)
855         (COMMAND_ATTRIBUTES) (CC_EC_Ephemeral * // 0x018E
856         (IS_IMPLEMENTED+ENCRYPT_2)),

```



```

857 #endif
858 #if (PAD_LIST || CC_PolicyNvWritten)
859     (COMMAND_ATTRIBUTES)(CC_PolicyNvWritten * // 0x018F
860     (IS_IMPLEMENTED+ALLOW_TRIAL)),
861 #endif
862 #if (PAD_LIST || CC_PolicyTemplate)
863     (COMMAND_ATTRIBUTES)(CC_PolicyTemplate * // 0x0190
864     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
865 #endif
866 #if (PAD_LIST || CC_CreateLoaded)
867     (COMMAND_ATTRIBUTES)(CC_CreateLoaded * // 0x0191
868     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
869 #endif
870 #if (PAD_LIST || CC_PolicyAuthorizeNV)
871     (COMMAND_ATTRIBUTES)(CC_PolicyAuthorizeNV * // 0x0192
872     (IS_IMPLEMENTED+HANDLE_1_USER+ALLOW_TRIAL)),
873 #endif
874 #if (PAD_LIST || CC_EncryptDecrypt2)
875     (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt2 * // 0x0193
876     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
877 #endif
878 #if (PAD_LIST || CC_AC_GetCapability)
879     (COMMAND_ATTRIBUTES)(CC_AC_GetCapability * // 0x0194
880     (IS_IMPLEMENTED)),
881 #endif
882 #if (PAD_LIST || CC_AC_Send)
883     (COMMAND_ATTRIBUTES)(CC_AC_Send * // 0x0195
884     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+HANDLE_2_USER)),
885 #endif
886 #if (PAD_LIST || CC_Policy_AC_SendSelect)
887     (COMMAND_ATTRIBUTES)(CC_Policy_AC_SendSelect * // 0x0196
888     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
889 #endif
890 #if (PAD_LIST || CC_CertifyX509)
891     (COMMAND_ATTRIBUTES)(CC_CertifyX509 * // 0x0197
892     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
893 #endif
894 #if (PAD_LIST || CC_ACT_SetTimeout)
895     (COMMAND_ATTRIBUTES)(CC_ACT_SetTimeout * // 0x0198
896     (IS_IMPLEMENTED+HANDLE_1_USER)),
897 #endif
898 #if (PAD_LIST || CC_ECC_Encrypt)
899     (COMMAND_ATTRIBUTES)(CC_ECC_Encrypt * // 0x0199
900     (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
901 #endif
902 #if (PAD_LIST || CC_ECC_Decrypt)
903     (COMMAND_ATTRIBUTES)(CC_ECC_Decrypt * // 0x019A
904     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
905 #endif
906 #if (PAD_LIST || CC_PolicyCapability)
907     (COMMAND_ATTRIBUTES)(CC_PolicyCapability * // 0x019B
908     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
909 #endif
910 #if (PAD_LIST || CC_PolicyParameters)
911     (COMMAND_ATTRIBUTES)(CC_PolicyParameters * // 0x019C
912     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
913 #endif
914 #if (PAD_LIST || CC_NV_DefineSpace2)
915     (COMMAND_ATTRIBUTES)(CC_NV_DefineSpace2 * // 0x019D
916     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
917 #endif
918 #if (PAD_LIST || CC_NV_ReadPublic2)
919     (COMMAND_ATTRIBUTES)(CC_NV_ReadPublic2 * // 0x019E
920     (IS_IMPLEMENTED+ENCRYPT_2)),
921 #endif
922 #if (PAD_LIST || CC_SetCapability)

```

```

923         (COMMAND_ATTRIBUTES) (CC_SetCapability * // 0x019F
924         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
925     #endif
926     #if (PAD_LIST || CC_Vendor_TCG_Test)
927         (COMMAND_ATTRIBUTES) (CC_Vendor_TCG_Test * // 0x0000
928         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
929     #endif
930     0
931 };
932
933 #endif // _COMMAND_CODE_ATTRIBUTES_

```

6.12 /tpm/include/private/CommandAttributes.h

```

1  /* (Auto-generated)
2  *   Created by TpmStructures; Version 4.4 Mar 26, 2019
3  *   Date: Aug 30, 2019   Time: 02:11:52PM
4  */
5
6  // The attributes defined in this file are produced by the parser that
7  // creates the structure definitions from Part 3. The attributes are defined
8  // in that parser and should track the attributes being tested in
9  // CommandCodeAttributes.c. Generally, when an attribute is added to this list,
10 // new code will be needed in CommandCodeAttributes.c to test it.
11
12 #ifndef COMMAND_ATTRIBUTES_H
13 #define COMMAND_ATTRIBUTES_H
14
15 typedef UINT16 COMMAND_ATTRIBUTES;
16 #define NOT_IMPLEMENTED ((COMMAND_ATTRIBUTES) 0)
17 #define ENCRYPT_2        ((COMMAND_ATTRIBUTES) 1 << 0)
18 #define ENCRYPT_4        ((COMMAND_ATTRIBUTES) 1 << 1)
19 #define DECRYPT_2        ((COMMAND_ATTRIBUTES) 1 << 2)
20 #define DECRYPT_4        ((COMMAND_ATTRIBUTES) 1 << 3)
21 #define HANDLE_1_USER   ((COMMAND_ATTRIBUTES) 1 << 4)
22 #define HANDLE_1_ADMIN  ((COMMAND_ATTRIBUTES) 1 << 5)
23 #define HANDLE_1_DUP    ((COMMAND_ATTRIBUTES) 1 << 6)
24 #define HANDLE_2_USER   ((COMMAND_ATTRIBUTES) 1 << 7)
25 #define PP_COMMAND      ((COMMAND_ATTRIBUTES) 1 << 8)
26 #define IS_IMPLEMENTED  ((COMMAND_ATTRIBUTES) 1 << 9)
27 #define NO_SESSIONS     ((COMMAND_ATTRIBUTES) 1 << 10)
28 #define NV_COMMAND      ((COMMAND_ATTRIBUTES) 1 << 11)
29 #define PP_REQUIRED     ((COMMAND_ATTRIBUTES) 1 << 12)
30 #define R_HANDLE        ((COMMAND_ATTRIBUTES) 1 << 13)
31 #define ALLOW_TRIAL     ((COMMAND_ATTRIBUTES) 1 << 14)
32
33 #endif // COMMAND_ATTRIBUTES_H

```

6.13 /tpm/include/private/CommandDispatchData.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2  // clang-format off
3
4  // This file should only be included by CommandCodeAttributes.c
5  #ifdef _COMMAND_TABLE_DISPATCH_
6
7  // Define the stop value
8  #define END_OF_LIST      0xff
9  #define ADD_FLAG         0x80
10
11 // These macros provide some variability in how the data is encoded. They also
12 // make the lines a little shorter. :-)
13 // When TABLE_DRIVEN_MARSHAL is 'NO', the un/marshaling of parameters uses
14 // calls to the function that does the type-specific un/marshaling. When

```

```

15 // TABLE_DRIVEN_MARSHAL is 'YES', the un/marshaling of parameters calls the
16 // singular code with a value that is the offset of the data descriptor of the
17 // type.
18 #if TABLE_DRIVEN_MARSHAL
19 # define UNMARSHAL_DISPATCH(name) (marshalIndex_t)name##_MARSHAL_REF
20 # define MARSHAL_DISPATCH(name) (marshalIndex_t)name##_MARSHAL_REF
21 # define _UNMARSHAL_T marshalIndex_t
22 # define _MARSHAL_T marshalIndex_t
23 #else
24 # define UNMARSHAL_DISPATCH(name) (UNMARSHAL_t)name##_Unmarshal
25 # define MARSHAL_DISPATCH(name) (MARSHAL_t)name##_Marshal
26 # define _UNMARSHAL_T UNMARSHAL_t
27 # define _MARSHAL_T MARSHAL_t
28 #endif
29
30 // The unmarshalArray contains the dispatch functions for the unmarshaling
31 // code. The defines in this array are used to make it easier to cross
32 // reference the unmarshaling values in the types array of each command
33
34 const _UNMARSHAL_T unmarshalArray[] = {
35 #define TPMI_DH_CONTEXT H_UNMARSHAL 0
36 UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
37 #define TPMI_RH_AC_H_UNMARSHAL (TPMI_DH_CONTEXT_H_UNMARSHAL + 1)
38 UNMARSHAL_DISPATCH(TPMI_RH_AC),
39 #define TPMI_RH_ACT_H_UNMARSHAL (TPMI_RH_AC_H_UNMARSHAL + 1)
40 UNMARSHAL_DISPATCH(TPMI_RH_ACT),
41 #define TPMI_RH_CLEAR_H_UNMARSHAL (TPMI_RH_ACT_H_UNMARSHAL + 1)
42 UNMARSHAL_DISPATCH(TPMI_RH_CLEAR),
43 #define TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL (TPMI_RH_CLEAR_H_UNMARSHAL + 1)
44 UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY_AUTH),
45 #define TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL (TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL + 1)
46 UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY_POLICY),
47 #define TPMI_RH_BASE_HIERARCHY_H_UNMARSHAL (TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL + 1)
48 UNMARSHAL_DISPATCH(TPMI_RH_BASE_HIERARCHY),
49 #define TPMI_RH_LOCKOUT_H_UNMARSHAL (TPMI_RH_BASE_HIERARCHY_H_UNMARSHAL + 1)
50 UNMARSHAL_DISPATCH(TPMI_RH_LOCKOUT),
51 #define TPMI_RH_NV_AUTH_H_UNMARSHAL (TPMI_RH_LOCKOUT_H_UNMARSHAL + 1)
52 UNMARSHAL_DISPATCH(TPMI_RH_NV_AUTH),
53 #define TPMI_RH_NV_DEFINED_INDEX_H_UNMARSHAL (TPMI_RH_NV_AUTH_H_UNMARSHAL + 1)
54 UNMARSHAL_DISPATCH(TPMI_RH_NV_DEFINED_INDEX),
55 #define TPMI_RH_NV_INDEX_H_UNMARSHAL (TPMI_RH_NV_DEFINED_INDEX_H_UNMARSHAL + 1)
56 UNMARSHAL_DISPATCH(TPMI_RH_NV_INDEX),
57 #define TPMI_RH_PLATFORM_H_UNMARSHAL (TPMI_RH_NV_INDEX_H_UNMARSHAL + 1)
58 UNMARSHAL_DISPATCH(TPMI_RH_PLATFORM),
59 #define TPMI_RH_PROVISION_H_UNMARSHAL (TPMI_RH_PLATFORM_H_UNMARSHAL + 1)
60 UNMARSHAL_DISPATCH(TPMI_RH_PROVISION),
61 #define TPMI_SH_HMAC_H_UNMARSHAL (TPMI_RH_PROVISION_H_UNMARSHAL + 1)
62 UNMARSHAL_DISPATCH(TPMI_SH_HMAC),
63 #define TPMI_SH_POLICY_H_UNMARSHAL (TPMI_SH_HMAC_H_UNMARSHAL + 1)
64 UNMARSHAL_DISPATCH(TPMI_SH_POLICY),
65 // HANDLE_FIRST_FLAG_TYPE is the first handle that needs a flag when called.
66 #define HANDLE_FIRST_FLAG_TYPE (TPMI_SH_POLICY_H_UNMARSHAL + 1)
67 #define TPMI_DH_ENTITY_H_UNMARSHAL (TPMI_SH_POLICY_H_UNMARSHAL + 1)
68 UNMARSHAL_DISPATCH(TPMI_DH_ENTITY),
69 #define TPMI_DH_OBJECT_H_UNMARSHAL (TPMI_DH_ENTITY_H_UNMARSHAL + 1)
70 UNMARSHAL_DISPATCH(TPMI_DH_OBJECT),
71 #define TPMI_DH_PARENT_H_UNMARSHAL (TPMI_DH_OBJECT_H_UNMARSHAL + 1)
72 UNMARSHAL_DISPATCH(TPMI_DH_PARENT),
73 #define TPMI_DH_PCR_H_UNMARSHAL (TPMI_DH_PARENT_H_UNMARSHAL + 1)
74 UNMARSHAL_DISPATCH(TPMI_DH_PCR),
75 #define TPMI_RH_ENDORSEMENT_H_UNMARSHAL (TPMI_DH_PCR_H_UNMARSHAL + 1)
76 UNMARSHAL_DISPATCH(TPMI_RH_ENDORSEMENT),
77 #define TPMI_RH_HIERARCHY_H_UNMARSHAL (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)
78 UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
79
80

```



```

81 // PARAMETER_FIRST_TYPE marks the end of the handle list.
82 #define PARAMETER_FIRST_TYPE (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
83 #define TPM_AT_P_UNMARSHAL (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
84     UNMARSHAL_DISPATCH(TPM_AT),
85 #define TPM_CAP_P_UNMARSHAL (TPM_AT_P_UNMARSHAL + 1)
86     UNMARSHAL_DISPATCH(TPM_CAP),
87 #define TPM_CLOCK_ADJUST_P_UNMARSHAL (TPM_CAP_P_UNMARSHAL + 1)
88     UNMARSHAL_DISPATCH(TPM_CLOCK_ADJUST),
89 #define TPM_EO_P_UNMARSHAL (TPM_CLOCK_ADJUST_P_UNMARSHAL + 1)
90     UNMARSHAL_DISPATCH(TPM_EO),
91 #define TPM_SE_P_UNMARSHAL (TPM_EO_P_UNMARSHAL + 1)
92     UNMARSHAL_DISPATCH(TPM_SE),
93 #define TPM_SU_P_UNMARSHAL (TPM_SE_P_UNMARSHAL + 1)
94     UNMARSHAL_DISPATCH(TPM_SU),
95 #define TPM2B_DATA_P_UNMARSHAL (TPM_SU_P_UNMARSHAL + 1)
96     UNMARSHAL_DISPATCH(TPM2B_DATA),
97 #define TPM2B_DIGEST_P_UNMARSHAL (TPM2B_DATA_P_UNMARSHAL + 1)
98     UNMARSHAL_DISPATCH(TPM2B_DIGEST),
99 #define TPM2B_ECC_PARAMETER_P_UNMARSHAL (TPM2B_DIGEST_P_UNMARSHAL + 1)
100     UNMARSHAL_DISPATCH(TPM2B_ECC_PARAMETER),
101 #define TPM2B_ECC_POINT_P_UNMARSHAL (TPM2B_ECC_PARAMETER_P_UNMARSHAL + 1)
102     UNMARSHAL_DISPATCH(TPM2B_ECC_POINT),
103 #define TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL (TPM2B_ECC_POINT_P_UNMARSHAL + 1)
104     UNMARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
105 #define TPM2B_EVENT_P_UNMARSHAL (TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL + 1)
106     UNMARSHAL_DISPATCH(TPM2B_EVENT),
107 #define TPM2B_ID_OBJECT_P_UNMARSHAL (TPM2B_EVENT_P_UNMARSHAL + 1)
108     UNMARSHAL_DISPATCH(TPM2B_ID_OBJECT),
109 #define TPM2B_IV_P_UNMARSHAL (TPM2B_ID_OBJECT_P_UNMARSHAL + 1)
110     UNMARSHAL_DISPATCH(TPM2B_IV),
111 #define TPM2B_MAX_BUFFER_P_UNMARSHAL (TPM2B_IV_P_UNMARSHAL + 1)
112     UNMARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
113 #define TPM2B_MAX_NV_BUFFER_P_UNMARSHAL (TPM2B_MAX_BUFFER_P_UNMARSHAL + 1)
114     UNMARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
115 #define TPM2B_NAME_P_UNMARSHAL (TPM2B_MAX_NV_BUFFER_P_UNMARSHAL + 1)
116     UNMARSHAL_DISPATCH(TPM2B_NAME),
117 #define TPM2B_NV_PUBLIC_P_UNMARSHAL (TPM2B_NAME_P_UNMARSHAL + 1)
118     UNMARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
119 #define TPM2B_NV_PUBLIC_2_P_UNMARSHAL (TPM2B_NV_PUBLIC_P_UNMARSHAL + 1)
120     UNMARSHAL_DISPATCH(TPM2B_NV_PUBLIC_2),
121 #define TPM2B_PRIVATE_P_UNMARSHAL (TPM2B_NV_PUBLIC_2_P_UNMARSHAL + 1)
122     UNMARSHAL_DISPATCH(TPM2B_PRIVATE),
123 #define TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL (TPM2B_PRIVATE_P_UNMARSHAL + 1)
124     UNMARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
125 #define TPM2B_SENSITIVE_P_UNMARSHAL (TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL + 1)
126     UNMARSHAL_DISPATCH(TPM2B_SENSITIVE),
127 #define TPM2B_SENSITIVE_CREATE_P_UNMARSHAL (TPM2B_SENSITIVE_P_UNMARSHAL + 1)
128     UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_CREATE),
129 #define TPM2B_SENSITIVE_DATA_P_UNMARSHAL (TPM2B_SENSITIVE_CREATE_P_UNMARSHAL + 1)
130     UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
131 #define TPM2B_SET_CAPABILITY_DATA_P_UNMARSHAL (TPM2B_SENSITIVE_DATA_P_UNMARSHAL + 1)
132     UNMARSHAL_DISPATCH(TPM2B_SET_CAPABILITY_DATA),
133 #define TPM2B_TEMPLATE_P_UNMARSHAL (TPM2B_SET_CAPABILITY_DATA_P_UNMARSHAL +
134     1)
135     UNMARSHAL_DISPATCH(TPM2B_TEMPLATE),
136 #define TPM2B_TIMEOUT_P_UNMARSHAL (TPM2B_TEMPLATE_P_UNMARSHAL + 1)
137     UNMARSHAL_DISPATCH(TPM2B_TIMEOUT),
138 #define TPMI_DH_CONTEXT_P_UNMARSHAL (TPM2B_TIMEOUT_P_UNMARSHAL + 1)
139     UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
140 #define TPMI_DH_PERSISTENT_P_UNMARSHAL (TPMI_DH_CONTEXT_P_UNMARSHAL + 1)
141     UNMARSHAL_DISPATCH(TPMI_DH_PERSISTENT),
142 #define TPMI_YES_NO_P_UNMARSHAL (TPMI_DH_PERSISTENT_P_UNMARSHAL + 1)
143     UNMARSHAL_DISPATCH(TPMI_YES_NO),
144 #define TPML_ALG_P_UNMARSHAL (TPMI_YES_NO_P_UNMARSHAL + 1)
145     UNMARSHAL_DISPATCH(TPML_ALG),
146 #define TPML_CC_P_UNMARSHAL (TPML_ALG_P_UNMARSHAL + 1)

```

```

146         UNMARSHAL_DISPATCH(TPML_CC),
147 #define TPML_DIGEST_P_UNMARSHAL (TPML_CC_P_UNMARSHAL + 1)
148         UNMARSHAL_DISPATCH(TPML_DIGEST),
149 #define TPML_DIGEST_VALUES_P_UNMARSHAL (TPML_DIGEST_P_UNMARSHAL + 1)
150         UNMARSHAL_DISPATCH(TPML_DIGEST_VALUES),
151 #define TPML_PCR_SELECTION_P_UNMARSHAL (TPML_DIGEST_VALUES_P_UNMARSHAL + 1)
152         UNMARSHAL_DISPATCH(TPML_PCR_SELECTION),
153 #define TPMS_CONTEXT_P_UNMARSHAL (TPML_PCR_SELECTION_P_UNMARSHAL + 1)
154         UNMARSHAL_DISPATCH(TPMS_CONTEXT),
155 #define TPMT_PUBLIC_PARMS_P_UNMARSHAL (TPMS_CONTEXT_P_UNMARSHAL + 1)
156         UNMARSHAL_DISPATCH(TPMT_PUBLIC_PARMS),
157 #define TPMT_TK_AUTH_P_UNMARSHAL (TPMT_PUBLIC_PARMS_P_UNMARSHAL + 1)
158         UNMARSHAL_DISPATCH(TPMT_TK_AUTH),
159 #define TPMT_TK_CREATION_P_UNMARSHAL (TPMT_TK_AUTH_P_UNMARSHAL + 1)
160         UNMARSHAL_DISPATCH(TPMT_TK_CREATION),
161 #define TPMT_TK_HASHCHECK_P_UNMARSHAL (TPMT_TK_CREATION_P_UNMARSHAL + 1)
162         UNMARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
163 #define TPMT_TK_VERIFIED_P_UNMARSHAL (TPMT_TK_HASHCHECK_P_UNMARSHAL + 1)
164         UNMARSHAL_DISPATCH(TPMT_TK_VERIFIED),
165 #define UINT16_P_UNMARSHAL (TPMT_TK_VERIFIED_P_UNMARSHAL + 1)
166         UNMARSHAL_DISPATCH(UINT16),
167 #define UINT32_P_UNMARSHAL (UINT16_P_UNMARSHAL + 1)
168         UNMARSHAL_DISPATCH(UINT32),
169 #define UINT64_P_UNMARSHAL (UINT32_P_UNMARSHAL + 1)
170         UNMARSHAL_DISPATCH(UINT64),
171 #define UINT8_P_UNMARSHAL (UINT64_P_UNMARSHAL + 1)
172         UNMARSHAL_DISPATCH(UINT8),
173 // PARAMETER_FIRST_FLAG_TYPE is the first parameter to need a flag.
174 #define PARAMETER_FIRST_FLAG_TYPE (UINT8_P_UNMARSHAL + 1)
175 #define TPM2B_PUBLIC_P_UNMARSHAL (UINT8_P_UNMARSHAL + 1)
176         UNMARSHAL_DISPATCH(TPM2B_PUBLIC),
177 #define TPMI_ALG_CIPHER_MODE_P_UNMARSHAL (TPM2B_PUBLIC_P_UNMARSHAL + 1)
178         UNMARSHAL_DISPATCH(TPMI_ALG_CIPHER_MODE),
179 #define TPMI_ALG_HASH_P_UNMARSHAL (TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + 1)
180         UNMARSHAL_DISPATCH(TPMI_ALG_HASH),
181 #define TPMI_ALG_MAC_SCHEME_P_UNMARSHAL (TPMI_ALG_HASH_P_UNMARSHAL + 1)
182         UNMARSHAL_DISPATCH(TPMI_ALG_MAC_SCHEME),
183 #define TPMI_DH_PCR_P_UNMARSHAL (TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + 1)
184         UNMARSHAL_DISPATCH(TPMI_DH_PCR),
185 #define TPMI_ECC_CURVE_P_UNMARSHAL (TPMI_DH_PCR_P_UNMARSHAL + 1)
186         UNMARSHAL_DISPATCH(TPMI_ECC_CURVE),
187 #define TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL (TPMI_ECC_CURVE_P_UNMARSHAL + 1)
188         UNMARSHAL_DISPATCH(TPMI_ECC_KEY_EXCHANGE),
189 #define TPMI_RH_ENABLES_P_UNMARSHAL (TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL + 1)
190         UNMARSHAL_DISPATCH(TPMI_RH_ENABLES),
191 #define TPMI_RH_HIERARCHY_P_UNMARSHAL (TPMI_RH_ENABLES_P_UNMARSHAL + 1)
192         UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
193 #define TPMT_KDF_SCHEME_P_UNMARSHAL (TPMI_RH_HIERARCHY_P_UNMARSHAL + 1)
194         UNMARSHAL_DISPATCH(TPMT_KDF_SCHEME),
195 #define TPMT_RSA_DECRYPT_P_UNMARSHAL (TPMT_KDF_SCHEME_P_UNMARSHAL + 1)
196         UNMARSHAL_DISPATCH(TPMT_RSA_DECRYPT),
197 #define TPMT_SIG_SCHEME_P_UNMARSHAL (TPMT_RSA_DECRYPT_P_UNMARSHAL + 1)
198         UNMARSHAL_DISPATCH(TPMT_SIG_SCHEME),
199 #define TPMT_SIGNATURE_P_UNMARSHAL (TPMT_SIG_SCHEME_P_UNMARSHAL + 1)
200         UNMARSHAL_DISPATCH(TPMT_SIGNATURE),
201 #define TPMT_SYM_DEF_P_UNMARSHAL (TPMT_SIGNATURE_P_UNMARSHAL + 1)
202         UNMARSHAL_DISPATCH(TPMT_SYM_DEF),
203 #define TPMT_SYM_DEF_OBJECT_P_UNMARSHAL (TPMT_SYM_DEF_P_UNMARSHAL + 1)
204         UNMARSHAL_DISPATCH(TPMT_SYM_DEF_OBJECT)
205 // PARAMETER_LAST_TYPE is the index of the last command parameter.
206 #define PARAMETER_LAST_TYPE (TPMT_SYM_DEF_OBJECT_P_UNMARSHAL)
207 };
208
209 // The marshalArray contains the dispatch functions for the marshaling code.
210 // The defines in this array are used to make it easier to cross reference the
211 // marshaling values in the types array of each command

```

```

212 const _MARSHAL_T marshalArray[] = {
213 #define UINT32_H_MARSHAL                                0
214     MARSHAL_DISPATCH(UINT32),
215 // RESPONSE_PARAMETER_FIRST_TYPE marks the end of the response handles.
216 #define RESPONSE_PARAMETER_FIRST_TYPE                    (UINT32_H_MARSHAL + 1)
217 #define TPM2B_ATTEST_P_MARSHAL                          (UINT32_H_MARSHAL + 1)
218     MARSHAL_DISPATCH(TPM2B_ATTEST),
219 #define TPM2B_CREATION_DATA_P_MARSHAL                    (TPM2B_ATTEST_P_MARSHAL + 1)
220     MARSHAL_DISPATCH(TPM2B_CREATION_DATA),
221 #define TPM2B_DATA_P_MARSHAL                            (TPM2B_CREATION_DATA_P_MARSHAL + 1)
222     MARSHAL_DISPATCH(TPM2B_DATA),
223 #define TPM2B_DIGEST_P_MARSHAL                          (TPM2B_DATA_P_MARSHAL + 1)
224     MARSHAL_DISPATCH(TPM2B_DIGEST),
225 #define TPM2B_ECC_POINT_P_MARSHAL                       (TPM2B_DIGEST_P_MARSHAL + 1)
226     MARSHAL_DISPATCH(TPM2B_ECC_POINT),
227 #define TPM2B_ENCRYPTED_SECRET_P_MARSHAL                 (TPM2B_ECC_POINT_P_MARSHAL + 1)
228     MARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
229 #define TPM2B_ID_OBJECT_P_MARSHAL                      (TPM2B_ENCRYPTED_SECRET_P_MARSHAL + 1)
230     MARSHAL_DISPATCH(TPM2B_ID_OBJECT),
231 #define TPM2B_IV_P_MARSHAL                              (TPM2B_ID_OBJECT_P_MARSHAL + 1)
232     MARSHAL_DISPATCH(TPM2B_IV),
233 #define TPM2B_MAX_BUFFER_P_MARSHAL                     (TPM2B_IV_P_MARSHAL + 1)
234     MARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
235 #define TPM2B_MAX_NV_BUFFER_P_MARSHAL                   (TPM2B_MAX_BUFFER_P_MARSHAL + 1)
236     MARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
237 #define TPM2B_NAME_P_MARSHAL                           (TPM2B_MAX_NV_BUFFER_P_MARSHAL + 1)
238     MARSHAL_DISPATCH(TPM2B_NAME),
239 #define TPM2B_NV_PUBLIC_P_MARSHAL                       (TPM2B_NAME_P_MARSHAL + 1)
240     MARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
241 #define TPM2B_NV_PUBLIC_2_P_MARSHAL                    (TPM2B_NV_PUBLIC_P_MARSHAL + 1)
242     MARSHAL_DISPATCH(TPM2B_NV_PUBLIC_2),
243 #define TPM2B_PRIVATE_P_MARSHAL                        (TPM2B_NV_PUBLIC_2_P_MARSHAL + 1)
244     MARSHAL_DISPATCH(TPM2B_PRIVATE),
245 #define TPM2B_PUBLIC_P_MARSHAL                         (TPM2B_PRIVATE_P_MARSHAL + 1)
246     MARSHAL_DISPATCH(TPM2B_PUBLIC),
247 #define TPM2B_PUBLIC_KEY_RSA_P_MARSHAL                  (TPM2B_PUBLIC_P_MARSHAL + 1)
248     MARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
249 #define TPM2B_SENSITIVE_DATA_P_MARSHAL                  (TPM2B_PUBLIC_KEY_RSA_P_MARSHAL + 1)
250     MARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
251 #define TPM2B_TIMEOUT_P_MARSHAL                        (TPM2B_SENSITIVE_DATA_P_MARSHAL + 1)
252     MARSHAL_DISPATCH(TPM2B_TIMEOUT),
253 #define TPML_AC_CAPABILITIES_P_MARSHAL                  (TPM2B_TIMEOUT_P_MARSHAL + 1)
254     MARSHAL_DISPATCH(TPML_AC_CAPABILITIES),
255 #define TPML_ALG_P_MARSHAL                             (TPML_AC_CAPABILITIES_P_MARSHAL + 1)
256     MARSHAL_DISPATCH(TPML_ALG),
257 #define TPML_DIGEST_P_MARSHAL                          (TPML_ALG_P_MARSHAL + 1)
258     MARSHAL_DISPATCH(TPML_DIGEST),
259 #define TPML_DIGEST_VALUES_P_MARSHAL                   (TPML_DIGEST_P_MARSHAL + 1)
260     MARSHAL_DISPATCH(TPML_DIGEST_VALUES),
261 #define TPML_PCR_SELECTION_P_MARSHAL                   (TPML_DIGEST_VALUES_P_MARSHAL + 1)
262     MARSHAL_DISPATCH(TPML_PCR_SELECTION),
263 #define TPMS_AC_OUTPUT_P_MARSHAL                       (TPML_PCR_SELECTION_P_MARSHAL + 1)
264     MARSHAL_DISPATCH(TPMS_AC_OUTPUT),
265 #define TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL            (TPMS_AC_OUTPUT_P_MARSHAL + 1)
266     MARSHAL_DISPATCH(TPMS_ALGORITHM_DETAIL_ECC),
267 #define TPMS_CAPABILITY_DATA_P_MARSHAL                  (TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL + 1)
268     MARSHAL_DISPATCH(TPMS_CAPABILITY_DATA),
269 #define TPMS_CONTEXT_P_MARSHAL                         (TPMS_CAPABILITY_DATA_P_MARSHAL + 1)
270     MARSHAL_DISPATCH(TPMS_CONTEXT),
271 #define TPMS_TIME_INFO_P_MARSHAL                      (TPMS_CONTEXT_P_MARSHAL + 1)
272     MARSHAL_DISPATCH(TPMS_TIME_INFO),
273 #define TPMT_HA_P_MARSHAL                              (TPMS_TIME_INFO_P_MARSHAL + 1)
274     MARSHAL_DISPATCH(TPMT_HA),
275 #define TPMT_SIGNATURE_P_MARSHAL                      (TPMT_HA_P_MARSHAL + 1)
276     MARSHAL_DISPATCH(TPMT_SIGNATURE),
277 #define TPMT_TK_AUTH_P_MARSHAL                        (TPMT_SIGNATURE_P_MARSHAL + 1)

```

```

278     MARSHAL_DISPATCH(TPMT TK_AUTH),
279 #define TPMT_TK_CREATION_P_MARSHAL (TPMT_TK_AUTH_P_MARSHAL + 1)
280     MARSHAL_DISPATCH(TPMT_TK_CREATION),
281 #define TPMT_TK_HASHCHECK_P_MARSHAL (TPMT_TK_CREATION_P_MARSHAL + 1)
282     MARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
283 #define TPMT_TK_VERIFIED_P_MARSHAL (TPMT_TK_HASHCHECK_P_MARSHAL + 1)
284     MARSHAL_DISPATCH(TPMT_TK_VERIFIED),
285 #define UINT16_P_MARSHAL (TPMT_TK_VERIFIED_P_MARSHAL + 1)
286     MARSHAL_DISPATCH(UINT16),
287 #define UINT32_P_MARSHAL (UINT16_P_MARSHAL + 1)
288     MARSHAL_DISPATCH(UINT32),
289 #define UINT8_P_MARSHAL (UINT32_P_MARSHAL + 1)
290     MARSHAL_DISPATCH(UINT8)
291 // RESPONSE_PARAMETER_LAST_TYPE is the index of the last response parameter.
292 #define RESPONSE_PARAMETER_LAST_TYPE (UINT8_P_MARSHAL)
293 };
294
295 // This list of aliases allows the types in the COMMAND_DESCRIPTOR_t to match
296 // the types in the command/response templates of part 3.
297 #define TPM2B_NONCE_P_UNMARSHAL TPM2B_DIGEST_P_UNMARSHAL
298 #define TPM2B_AUTH_P_UNMARSHAL TPM2B_DIGEST_P_UNMARSHAL
299 #define TPM2B_OPERAND_P_UNMARSHAL TPM2B_DIGEST_P_UNMARSHAL
300 #define INT32_P_UNMARSHAL UINT32_P_UNMARSHAL
301 #define TPM_CC_P_UNMARSHAL UINT32_P_UNMARSHAL
302 #define TPMA_LOCALITY_P_UNMARSHAL UINT8_P_UNMARSHAL
303 #define TPMI_SH_AUTH_SESSION_H_MARSHAL UINT32_H_MARSHAL
304 #define TPM_HANDLE_H_MARSHAL UINT32_H_MARSHAL
305 #define TPMI_DH_OBJECT_H_MARSHAL UINT32_H_MARSHAL
306 #define TPMI_DH_CONTEXT_H_MARSHAL UINT32_H_MARSHAL
307 #define TPM2B_NONCE_P_MARSHAL TPM2B_DIGEST_P_MARSHAL
308 #define TPM_RC_P_MARSHAL UINT32_P_MARSHAL
309 #define TPMI_YES_NO_P_MARSHAL UINT8_P_MARSHAL
310
311 // Per-command un/marshaling tables
312
313 #if CC_Startup
314 #include "Startup_fp.h"
315
316 typedef TPM_RC (Startup_Entry)(
317     Startup_In* in
318 );
319
320
321 typedef const struct
322 {
323     Startup_Entry *entry;
324     UINT16 inSize;
325     UINT16 outSize;
326     UINT16 offsetOfTypes;
327     BYTE types[3];
328 } Startup_COMMAND_DESCRIPTOR_t;
329
330 Startup_COMMAND_DESCRIPTOR_t _StartupData = {
331     /* entry */ &TPM2_Startup,
332     /* inSize */ (UINT16)(sizeof(Startup_In)),
333     /* outSize */ 0,
334     /* offsetOfTypes */ offsetof(Startup_COMMAND_DESCRIPTOR_t, types),
335     /* offsets */ // No parameter offsets
336     /* types */ {TPM_SU_P_UNMARSHAL,
337                 END_OF_LIST,
338                 END_OF_LIST}
339 };
340
341 #define _StartupDataAddress (&_StartupData)
342 #else
343

```



```

344 #define _StartupDataAddress 0
345 #endif // CC_Startup
346
347 #if CC_Shutdown
348 #include "Shutdown_fp.h"
349
350 typedef TPM_RC (Shutdown_Entry)(
351     Shutdown_In* in
352 );
353
354
355 typedef const struct
356 {
357     Shutdown_Entry *entry;
358     UINT16 inSize;
359     UINT16 outSize;
360     UINT16 offsetOfTypes;
361     BYTE types[3];
362 } Shutdown_COMMAND_DESCRIPTOR_t;
363
364 Shutdown_COMMAND_DESCRIPTOR_t _ShutdownData = {
365     /* entry */ &TPM2_Shutdown,
366     /* inSize */ (UINT16)(sizeof(Shutdown_In)),
367     /* outSize */ 0,
368     /* offsetOfTypes */ offsetof(Shutdown_COMMAND_DESCRIPTOR_t, types),
369     /* offsets */ // No parameter offsets
370     /* types */ {TPM_SU_P_UNMARSHAL,
371                 END_OF_LIST,
372                 END_OF_LIST}
373 };
374
375 #define _ShutdownDataAddress (&_ShutdownData)
376 #else
377 #define _ShutdownDataAddress 0
378 #endif // CC_Shutdown
379
380 #if CC_SelfTest
381 #include "SelfTest_fp.h"
382
383 typedef TPM_RC (SelfTest_Entry)(
384     SelfTest_In* in
385 );
386
387
388 typedef const struct
389 {
390     SelfTest_Entry *entry;
391     UINT16 inSize;
392     UINT16 outSize;
393     UINT16 offsetOfTypes;
394     BYTE types[3];
395 } SelfTest_COMMAND_DESCRIPTOR_t;
396
397 SelfTest_COMMAND_DESCRIPTOR_t _SelfTestData = {
398     /* entry */ &TPM2_SelfTest,
399     /* inSize */ (UINT16)(sizeof(SelfTest_In)),
400     /* outSize */ 0,
401     /* offsetOfTypes */ offsetof(SelfTest_COMMAND_DESCRIPTOR_t, types),
402     /* offsets */ // No parameter offsets
403     /* types */ {TPMI_YES_NO_P_UNMARSHAL,
404                 END_OF_LIST,
405                 END_OF_LIST}
406 };
407
408 #define _SelfTestDataAddress (&_SelfTestData)
409 #else

```

```

410 #define _SelfTestDataAddress 0
411 #endif // CC_SelfTest
412
413 #if CC_IncrementalSelfTest
414 #include "IncrementalSelfTest_fp.h"
415
416 typedef TPM_RC (IncrementalSelfTest_Entry) (
417     IncrementalSelfTest_In* in,
418     IncrementalSelfTest_Out* out
419 );
420
421
422 typedef const struct
423 {
424     IncrementalSelfTest_Entry *entry;
425     UINT16 inSize;
426     UINT16 outSize;
427     UINT16 offsetOfTypes;
428     BYTE types[4];
429 } IncrementalSelfTest_COMMAND_DESCRIPTOR_t;
430
431 IncrementalSelfTest_COMMAND_DESCRIPTOR_t _IncrementalSelfTestData = {
432     /* entry */ &TPM2_IncrementalSelfTest,
433     /* inSize */ (UINT16) (sizeof(IncrementalSelfTest_In)),
434     /* outSize */ (UINT16) (sizeof(IncrementalSelfTest_Out)),
435     /* offsetOfTypes */ offsetof(IncrementalSelfTest_COMMAND_DESCRIPTOR_t,
types),
436     /* offsets */ // No parameter offsets
437     /* types */ {TPML_ALG_P_UNMARSHAL,
END_OF_LIST,
TPML_ALG_P_MARSHAL,
END_OF_LIST}
441 };
442
443 #define _IncrementalSelfTestDataAddress (&_IncrementalSelfTestData)
444 #else
445 #define _IncrementalSelfTestDataAddress 0
446 #endif // CC_IncrementalSelfTest
447
448 #if CC_GetTestResult
449 #include "GetTestResult_fp.h"
450
451 typedef TPM_RC (GetTestResult_Entry) (
452     GetTestResult_Out* out
453 );
454
455
456 typedef const struct
457 {
458     GetTestResult_Entry *entry;
459     UINT16 inSize;
460     UINT16 outSize;
461     UINT16 offsetOfTypes;
462     UINT16 paramOffsets[1];
463     BYTE types[4];
464 } GetTestResult_COMMAND_DESCRIPTOR_t;
465
466 GetTestResult_COMMAND_DESCRIPTOR_t _GetTestResultData = {
467     /* entry */ &TPM2_GetTestResult,
468     /* inSize */ 0,
469     /* outSize */ (UINT16) (sizeof(GetTestResult_Out)),
470     /* offsetOfTypes */ offsetof(GetTestResult_COMMAND_DESCRIPTOR_t, types),
471     /* offsets */ { (UINT16) (offsetof(GetTestResult_Out, testResult))},
472     /* types */ {END_OF_LIST,
TPM2B_MAX_BUFFER_P_MARSHAL,
TPM_RC_P_MARSHAL,

```

```

475                                     END_OF_LIST}
476 };
477
478 #define _GetTestResultDataAddress (&_GetTestResultData)
479 #else
480 #define _GetTestResultDataAddress 0
481 #endif // CC_GetTestResult
482
483 #if CC_StartAuthSession
484 #include "StartAuthSession_fp.h"
485
486 typedef TPM_RC (StartAuthSession_Entry) (
487     StartAuthSession_In*    in,
488     StartAuthSession_Out*   out
489 );
490
491
492 typedef const struct
493 {
494     StartAuthSession_Entry    *entry;
495     UINT16                    inSize;
496     UINT16                    outSize;
497     UINT16                    offsetOfTypes;
498     UINT16                    paramOffsets[7];
499     BYTE                      types[11];
500 } StartAuthSession_COMMAND_DESCRIPTOR_t;
501
502 StartAuthSession_COMMAND_DESCRIPTOR_t _StartAuthSessionData = {
503     /* entry */                &TPM2_StartAuthSession,
504     /* inSize */               (UINT16) (sizeof(StartAuthSession_In)),
505     /* outSize */              (UINT16) (sizeof(StartAuthSession_Out)),
506     /* offsetOfTypes */        offsetof(StartAuthSession_COMMAND_DESCRIPTOR_t,
507 types),
508     /* offsets */              { (UINT16) (offsetof(StartAuthSession_In, bind)),
509 (UINT16) (offsetof(StartAuthSession_In, nonceCaller)),
510 (UINT16) (offsetof(StartAuthSession_In,
511 encryptedSalt)),
512 (UINT16) (offsetof(StartAuthSession_In, sessionType)),
513 (UINT16) (offsetof(StartAuthSession_In, symmetric)),
514 (UINT16) (offsetof(StartAuthSession_In, authHash)),
515 (UINT16) (offsetof(StartAuthSession_Out, nonceTPM)) },
516     /* types */                {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
517 TPMI_DH_ENTITY_H_UNMARSHAL + ADD_FLAG,
518 TPM2B_NONCE_P_UNMARSHAL,
519 TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
520 TPM_SE_P_UNMARSHAL,
521 TPMT_SYM_DEF_P_UNMARSHAL + ADD_FLAG,
522 TPMI_ALG_HASH_P_UNMARSHAL,
523 END_OF_LIST,
524 TPMI_SH_AUTH_SESSION_H_MARSHAL,
525 TPM2B_NONCE_P_MARSHAL,
526 END_OF_LIST}
527 };
528
529 #define _StartAuthSessionDataAddress (&_StartAuthSessionData)
530 #else
531 #define _StartAuthSessionDataAddress 0
532 #endif // CC_StartAuthSession
533
534 #if CC_PolicyRestart
535 #include "PolicyRestart_fp.h"
536
537 typedef TPM_RC (PolicyRestart_Entry) (
538     PolicyRestart_In*        in
539 );
540

```

```

539
540 typedef const struct
541 {
542     PolicyRestart_Entry      *entry;
543     UINT16                   inSize;
544     UINT16                   outSize;
545     UINT16                   offsetOfTypes;
546     BYTE                     types[3];
547 } PolicyRestart_COMMAND_DESCRIPTOR_t;
548
549 PolicyRestart_COMMAND_DESCRIPTOR_t _PolicyRestartData = {
550     /* entry */           &TPM2_PolicyRestart,
551     /* inSize */          (UINT16) (sizeof(PolicyRestart_In)),
552     /* outSize */         0,
553     /* offsetOfTypes */   offsetof(PolicyRestart_COMMAND_DESCRIPTOR_t, types),
554     /* offsets */          // No parameter offsets
555     /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
556                           END_OF_LIST,
557                           END_OF_LIST};
558 };
559
560 #define _PolicyRestartDataAddress (&_PolicyRestartData)
561 #else
562 #define _PolicyRestartDataAddress 0
563 #endif // CC_PolicyRestart
564
565 #if CC_Create
566 #include "Create_fp.h"
567
568 typedef TPM_RC (Create_Entry) (
569     Create_In*           in,
570     Create_Out*          out
571 );
572
573 typedef const struct
574 {
575     Create_Entry          *entry;
576     UINT16                inSize;
577     UINT16                outSize;
578     UINT16                offsetOfTypes;
579     UINT16                paramOffsets[8];
580     BYTE                  types[12];
581 } Create_COMMAND_DESCRIPTOR_t;
582
583 Create_COMMAND_DESCRIPTOR_t _CreateData = {
584     /* entry */           &TPM2_Create,
585     /* inSize */          (UINT16) (sizeof(Create_In)),
586     /* outSize */         (UINT16) (sizeof(Create_Out)),
587     /* offsetOfTypes */   offsetof(Create_COMMAND_DESCRIPTOR_t, types),
588     /* offsets */          {(UINT16) (offsetof(Create_In, inSensitive)),
589                             (UINT16) (offsetof(Create_In, inPublic)),
590                             (UINT16) (offsetof(Create_In, outsideInfo)),
591                             (UINT16) (offsetof(Create_In, creationPCR)),
592                             (UINT16) (offsetof(Create_Out, outPublic)),
593                             (UINT16) (offsetof(Create_Out, creationData)),
594                             (UINT16) (offsetof(Create_Out, creationHash)),
595                             (UINT16) (offsetof(Create_Out, creationTicket))},
596     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
597                             TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
598                             TPM2B_PUBLIC_P_UNMARSHAL,
599                             TPM2B_DATA_P_UNMARSHAL,
600                             TPML_PCR_SELECTION_P_UNMARSHAL,
601                             END_OF_LIST,
602                             TPM2B_PRIVATE_P_MARSHAL,
603                             TPM2B_PUBLIC_P_MARSHAL,
604

```



```

605         TPM2B_CREATION_DATA_P_MARSHAL,
606         TPM2B_DIGEST_P_MARSHAL,
607         TPMT_TK_CREATION_P_MARSHAL,
608         END_OF_LIST}
609 };
610
611 #define _CreateDataAddress (&_CreateData)
612 #else
613 #define _CreateDataAddress 0
614 #endif // CC_Create
615
616 #if CC_Load
617 #include "Load_fp.h"
618
619 typedef TPM_RC (Load_Entry) (
620     Load_In*          in,
621     Load_Out*         out
622 );
623
624
625 typedef const struct
626 {
627     Load_Entry          *entry;
628     UINT16              inSize;
629     UINT16              outSize;
630     UINT16              offsetOfTypes;
631     UINT16              paramOffsets[3];
632     BYTE                types[7];
633 } Load_COMMAND_DESCRIPTOR_t;
634
635 Load_COMMAND_DESCRIPTOR_t _LoadData = {
636     /* entry */          &TPM2_Load,
637     /* inSize */         (UINT16) (sizeof(Load_In)),
638     /* outSize */        (UINT16) (sizeof(Load_Out)),
639     /* offsetOfTypes */  offsetof(Load_COMMAND_DESCRIPTOR_t, types),
640     /* offsets */        {(UINT16) (offsetof(Load_In, inPrivate)),
641                          (UINT16) (offsetof(Load_In, inPublic)),
642                          (UINT16) (offsetof(Load_Out, name))},
643     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
644                          TPM2B_PRIVATE_P_UNMARSHAL,
645                          TPM2B_PUBLIC_P_UNMARSHAL,
646                          END_OF_LIST,
647                          TPM_HANDLE_H_MARSHAL,
648                          TPM2B_NAME_P_MARSHAL,
649                          END_OF_LIST}
650 };
651
652 #define _LoadDataAddress (&_LoadData)
653 #else
654 #define _LoadDataAddress 0
655 #endif // CC_Load
656
657 #if CC_LoadExternal
658 #include "LoadExternal_fp.h"
659
660 typedef TPM_RC (LoadExternal_Entry) (
661     LoadExternal_In*   in,
662     LoadExternal_Out*  out
663 );
664
665
666 typedef const struct
667 {
668     LoadExternal_Entry  *entry;
669     UINT16              inSize;
670     UINT16              outSize;

```

```

671     UINT16                offsetOfTypes;
672     UINT16                paramOffsets[3];
673     BYTE                  types[7];
674 } LoadExternal_COMMAND_DESCRIPTOR_t;
675
676 LoadExternal_COMMAND_DESCRIPTOR_t _LoadExternalData = {
677     /* entry */           &TPM2_LoadExternal,
678     /* inSize */          (UINT16) (sizeof(LoadExternal_In)),
679     /* outSize */         (UINT16) (sizeof(LoadExternal_Out)),
680     /* offsetOfTypes */   offsetof(LoadExternal_COMMAND_DESCRIPTOR_t, types),
681     /* offsets */         {(UINT16) (offsetof(LoadExternal_In, inPublic)),
682                          (UINT16) (offsetof(LoadExternal_In, hierarchy)),
683                          (UINT16) (offsetof(LoadExternal_Out, name))},
684     /* types */           {TPM2B_SENSITIVE_P_UNMARSHAL,
685                          TPM2B_PUBLIC_P_UNMARSHAL + ADD_FLAG,
686                          TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
687                          END_OF_LIST,
688                          TPM_HANDLE_H_MARSHAL,
689                          TPM2B_NAME_P_MARSHAL,
690                          END_OF_LIST};
691 };
692
693 #define _LoadExternalDataAddress (&_LoadExternalData)
694 #else
695 #define _LoadExternalDataAddress 0
696 #endif // CC_LoadExternal
697
698 #if CC_ReadPublic
699 #include "ReadPublic_fp.h"
700
701 typedef TPM_RC (ReadPublic_Entry) (
702     ReadPublic_In*      in,
703     ReadPublic_Out*     out
704 );
705
706
707 typedef const struct
708 {
709     ReadPublic_Entry     *entry;
710     UINT16               inSize;
711     UINT16               outSize;
712     UINT16               offsetOfTypes;
713     UINT16               paramOffsets[2];
714     BYTE                 types[6];
715 } ReadPublic_COMMAND_DESCRIPTOR_t;
716
717 ReadPublic_COMMAND_DESCRIPTOR_t _ReadPublicData = {
718     /* entry */           &TPM2_ReadPublic,
719     /* inSize */          (UINT16) (sizeof(ReadPublic_In)),
720     /* outSize */         (UINT16) (sizeof(ReadPublic_Out)),
721     /* offsetOfTypes */   offsetof(ReadPublic_COMMAND_DESCRIPTOR_t, types),
722     /* offsets */         {(UINT16) (offsetof(ReadPublic_Out, name)),
723                          (UINT16) (offsetof(ReadPublic_Out, qualifiedName))},
724     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
725                          END_OF_LIST,
726                          TPM2B_PUBLIC_P_MARSHAL,
727                          TPM2B_NAME_P_MARSHAL,
728                          TPM2B_NAME_P_MARSHAL,
729                          END_OF_LIST};
730 };
731
732 #define _ReadPublicDataAddress (&_ReadPublicData)
733 #else
734 #define _ReadPublicDataAddress 0
735 #endif // CC_ReadPublic
736

```

```

737 #if CC_ActivateCredential
738 #include "ActivateCredential_fp.h"
739
740 typedef TPM_RC (ActivateCredential_Entry) (
741     ActivateCredential_In* in,
742     ActivateCredential_Out* out
743 );
744
745
746 typedef const struct
747 {
748     ActivateCredential_Entry *entry;
749     UINT16 inSize;
750     UINT16 outSize;
751     UINT16 offsetOfTypes;
752     UINT16 paramOffsets[3];
753     BYTE types[7];
754 } ActivateCredential_COMMAND_DESCRIPTOR_t;
755
756 ActivateCredential_COMMAND_DESCRIPTOR_t _ActivateCredentialData = {
757     /* entry */ &TPM2_ActivateCredential,
758     /* inSize */ (UINT16) (sizeof(ActivateCredential_In)),
759     /* outSize */ (UINT16) (sizeof(ActivateCredential_Out)),
760     /* offsetOfTypes */ offsetof(ActivateCredential_COMMAND_DESCRIPTOR_t,
types),
761     /* offsets */ { (UINT16) (offsetof(ActivateCredential_In, keyHandle)),
762                     (UINT16) (offsetof(ActivateCredential_In,
credentialBlob)),
763                     (UINT16) (offsetof(ActivateCredential_In, secret))},
764     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
765                  TPMI_DH_OBJECT_H_UNMARSHAL,
766                  TPM2B_ID_OBJECT_P_UNMARSHAL,
767                  TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
768                  END_OF_LIST,
769                  TPM2B_DIGEST_P_MARSHAL,
770                  END_OF_LIST}
771 };
772
773 #define _ActivateCredentialDataAddress (&_ActivateCredentialData)
774 #else
775 #define _ActivateCredentialDataAddress 0
776 #endif // CC_ActivateCredential
777
778 #if CC_MakeCredential
779 #include "MakeCredential_fp.h"
780
781 typedef TPM_RC (MakeCredential_Entry) (
782     MakeCredential_In* in,
783     MakeCredential_Out* out
784 );
785
786
787 typedef const struct
788 {
789     MakeCredential_Entry *entry;
790     UINT16 inSize;
791     UINT16 outSize;
792     UINT16 offsetOfTypes;
793     UINT16 paramOffsets[3];
794     BYTE types[7];
795 } MakeCredential_COMMAND_DESCRIPTOR_t;
796
797 MakeCredential_COMMAND_DESCRIPTOR_t _MakeCredentialData = {
798     /* entry */ &TPM2_MakeCredential,
799     /* inSize */ (UINT16) (sizeof(MakeCredential_In)),
800     /* outSize */ (UINT16) (sizeof(MakeCredential_Out)),

```

```

801     /* offsetOfTypes */           offsetof(MakeCredential_COMMAND_DESCRIPTOR_t, types),
802     /* offsets */                 {(UINT16) (offsetof(MakeCredential_In, credential)),
803                                   (UINT16) (offsetof(MakeCredential_In, objectName)),
804                                   (UINT16) (offsetof(MakeCredential_Out, secret))},
805     /* types */                   {TPMI_DH_OBJECT_H_UNMARSHAL,
806                                   TPM2B_DIGEST_P_UNMARSHAL,
807                                   TPM2B_NAME_P_UNMARSHAL,
808                                   END_OF_LIST,
809                                   TPM2B_ID_OBJECT_P_MARSHAL,
810                                   TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
811                                   END_OF_LIST}
812 };
813
814 #define _MakeCredentialDataAddress (&_MakeCredentialData)
815 #else
816 #define _MakeCredentialDataAddress 0
817 #endif // CC_MakeCredential
818
819 #if CC_Unseal
820 #include "Unseal_fp.h"
821
822 typedef TPM_RC (Unseal_Entry) (
823     Unseal_In*           in,
824     Unseal_Out*          out
825 );
826
827
828 typedef const struct
829 {
830     Unseal_Entry          *entry;
831     UINT16                inSize;
832     UINT16                outSize;
833     UINT16                offsetOfTypes;
834     BYTE                  types[4];
835 } Unseal_COMMAND_DESCRIPTOR_t;
836
837 Unseal_COMMAND_DESCRIPTOR_t _UnsealData = {
838     /* entry */             &TPM2_Unseal,
839     /* inSize */            (UINT16) (sizeof(Unseal_In)),
840     /* outSize */           (UINT16) (sizeof(Unseal_Out)),
841     /* offsetOfTypes */     offsetof(Unseal_COMMAND_DESCRIPTOR_t, types),
842     /* offsets */           // No parameter offsets
843     /* types */             {TPMI_DH_OBJECT_H_UNMARSHAL,
844                             END_OF_LIST,
845                             TPM2B_SENSITIVE_DATA_P_MARSHAL,
846                             END_OF_LIST}
847 };
848
849 #define _UnsealDataAddress (&_UnsealData)
850 #else
851 #define _UnsealDataAddress 0
852 #endif // CC_Unseal
853
854 #if CC_ObjectChangeAuth
855 #include "ObjectChangeAuth_fp.h"
856
857 typedef TPM_RC (ObjectChangeAuth_Entry) (
858     ObjectChangeAuth_In*   in,
859     ObjectChangeAuth_Out*  out
860 );
861
862
863 typedef const struct
864 {
865     ObjectChangeAuth_Entry *entry;
866     UINT16                inSize;

```

```

867     UINT16                outSize;
868     UINT16                offsetOfTypes;
869     UINT16                paramOffsets[2];
870     BYTE                  types[6];
871 } ObjectChangeAuth_COMMAND_DESCRIPTOR_t;
872
873 ObjectChangeAuth_COMMAND_DESCRIPTOR_t _ObjectChangeAuthData = {
874     /* entry */                &TPM2_ObjectChangeAuth,
875     /* inSize */              (UINT16) (sizeof(ObjectChangeAuth_In)),
876     /* outSize */             (UINT16) (sizeof(ObjectChangeAuth_Out)),
877     /* offsetOfTypes */       offsetof(ObjectChangeAuth_COMMAND_DESCRIPTOR_t,
types),
878     /* offsets */             { (UINT16) (offsetof(ObjectChangeAuth_In,
parentHandle)),
                                (UINT16) (offsetof(ObjectChangeAuth_In, newAuth))},
879     /* types */               {TPMI_DH_OBJECT_H_UNMARSHAL,
                                TPMI_DH_OBJECT_H_UNMARSHAL,
                                TPM2B_AUTH_P_UNMARSHAL,
                                END_OF_LIST,
                                TPM2B_PRIVATE_P_MARSHAL,
                                END_OF_LIST};
880
881 };
882
883 #define _ObjectChangeAuthDataAddress (&_ObjectChangeAuthData)
884 #else
885 #define _ObjectChangeAuthDataAddress 0
886 #endif // CC_ObjectChangeAuth
887
888 #if CC_CreateLoaded
889 #include "CreateLoaded_fp.h"
890
891 typedef TPM_RC (CreateLoaded_Entry) (
892     CreateLoaded_In*        in,
893     CreateLoaded_Out*       out
894 );
895
896 typedef const struct
897 {
898     CreateLoaded_Entry      *entry;
899     UINT16                  inSize;
900     UINT16                  outSize;
901     UINT16                  offsetOfTypes;
902     UINT16                  paramOffsets[5];
903     BYTE                    types[9];
904 } CreateLoaded_COMMAND_DESCRIPTOR_t;
905
906 CreateLoaded_COMMAND_DESCRIPTOR_t _CreateLoadedData = {
907     /* entry */                &TPM2_CreateLoaded,
908     /* inSize */              (UINT16) (sizeof(CreateLoaded_In)),
909     /* outSize */             (UINT16) (sizeof(CreateLoaded_Out)),
910     /* offsetOfTypes */       offsetof(CreateLoaded_COMMAND_DESCRIPTOR_t, types),
911     /* offsets */             { (UINT16) (offsetof(CreateLoaded_In, inSensitive)),
                                (UINT16) (offsetof(CreateLoaded_In, inPublic)),
                                (UINT16) (offsetof(CreateLoaded_Out, outPrivate)),
                                (UINT16) (offsetof(CreateLoaded_Out, outPublic)),
                                (UINT16) (offsetof(CreateLoaded_Out, name))},
912     /* types */               {TPMI_DH_PARENT_H_UNMARSHAL + ADD_FLAG,
                                TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
                                TPM2B_TEMPLATE_P_UNMARSHAL,
                                END_OF_LIST,
                                TPM_HANDLE_H_MARSHAL,
                                TPM2B_PRIVATE_P_MARSHAL,
                                TPM2B_PUBLIC_P_MARSHAL,
                                TPM2B_NAME_P_MARSHAL,
                                END_OF_LIST};
913
914 };
915
916 #endif
917
918 #endif
919
920 #endif
921
922 #endif
923
924 #endif
925
926 #endif
927
928 #endif
929
930 #endif

```

```

931 };
932
933 #define _CreateLoadedDataAddress (&_CreateLoadedData)
934 #else
935 #define _CreateLoadedDataAddress 0
936 #endif // CC_CreateLoaded
937
938 #if CC_Duplicate
939 #include "Duplicate_fp.h"
940
941 typedef TPM_RC (Duplicate_Entry) (
942     Duplicate_In*      in,
943     Duplicate_Out*     out
944 );
945
946
947 typedef const struct
948 {
949     Duplicate_Entry      *entry;
950     UINT16               inSize;
951     UINT16               outSize;
952     UINT16               offsetOfTypes;
953     UINT16               paramOffsets[5];
954     BYTE                 types[9];
955 } Duplicate_COMMAND_DESCRIPTOR_t;
956
957 Duplicate_COMMAND_DESCRIPTOR_t _DuplicateData = {
958     /* entry */           &TPM2_Duplicate,
959     /* inSize */          (UINT16) (sizeof(Duplicate_In)),
960     /* outSize */         (UINT16) (sizeof(Duplicate_Out)),
961     /* offsetOfTypes */   offsetof(Duplicate_COMMAND_DESCRIPTOR_t, types),
962     /* offsets */         { (UINT16) (offsetof(Duplicate_In, newParentHandle)),
963                           (UINT16) (offsetof(Duplicate_In, encryptionKeyIn)),
964                           (UINT16) (offsetof(Duplicate_In, symmetricAlg)),
965                           (UINT16) (offsetof(Duplicate_Out, duplicate)),
966                           (UINT16) (offsetof(Duplicate_Out, outSymSeed)) },
967     /* types */           { TPMI_DH_OBJECT_H_UNMARSHAL,
968                           TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
969                           TPM2B_DATA_P_UNMARSHAL,
970                           TPMI_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
971                           END_OF_LIST,
972                           TPM2B_DATA_P_MARSHAL,
973                           TPM2B_PRIVATE_P_MARSHAL,
974                           TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
975                           END_OF_LIST }
976 };
977
978 #define _DuplicateDataAddress (&_DuplicateData)
979 #else
980 #define _DuplicateDataAddress 0
981 #endif // CC_Duplicate
982
983 #if CC_Rewrap
984 #include "Rewrap_fp.h"
985
986 typedef TPM_RC (Rewrap_Entry) (
987     Rewrap_In*          in,
988     Rewrap_Out*         out
989 );
990
991
992 typedef const struct
993 {
994     Rewrap_Entry         *entry;
995     UINT16               inSize;
996     UINT16               outSize;

```



```

997     UINT16      offsetOfTypes;
998     UINT16      paramOffsets[5];
999     BYTE        types[9];
1000 } Rewrap_COMMAND_DESCRIPTOR_t;
1001
1002 Rewrap_COMMAND_DESCRIPTOR_t _RewrapData = {
1003     /* entry */          &TPM2_Rewrap,
1004     /* inSize */         (UINT16) (sizeof(Rewrap_In)),
1005     /* outSize */        (UINT16) (sizeof(Rewrap_Out)),
1006     /* offsetOfTypes */  offsetof(Rewrap_COMMAND_DESCRIPTOR_t, types),
1007     /* offsets */         { (UINT16) (offsetof(Rewrap_In, newParent)),
1008                           (UINT16) (offsetof(Rewrap_In, inDuplicate)),
1009                           (UINT16) (offsetof(Rewrap_In, name)),
1010                           (UINT16) (offsetof(Rewrap_In, inSymSeed)),
1011                           (UINT16) (offsetof(Rewrap_Out, outSymSeed)) },
1012     /* types */          { TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1013                           TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1014                           TPM2B_PRIVATE_P_UNMARSHAL,
1015                           TPM2B_NAME_P_UNMARSHAL,
1016                           TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
1017                           END_OF_LIST,
1018                           TPM2B_PRIVATE_P_MARSHAL,
1019                           TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
1020                           END_OF_LIST };
1021 };
1022
1023 #define _RewrapDataAddress (&_RewrapData)
1024 #else
1025 #define _RewrapDataAddress 0
1026 #endif // CC_Rewrap
1027
1028 #if CC_Import
1029 #include "Import_fp.h"
1030
1031 typedef TPM_RC (Import_Entry) (
1032     Import_In*      in,
1033     Import_Out*     out
1034 );
1035
1036
1037 typedef const struct
1038 {
1039     Import_Entry     *entry;
1040     UINT16           inSize;
1041     UINT16           outSize;
1042     UINT16           offsetOfTypes;
1043     UINT16           paramOffsets[5];
1044     BYTE            types[9];
1045 } Import_COMMAND_DESCRIPTOR_t;
1046
1047 Import_COMMAND_DESCRIPTOR_t _ImportData = {
1048     /* entry */          &TPM2_Import,
1049     /* inSize */         (UINT16) (sizeof(Import_In)),
1050     /* outSize */        (UINT16) (sizeof(Import_Out)),
1051     /* offsetOfTypes */  offsetof(Import_COMMAND_DESCRIPTOR_t, types),
1052     /* offsets */         { (UINT16) (offsetof(Import_In, encryptionKey)),
1053                           (UINT16) (offsetof(Import_In, objectPublic)),
1054                           (UINT16) (offsetof(Import_In, duplicate)),
1055                           (UINT16) (offsetof(Import_In, inSymSeed)),
1056                           (UINT16) (offsetof(Import_In, symmetricAlg)) },
1057     /* types */          { TPMI_DH_OBJECT_H_UNMARSHAL,
1058                           TPM2B_DATA_P_UNMARSHAL,
1059                           TPM2B_PUBLIC_P_UNMARSHAL,
1060                           TPM2B_PRIVATE_P_UNMARSHAL,
1061                           TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
1062                           TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,

```

```

1063                                     END_OF_LIST,
1064                                     TPM2B_PRIVATE_P_MARSHAL,
1065                                     END_OF_LIST}
1066 };
1067
1068 #define _ImportDataAddress (&_ImportData)
1069 #else
1070 #define _ImportDataAddress 0
1071 #endif // CC_Import
1072
1073 #if CC_RSA_Encrypt
1074 #include "RSA_Encrypt_fp.h"
1075
1076 typedef TPM_RC (RSA_Encrypt_Entry) (
1077     RSA_Encrypt_In*      in,
1078     RSA_Encrypt_Out*     out
1079 );
1080
1081
1082 typedef const struct
1083 {
1084     RSA_Encrypt_Entry      *entry;
1085     UINT16                 inSize;
1086     UINT16                 outSize;
1087     UINT16                 offsetOfTypes;
1088     UINT16                 paramOffsets[3];
1089     BYTE                   types[7];
1090 } RSA_Encrypt_COMMAND_DESCRIPTOR_t;
1091
1092 RSA_Encrypt_COMMAND_DESCRIPTOR_t _RSA_EncryptData = {
1093     /* entry */           &TPM2_RSA_Encrypt,
1094     /* inSize */          (UINT16) (sizeof(RSA_Encrypt_In)),
1095     /* outSize */         (UINT16) (sizeof(RSA_Encrypt_Out)),
1096     /* offsetOfTypes */   offsetof(RSA_Encrypt_COMMAND_DESCRIPTOR_t, types),
1097     /* offsets */         { (UINT16) (offsetof(RSA_Encrypt_In, message)),
1098                           (UINT16) (offsetof(RSA_Encrypt_In, inScheme)),
1099                           (UINT16) (offsetof(RSA_Encrypt_In, label)) },
1100     /* types */           { TPMI_DH_OBJECT_H_UNMARSHAL,
1101                           TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
1102                           TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
1103                           TPM2B_DATA_P_UNMARSHAL,
1104                           END_OF_LIST,
1105                           TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
1106                           END_OF_LIST }
1107 };
1108
1109 #define _RSA_EncryptDataAddress (&_RSA_EncryptData)
1110 #else
1111 #define _RSA_EncryptDataAddress 0
1112 #endif // CC_RSA_Encrypt
1113
1114 #if CC_RSA_Decrypt
1115 #include "RSA_Decrypt_fp.h"
1116
1117 typedef TPM_RC (RSA_Decrypt_Entry) (
1118     RSA_Decrypt_In*      in,
1119     RSA_Decrypt_Out*     out
1120 );
1121
1122
1123 typedef const struct
1124 {
1125     RSA_Decrypt_Entry      *entry;
1126     UINT16                 inSize;
1127     UINT16                 outSize;
1128     UINT16                 offsetOfTypes;

```



```

1129     UINT16                paramOffsets[3];
1130     BYTE                   types[7];
1131 } RSA_Decrypt_COMMAND_DESCRIPTOR_t;
1132
1133 RSA_Decrypt_COMMAND_DESCRIPTOR_t _RSA_DecryptData = {
1134     /* entry */           &TPM2_RSA_Decrypt,
1135     /* inSize */          (UINT16) (sizeof(RSA_Decrypt_In)),
1136     /* outSize */         (UINT16) (sizeof(RSA_Decrypt_Out)),
1137     /* offsetOfTypes */   offsetof(RSA_Decrypt_COMMAND_DESCRIPTOR_t, types),
1138     /* offsets */         { (UINT16) (offsetof(RSA_Decrypt_In, cipherText)),
1139                           (UINT16) (offsetof(RSA_Decrypt_In, inScheme)),
1140                           (UINT16) (offsetof(RSA_Decrypt_In, label)) },
1141     /* types */           { TPMI_DH_OBJECT_H_UNMARSHAL,
1142                           TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
1143                           TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
1144                           TPM2B_DATA_P_UNMARSHAL,
1145                           END_OF_LIST,
1146                           TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
1147                           END_OF_LIST };
1148 };
1149
1150 #define _RSA_DecryptDataAddress (&_RSA_DecryptData)
1151 #else
1152 #define _RSA_DecryptDataAddress 0
1153 #endif // CC_RSA_Decrypt
1154
1155 #if CC_ECDH_KeyGen
1156 #include "ECDH_KeyGen_fp.h"
1157
1158 typedef TPM_RC (ECDH_KeyGen_Entry) (
1159     ECDH_KeyGen_In*      in,
1160     ECDH_KeyGen_Out*     out
1161 );
1162
1163 typedef const struct
1164 {
1165     ECDH_KeyGen_Entry     *entry;
1166     UINT16                inSize;
1167     UINT16                outSize;
1168     UINT16                offsetOfTypes;
1169     UINT16                paramOffsets[1];
1170     BYTE                   types[5];
1171 } ECDH_KeyGen_COMMAND_DESCRIPTOR_t;
1172
1173 ECDH_KeyGen_COMMAND_DESCRIPTOR_t _ECDH_KeyGenData = {
1174     /* entry */           &TPM2_ECDH_KeyGen,
1175     /* inSize */          (UINT16) (sizeof(ECDH_KeyGen_In)),
1176     /* outSize */         (UINT16) (sizeof(ECDH_KeyGen_Out)),
1177     /* offsetOfTypes */   offsetof(ECDH_KeyGen_COMMAND_DESCRIPTOR_t, types),
1178     /* offsets */         { (UINT16) (offsetof(ECDH_KeyGen_Out, pubPoint)) },
1179     /* types */           { TPMI_DH_OBJECT_H_UNMARSHAL,
1180                           END_OF_LIST,
1181                           TPM2B_ECC_POINT_P_MARSHAL,
1182                           TPM2B_ECC_POINT_P_MARSHAL,
1183                           END_OF_LIST };
1184 };
1185
1186 #define _ECDH_KeyGenDataAddress (&_ECDH_KeyGenData)
1187 #else
1188 #define _ECDH_KeyGenDataAddress 0
1189 #endif // CC_ECDH_KeyGen
1190
1191 #if CC_ECDH_ZGen
1192 #include "ECDH_ZGen_fp.h"
1193
1194

```

```

1195 typedef TPM_RC (ECDH_ZGen_Entry) (
1196     ECDH_ZGen_In*          in,
1197     ECDH_ZGen_Out*         out
1198 );
1199
1200
1201 typedef const struct
1202 {
1203     ECDH_ZGen_Entry          *entry;
1204     UINT16                   inSize;
1205     UINT16                   outSize;
1206     UINT16                   offsetOfTypes;
1207     UINT16                   paramOffsets[1];
1208     BYTE                     types[5];
1209 } ECDH_ZGen_COMMAND_DESCRIPTOR_t;
1210
1211 ECDH_ZGen_COMMAND_DESCRIPTOR_t _ECDH_ZGenData = {
1212     /* entry */           &TPM2_ECDH_ZGen,
1213     /* inSize */          (UINT16) (sizeof(ECDH_ZGen_In)),
1214     /* outSize */         (UINT16) (sizeof(ECDH_ZGen_Out)),
1215     /* offsetOfTypes */   offsetof(ECDH_ZGen_COMMAND_DESCRIPTOR_t, types),
1216     /* offsets */         {(UINT16) (offsetof(ECDH_ZGen_In, inPoint))},
1217     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1218                           TPM2B_ECC_POINT_P_UNMARSHAL,
1219                           END_OF_LIST,
1220                           TPM2B_ECC_POINT_P_MARSHAL,
1221                           END_OF_LIST};
1222 };
1223
1224 #define _ECDH_ZGenDataAddress (&_ECDH_ZGenData)
1225 #else
1226 #define _ECDH_ZGenDataAddress 0
1227 #endif // CC_ECDH_ZGen
1228
1229 #if CC_ECC_Parameters
1230 #include "ECC_Parameters_fp.h"
1231
1232 typedef TPM_RC (ECC_Parameters_Entry) (
1233     ECC_Parameters_In*      in,
1234     ECC_Parameters_Out*     out
1235 );
1236
1237
1238 typedef const struct
1239 {
1240     ECC_Parameters_Entry      *entry;
1241     UINT16                   inSize;
1242     UINT16                   outSize;
1243     UINT16                   offsetOfTypes;
1244     BYTE                     types[4];
1245 } ECC_Parameters_COMMAND_DESCRIPTOR_t;
1246
1247 ECC_Parameters_COMMAND_DESCRIPTOR_t _ECC_ParametersData = {
1248     /* entry */           &TPM2_ECC_Parameters,
1249     /* inSize */          (UINT16) (sizeof(ECC_Parameters_In)),
1250     /* outSize */         (UINT16) (sizeof(ECC_Parameters_Out)),
1251     /* offsetOfTypes */   offsetof(ECC_Parameters_COMMAND_DESCRIPTOR_t, types),
1252     /* offsets */         // No parameter offsets
1253     /* types */           {TPMI_ECC_CURVE_P_UNMARSHAL,
1254                           END_OF_LIST,
1255                           TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL,
1256                           END_OF_LIST};
1257 };
1258
1259 #define _ECC_ParametersDataAddress (&_ECC_ParametersData)
1260 #else

```

```

1261 #define _ECC_ParametersDataAddress 0
1262 #endif // CC_ECC_Parameters
1263
1264 #if CC_ZGen_2Phase
1265 #include "ZGen_2Phase_fp.h"
1266
1267 typedef TPM_RC (ZGen_2Phase_Entry) (
1268     ZGen_2Phase_In* in,
1269     ZGen_2Phase_Out* out
1270 );
1271
1272
1273 typedef const struct
1274 {
1275     ZGen_2Phase_Entry *entry;
1276     UINT16 inSize;
1277     UINT16 outSize;
1278     UINT16 offsetOfTypes;
1279     UINT16 paramOffsets[5];
1280     BYTE types[9];
1281 } ZGen_2Phase_COMMAND_DESCRIPTOR_t;
1282
1283 ZGen_2Phase_COMMAND_DESCRIPTOR_t _ZGen_2PhaseData = {
1284     /* entry */ &TPM2_ZGen_2Phase,
1285     /* inSize */ (UINT16) (sizeof(ZGen_2Phase_In)),
1286     /* outSize */ (UINT16) (sizeof(ZGen_2Phase_Out)),
1287     /* offsetOfTypes */ offsetof(ZGen_2Phase_COMMAND_DESCRIPTOR_t, types),
1288     /* offsets */ { (UINT16) (offsetof(ZGen_2Phase_In, inQsB)),
1289                   (UINT16) (offsetof(ZGen_2Phase_In, inQeB)),
1290                   (UINT16) (offsetof(ZGen_2Phase_In, inScheme)),
1291                   (UINT16) (offsetof(ZGen_2Phase_In, counter)),
1292                   (UINT16) (offsetof(ZGen_2Phase_Out, outZ2)) },
1293     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
1294                 TPM2B_ECC_POINT_P_UNMARSHAL,
1295                 TPM2B_ECC_POINT_P_UNMARSHAL,
1296                 TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL,
1297                 UINT16_P_UNMARSHAL,
1298                 END_OF_LIST,
1299                 TPM2B_ECC_POINT_P_MARSHAL,
1300                 TPM2B_ECC_POINT_P_MARSHAL,
1301                 END_OF_LIST };
1302 };
1303
1304 #define _ZGen_2PhaseDataAddress (&_ZGen_2PhaseData)
1305 #else
1306 #define _ZGen_2PhaseDataAddress 0
1307 #endif // CC_ZGen_2Phase
1308
1309 #if CC_ECC_Encrypt
1310 #include "ECC_Encrypt_fp.h"
1311
1312 typedef TPM_RC (ECC_Encrypt_Entry) (
1313     ECC_Encrypt_In* in,
1314     ECC_Encrypt_Out* out
1315 );
1316
1317
1318 typedef const struct
1319 {
1320     ECC_Encrypt_Entry *entry;
1321     UINT16 inSize;
1322     UINT16 outSize;
1323     UINT16 offsetOfTypes;
1324     UINT16 paramOffsets[4];
1325     BYTE types[8];
1326 } ECC_Encrypt_COMMAND_DESCRIPTOR_t;

```

```

1327
1328 ECC_Encrypt_COMMAND_DESCRIPTOR_t _ECC_EncryptData = {
1329     /* entry */ &TPM2_ECC_Encrypt,
1330     /* inSize */ (UINT16) (sizeof(ECC_Encrypt_In)),
1331     /* outSize */ (UINT16) (sizeof(ECC_Encrypt_Out)),
1332     /* offsetOfTypes */ offsetof(ECC_Encrypt_COMMAND_DESCRIPTOR_t, types),
1333     /* offsets */ { (UINT16) (offsetof(ECC_Encrypt_In, plainText)),
1334                    (UINT16) (offsetof(ECC_Encrypt_In, inScheme)),
1335                    (UINT16) (offsetof(ECC_Encrypt_Out, C2)),
1336                    (UINT16) (offsetof(ECC_Encrypt_Out, C3)) },
1337     /* types */ { TPM1_DH_OBJECT_H_UNMARSHAL,
1338                  TPM2B_MAX_BUFFER_P_UNMARSHAL,
1339                  TPMT_KDF_SCHEME_P_UNMARSHAL + ADD_FLAG,
1340                  END_OF_LIST,
1341                  TPM2B_ECC_POINT_P_MARSHAL,
1342                  TPM2B_MAX_BUFFER_P_MARSHAL,
1343                  TPM2B_DIGEST_P_MARSHAL,
1344                  END_OF_LIST };
1345 };
1346
1347 #define _ECC_EncryptDataAddress (&_ECC_EncryptData)
1348 #else
1349 #define _ECC_EncryptDataAddress 0
1350 #endif // CC_ECC_Encrypt
1351
1352 #if CC_ECC_Decrypt
1353 #include "ECC_Decrypt_fp.h"
1354
1355 typedef TPM_RC (ECC_Decrypt_Entry) (
1356     ECC_Decrypt_In* in,
1357     ECC_Decrypt_Out* out
1358 );
1359
1360 typedef const struct
1361 {
1362     ECC_Decrypt_Entry *entry;
1363     UINT16 inSize;
1364     UINT16 outSize;
1365     UINT16 offsetOfTypes;
1366     UINT16 paramOffsets[4];
1367     BYTE types[8];
1368 } ECC_Decrypt_COMMAND_DESCRIPTOR_t;
1369
1370
1371 ECC_Decrypt_COMMAND_DESCRIPTOR_t _ECC_DecryptData = {
1372     /* entry */ &TPM2_ECC_Decrypt,
1373     /* inSize */ (UINT16) (sizeof(ECC_Decrypt_In)),
1374     /* outSize */ (UINT16) (sizeof(ECC_Decrypt_Out)),
1375     /* offsetOfTypes */ offsetof(ECC_Decrypt_COMMAND_DESCRIPTOR_t, types),
1376     /* offsets */ { (UINT16) (offsetof(ECC_Decrypt_In, C1)),
1377                    (UINT16) (offsetof(ECC_Decrypt_In, C2)),
1378                    (UINT16) (offsetof(ECC_Decrypt_In, C3)),
1379                    (UINT16) (offsetof(ECC_Decrypt_In, inScheme)) },
1380     /* types */ { TPM1_DH_OBJECT_H_UNMARSHAL,
1381                  TPM2B_ECC_POINT_P_UNMARSHAL,
1382                  TPM2B_MAX_BUFFER_P_UNMARSHAL,
1383                  TPM2B_DIGEST_P_UNMARSHAL,
1384                  TPMT_KDF_SCHEME_P_UNMARSHAL + ADD_FLAG,
1385                  END_OF_LIST,
1386                  TPM2B_MAX_BUFFER_P_MARSHAL,
1387                  END_OF_LIST };
1388 };
1389
1390 #define _ECC_DecryptDataAddress (&_ECC_DecryptData)
1391 #else
1392 #define _ECC_DecryptDataAddress 0

```

```

1393 #endif // CC_ECC_Decrypt
1394
1395 #if CC_EncryptDecrypt
1396 #include "EncryptDecrypt_fp.h"
1397
1398 typedef TPM_RC (EncryptDecrypt_Entry) (
1399     EncryptDecrypt_In* in,
1400     EncryptDecrypt_Out* out
1401 );
1402
1403
1404 typedef const struct
1405 {
1406     EncryptDecrypt_Entry *entry;
1407     UINT16 inSize;
1408     UINT16 outSize;
1409     UINT16 offsetOfTypes;
1410     UINT16 paramOffsets[5];
1411     BYTE types[9];
1412 } EncryptDecrypt_COMMAND_DESCRIPTOR_t;
1413
1414 EncryptDecrypt_COMMAND_DESCRIPTOR_t _EncryptDecryptData = {
1415     /* entry */ &TPM2_EncryptDecrypt,
1416     /* inSize */ (UINT16) (sizeof(EncryptDecrypt_In)),
1417     /* outSize */ (UINT16) (sizeof(EncryptDecrypt_Out)),
1418     /* offsetOfTypes */ offsetof(EncryptDecrypt_COMMAND_DESCRIPTOR_t, types),
1419     /* offsets */ { (UINT16) (offsetof(EncryptDecrypt_In, decrypt)),
1420                   (UINT16) (offsetof(EncryptDecrypt_In, mode)),
1421                   (UINT16) (offsetof(EncryptDecrypt_In, ivIn)),
1422                   (UINT16) (offsetof(EncryptDecrypt_In, inData)),
1423                   (UINT16) (offsetof(EncryptDecrypt_Out, ivOut)) },
1424     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
1425                 TPMI_YES_NO_P_UNMARSHAL,
1426                 TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1427                 TPM2B_IV_P_UNMARSHAL,
1428                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1429                 END_OF_LIST,
1430                 TPM2B_MAX_BUFFER_P_MARSHAL,
1431                 TPM2B_IV_P_MARSHAL,
1432                 END_OF_LIST }
1433 };
1434
1435 #define _EncryptDecryptDataAddress (&_EncryptDecryptData)
1436 #else
1437 #define _EncryptDecryptDataAddress 0
1438 #endif // CC_EncryptDecrypt
1439
1440 #if CC_EncryptDecrypt2
1441 #include "EncryptDecrypt2_fp.h"
1442
1443 typedef TPM_RC (EncryptDecrypt2_Entry) (
1444     EncryptDecrypt2_In* in,
1445     EncryptDecrypt2_Out* out
1446 );
1447
1448
1449 typedef const struct
1450 {
1451     EncryptDecrypt2_Entry *entry;
1452     UINT16 inSize;
1453     UINT16 outSize;
1454     UINT16 offsetOfTypes;
1455     UINT16 paramOffsets[5];
1456     BYTE types[9];
1457 } EncryptDecrypt2_COMMAND_DESCRIPTOR_t;
1458

```

```

1459 EncryptDecrypt2_COMMAND_DESCRIPTOR t_EncryptDecrypt2Data = {
1460     /* entry */ &TPM2_EncryptDecrypt2,
1461     /* inSize */ (UINT16) (sizeof(EncryptDecrypt2_In)),
1462     /* outSize */ (UINT16) (sizeof(EncryptDecrypt2_Out)),
1463     /* offsetOfTypes */ offsetof(EncryptDecrypt2_COMMAND_DESCRIPTOR t, types),
1464     /* offsets */ { (UINT16) (offsetof(EncryptDecrypt2_In, inData)),
1465                    (UINT16) (offsetof(EncryptDecrypt2_In, decrypt)),
1466                    (UINT16) (offsetof(EncryptDecrypt2_In, mode)),
1467                    (UINT16) (offsetof(EncryptDecrypt2_In, ivIn)),
1468                    (UINT16) (offsetof(EncryptDecrypt2_Out, ivOut)) },
1469     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
1470                  TPM2B_MAX_BUFFER_P_UNMARSHAL,
1471                  TPMI_YES_NO_P_UNMARSHAL,
1472                  TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1473                  TPM2B_IV_P_UNMARSHAL,
1474                  END_OF_LIST,
1475                  TPM2B_MAX_BUFFER_P_MARSHAL,
1476                  TPM2B_IV_P_MARSHAL,
1477                  END_OF_LIST };
1478 };
1479
1480 #define _EncryptDecrypt2DataAddress (&_EncryptDecrypt2Data)
1481 #else
1482 #define _EncryptDecrypt2DataAddress 0
1483 #endif // CC_EncryptDecrypt2
1484
1485 #if CC_Hash
1486 #include "Hash_fp.h"
1487
1488 typedef TPM_RC (Hash_Entry) (
1489     Hash_In* in,
1490     Hash_Out* out
1491 );
1492
1493 typedef const struct
1494 {
1495     Hash_Entry *entry;
1496     UINT16 inSize;
1497     UINT16 outSize;
1498     UINT16 offsetOfTypes;
1499     UINT16 paramOffsets[3];
1500     BYTE types[7];
1501 } Hash_COMMAND_DESCRIPTOR_t;
1502
1503 Hash_COMMAND_DESCRIPTOR_t _HashData = {
1504     /* entry */ &TPM2_Hash,
1505     /* inSize */ (UINT16) (sizeof(Hash_In)),
1506     /* outSize */ (UINT16) (sizeof(Hash_Out)),
1507     /* offsetOfTypes */ offsetof(Hash_COMMAND_DESCRIPTOR_t, types),
1508     /* offsets */ { (UINT16) (offsetof(Hash_In, hashAlg)),
1509                    (UINT16) (offsetof(Hash_In, hierarchy)),
1510                    (UINT16) (offsetof(Hash_Out, validation)) },
1511     /* types */ { TPM2B_MAX_BUFFER_P_UNMARSHAL,
1512                  TPMI_ALG_HASH_P_UNMARSHAL,
1513                  TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1514                  END_OF_LIST,
1515                  TPM2B_DIGEST_P_MARSHAL,
1516                  TPMT_TK_HASHCHECK_P_MARSHAL,
1517                  END_OF_LIST };
1518 };
1519
1520 #define _HashDataAddress (&_HashData)
1521 #else
1522 #define _HashDataAddress 0
1523 #endif // CC_Hash
1524

```



```

1525
1526 #if CC_HMAC
1527 #include "HMAC_fp.h"
1528
1529 typedef TPM_RC (HMAC_Entry) (
1530     HMAC_In*           in,
1531     HMAC_Out*          out
1532 );
1533
1534
1535 typedef const struct
1536 {
1537     HMAC_Entry          *entry;
1538     UINT16              inSize;
1539     UINT16              outSize;
1540     UINT16              offsetOfTypes;
1541     UINT16              paramOffsets[2];
1542     BYTE                types[6];
1543 } HMAC_COMMAND_DESCRIPTOR_t;
1544
1545 HMAC_COMMAND_DESCRIPTOR_t _HMACData = {
1546     /* entry */          &TPM2_HMAC,
1547     /* inSize */         (UINT16) (sizeof(HMAC_In)),
1548     /* outSize */        (UINT16) (sizeof(HMAC_Out)),
1549     /* offsetOfTypes */  offsetof(HMAC_COMMAND_DESCRIPTOR_t, types),
1550     /* offsets */        { (UINT16) (offsetof(HMAC_In, buffer)),
1551                          (UINT16) (offsetof(HMAC_In, hashAlg)) },
1552     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1553                          TPM2B_MAX_BUFFER_P_UNMARSHAL,
1554                          TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1555                          END_OF_LIST,
1556                          TPM2B_DIGEST_P_MARSHAL,
1557                          END_OF_LIST};
1558 };
1559
1560 #define _HMACDataAddress (&_HMACData)
1561 #else
1562 #define _HMACDataAddress 0
1563 #endif // CC_HMAC
1564
1565 #if CC_MAC
1566 #include "MAC_fp.h"
1567
1568 typedef TPM_RC (MAC_Entry) (
1569     MAC_In*             in,
1570     MAC_Out*            out
1571 );
1572
1573
1574 typedef const struct
1575 {
1576     MAC_Entry           *entry;
1577     UINT16              inSize;
1578     UINT16              outSize;
1579     UINT16              offsetOfTypes;
1580     UINT16              paramOffsets[2];
1581     BYTE                types[6];
1582 } MAC_COMMAND_DESCRIPTOR_t;
1583
1584 MAC_COMMAND_DESCRIPTOR_t _MACData = {
1585     /* entry */          &TPM2_MAC,
1586     /* inSize */         (UINT16) (sizeof(MAC_In)),
1587     /* outSize */        (UINT16) (sizeof(MAC_Out)),
1588     /* offsetOfTypes */  offsetof(MAC_COMMAND_DESCRIPTOR_t, types),
1589     /* offsets */        { (UINT16) (offsetof(MAC_In, buffer)),
1590                          (UINT16) (offsetof(MAC_In, inScheme)) },

```



```

1591     /* types */
1592     TPM2B_MAX_BUFFER_P_UNMARSHAL,
1593     TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1594     END_OF_LIST,
1595     TPM2B_DIGEST_P_MARSHAL,
1596     END_OF_LIST}
1597 };
1598
1599 #define _MACDataAddress (&_MACData)
1600 #else
1601 #define _MACDataAddress 0
1602 #endif // CC_MAC
1603
1604 #if CC_GetRandom
1605 #include "GetRandom_fp.h"
1606
1607 typedef TPM_RC (GetRandom_Entry) (
1608     GetRandom_In* in,
1609     GetRandom_Out* out
1610 );
1611
1612
1613 typedef const struct
1614 {
1615     GetRandom_Entry *entry;
1616     UINT16 inSize;
1617     UINT16 outSize;
1618     UINT16 offsetOfTypes;
1619     BYTE types[4];
1620 } GetRandom_COMMAND_DESCRIPTOR_t;
1621
1622 GetRandom_COMMAND_DESCRIPTOR_t _GetRandomData = {
1623     /* entry */ &TPM2_GetRandom,
1624     /* inSize */ (UINT16) (sizeof (GetRandom_In)),
1625     /* outSize */ (UINT16) (sizeof (GetRandom_Out)),
1626     /* offsetOfTypes */ offsetOf (GetRandom_COMMAND_DESCRIPTOR_t, types),
1627     /* offsets */ // No parameter offsets
1628     /* types */ {UINT16_P_UNMARSHAL,
1629                 END_OF_LIST,
1630                 TPM2B_DIGEST_P_MARSHAL,
1631                 END_OF_LIST}
1632 };
1633
1634 #define _GetRandomDataAddress (&_GetRandomData)
1635 #else
1636 #define _GetRandomDataAddress 0
1637 #endif // CC_GetRandom
1638
1639 #if CC_StirRandom
1640 #include "StirRandom_fp.h"
1641
1642 typedef TPM_RC (StirRandom_Entry) (
1643     StirRandom_In* in
1644 );
1645
1646
1647 typedef const struct
1648 {
1649     StirRandom_Entry *entry;
1650     UINT16 inSize;
1651     UINT16 outSize;
1652     UINT16 offsetOfTypes;
1653     BYTE types[3];
1654 } StirRandom_COMMAND_DESCRIPTOR_t;
1655
1656 StirRandom_COMMAND_DESCRIPTOR_t _StirRandomData = {

```

```

1657     /* entry          */      &TPM2_StirRandom,
1658     /* inSize         */      (UINT16) (sizeof(StirRandom_In)),
1659     /* outSize        */      0,
1660     /* offsetOfTypes  */      offsetof(StirRandom_COMMAND_DESCRIPTOR_t, types),
1661     /* offsets        */      // No parameter offsets
1662     /* types          */      {TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1663                               END_OF_LIST,
1664                               END_OF_LIST};
1665 };
1666
1667 #define _StirRandomDataAddress (&_StirRandomData)
1668 #else
1669 #define _StirRandomDataAddress 0
1670 #endif // CC_StirRandom
1671
1672 #if CC_HMAC_Start
1673 #include "HMAC_Start_fp.h"
1674
1675 typedef TPM_RC (HMAC_Start_Entry) (
1676     HMAC_Start_In*      in,
1677     HMAC_Start_Out*     out
1678 );
1679
1680
1681 typedef const struct
1682 {
1683     HMAC_Start_Entry      *entry;
1684     UINT16                inSize;
1685     UINT16                outSize;
1686     UINT16                offsetOfTypes;
1687     UINT16                paramOffsets[2];
1688     BYTE                  types[6];
1689 } HMAC_Start_COMMAND_DESCRIPTOR_t;
1690
1691 HMAC_Start_COMMAND_DESCRIPTOR_t HMAC_StartData = {
1692     /* entry          */      &TPM2_HMAC_Start,
1693     /* inSize         */      (UINT16) (sizeof(HMAC_Start_In)),
1694     /* outSize        */      (UINT16) (sizeof(HMAC_Start_Out)),
1695     /* offsetOfTypes  */      offsetof(HMAC_Start_COMMAND_DESCRIPTOR_t, types),
1696     /* offsets        */      {(UINT16) (offsetof(HMAC_Start_In, auth)),
1697                               (UINT16) (offsetof(HMAC_Start_In, hashAlg))},
1698     /* types          */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1699                               TPM2B_AUTH_P_UNMARSHAL,
1700                               TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1701                               END_OF_LIST,
1702                               TPMI_DH_OBJECT_H_MARSHAL,
1703                               END_OF_LIST};
1704 };
1705
1706 #define _HMAC_StartDataAddress (&_HMAC_StartData)
1707 #else
1708 #define _HMAC_StartDataAddress 0
1709 #endif // CC_HMAC_Start
1710
1711 #if CC_MAC_Start
1712 #include "MAC_Start_fp.h"
1713
1714 typedef TPM_RC (MAC_Start_Entry) (
1715     MAC_Start_In*      in,
1716     MAC_Start_Out*     out
1717 );
1718
1719
1720 typedef const struct
1721 {
1722     MAC_Start_Entry      *entry;

```

```

1723     UINT16      inSize;
1724     UINT16      outSize;
1725     UINT16      offsetOfTypes;
1726     UINT16      paramOffsets[2];
1727     BYTE        types[6];
1728 } MAC_Start_COMMAND_DESCRIPTOR_t;
1729
1730 MAC_Start_COMMAND_DESCRIPTOR_t _MAC_StartData = {
1731     /* entry */      &TPM2_MAC_Start,
1732     /* inSize */     (UINT16) (sizeof(MAC_Start_In)),
1733     /* outSize */    (UINT16) (sizeof(MAC_Start_Out)),
1734     /* offsetOfTypes */ offsetof(MAC_Start_COMMAND_DESCRIPTOR_t, types),
1735     /* offsets */    {(UINT16) (offsetof(MAC_Start_In, auth)),
1736                     (UINT16) (offsetof(MAC_Start_In, inScheme))},
1737     /* types */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1738                     TPM2B_AUTH_P_UNMARSHAL,
1739                     TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1740                     END_OF_LIST,
1741                     TPMI_DH_OBJECT_H_MARSHAL,
1742                     END_OF_LIST};
1743 };
1744
1745 #define _MAC_StartDataAddress (&_MAC_StartData)
1746 #else
1747 #define _MAC_StartDataAddress 0
1748 #endif // CC_MAC_Start
1749
1750 #if CC_HashSequenceStart
1751 #include "HashSequenceStart_fp.h"
1752
1753 typedef TPM_RC (HashSequenceStart_Entry) (
1754     HashSequenceStart_In* in,
1755     HashSequenceStart_Out* out
1756 );
1757
1758 typedef const struct
1759 {
1760     HashSequenceStart_Entry *entry;
1761     UINT16 inSize;
1762     UINT16 outSize;
1763     UINT16 offsetOfTypes;
1764     UINT16 paramOffsets[1];
1765     BYTE types[5];
1766 } HashSequenceStart_COMMAND_DESCRIPTOR_t;
1767
1768 HashSequenceStart_COMMAND_DESCRIPTOR_t _HashSequenceStartData = {
1769     /* entry */      &TPM2_HashSequenceStart,
1770     /* inSize */     (UINT16) (sizeof(HashSequenceStart_In)),
1771     /* outSize */    (UINT16) (sizeof(HashSequenceStart_Out)),
1772     /* offsetOfTypes */ offsetof(HashSequenceStart_COMMAND_DESCRIPTOR_t,
1773     types),
1774     /* offsets */    {(UINT16) (offsetof(HashSequenceStart_In, hashAlg))},
1775     /* types */      {TPM2B_AUTH_P_UNMARSHAL,
1776                     TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1777                     END_OF_LIST,
1778                     TPMI_DH_OBJECT_H_MARSHAL,
1779                     END_OF_LIST};
1780 };
1781
1782 #define _HashSequenceStartDataAddress (&_HashSequenceStartData)
1783 #else
1784 #define _HashSequenceStartDataAddress 0
1785 #endif // CC_HashSequenceStart
1786
1787 #if CC_SequenceUpdate

```

```

1788 #include "SequenceUpdate_fp.h"
1789
1790 typedef TPM_RC (SequenceUpdate_Entry) (
1791     SequenceUpdate_In* in
1792 );
1793
1794
1795 typedef const struct
1796 {
1797     SequenceUpdate_Entry *entry;
1798     UINT16 inSize;
1799     UINT16 outSize;
1800     UINT16 offsetOfTypes;
1801     UINT16 paramOffsets[1];
1802     BYTE types[4];
1803 } SequenceUpdate_COMMAND_DESCRIPTOR_t;
1804
1805 SequenceUpdate_COMMAND_DESCRIPTOR_t _SequenceUpdateData = {
1806     /* entry */ &TPM2_SequenceUpdate,
1807     /* inSize */ (UINT16) (sizeof(SequenceUpdate_In)),
1808     /* outSize */ 0,
1809     /* offsetOfTypes */ offsetof(SequenceUpdate_COMMAND_DESCRIPTOR_t, types),
1810     /* offsets */ { (UINT16) (offsetof(SequenceUpdate_In, buffer)) },
1811     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
1812                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1813                 END_OF_LIST,
1814                 END_OF_LIST }
1815 };
1816
1817 #define _SequenceUpdateDataAddress (&_SequenceUpdateData)
1818 #else
1819 #define SequenceUpdateDataAddress 0
1820 #endif // CC_SequenceUpdate
1821
1822 #if CC_SequenceComplete
1823 #include "SequenceComplete_fp.h"
1824
1825 typedef TPM_RC (SequenceComplete_Entry) (
1826     SequenceComplete_In* in,
1827     SequenceComplete_Out* out
1828 );
1829
1830
1831 typedef const struct
1832 {
1833     SequenceComplete_Entry *entry;
1834     UINT16 inSize;
1835     UINT16 outSize;
1836     UINT16 offsetOfTypes;
1837     UINT16 paramOffsets[3];
1838     BYTE types[7];
1839 } SequenceComplete_COMMAND_DESCRIPTOR_t;
1840
1841 SequenceComplete_COMMAND_DESCRIPTOR_t _SequenceCompleteData = {
1842     /* entry */ &TPM2_SequenceComplete,
1843     /* inSize */ (UINT16) (sizeof(SequenceComplete_In)),
1844     /* outSize */ (UINT16) (sizeof(SequenceComplete_Out)),
1845     /* offsetOfTypes */ offsetof(SequenceComplete_COMMAND_DESCRIPTOR_t,
1846     types),
1847     /* offsets */ { (UINT16) (offsetof(SequenceComplete_In, buffer)),
1848                   (UINT16) (offsetof(SequenceComplete_In, hierarchy)),
1849                   (UINT16) (offsetof(SequenceComplete_Out,
1850 validation)) },
1851     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
1852                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1853                 TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,

```

```

1852                                     END_OF_LIST,
1853                                     TPM2B_DIGEST_P_MARSHAL,
1854                                     TPMT_TK_HASHCHECK_P_MARSHAL,
1855                                     END_OF_LIST}
1856 };
1857
1858 #define _SequenceCompleteDataAddress (&_SequenceCompleteData)
1859 #else
1860 #define _SequenceCompleteDataAddress 0
1861 #endif // CC_SequenceComplete
1862
1863 #if CC_EventSequenceComplete
1864 #include "EventSequenceComplete_fp.h"
1865
1866 typedef TPM_RC (EventSequenceComplete_Entry) (
1867     EventSequenceComplete_In* in,
1868     EventSequenceComplete_Out* out
1869 );
1870
1871
1872 typedef const struct
1873 {
1874     EventSequenceComplete_Entry *entry;
1875     UINT16 inSize;
1876     UINT16 outSize;
1877     UINT16 offsetOfTypes;
1878     UINT16 paramOffsets[2];
1879     BYTE types[6];
1880 } EventSequenceComplete_COMMAND_DESCRIPTOR_t;
1881
1882 EventSequenceComplete_COMMAND_DESCRIPTOR_t EventSequenceCompleteData = {
1883     /* entry */ &TPM2_EventSequenceComplete,
1884     /* inSize */ (UINT16) (sizeof(EventSequenceComplete_In)),
1885     /* outSize */ (UINT16) (sizeof(EventSequenceComplete_Out)),
1886     /* offsetOfTypes */ offsetof(EventSequenceComplete_COMMAND_DESCRIPTOR_t,
1887     types),
1888     /* offsets */ { (UINT16) (offsetof(EventSequenceComplete_In,
1889     sequenceHandle)),
1890     (UINT16) (offsetof(EventSequenceComplete_In,
1891     buffer)) },
1892     /* types */ {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1893     TPMI_DH_OBJECT_H_UNMARSHAL,
1894     TPM2B_MAX_BUFFER_P_UNMARSHAL,
1895     END_OF_LIST,
1896     TPML_DIGEST_VALUES_P_MARSHAL,
1897     END_OF_LIST}
1898 };
1899
1900 #define _EventSequenceCompleteDataAddress (&_EventSequenceCompleteData)
1901 #else
1902 #define _EventSequenceCompleteDataAddress 0
1903 #endif // CC_EventSequenceComplete
1904
1905 #if CC_Certify
1906 #include "Certify_fp.h"
1907
1908 typedef TPM_RC (Certify_Entry) (
1909     Certify_In* in,
1910     Certify_Out* out
1911 );
1912
1913 typedef const struct
1914 {
1915     Certify_Entry *entry;
1916     UINT16 inSize;

```

```

1915     UINT16      outSize;
1916     UINT16      offsetOfTypes;
1917     UINT16      paramOffsets[4];
1918     BYTE        types[8];
1919 } Certify_COMMAND_DESCRIPTOR_t;
1920
1921 Certify_COMMAND_DESCRIPTOR_t _CertifyData = {
1922     /* entry      */      &TPM2_Certify,
1923     /* inSize     */      (UINT16) (sizeof(Certify_In)),
1924     /* outSize    */      (UINT16) (sizeof(Certify_Out)),
1925     /* offsetOfTypes */      offsetof(Certify_COMMAND_DESCRIPTOR_t, types),
1926     /* offsets    */      { (UINT16) (offsetof(Certify_In, signHandle)),
1927                           (UINT16) (offsetof(Certify_In, qualifyingData)),
1928                           (UINT16) (offsetof(Certify_In, inScheme)),
1929                           (UINT16) (offsetof(Certify_Out, signature)) },
1930     /* types      */      {TPMI_DH_OBJECT_H_UNMARSHAL,
1931                           TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1932                           TPM2B_DATA_P_UNMARSHAL,
1933                           TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1934                           END_OF_LIST,
1935                           TPM2B_ATTEST_P_UNMARSHAL,
1936                           TPMT_SIGNATURE_P_UNMARSHAL,
1937                           END_OF_LIST}
1938 };
1939
1940 #define _CertifyDataAddress (&_CertifyData)
1941 #else
1942 #define _CertifyDataAddress 0
1943 #endif // CC_Certify
1944
1945 #if CC_CertifyCreation
1946 #include "CertifyCreation_fp.h"
1947
1948 typedef TPM_RC (CertifyCreation_Entry) (
1949     CertifyCreation_In*   in,
1950     CertifyCreation_Out*  out
1951 );
1952
1953
1954 typedef const struct
1955 {
1956     CertifyCreation_Entry  *entry;
1957     UINT16                 inSize;
1958     UINT16                 outSize;
1959     UINT16                 offsetOfTypes;
1960     UINT16                 paramOffsets[6];
1961     BYTE                   types[10];
1962 } CertifyCreation_COMMAND_DESCRIPTOR_t;
1963
1964 CertifyCreation_COMMAND_DESCRIPTOR_t _CertifyCreationData = {
1965     /* entry      */      &TPM2_CertifyCreation,
1966     /* inSize     */      (UINT16) (sizeof(CertifyCreation_In)),
1967     /* outSize    */      (UINT16) (sizeof(CertifyCreation_Out)),
1968     /* offsetOfTypes */      offsetof(CertifyCreation_COMMAND_DESCRIPTOR_t, types),
1969     /* offsets    */      { (UINT16) (offsetof(CertifyCreation_In, objectHandle)),
1970                           (UINT16) (offsetof(CertifyCreation_In,
1971     qualifyingData)),
1972                           (UINT16) (offsetof(CertifyCreation_In, creationHash)),
1973                           (UINT16) (offsetof(CertifyCreation_In, inScheme)),
1974                           (UINT16) (offsetof(CertifyCreation_In,
1975     creationTicket)),
1976                           (UINT16) (offsetof(CertifyCreation_Out, signature)) },
1977     /* types      */      {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1978                           TPMI_DH_OBJECT_H_UNMARSHAL,
1979                           TPM2B_DATA_P_UNMARSHAL,
1980                           TPM2B_DIGEST_P_UNMARSHAL,

```



```

1979         TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1980         TPMT_TK_CREATION_P_UNMARSHAL,
1981         END_OF_LIST,
1982         TPM2B_ATTEST_P_MARSHAL,
1983         TPMT_SIGNATURE_P_MARSHAL,
1984         END_OF_LIST}
1985     };
1986
1987     #define _CertifyCreationDataAddress (&_CertifyCreationData)
1988     #else
1989     #define _CertifyCreationDataAddress 0
1990     #endif // CC_CertifyCreation
1991
1992     #if CC_Quote
1993     #include "Quote_fp.h"
1994
1995     typedef TPM_RC (Quote_Entry) (
1996         Quote_In*      in,
1997         Quote_Out*     out
1998     );
1999
2000
2001     typedef const struct
2002     {
2003         Quote_Entry      *entry;
2004         UINT16           inSize;
2005         UINT16           outSize;
2006         UINT16           offsetOfTypes;
2007         UINT16           paramOffsets[4];
2008         BYTE             types[8];
2009     } Quote_COMMAND_DESCRIPTOR_t;
2010
2011     Quote_COMMAND_DESCRIPTOR_t _QuoteData = {
2012         /* entry */           &TPM2_Quote,
2013         /* inSize */         (UINT16) (sizeof(Quote_In)),
2014         /* outSize */        (UINT16) (sizeof(Quote_Out)),
2015         /* offsetOfTypes */  offsetof(Quote_COMMAND_DESCRIPTOR_t, types),
2016         /* offsets */        { (UINT16) (offsetof(Quote_In, qualifyingData)),
2017                             (UINT16) (offsetof(Quote_In, inScheme)),
2018                             (UINT16) (offsetof(Quote_In, PCRselect)),
2019                             (UINT16) (offsetof(Quote_Out, signature)) },
2020         /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
2021                             TPM2B_DATA_P_UNMARSHAL,
2022                             TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2023                             TPML_PCR_SELECTION_P_UNMARSHAL,
2024                             END_OF_LIST,
2025                             TPM2B_ATTEST_P_MARSHAL,
2026                             TPMT_SIGNATURE_P_MARSHAL,
2027                             END_OF_LIST}
2028     };
2029
2030     #define _QuoteDataAddress (&_QuoteData)
2031     #else
2032     #define _QuoteDataAddress 0
2033     #endif // CC_Quote
2034
2035     #if CC_GetSessionAuditDigest
2036     #include "GetSessionAuditDigest_fp.h"
2037
2038     typedef TPM_RC (GetSessionAuditDigest_Entry) (
2039         GetSessionAuditDigest_In*  in,
2040         GetSessionAuditDigest_Out* out
2041     );
2042
2043
2044     typedef const struct

```

```

2045 {
2046     GetSessionAuditDigest_Entry *entry;
2047     UINT16 inSize;
2048     UINT16 outSize;
2049     UINT16 offsetOfTypes;
2050     UINT16 paramOffsets[5];
2051     BYTE types[9];
2052 } GetSessionAuditDigest_COMMAND_DESCRIPTOR_t;
2053
2054 GetSessionAuditDigest_COMMAND_DESCRIPTOR_t_GetSessionAuditDigestData = {
2055     /* entry */ &TPM2_GetSessionAuditDigest,
2056     /* inSize */ (UINT16) (sizeof(GetSessionAuditDigest_In)),
2057     /* outSize */ (UINT16) (sizeof(GetSessionAuditDigest_Out)),
2058     /* offsetOfTypes */ offsetof(GetSessionAuditDigest_COMMAND_DESCRIPTOR_t,
types),
2059     /* offsets */ { (UINT16) (offsetof(GetSessionAuditDigest_In,
signHandle)),
2060 (UINT16) (offsetof(GetSessionAuditDigest_In,
sessionHandle)),
2061 (UINT16) (offsetof(GetSessionAuditDigest_In,
qualifyingData)),
2062 (UINT16) (offsetof(GetSessionAuditDigest_In,
inScheme)),
2063 (UINT16) (offsetof(GetSessionAuditDigest_Out,
signature))},
2064     /* types */ {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
2065 TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
2066 TPMI_SH_HMAC_H_UNMARSHAL,
2067 TPM2B_DATA_P_UNMARSHAL,
2068 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2069 END_OF_LIST,
2070 TPM2B_ATTEST_P_MARSHAL,
2071 TPMT_SIGNATURE_P_MARSHAL,
2072 END_OF_LIST};
2073 };
2074
2075 #define _GetSessionAuditDigestDataAddress (&_GetSessionAuditDigestData)
2076 #else
2077 #define _GetSessionAuditDigestDataAddress 0
2078 #endif // CC_GetSessionAuditDigest
2079
2080 #if CC_GetCommandAuditDigest
2081 #include "GetCommandAuditDigest_fp.h"
2082
2083 typedef TPM_RC (GetCommandAuditDigest_Entry) (
2084     GetCommandAuditDigest_In* in,
2085     GetCommandAuditDigest_Out* out
2086 );
2087
2088
2089 typedef const struct
2090 {
2091     GetCommandAuditDigest_Entry *entry;
2092     UINT16 inSize;
2093     UINT16 outSize;
2094     UINT16 offsetOfTypes;
2095     UINT16 paramOffsets[4];
2096     BYTE types[8];
2097 } GetCommandAuditDigest_COMMAND_DESCRIPTOR_t;
2098
2099 GetCommandAuditDigest_COMMAND_DESCRIPTOR_t_GetCommandAuditDigestData = {
2100     /* entry */ &TPM2_GetCommandAuditDigest,
2101     /* inSize */ (UINT16) (sizeof(GetCommandAuditDigest_In)),
2102     /* outSize */ (UINT16) (sizeof(GetCommandAuditDigest_Out)),
2103     /* offsetOfTypes */ offsetof(GetCommandAuditDigest_COMMAND_DESCRIPTOR_t,
types),

```

```

2104     /* offsets */
2105     signHandle)),
2106     qualifyingData)),
2107     inScheme)),
2108     signature))),
2109     /* types */
2110     { (UINT16) (offsetof(GetCommandAuditDigest_In,
2111     TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
2112     TPM2B_DATA_P_UNMARSHAL,
2113     TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2114     END_OF_LIST,
2115     TPM2B_ATTEST_P_MARSHAL,
2116     TPMT_SIGNATURE_P_MARSHAL,
2117     END_OF_LIST)
2118     };
2119
2120 #define _GetCommandAuditDigestDataAddress (&_GetCommandAuditDigestData)
2121 #else
2122 #define _GetCommandAuditDigestDataAddress 0
2123 #endif // CC_GetCommandAuditDigest
2124
2125 #if CC_GetTime
2126 #include "GetTime_fp.h"
2127
2128 typedef TPM_RC (GetTime_Entry) (
2129     GetTime_In* in,
2130     GetTime_Out* out
2131 );
2132
2133 typedef const struct
2134 {
2135     GetTime_Entry *entry;
2136     UINT16 inSize;
2137     UINT16 outSize;
2138     UINT16 offsetOfTypes;
2139     UINT16 paramOffsets[4];
2140     BYTE types[8];
2141 } GetTime_COMMAND_DESCRIPTOR_t;
2142
2143 GetTime_COMMAND_DESCRIPTOR_t _GetTimeData = {
2144     /* entry */ &TPM2_GetTime,
2145     /* inSize */ (UINT16) (sizeof(GetTime_In)),
2146     /* outSize */ (UINT16) (sizeof(GetTime_Out)),
2147     /* offsetOfTypes */ offsetof(GetTime_COMMAND_DESCRIPTOR_t, types),
2148     /* offsets */ { (UINT16) (offsetof(GetTime_In, signHandle)),
2149     (UINT16) (offsetof(GetTime_In, qualifyingData)),
2150     (UINT16) (offsetof(GetTime_In, inScheme)),
2151     (UINT16) (offsetof(GetTime_Out, signature)) },
2152     /* types */ { TPMI_DH_ENDORSEMENT_H_UNMARSHAL,
2153     TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
2154     TPM2B_DATA_P_UNMARSHAL,
2155     TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2156     END_OF_LIST,
2157     TPM2B_ATTEST_P_MARSHAL,
2158     TPMT_SIGNATURE_P_MARSHAL,
2159     END_OF_LIST }
2160     };
2161
2162 #define _GetTimeDataAddress (&_GetTimeData)
2163 #else
2164 #define _GetTimeDataAddress 0
2165 #endif // CC_GetTime

```

```

2166 #if CC_CertifyX509
2167 #include "CertifyX509_fp.h"
2168
2169 typedef TPM_RC (CertifyX509_Entry) (
2170     CertifyX509_In*      in,
2171     CertifyX509_Out*     out
2172 );
2173
2174
2175 typedef const struct
2176 {
2177     CertifyX509_Entry      *entry;
2178     UINT16                 inSize;
2179     UINT16                 outSize;
2180     UINT16                 offsetOfTypes;
2181     UINT16                 paramOffsets[6];
2182     BYTE                   types[10];
2183 } CertifyX509_COMMAND_DESCRIPTOR_t;
2184
2185 CertifyX509_COMMAND_DESCRIPTOR_t _CertifyX509Data = {
2186     /* entry */ &TPM2_CertifyX509,
2187     /* inSize */ (UINT16) (sizeof(CertifyX509_In)),
2188     /* outSize */ (UINT16) (sizeof(CertifyX509_Out)),
2189     /* offsetOfTypes */ offsetof(CertifyX509_COMMAND_DESCRIPTOR_t, types),
2190     /* offsets */ { (UINT16) (offsetof(CertifyX509_In, signHandle)),
2191                    (UINT16) (offsetof(CertifyX509_In, reserved)),
2192                    (UINT16) (offsetof(CertifyX509_In, inScheme)),
2193                    (UINT16) (offsetof(CertifyX509_In,
2194
2195                    (UINT16) (offsetof(CertifyX509_Out, tbsDigest)),
2196                    (UINT16) (offsetof(CertifyX509_Out, signature)))},
2197     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
2198                 TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
2199                 TPM2B_DATA_P_UNMARSHAL,
2200                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2201                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
2202                 END_OF_LIST,
2203                 TPM2B_MAX_BUFFER_P_MARSHAL,
2204                 TPM2B_DIGEST_P_MARSHAL,
2205                 TPMT_SIGNATURE_P_MARSHAL,
2206                 END_OF_LIST}
2207 };
2208 #define _CertifyX509DataAddress (&_CertifyX509Data)
2209 #else
2210 #define _CertifyX509DataAddress 0
2211 #endif // CC_CertifyX509
2212
2213 #if CC_Commit
2214 #include "Commit_fp.h"
2215
2216 typedef TPM_RC (Commit_Entry) (
2217     Commit_In*      in,
2218     Commit_Out*     out
2219 );
2220
2221
2222 typedef const struct
2223 {
2224     Commit_Entry      *entry;
2225     UINT16             inSize;
2226     UINT16             outSize;
2227     UINT16             offsetOfTypes;
2228     UINT16             paramOffsets[6];
2229     BYTE               types[10];
2230 } Commit_COMMAND_DESCRIPTOR_t;

```

```

2231
2232 Commit_COMMAND_DESCRIPTOR_t _CommitData = {
2233     /* entry */           &TPM2_Commit,
2234     /* inSize */          (UINT16) (sizeof(Commit_In)),
2235     /* outSize */         (UINT16) (sizeof(Commit_Out)),
2236     /* offsetOfTypes */   offsetof(Commit_COMMAND_DESCRIPTOR_t, types),
2237     /* offsets */          { (UINT16) (offsetof(Commit_In, P1)),
2238                             (UINT16) (offsetof(Commit_In, s2)),
2239                             (UINT16) (offsetof(Commit_In, y2)),
2240                             (UINT16) (offsetof(Commit_Out, L)),
2241                             (UINT16) (offsetof(Commit_Out, E)),
2242                             (UINT16) (offsetof(Commit_Out, counter))},
2243     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
2244                             TPM2B_ECC_POINT_P_UNMARSHAL,
2245                             TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
2246                             TPM2B_ECC_PARAMETER_P_UNMARSHAL,
2247                             END_OF_LIST,
2248                             TPM2B_ECC_POINT_P_MARSHAL,
2249                             TPM2B_ECC_POINT_P_MARSHAL,
2250                             TPM2B_ECC_POINT_P_MARSHAL,
2251                             UINT16_P_MARSHAL,
2252                             END_OF_LIST}
2253 };
2254
2255 #define _CommitDataAddress (&_CommitData)
2256 #else
2257 #define _CommitDataAddress 0
2258 #endif // CC_Commit
2259
2260 #if CC_EC_Ephemeral
2261 #include "EC_Ephemeral_fp.h"
2262
2263 typedef TPM_RC (EC_Ephemeral_Entry) (
2264     EC_Ephemeral_In* in,
2265     EC_Ephemeral_Out* out
2266 );
2267
2268 typedef const struct
2269 {
2270     EC_Ephemeral_Entry *entry;
2271     UINT16 inSize;
2272     UINT16 outSize;
2273     UINT16 offsetOfTypes;
2274     UINT16 paramOffsets[1];
2275     BYTE types[5];
2276 } EC_Ephemeral_COMMAND_DESCRIPTOR_t;
2277
2278 EC_Ephemeral_COMMAND_DESCRIPTOR_t _EC_EphemeralData = {
2279     /* entry */           &TPM2_EC_Ephemeral,
2280     /* inSize */          (UINT16) (sizeof(EC_Ephemeral_In)),
2281     /* outSize */         (UINT16) (sizeof(EC_Ephemeral_Out)),
2282     /* offsetOfTypes */   offsetof(EC_Ephemeral_COMMAND_DESCRIPTOR_t, types),
2283     /* offsets */          { (UINT16) (offsetof(EC_Ephemeral_Out, counter))},
2284     /* types */           {TPMI_ECC_CURVE_P_UNMARSHAL,
2285                             END_OF_LIST,
2286                             TPM2B_ECC_POINT_P_MARSHAL,
2287                             UINT16_P_MARSHAL,
2288                             END_OF_LIST}
2289 };
2290
2291 #define _EC_EphemeralDataAddress (&_EC_EphemeralData)
2292 #else
2293 #define _EC_EphemeralDataAddress 0
2294 #endif // CC_EC_Ephemeral
2295
2296

```

```

2297 #if CC_VerifySignature
2298 #include "VerifySignature_fp.h"
2299
2300 typedef TPM_RC (VerifySignature_Entry) (
2301     VerifySignature_In* in,
2302     VerifySignature_Out* out
2303 );
2304
2305
2306 typedef const struct
2307 {
2308     VerifySignature_Entry *entry;
2309     UINT16 inSize;
2310     UINT16 outSize;
2311     UINT16 offsetOfTypes;
2312     UINT16 paramOffsets[2];
2313     BYTE types[6];
2314 } VerifySignature_COMMAND_DESCRIPTOR_t;
2315
2316 VerifySignature_COMMAND_DESCRIPTOR_t _VerifySignatureData = {
2317     /* entry */ &TPM2_VerifySignature,
2318     /* inSize */ (UINT16) (sizeof(VerifySignature_In)),
2319     /* outSize */ (UINT16) (sizeof(VerifySignature_Out)),
2320     /* offsetOfTypes */ offsetof(VerifySignature_COMMAND_DESCRIPTOR_t, types),
2321     /* offsets */ { (UINT16) (offsetof(VerifySignature_In, digest)),
2322                   (UINT16) (offsetof(VerifySignature_In, signature)) },
2323     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
2324                 TPM2B_DIGEST_P_UNMARSHAL,
2325                 TPMT_SIGNATURE_P_UNMARSHAL,
2326                 END_OF_LIST,
2327                 TPMT_TK_VERIFIED_P_MARSHAL,
2328                 END_OF_LIST }
2329 };
2330
2331 #define _VerifySignatureDataAddress (&_VerifySignatureData)
2332 #else
2333 #define _VerifySignatureDataAddress 0
2334 #endif // CC_VerifySignature
2335
2336 #if CC_Sign
2337 #include "Sign_fp.h"
2338
2339 typedef TPM_RC (Sign_Entry) (
2340     Sign_In* in,
2341     Sign_Out* out
2342 );
2343
2344
2345 typedef const struct
2346 {
2347     Sign_Entry *entry;
2348     UINT16 inSize;
2349     UINT16 outSize;
2350     UINT16 offsetOfTypes;
2351     UINT16 paramOffsets[3];
2352     BYTE types[7];
2353 } Sign_COMMAND_DESCRIPTOR_t;
2354
2355 Sign_COMMAND_DESCRIPTOR_t _SignData = {
2356     /* entry */ &TPM2_Sign,
2357     /* inSize */ (UINT16) (sizeof(Sign_In)),
2358     /* outSize */ (UINT16) (sizeof(Sign_Out)),
2359     /* offsetOfTypes */ offsetof(Sign_COMMAND_DESCRIPTOR_t, types),
2360     /* offsets */ { (UINT16) (offsetof(Sign_In, digest)),
2361                   (UINT16) (offsetof(Sign_In, inScheme)),
2362                   (UINT16) (offsetof(Sign_In, validation)) },

```



```

2363     /* types */
2364
2365     {TPMI_DH_OBJECT_H_UNMARSHAL,
2366     TPM2B_DIGEST_P_UNMARSHAL,
2367     TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2368     TPMT_TK_HASHCHECK_P_UNMARSHAL,
2369     END_OF_LIST,
2370     TPMT_SIGNATURE_P_MARSHAL,
2371     END_OF_LIST}
2372 };
2373
2374 #define _SignDataAddress (&_SignData)
2375 #else
2376 #define _SignDataAddress 0
2377 #endif // CC_Sign
2378
2379 #if CC_SetCommandCodeAuditStatus
2380 #include "SetCommandCodeAuditStatus_fp.h"
2381
2382 typedef TPM_RC (SetCommandCodeAuditStatus_Entry) (
2383     SetCommandCodeAuditStatus_In* in
2384 );
2385
2386 typedef const struct
2387 {
2388     SetCommandCodeAuditStatus_Entry *entry;
2389     UINT16 inSize;
2390     UINT16 outSize;
2391     UINT16 offsetOfTypes;
2392     paramOffsets[3];
2393     BYTE types[6];
2394 } SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t;
2395
2396 SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t _SetCommandCodeAuditStatusData = {
2397     /* entry */ &TPM2_SetCommandCodeAuditStatus,
2398     /* inSize */ (UINT16) (sizeof (SetCommandCodeAuditStatus_In)),
2399     /* outSize */ 0,
2400     /* offsetOfTypes */
2401     offsetof (SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t, types),
2402     /* offsets */ { (UINT16) (offsetof (SetCommandCodeAuditStatus_In,
2403     auditAlg)),
2404     (UINT16) (offsetof (SetCommandCodeAuditStatus_In,
2405     setList)),
2406     (UINT16) (offsetof (SetCommandCodeAuditStatus_In,
2407     clearList)) },
2408     /* types */
2409     {TPMI_RH_PROVISION_H_UNMARSHAL,
2410     TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2411     TPML_CC_P_UNMARSHAL,
2412     TPML_CC_P_UNMARSHAL,
2413     END_OF_LIST,
2414     END_OF_LIST}
2415 };
2416
2417 #define _SetCommandCodeAuditStatusDataAddress (&_SetCommandCodeAuditStatusData)
2418 #else
2419 #define _SetCommandCodeAuditStatusDataAddress 0
2420 #endif // CC_SetCommandCodeAuditStatus
2421
2422 #if CC_PCR_Extend
2423 #include "PCR_Extend_fp.h"
2424
2425 typedef TPM_RC (PCR_Extend_Entry) (
2426     PCR_Extend_In* in
2427 );
2428
2429 typedef const struct

```

```

2425 {
2426     PCR_Extend_Entry      *entry;
2427     UINT16                inSize;
2428     UINT16                outSize;
2429     UINT16                offsetOfTypes;
2430     UINT16                paramOffsets[1];
2431     BYTE                  types[4];
2432 } PCR_Extend_COMMAND_DESCRIPTOR_t;
2433
2434 PCR_Extend_COMMAND_DESCRIPTOR_t _PCR_ExtendData = {
2435     /* entry          */      &TPM2_PCR_Extend,
2436     /* inSize         */      (UINT16) (sizeof(PCR_Extend_In)),
2437     /* outSize        */      0,
2438     /* offsetOfTypes  */      offsetof(PCR_Extend_COMMAND_DESCRIPTOR_t, types),
2439     /* offsets        */      {(UINT16) (offsetof(PCR_Extend_In, digests))},
2440     /* types          */      {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
2441                             TPML_DIGEST_VALUES_P_UNMARSHAL,
2442                             END_OF_LIST,
2443                             END_OF_LIST};
2444 };
2445
2446 #define _PCR_ExtendDataAddress (&_PCR_ExtendData)
2447 #else
2448 #define _PCR_ExtendDataAddress 0
2449 #endif // CC_PCR_Extend
2450
2451 #if CC_PCR_Event
2452 #include "PCR_Event_fp.h"
2453
2454 typedef TPM_RC (PCR_Event_Entry) (
2455     PCR_Event_In*          in,
2456     PCR_Event_Out*         out
2457 );
2458
2459 typedef const struct
2460 {
2461     PCR_Event_Entry      *entry;
2462     UINT16                inSize;
2463     UINT16                outSize;
2464     UINT16                offsetOfTypes;
2465     UINT16                paramOffsets[1];
2466     BYTE                  types[5];
2467 } PCR_Event_COMMAND_DESCRIPTOR_t;
2468
2469 PCR_Event_COMMAND_DESCRIPTOR_t _PCR_EventData = {
2470     /* entry          */      &TPM2_PCR_Event,
2471     /* inSize         */      (UINT16) (sizeof(PCR_Event_In)),
2472     /* outSize        */      (UINT16) (sizeof(PCR_Event_Out)),
2473     /* offsetOfTypes  */      offsetof(PCR_Event_COMMAND_DESCRIPTOR_t, types),
2474     /* offsets        */      {(UINT16) (offsetof(PCR_Event_In, eventData))},
2475     /* types          */      {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
2476                             TPM2B_EVENT_P_UNMARSHAL,
2477                             END_OF_LIST,
2478                             TPML_DIGEST_VALUES_P_MARSHAL,
2479                             END_OF_LIST};
2480 };
2481
2482 #define _PCR_EventDataAddress (&_PCR_EventData)
2483 #else
2484 #define _PCR_EventDataAddress 0
2485 #endif // CC_PCR_Event
2486
2487 #if CC_PCR_Read
2488 #include "PCR_Read_fp.h"
2489
2490

```

```

2491 typedef TPM_RC (PCR_Read_Entry) (
2492     PCR_Read_In*           in,
2493     PCR_Read_Out*          out
2494 );
2495
2496
2497 typedef const struct
2498 {
2499     PCR_Read_Entry          *entry;
2500     UINT16                  inSize;
2501     UINT16                  outSize;
2502     UINT16                  offsetOfTypes;
2503     UINT16                  paramOffsets[2];
2504     BYTE                    types[6];
2505 } PCR_Read_COMMAND_DESCRIPTOR_t;
2506
2507 PCR_Read_COMMAND_DESCRIPTOR_t _PCR_ReadData = {
2508     /* entry          */      &TPM2_PCR_Read,
2509     /* inSize         */      (UINT16) (sizeof(PCR_Read_In)),
2510     /* outSize        */      (UINT16) (sizeof(PCR_Read_Out)),
2511     /* offsetOfTypes  */      offsetof(PCR_Read_COMMAND_DESCRIPTOR_t, types),
2512     /* offsets        */      {(UINT16) (offsetof(PCR_Read_Out, pcrSelectionOut)),
2513                               (UINT16) (offsetof(PCR_Read_Out, pcrValues))},
2514     /* types          */      {TPML_PCR_SELECTION_P_UNMARSHAL,
2515                               END_OF_LIST,
2516                               UINT32_P_MARSHAL,
2517                               TPML_PCR_SELECTION_P_MARSHAL,
2518                               TPML_DIGEST_P_MARSHAL,
2519                               END_OF_LIST}
2520 };
2521
2522 #define _PCR_ReadDataAddress (&_PCR_ReadData)
2523 #else
2524 #define _PCR_ReadDataAddress 0
2525 #endif // CC_PCR_Read
2526
2527 #if CC_PCR_Allocate
2528 #include "PCR_Allocate_fp.h"
2529
2530 typedef TPM_RC (PCR_Allocate_Entry) (
2531     PCR_Allocate_In*        in,
2532     PCR_Allocate_Out*       out
2533 );
2534
2535
2536 typedef const struct
2537 {
2538     PCR_Allocate_Entry      *entry;
2539     UINT16                  inSize;
2540     UINT16                  outSize;
2541     UINT16                  offsetOfTypes;
2542     UINT16                  paramOffsets[4];
2543     BYTE                    types[8];
2544 } PCR_Allocate_COMMAND_DESCRIPTOR_t;
2545
2546 PCR_Allocate_COMMAND_DESCRIPTOR_t _PCR_AllocateData = {
2547     /* entry          */      &TPM2_PCR_Allocate,
2548     /* inSize         */      (UINT16) (sizeof(PCR_Allocate_In)),
2549     /* outSize        */      (UINT16) (sizeof(PCR_Allocate_Out)),
2550     /* offsetOfTypes  */      offsetof(PCR_Allocate_COMMAND_DESCRIPTOR_t, types),
2551     /* offsets        */      {(UINT16) (offsetof(PCR_Allocate_In, pcrAllocation)),
2552                               (UINT16) (offsetof(PCR_Allocate_Out, maxPCR)),
2553                               (UINT16) (offsetof(PCR_Allocate_Out, sizeNeeded)),
2554                               (UINT16) (offsetof(PCR_Allocate_Out, sizeAvailable))},
2555     /* types          */      {TPMI_RH_PLATFORM_H_UNMARSHAL,
2556                               TPML_PCR_SELECTION_P_UNMARSHAL,

```

```

2557         END_OF_LIST,
2558         TPMI_YES_NO_P_MARSHAL,
2559         UINT32_P_MARSHAL,
2560         UINT32_P_MARSHAL,
2561         UINT32_P_MARSHAL,
2562         END_OF_LIST}
2563 };
2564
2565 #define _PCR_AllocateDataAddress (&_PCR_AllocateData)
2566 #else
2567 #define _PCR_AllocateDataAddress 0
2568 #endif // CC_PCR_Allocate
2569
2570 #if CC_PCR_SetAuthPolicy
2571 #include "PCR_SetAuthPolicy_fp.h"
2572
2573 typedef TPM_RC (PCR_SetAuthPolicy_Entry) (
2574     PCR_SetAuthPolicy_In* in
2575 );
2576
2577 typedef const struct
2578 {
2579     PCR_SetAuthPolicy_Entry *entry;
2580     UINT16 inSize;
2581     UINT16 outSize;
2582     UINT16 offsetOfTypes;
2583     UINT16 paramOffsets[3];
2584     BYTE types[6];
2585 } PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t;
2586
2587 PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t _PCR_SetAuthPolicyData = {
2588     /* entry */ &TPM2_PCR_SetAuthPolicy,
2589     /* inSize */ (UINT16)(sizeof(PCR_SetAuthPolicy_In)),
2590     /* outSize */ 0,
2591     /* offsetOfTypes */ offsetof(PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t,
2592 types),
2593     /* offsets */ { (UINT16)(offsetof(PCR_SetAuthPolicy_In, authPolicy)),
2594 (UINT16)(offsetof(PCR_SetAuthPolicy_In, hashAlg)),
2595 (UINT16)(offsetof(PCR_SetAuthPolicy_In, pcrNum)) },
2596     /* types */ {TPMI_RH_PLATFORM_H_UNMARSHAL,
2597 TPM2B_DIGEST_P_UNMARSHAL,
2598 TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2599 TPMI_DH_PCR_P_UNMARSHAL,
2600 END_OF_LIST,
2601 END_OF_LIST}
2602 };
2603
2604 #define _PCR_SetAuthPolicyDataAddress (&_PCR_SetAuthPolicyData)
2605 #else
2606 #define _PCR_SetAuthPolicyDataAddress 0
2607 #endif // CC_PCR_SetAuthPolicy
2608
2609 #if CC_PCR_SetAuthValue
2610 #include "PCR_SetAuthValue_fp.h"
2611
2612 typedef TPM_RC (PCR_SetAuthValue_Entry) (
2613     PCR_SetAuthValue_In* in
2614 );
2615
2616 typedef const struct
2617 {
2618     PCR_SetAuthValue_Entry *entry;
2619     UINT16 inSize;
2620     UINT16 outSize;

```

```

2622     UINT16                offsetOfTypes;
2623     UINT16                paramOffsets[1];
2624     BYTE                  types[4];
2625 } PCR_SetAuthValue_COMMAND_DESCRIPTOR_t;
2626
2627 PCR_SetAuthValue_COMMAND_DESCRIPTOR_t _PCR_SetAuthValueData = {
2628     /* entry                */      &TPM2_PCR_SetAuthValue,
2629     /* inSize              */      (UINT16) (sizeof(PCR_SetAuthValue_In)),
2630     /* outSize             */      0,
2631     /* offsetOfTypes      */      offsetof(PCR_SetAuthValue_COMMAND_DESCRIPTOR_t,
types),
2632     /* offsets            */      {(UINT16) (offsetof(PCR_SetAuthValue_In, auth))},
2633     /* types              */      {TPMI_DH_PCR_H_UNMARSHAL,
2634     TPM2B_DIGEST_P_UNMARSHAL,
2635     END_OF_LIST,
2636     END_OF_LIST};
2637 };
2638
2639 #define _PCR_SetAuthValueDataAddress (&_PCR_SetAuthValueData)
2640 #else
2641 #define _PCR_SetAuthValueDataAddress 0
2642 #endif // CC_PCR_SetAuthValue
2643
2644 #if CC_PCR_Reset
2645 #include "PCR_Reset_fp.h"
2646
2647 typedef TPM_RC (PCR_Reset_Entry) (
2648     PCR_Reset_In*          in
2649 );
2650
2651
2652 typedef const struct
2653 {
2654     PCR_Reset_Entry        *entry;
2655     UINT16                 inSize;
2656     UINT16                 outSize;
2657     UINT16                 offsetOfTypes;
2658     BYTE                   types[3];
2659 } PCR_Reset_COMMAND_DESCRIPTOR_t;
2660
2661 PCR_Reset_COMMAND_DESCRIPTOR_t _PCR_ResetData = {
2662     /* entry                */      &TPM2_PCR_Reset,
2663     /* inSize              */      (UINT16) (sizeof(PCR_Reset_In)),
2664     /* outSize             */      0,
2665     /* offsetOfTypes      */      offsetof(PCR_Reset_COMMAND_DESCRIPTOR_t, types),
2666     /* offsets            */      // No parameter offsets
2667     /* types              */      {TPMI_DH_PCR_H_UNMARSHAL,
2668     END_OF_LIST,
2669     END_OF_LIST};
2670 };
2671
2672 #define _PCR_ResetDataAddress (&_PCR_ResetData)
2673 #else
2674 #define _PCR_ResetDataAddress 0
2675 #endif // CC_PCR_Reset
2676
2677 #if CC_PolicySigned
2678 #include "PolicySigned_fp.h"
2679
2680 typedef TPM_RC (PolicySigned_Entry) (
2681     PolicySigned_In*        in,
2682     PolicySigned_Out*       out
2683 );
2684
2685
2686 typedef const struct

```

```

2687 {
2688     PolicySigned_Entry      *entry;
2689     UINT16                  inSize;
2690     UINT16                  outSize;
2691     UINT16                  offsetOfTypes;
2692     UINT16                  paramOffsets[7];
2693     BYTE                    types[11];
2694 } PolicySigned_COMMAND_DESCRIPTOR_t;
2695
2696 PolicySigned_COMMAND_DESCRIPTOR_t _PolicySignedData = {
2697     /* entry          */      &TPM2_PolicySigned,
2698     /* inSize         */      (UINT16) (sizeof(PolicySigned_In)),
2699     /* outSize        */      (UINT16) (sizeof(PolicySigned_Out)),
2700     /* offsetOfTypes  */      offsetof(PolicySigned_COMMAND_DESCRIPTOR_t, types),
2701     /* offsets        */      { (UINT16) (offsetof(PolicySigned_In, policySession)),
2702                               (UINT16) (offsetof(PolicySigned_In, nonceTPM)),
2703                               (UINT16) (offsetof(PolicySigned_In, cpHashA)),
2704                               (UINT16) (offsetof(PolicySigned_In, policyRef)),
2705                               (UINT16) (offsetof(PolicySigned_In, expiration)),
2706                               (UINT16) (offsetof(PolicySigned_In, auth)),
2707                               (UINT16) (offsetof(PolicySigned_Out, policyTicket))},
2708     /* types          */      {TPMI_DH_OBJECT_H_UNMARSHAL,
2709                               TPMI_SH_POLICY_H_UNMARSHAL,
2710                               TPM2B_NONCE_P_UNMARSHAL,
2711                               TPM2B_DIGEST_P_UNMARSHAL,
2712                               TPM2B_NONCE_P_UNMARSHAL,
2713                               INT32_P_UNMARSHAL,
2714                               TPMT_SIGNATURE_P_UNMARSHAL,
2715                               END_OF_LIST,
2716                               TPM2B_TIMEOUT_P_MARSHAL,
2717                               TPMT_TK_AUTH_P_MARSHAL,
2718                               END_OF_LIST};
2719 };
2720
2721 #define _PolicySignedDataAddress (&_PolicySignedData)
2722 #else
2723 #define _PolicySignedDataAddress 0
2724 #endif // CC_PolicySigned
2725
2726 #if CC_PolicySecret
2727 #include "PolicySecret_fp.h"
2728
2729 typedef TPM_RC (PolicySecret_Entry) (
2730     PolicySecret_In*      in,
2731     PolicySecret_Out*     out
2732 );
2733
2734 typedef const struct
2735 {
2736     PolicySecret_Entry      *entry;
2737     UINT16                  inSize;
2738     UINT16                  outSize;
2739     UINT16                  offsetOfTypes;
2740     UINT16                  paramOffsets[6];
2741     BYTE                    types[10];
2742 } PolicySecret_COMMAND_DESCRIPTOR_t;
2743
2744 PolicySecret_COMMAND_DESCRIPTOR_t _PolicySecretData = {
2745     /* entry          */      &TPM2_PolicySecret,
2746     /* inSize         */      (UINT16) (sizeof(PolicySecret_In)),
2747     /* outSize        */      (UINT16) (sizeof(PolicySecret_Out)),
2748     /* offsetOfTypes  */      offsetof(PolicySecret_COMMAND_DESCRIPTOR_t, types),
2749     /* offsets        */      { (UINT16) (offsetof(PolicySecret_In, policySession)),
2750                               (UINT16) (offsetof(PolicySecret_In, nonceTPM)),
2751                               (UINT16) (offsetof(PolicySecret_In, cpHashA)),

```



```

2753         (UINT16) (offsetof(PolicySecret_In, policyRef)),
2754         (UINT16) (offsetof(PolicySecret_In, expiration)),
2755         (UINT16) (offsetof(PolicySecret_Out, policyTicket))),
2756     /* types */
2757     {TPMI_DH_ENTITY_H_UNMARSHAL,
2758     TPMI_SH_POLICY_H_UNMARSHAL,
2759     TPM2B_NONCE_P_UNMARSHAL,
2760     TPM2B_DIGEST_P_UNMARSHAL,
2761     TPM2B_NONCE_P_UNMARSHAL,
2762     INT32_P_UNMARSHAL,
2763     END_OF_LIST,
2764     TPM2B_TIMEOUT_P_MARSHAL,
2765     TPMT_TK_AUTH_P_MARSHAL,
2766     END_OF_LIST};
2767 };
2768 #define _PolicySecretDataAddress (&_PolicySecretData)
2769 #else
2770 #define _PolicySecretDataAddress 0
2771 #endif // CC_PolicySecret
2772
2773 #if CC_PolicyTicket
2774 #include "PolicyTicket_fp.h"
2775
2776 typedef TPM_RC (PolicyTicket_Entry) (
2777     PolicyTicket_In* in
2778 );
2779
2780 typedef const struct
2781 {
2782     PolicyTicket_Entry *entry;
2783     UINT16 inSize;
2784     UINT16 outSize;
2785     UINT16 offsetOfTypes;
2786     UINT16 paramOffsets[5];
2787     BYTE types[8];
2788 } PolicyTicket_COMMAND_DESCRIPTOR_t;
2789
2790 PolicyTicket_COMMAND_DESCRIPTOR_t _PolicyTicketData = {
2791     /* entry */ &TPM2_PolicyTicket,
2792     /* inSize */ (UINT16) (sizeof(PolicyTicket_In)),
2793     /* outSize */ 0,
2794     /* offsetOfTypes */ offsetof(PolicyTicket_COMMAND_DESCRIPTOR_t, types),
2795     /* offsets */ { (UINT16) (offsetof(PolicyTicket_In, timeout)),
2796     (UINT16) (offsetof(PolicyTicket_In, cpHashA)),
2797     (UINT16) (offsetof(PolicyTicket_In, policyRef)),
2798     (UINT16) (offsetof(PolicyTicket_In, authName)),
2799     (UINT16) (offsetof(PolicyTicket_In, ticket))},
2800     /* types */
2801     {TPMI_SH_POLICY_H_UNMARSHAL,
2802     TPM2B_TIMEOUT_P_UNMARSHAL,
2803     TPM2B_DIGEST_P_UNMARSHAL,
2804     TPM2B_NONCE_P_UNMARSHAL,
2805     TPM2B_NAME_P_UNMARSHAL,
2806     TPMT_TK_AUTH_P_UNMARSHAL,
2807     END_OF_LIST,
2808     END_OF_LIST};
2809 };
2810
2811 #define _PolicyTicketDataAddress (&_PolicyTicketData)
2812 #else
2813 #define _PolicyTicketDataAddress 0
2814 #endif // CC_PolicyTicket
2815
2816 #if CC_PolicyOR
2817 #include "PolicyOR_fp.h"
2818

```

```

2819 typedef TPM_RC (PolicyOR_Entry) (
2820     PolicyOR_In*          in
2821 );
2822
2823
2824 typedef const struct
2825 {
2826     PolicyOR_Entry          *entry;
2827     UINT16                  inSize;
2828     UINT16                  outSize;
2829     UINT16                  offsetOfTypes;
2830     UINT16                  paramOffsets[1];
2831     BYTE                    types[4];
2832 } PolicyOR_COMMAND_DESCRIPTOR_t;
2833
2834 PolicyOR_COMMAND_DESCRIPTOR_t _PolicyORData = {
2835     /* entry          */      &TPM2_PolicyOR,
2836     /* inSize         */      (UINT16) (sizeof(PolicyOR_In)),
2837     /* outSize        */      0,
2838     /* offsetOfTypes  */      offsetof(PolicyOR_COMMAND_DESCRIPTOR_t, types),
2839     /* offsets        */      {(UINT16) (offsetof(PolicyOR_In, pHashList))},
2840     /* types          */      {TPMI_SH_POLICY_H_UNMARSHAL,
2841                               TPML_DIGEST_P_UNMARSHAL,
2842                               END_OF_LIST,
2843                               END_OF_LIST};
2844 };
2845
2846 #define _PolicyORDataAddress (&_PolicyORData)
2847 #else
2848 #define _PolicyORDataAddress 0
2849 #endif // CC_PolicyOR
2850
2851 #if CC_PolicyPCR
2852 #include "PolicyPCR_fp.h"
2853
2854 typedef TPM_RC (PolicyPCR_Entry) (
2855     PolicyPCR_In*          in
2856 );
2857
2858
2859 typedef const struct
2860 {
2861     PolicyPCR_Entry          *entry;
2862     UINT16                  inSize;
2863     UINT16                  outSize;
2864     UINT16                  offsetOfTypes;
2865     UINT16                  paramOffsets[2];
2866     BYTE                    types[5];
2867 } PolicyPCR_COMMAND_DESCRIPTOR_t;
2868
2869 PolicyPCR_COMMAND_DESCRIPTOR_t _PolicyPCRData = {
2870     /* entry          */      &TPM2_PolicyPCR,
2871     /* inSize         */      (UINT16) (sizeof(PolicyPCR_In)),
2872     /* outSize        */      0,
2873     /* offsetOfTypes  */      offsetof(PolicyPCR_COMMAND_DESCRIPTOR_t, types),
2874     /* offsets        */      {(UINT16) (offsetof(PolicyPCR_In, pcrDigest)),
2875                               (UINT16) (offsetof(PolicyPCR_In, pcrs))},
2876     /* types          */      {TPMI_SH_POLICY_H_UNMARSHAL,
2877                               TPM2B_DIGEST_P_UNMARSHAL,
2878                               TPML_PCR_SELECTION_P_UNMARSHAL,
2879                               END_OF_LIST,
2880                               END_OF_LIST};
2881 };
2882
2883 #define _PolicyPCRDataAddress (&_PolicyPCRData)
2884 #else

```

```

2885 #define _PolicyPCRDataAddress 0
2886 #endif // CC_PolicyPCR
2887
2888 #if CC_PolicyLocality
2889 #include "PolicyLocality_fp.h"
2890
2891 typedef TPM_RC (PolicyLocality_Entry)(
2892     PolicyLocality_In* in
2893 );
2894
2895
2896 typedef const struct
2897 {
2898     PolicyLocality_Entry *entry;
2899     UINT16 inSize;
2900     UINT16 outSize;
2901     UINT16 offsetOfTypes;
2902     UINT16 paramOffsets[1];
2903     BYTE types[4];
2904 } PolicyLocality_COMMAND_DESCRIPTOR_t;
2905
2906 PolicyLocality_COMMAND_DESCRIPTOR_t _PolicyLocalityData = {
2907     /* entry */ &TPM2_PolicyLocality,
2908     /* inSize */ (UINT16) (sizeof(PolicyLocality_In)),
2909     /* outSize */ 0,
2910     /* offsetOfTypes */ offsetof(PolicyLocality_COMMAND_DESCRIPTOR_t, types),
2911     /* offsets */ {(UINT16) (offsetof(PolicyLocality_In, locality))},
2912     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2913                 TPMA_LOCALITY_P_UNMARSHAL,
2914                 END_OF_LIST,
2915                 END_OF_LIST}
2916 };
2917
2918 #define _PolicyLocalityDataAddress (&_PolicyLocalityData)
2919 #else
2920 #define _PolicyLocalityDataAddress 0
2921 #endif // CC_PolicyLocality
2922
2923 #if CC_PolicyNV
2924 #include "PolicyNV_fp.h"
2925
2926 typedef TPM_RC (PolicyNV_Entry)(
2927     PolicyNV_In* in
2928 );
2929
2930
2931 typedef const struct
2932 {
2933     PolicyNV_Entry *entry;
2934     UINT16 inSize;
2935     UINT16 outSize;
2936     UINT16 offsetOfTypes;
2937     UINT16 paramOffsets[5];
2938     BYTE types[8];
2939 } PolicyNV_COMMAND_DESCRIPTOR_t;
2940
2941 PolicyNV_COMMAND_DESCRIPTOR_t _PolicyNVData = {
2942     /* entry */ &TPM2_PolicyNV,
2943     /* inSize */ (UINT16) (sizeof(PolicyNV_In)),
2944     /* outSize */ 0,
2945     /* offsetOfTypes */ offsetof(PolicyNV_COMMAND_DESCRIPTOR_t, types),
2946     /* offsets */ {(UINT16) (offsetof(PolicyNV_In, nvIndex)),
2947                 (UINT16) (offsetof(PolicyNV_In, policySession)),
2948                 (UINT16) (offsetof(PolicyNV_In, operandB)),
2949                 (UINT16) (offsetof(PolicyNV_In, offset)),
2950                 (UINT16) (offsetof(PolicyNV_In, operation))},

```

```

2951     /* types */
2952
2953     {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2954     TPMI_RH_NV_INDEX_H_UNMARSHAL,
2955     TPMI_SH_POLICY_H_UNMARSHAL,
2956     TPM2B_OPERAND_P_UNMARSHAL,
2957     UINT16_P_UNMARSHAL,
2958     TPM_EO_P_UNMARSHAL,
2959     END_OF_LIST,
2960     END_OF_LIST}
2961 };
2962
2963 #define _PolicyNVDDataAddress (&_PolicyNVDData)
2964 #else
2965 #define _PolicyNVDDataAddress 0
2966 #endif // CC_PolicyNV
2967
2968 #if CC_PolicyCounterTimer
2969 #include "PolicyCounterTimer_fp.h"
2970
2971 typedef TPM_RC (PolicyCounterTimer_Entry) (
2972     PolicyCounterTimer_In* in
2973 );
2974
2975 typedef const struct
2976 {
2977     PolicyCounterTimer_Entry *entry;
2978     UINT16 inSize;
2979     UINT16 outSize;
2980     UINT16 offsetOfTypes;
2981     paramOffsets[3];
2982     BYTE types[6];
2983 } PolicyCounterTimer_COMMAND_DESCRIPTOR_t;
2984
2985 PolicyCounterTimer_COMMAND_DESCRIPTOR_t _PolicyCounterTimerData = {
2986     /* entry */ &TPM2_PolicyCounterTimer,
2987     /* inSize */ (UINT16) (sizeof(PolicyCounterTimer_In)),
2988     /* outSize */ 0,
2989     /* offsetOfTypes */ offsetof(PolicyCounterTimer_COMMAND_DESCRIPTOR_t,
2990     types),
2991     /* offsets */ { (UINT16) (offsetof(PolicyCounterTimer_In, operandB)),
2992     (UINT16) (offsetof(PolicyCounterTimer_In, offset)),
2993     (UINT16) (offsetof(PolicyCounterTimer_In,
2994     operation)) },
2995     /* types */
2996     {TPMI_SH_POLICY_H_UNMARSHAL,
2997     TPM2B_OPERAND_P_UNMARSHAL,
2998     UINT16_P_UNMARSHAL,
2999     TPM_EO_P_UNMARSHAL,
3000     END_OF_LIST,
3001     END_OF_LIST}
3002 };
3003
3004 #define _PolicyCounterTimerDataAddress (&_PolicyCounterTimerData)
3005 #else
3006 #define _PolicyCounterTimerDataAddress 0
3007 #endif // CC_PolicyCounterTimer
3008
3009 #if CC_PolicyCommandCode
3010 #include "PolicyCommandCode_fp.h"
3011
3012 typedef TPM_RC (PolicyCommandCode_Entry) (
3013     PolicyCommandCode_In* in
3014 );
3015
3016 typedef const struct
3017 {

```

```

3015     PolicyCommandCode_Entry      *entry;
3016     UINT16                        inSize;
3017     UINT16                        outSize;
3018     UINT16                        offsetOfTypes;
3019     UINT16                        paramOffsets[1];
3020     BYTE                           types[4];
3021 } PolicyCommandCode_COMMAND_DESCRIPTOR_t;
3022
3023 PolicyCommandCode_COMMAND_DESCRIPTOR_t _PolicyCommandCodeData = {
3024     /* entry */ &TPM2_PolicyCommandCode,
3025     /* inSize */ (UINT16) (sizeof(PolicyCommandCode_In)),
3026     /* outSize */ 0,
3027     /* offsetOfTypes */ offsetof(PolicyCommandCode_COMMAND_DESCRIPTOR_t,
types),
3028     /* offsets */ { (UINT16) (offsetof(PolicyCommandCode_In, code)) },
3029     /* types */ { TPMI_SH_POLICY_H_UNMARSHAL,
TPM_CC_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST }
3030 };
3031
3032 #define _PolicyCommandCodeDataAddress (&_PolicyCommandCodeData)
3033 #else
3034 #define _PolicyCommandCodeDataAddress 0
3035 #endif // CC_PolicyCommandCode
3036
3037 #if CC_PolicyPhysicalPresence
3038 #include "PolicyPhysicalPresence_fp.h"
3039
3040 typedef TPM_RC (PolicyPhysicalPresence_Entry) (
3041     PolicyPhysicalPresence_In* in
3042 );
3043
3044 typedef const struct
3045 {
3046     PolicyPhysicalPresence_Entry *entry;
3047     UINT16 inSize;
3048     UINT16 outSize;
3049     UINT16 offsetOfTypes;
3050     BYTE types[3];
3051 } PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t;
3052
3053 PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t _PolicyPhysicalPresenceData = {
3054     /* entry */ &TPM2_PolicyPhysicalPresence,
3055     /* inSize */ (UINT16) (sizeof(PolicyPhysicalPresence_In)),
3056     /* outSize */ 0,
3057     /* offsetOfTypes */ offsetof(PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t,
types),
3058     /* offsets */ // No parameter offsets
3059     /* types */ { TPMI_SH_POLICY_H_UNMARSHAL,
END_OF_LIST,
END_OF_LIST }
3060 };
3061
3062 #define _PolicyPhysicalPresenceDataAddress (&_PolicyPhysicalPresenceData)
3063 #else
3064 #define _PolicyPhysicalPresenceDataAddress 0
3065 #endif // CC_PolicyPhysicalPresence
3066
3067 #if CC_PolicyCpHash
3068 #include "PolicyCpHash_fp.h"
3069
3070 typedef TPM_RC (PolicyCpHash_Entry) (
3071     PolicyCpHash_In* in
3072 );

```

```

3079
3080
3081 typedef const struct
3082 {
3083     PolicyCpHash_Entry      *entry;
3084     UINT16                  inSize;
3085     UINT16                  outSize;
3086     UINT16                  offsetOfTypes;
3087     UINT16                  paramOffsets[1];
3088     BYTE                    types[4];
3089 } PolicyCpHash_COMMAND_DESCRIPTOR_t;
3090
3091 PolicyCpHash_COMMAND_DESCRIPTOR_t _PolicyCpHashData = {
3092     /* entry */          &TPM2_PolicyCpHash,
3093     /* inSize */         (UINT16) (sizeof(PolicyCpHash_In)),
3094     /* outSize */        0,
3095     /* offsetOfTypes */  offsetof(PolicyCpHash_COMMAND_DESCRIPTOR_t, types),
3096     /* offsets */        {(UINT16) (offsetof(PolicyCpHash_In, cpHashA))},
3097     /* types */          {TPMI_SH_POLICY_H_UNMARSHAL,
3098                          TPM2B_DIGEST_P_UNMARSHAL,
3099                          END_OF_LIST,
3100                          END_OF_LIST};
3101 };
3102
3103 #define _PolicyCpHashDataAddress (&_PolicyCpHashData)
3104 #else
3105 #define _PolicyCpHashDataAddress 0
3106 #endif // CC_PolicyCpHash
3107
3108 #if CC_PolicyNameHash
3109 #include "PolicyNameHash_fp.h"
3110
3111 typedef TPM_RC (PolicyNameHash_Entry)(
3112     PolicyNameHash_In* in
3113 );
3114
3115
3116 typedef const struct
3117 {
3118     PolicyNameHash_Entry      *entry;
3119     UINT16                  inSize;
3120     UINT16                  outSize;
3121     UINT16                  offsetOfTypes;
3122     UINT16                  paramOffsets[1];
3123     BYTE                    types[4];
3124 } PolicyNameHash_COMMAND_DESCRIPTOR_t;
3125
3126 PolicyNameHash_COMMAND_DESCRIPTOR_t _PolicyNameHashData = {
3127     /* entry */          &TPM2_PolicyNameHash,
3128     /* inSize */         (UINT16) (sizeof(PolicyNameHash_In)),
3129     /* outSize */        0,
3130     /* offsetOfTypes */  offsetof(PolicyNameHash_COMMAND_DESCRIPTOR_t, types),
3131     /* offsets */        {(UINT16) (offsetof(PolicyNameHash_In, nameHash))},
3132     /* types */          {TPMI_SH_POLICY_H_UNMARSHAL,
3133                          TPM2B_DIGEST_P_UNMARSHAL,
3134                          END_OF_LIST,
3135                          END_OF_LIST};
3136 };
3137
3138 #define _PolicyNameHashDataAddress (&_PolicyNameHashData)
3139 #else
3140 #define _PolicyNameHashDataAddress 0
3141 #endif // CC_PolicyNameHash
3142
3143 #if CC_PolicyDuplicationSelect
3144 #include "PolicyDuplicationSelect_fp.h"

```



```

3145
3146 typedef TPM_RC (PolicyDuplicationSelect_Entry) (
3147     PolicyDuplicationSelect_In* in
3148 );
3149
3150
3151 typedef const struct
3152 {
3153     PolicyDuplicationSelect_Entry    *entry;
3154     UINT16                           inSize;
3155     UINT16                           outSize;
3156     UINT16                           offsetOfTypes;
3157     UINT16                           paramOffsets[3];
3158     BYTE                             types[6];
3159 } PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t;
3160
3161 PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t _PolicyDuplicationSelectData = {
3162     /* entry          */      &TPM2_PolicyDuplicationSelect,
3163     /* inSize         */      (UINT16) (sizeof(PolicyDuplicationSelect_In)),
3164     /* outSize        */      0,
3165     /* offsetOfTypes  */      offsetof(PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t,
types),
3166     /* offsets        */      { (UINT16) (offsetof(PolicyDuplicationSelect_In,
objectName)),
                                (UINT16) (offsetof(PolicyDuplicationSelect_In,
newParentName)),
                                (UINT16) (offsetof(PolicyDuplicationSelect_In,
includeObject)) },
3167     /* types          */      { TPMI_SH_POLICY_H_UNMARSHAL,
                                TPM2B_NAME_P_UNMARSHAL,
                                TPM2B_NAME_P_UNMARSHAL,
                                TPMI_YES_NO_P_UNMARSHAL,
                                END_OF_LIST,
                                END_OF_LIST }
3168 };
3169
3170 #define _PolicyDuplicationSelectDataAddress (&_PolicyDuplicationSelectData)
3171 #else
3172 #define _PolicyDuplicationSelectDataAddress 0
3173 #endif // CC_PolicyDuplicationSelect
3174
3175 #if CC_PolicyAuthorize
3176 #include "PolicyAuthorize_fp.h"
3177
3178 typedef TPM_RC (PolicyAuthorize_Entry) (
3179     PolicyAuthorize_In* in
3180 );
3181
3182 typedef const struct
3183 {
3184     PolicyAuthorize_Entry    *entry;
3185     UINT16                   inSize;
3186     UINT16                   outSize;
3187     UINT16                   offsetOfTypes;
3188     UINT16                   paramOffsets[4];
3189     BYTE                     types[7];
3190 } PolicyAuthorize_COMMAND_DESCRIPTOR_t;
3191
3192 PolicyAuthorize_COMMAND_DESCRIPTOR_t _PolicyAuthorizeData = {
3193     /* entry          */      &TPM2_PolicyAuthorize,
3194     /* inSize         */      (UINT16) (sizeof(PolicyAuthorize_In)),
3195     /* outSize        */      0,
3196     /* offsetOfTypes  */      offsetof(PolicyAuthorize_COMMAND_DESCRIPTOR_t, types),
3197     /* offsets        */      { (UINT16) (offsetof(PolicyAuthorize_In,
approvedPolicy)),

```

```

3206             (UINT16) (offsetof(PolicyAuthorize_In, policyRef)),
3207             (UINT16) (offsetof(PolicyAuthorize_In, keySign)),
3208             (UINT16) (offsetof(PolicyAuthorize_In, checkTicket))),
3209     /* types */
3210     {TPMI_SH_POLICY_H_UNMARSHAL,
3211     TPM2B_DIGEST_P_UNMARSHAL,
3212     TPM2B_NONCE_P_UNMARSHAL,
3213     TPM2B_NAME_P_UNMARSHAL,
3214     TPMT_TK_VERIFIED_P_UNMARSHAL,
3215     END_OF_LIST,
3216     END_OF_LIST}
3217 };
3218 #define _PolicyAuthorizeDataAddress (&_PolicyAuthorizeData)
3219 #else
3220 #define _PolicyAuthorizeDataAddress 0
3221 #endif // CC_PolicyAuthorize
3222
3223 #if CC_PolicyAuthValue
3224 #include "PolicyAuthValue_fp.h"
3225
3226 typedef TPM_RC (PolicyAuthValue_Entry) (
3227     PolicyAuthValue_In* in
3228 );
3229
3230
3231 typedef const struct
3232 {
3233     PolicyAuthValue_Entry *entry;
3234     UINT16 inSize;
3235     UINT16 outSize;
3236     UINT16 offsetOfTypes;
3237     BYTE types[3];
3238 } PolicyAuthValue_COMMAND_DESCRIPTOR_t;
3239
3240 PolicyAuthValue_COMMAND_DESCRIPTOR_t _PolicyAuthValueData = {
3241     /* entry */ &TPM2_PolicyAuthValue,
3242     /* inSize */ (UINT16) (sizeof(PolicyAuthValue_In)),
3243     /* outSize */ 0,
3244     /* offsetOfTypes */ offsetof(PolicyAuthValue_COMMAND_DESCRIPTOR_t, types),
3245     /* offsets */ // No parameter offsets
3246     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
3247     END_OF_LIST,
3248     END_OF_LIST}
3249 };
3250
3251 #define _PolicyAuthValueDataAddress (&_PolicyAuthValueData)
3252 #else
3253 #define _PolicyAuthValueDataAddress 0
3254 #endif // CC_PolicyAuthValue
3255
3256 #if CC_PolicyPassword
3257 #include "PolicyPassword_fp.h"
3258
3259 typedef TPM_RC (PolicyPassword_Entry) (
3260     PolicyPassword_In* in
3261 );
3262
3263
3264 typedef const struct
3265 {
3266     PolicyPassword_Entry *entry;
3267     UINT16 inSize;
3268     UINT16 outSize;
3269     UINT16 offsetOfTypes;
3270     BYTE types[3];
3271 } PolicyPassword_COMMAND_DESCRIPTOR_t;

```

```

3272
3273 PolicyPassword_COMMAND_DESCRIPTOR_t _PolicyPasswordData = {
3274     /* entry */           &TPM2_PolicyPassword,
3275     /* inSize */          (UINT16) (sizeof(PolicyPassword_In)),
3276     /* outSize */         0,
3277     /* offsetOfTypes */   offsetof(PolicyPassword_COMMAND_DESCRIPTOR_t, types),
3278     /* offsets */          // No parameter offsets
3279     /* types */            {TPMI_SH_POLICY_H_UNMARSHAL,
3280                           END_OF_LIST,
3281                           END_OF_LIST}
3282 };
3283
3284 #define _PolicyPasswordDataAddress (&_PolicyPasswordData)
3285 #else
3286 #define _PolicyPasswordDataAddress 0
3287 #endif // CC_PolicyPassword
3288
3289 #if CC_PolicyGetDigest
3290 #include "PolicyGetDigest_fp.h"
3291
3292 typedef TPM_RC (PolicyGetDigest_Entry) (
3293     PolicyGetDigest_In*      in,
3294     PolicyGetDigest_Out*     out
3295 );
3296
3297
3298 typedef const struct
3299 {
3300     PolicyGetDigest_Entry      *entry;
3301     UINT16                     inSize;
3302     UINT16                     outSize;
3303     UINT16                     offsetOfTypes;
3304     BYTE                       types[4];
3305 } PolicyGetDigest_COMMAND_DESCRIPTOR_t;
3306
3307 PolicyGetDigest_COMMAND_DESCRIPTOR_t _PolicyGetDigestData = {
3308     /* entry */           &TPM2_PolicyGetDigest,
3309     /* inSize */          (UINT16) (sizeof(PolicyGetDigest_In)),
3310     /* outSize */         (UINT16) (sizeof(PolicyGetDigest_Out)),
3311     /* offsetOfTypes */   offsetof(PolicyGetDigest_COMMAND_DESCRIPTOR_t, types),
3312     /* offsets */          // No parameter offsets
3313     /* types */            {TPMI_SH_POLICY_H_UNMARSHAL,
3314                           END_OF_LIST,
3315                           TPM2B_DIGEST_P_MARSHAL,
3316                           END_OF_LIST}
3317 };
3318
3319 #define _PolicyGetDigestDataAddress (&_PolicyGetDigestData)
3320 #else
3321 #define _PolicyGetDigestDataAddress 0
3322 #endif // CC_PolicyGetDigest
3323
3324 #if CC_PolicyNvWritten
3325 #include "PolicyNvWritten_fp.h"
3326
3327 typedef TPM_RC (PolicyNvWritten_Entry) (
3328     PolicyNvWritten_In*      in
3329 );
3330
3331
3332 typedef const struct
3333 {
3334     PolicyNvWritten_Entry      *entry;
3335     UINT16                     inSize;
3336     UINT16                     outSize;
3337     UINT16                     offsetOfTypes;

```

```

3338     UINT16                paramOffsets[1];
3339     BYTE                  types[4];
3340 } PolicyNvWritten_COMMAND_DESCRIPTOR_t;
3341
3342 PolicyNvWritten_COMMAND_DESCRIPTOR_t _PolicyNvWrittenData = {
3343     /* entry                */ &TPM2_PolicyNvWritten,
3344     /* inSize               */ (UINT16) (sizeof(PolicyNvWritten_In)),
3345     /* outSize              */ 0,
3346     /* offsetOfTypes        */ offsetof(PolicyNvWritten_COMMAND_DESCRIPTOR_t, types),
3347     /* offsets              */ {(UINT16) (offsetof(PolicyNvWritten_In, writtenSet))},
3348     /* types                */ {TPMI_SH_POLICY_H_UNMARSHAL,
3349                                TPMI_YES_NO_P_UNMARSHAL,
3350                                END_OF_LIST,
3351                                END_OF_LIST};
3352 };
3353
3354 #define _PolicyNvWrittenDataAddress (&_PolicyNvWrittenData)
3355 #else
3356 #define _PolicyNvWrittenDataAddress 0
3357 #endif // CC_PolicyNvWritten
3358
3359 #if CC_PolicyTemplate
3360 #include "PolicyTemplate_fp.h"
3361
3362 typedef TPM_RC (PolicyTemplate_Entry) (
3363     PolicyTemplate_In* in
3364 );
3365
3366 typedef const struct
3367 {
3368     PolicyTemplate_Entry *entry;
3369     UINT16 inSize;
3370     UINT16 outSize;
3371     UINT16 offsetOfTypes;
3372     UINT16 paramOffsets[1];
3373     BYTE types[4];
3374 } PolicyTemplate_COMMAND_DESCRIPTOR_t;
3375
3376 PolicyTemplate_COMMAND_DESCRIPTOR_t _PolicyTemplateData = {
3377     /* entry                */ &TPM2_PolicyTemplate,
3378     /* inSize               */ (UINT16) (sizeof(PolicyTemplate_In)),
3379     /* outSize              */ 0,
3380     /* offsetOfTypes        */ offsetof(PolicyTemplate_COMMAND_DESCRIPTOR_t, types),
3381     /* offsets              */ {(UINT16) (offsetof(PolicyTemplate_In, templateHash))},
3382     /* types                */ {TPMI_SH_POLICY_H_UNMARSHAL,
3383                                TPM2B_DIGEST_P_UNMARSHAL,
3384                                END_OF_LIST,
3385                                END_OF_LIST};
3386 };
3387
3388 #define _PolicyTemplateDataAddress (&_PolicyTemplateData)
3389 #else
3390 #define _PolicyTemplateDataAddress 0
3391 #endif // CC_PolicyTemplate
3392
3393 #if CC_PolicyAuthorizeNV
3394 #include "PolicyAuthorizeNV_fp.h"
3395
3396 typedef TPM_RC (PolicyAuthorizeNV_Entry) (
3397     PolicyAuthorizeNV_In* in
3398 );
3399
3400 typedef const struct
3401 {

```

```

3404     PolicyAuthorizeNV_Entry      *entry;
3405     UINT16                       inSize;
3406     UINT16                       outSize;
3407     UINT16                       offsetOfTypes;
3408     UINT16                       paramOffsets[2];
3409     BYTE                         types[5];
3410 } PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t;
3411
3412 PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t _PolicyAuthorizeNVData = {
3413     /* entry */                &TPM2_PolicyAuthorizeNV,
3414     /* inSize */               (UINT16) (sizeof(PolicyAuthorizeNV_In)),
3415     /* outSize */              0,
3416     /* offsetOfTypes */        offsetof(PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t,
types),
3417     /* offsets */              { (UINT16) (offsetof(PolicyAuthorizeNV_In, nvIndex)),
3418                               (UINT16) (offsetof(PolicyAuthorizeNV_In,
policySession)) },
3419     /* types */                { TPMI_RH_NV_AUTH_H_UNMARSHAL,
3420                               TPMI_RH_NV_INDEX_H_UNMARSHAL,
3421                               TPMI_SH_POLICY_H_UNMARSHAL,
3422                               END_OF_LIST,
3423                               END_OF_LIST };
3424 };
3425
3426 #define _PolicyAuthorizeNVDataAddress (&_PolicyAuthorizeNVData)
3427 #else
3428 #define _PolicyAuthorizeNVDataAddress 0
3429 #endif // CC_PolicyAuthorizeNV
3430
3431 #if CC_PolicyCapability
3432 #include "PolicyCapability_fp.h"
3433
3434 typedef TPM_RC (PolicyCapability_Entry) (
3435     PolicyCapability_In* in
3436 );
3437
3438
3439 typedef const struct
3440 {
3441     PolicyCapability_Entry      *entry;
3442     UINT16                       inSize;
3443     UINT16                       outSize;
3444     UINT16                       offsetOfTypes;
3445     UINT16                       paramOffsets[5];
3446     BYTE                         types[8];
3447 } PolicyCapability_COMMAND_DESCRIPTOR_t;
3448
3449 PolicyCapability_COMMAND_DESCRIPTOR_t _PolicyCapabilityData = {
3450     /* entry */                &TPM2_PolicyCapability,
3451     /* inSize */               (UINT16) (sizeof(PolicyCapability_In)),
3452     /* outSize */              0,
3453     /* offsetOfTypes */        offsetof(PolicyCapability_COMMAND_DESCRIPTOR_t,
types),
3454     /* offsets */              { (UINT16) (offsetof(PolicyCapability_In, operandB)),
3455                               (UINT16) (offsetof(PolicyCapability_In, offset)),
3456                               (UINT16) (offsetof(PolicyCapability_In, operation)),
3457                               (UINT16) (offsetof(PolicyCapability_In, capability)),
3458                               (UINT16) (offsetof(PolicyCapability_In, property)) },
3459     /* types */                { TPMI_SH_POLICY_H_UNMARSHAL,
3460                               TPM2B_OPERAND_P_UNMARSHAL,
3461                               UINT16_P_UNMARSHAL,
3462                               TPM_EO_P_UNMARSHAL,
3463                               TPM_CAP_P_UNMARSHAL,
3464                               UINT32_P_UNMARSHAL,
3465                               END_OF_LIST,
3466                               END_OF_LIST };

```

```

3467 };
3468
3469 #define _PolicyCapabilityDataAddress (&_PolicyCapabilityData)
3470 #else
3471 #define _PolicyCapabilityDataAddress 0
3472 #endif // CC_PolicyCapability
3473
3474 #if CC_PolicyParameters
3475 #include "PolicyParameters_fp.h"
3476
3477 typedef TPM_RC (PolicyParameters_Entry) (
3478     PolicyParameters_In* in
3479 );
3480
3481
3482 typedef const struct
3483 {
3484     PolicyParameters_Entry *entry;
3485     UINT16 inSize;
3486     UINT16 outSize;
3487     UINT16 offsetOfTypes;
3488     UINT16 paramOffsets[1];
3489     BYTE types[4];
3490 } PolicyParameters_COMMAND_DESCRIPTOR_t;
3491
3492 PolicyParameters_COMMAND_DESCRIPTOR_t _PolicyParametersData = {
3493     /* entry */ &TPM2_PolicyParameters,
3494     /* inSize */ (UINT16) (sizeof(PolicyParameters_In)),
3495     /* outSize */ 0,
3496     /* offsetOfTypes */ offsetof(PolicyParameters_COMMAND_DESCRIPTOR_t,
types),
3497     /* offsets */ {(UINT16) (offsetof(PolicyParameters_In, pHash))},
3498     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
TPM2B_DIGEST_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST}
3502 };
3503
3504 #define _PolicyParametersDataAddress (&_PolicyParametersData)
3505 #else
3506 #define _PolicyParametersDataAddress 0
3507 #endif // CC_PolicyParameters
3508
3509 #if CC_CreatePrimary
3510 #include "CreatePrimary_fp.h"
3511
3512 typedef TPM_RC (CreatePrimary_Entry) (
3513     CreatePrimary_In* in,
3514     CreatePrimary_Out* out
3515 );
3516
3517
3518 typedef const struct
3519 {
3520     CreatePrimary_Entry *entry;
3521     UINT16 inSize;
3522     UINT16 outSize;
3523     UINT16 offsetOfTypes;
3524     UINT16 paramOffsets[9];
3525     BYTE types[13];
3526 } CreatePrimary_COMMAND_DESCRIPTOR_t;
3527
3528 CreatePrimary_COMMAND_DESCRIPTOR_t _CreatePrimaryData = {
3529     /* entry */ &TPM2_CreatePrimary,
3530     /* inSize */ (UINT16) (sizeof(CreatePrimary_In)),
3531     /* outSize */ (UINT16) (sizeof(CreatePrimary_Out)),

```



```

3532      /* offsetOfTypes */      offsetof(CreatePrimary_COMMAND_DESCRIPTOR_t, types),
3533      /* offsets */            {(UINT16) (offsetof(CreatePrimary_In, inSensitive)),
3534                                (UINT16) (offsetof(CreatePrimary_In, inPublic)),
3535                                (UINT16) (offsetof(CreatePrimary_In, outsideInfo)),
3536                                (UINT16) (offsetof(CreatePrimary_In, creationPCR)),
3537                                (UINT16) (offsetof(CreatePrimary_Out, outPublic)),
3538                                (UINT16) (offsetof(CreatePrimary_Out, creationData)),
3539                                (UINT16) (offsetof(CreatePrimary_Out, creationHash)),
3540                                (UINT16) (offsetof(CreatePrimary_Out,
creationTicket))),
3541                                (UINT16) (offsetof(CreatePrimary_Out, name))},
3542      /* types */              {TPMI_RH_HIERARCHY_H_UNMARSHAL + ADD_FLAG,
3543                                TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
3544                                TPM2B_PUBLIC_P_UNMARSHAL,
3545                                TPM2B_DATA_P_UNMARSHAL,
3546                                TPML_PCR_SELECTION_P_UNMARSHAL,
3547                                END_OF_LIST,
3548                                TPM_HANDLE_H_MARSHAL,
3549                                TPM2B_PUBLIC_P_MARSHAL,
3550                                TPM2B_CREATION_DATA_P_MARSHAL,
3551                                TPM2B_DIGEST_P_MARSHAL,
3552                                TPMT_TK_CREATION_P_MARSHAL,
3553                                TPM2B_NAME_P_MARSHAL,
3554                                END_OF_LIST}
3555  };
3556
3557  #define _CreatePrimaryDataAddress (&_CreatePrimaryData)
3558  #else
3559  #define _CreatePrimaryDataAddress 0
3560  #endif // CC_CreatePrimary
3561
3562  #if CC_HierarchyControl
3563  #include "HierarchyControl_fp.h"
3564
3565  typedef TPM_RC (HierarchyControl_Entry) (
3566      HierarchyControl_In* in
3567  );
3568
3569
3570  typedef const struct
3571  {
3572      HierarchyControl_Entry *entry;
3573      UINT16 inSize;
3574      UINT16 outSize;
3575      UINT16 offsetOfTypes;
3576      UINT16 paramOffsets[2];
3577      BYTE types[5];
3578  } HierarchyControl_COMMAND_DESCRIPTOR_t;
3579
3580  HierarchyControl_COMMAND_DESCRIPTOR_t _HierarchyControlData = {
3581      /* entry */                &TPM2_HierarchyControl,
3582      /* inSize */               (UINT16) (sizeof(HierarchyControl_In)),
3583      /* outSize */              0,
3584      /* offsetOfTypes */        offsetof(HierarchyControl_COMMAND_DESCRIPTOR_t,
types),
3585      /* offsets */              {(UINT16) (offsetof(HierarchyControl_In, enable)),
3586                                (UINT16) (offsetof(HierarchyControl_In, state))},
3587      /* types */                {TPMI_RH_HIERARCHY_H_UNMARSHAL,
3588                                TPMI_RH_ENABLES_P_UNMARSHAL,
3589                                TPMI_YES_NO_P_UNMARSHAL,
3590                                END_OF_LIST,
3591                                END_OF_LIST}
3592  };
3593
3594  #define _HierarchyControlDataAddress (&_HierarchyControlData)
3595  #else

```

```

3596 #define HierarchyControlDataAddress 0
3597 #endif // CC_HierarchyControl
3598
3599 #if CC_SetPrimaryPolicy
3600 #include "SetPrimaryPolicy_fp.h"
3601
3602 typedef TPM_RC (SetPrimaryPolicy_Entry) (
3603     SetPrimaryPolicy_In* in
3604 );
3605
3606
3607 typedef const struct
3608 {
3609     SetPrimaryPolicy_Entry *entry;
3610     UINT16 inSize;
3611     UINT16 outSize;
3612     UINT16 offsetOfTypes;
3613     UINT16 paramOffsets[2];
3614     BYTE types[5];
3615 } SetPrimaryPolicy_COMMAND_DESCRIPTOR_t;
3616
3617 SetPrimaryPolicy_COMMAND_DESCRIPTOR_t _SetPrimaryPolicyData = {
3618     /* entry */ &TPM2_SetPrimaryPolicy,
3619     /* inSize */ (UINT16) (sizeof(SetPrimaryPolicy_In)),
3620     /* outSize */ 0,
3621     /* offsetOfTypes */ offsetof(SetPrimaryPolicy_COMMAND_DESCRIPTOR_t,
3622     types),
3623     /* offsets */ { (UINT16) (offsetof(SetPrimaryPolicy_In, authPolicy)),
3624     (UINT16) (offsetof(SetPrimaryPolicy_In, hashAlg)) },
3625     /* types */ { TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL,
3626     TPM2B_DIGEST_P_UNMARSHAL,
3627     TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
3628     END_OF_LIST,
3629     END_OF_LIST };
3630
3631 #define _SetPrimaryPolicyDataAddress (&_SetPrimaryPolicyData)
3632 #else
3633 #define _SetPrimaryPolicyDataAddress 0
3634 #endif // CC_SetPrimaryPolicy
3635
3636 #if CC_ChangePPS
3637 #include "ChangePPS_fp.h"
3638
3639 typedef TPM_RC (ChangePPS_Entry) (
3640     ChangePPS_In* in
3641 );
3642
3643
3644 typedef const struct
3645 {
3646     ChangePPS_Entry *entry;
3647     UINT16 inSize;
3648     UINT16 outSize;
3649     UINT16 offsetOfTypes;
3650     BYTE types[3];
3651 } ChangePPS_COMMAND_DESCRIPTOR_t;
3652
3653 ChangePPS_COMMAND_DESCRIPTOR_t _ChangePPSData = {
3654     /* entry */ &TPM2_ChangePPS,
3655     /* inSize */ (UINT16) (sizeof(ChangePPS_In)),
3656     /* outSize */ 0,
3657     /* offsetOfTypes */ offsetof(ChangePPS_COMMAND_DESCRIPTOR_t, types),
3658     /* offsets */ // No parameter offsets
3659     /* types */ { TPMI_RH_PLATFORM_H_UNMARSHAL,
3660     END_OF_LIST,

```

```

3661                                     END_OF_LIST}
3662 };
3663
3664 #define _ChangePPSDataAddress (&_ChangePPSData)
3665 #else
3666 #define _ChangePPSDataAddress 0
3667 #endif // CC_ChangePPS
3668
3669 #if CC_ChangeEPS
3670 #include "ChangeEPS_fp.h"
3671
3672 typedef TPM_RC (ChangeEPS_Entry) (
3673     ChangeEPS_In*          in
3674 );
3675
3676
3677 typedef const struct
3678 {
3679     ChangeEPS_Entry        *entry;
3680     UINT16                 inSize;
3681     UINT16                 outSize;
3682     UINT16                 offsetOfTypes;
3683     BYTE                   types[3];
3684 } ChangeEPS_COMMAND_DESCRIPTOR_t;
3685
3686 ChangeEPS_COMMAND_DESCRIPTOR_t _ChangeEPSData = {
3687     /* entry */                &TPM2_ChangeEPS,
3688     /* inSize */               (UINT16) (sizeof(ChangeEPS_In)),
3689     /* outSize */              0,
3690     /* offsetOfTypes */        offsetof(ChangeEPS_COMMAND_DESCRIPTOR_t, types),
3691     /* offsets */              // No parameter offsets
3692     /* types */                {TPMI_RH_PLATFORM_H_UNMARSHAL,
3693                                END_OF_LIST,
3694                                END_OF_LIST}
3695 };
3696
3697 #define _ChangeEPSDataAddress (&_ChangeEPSData)
3698 #else
3699 #define _ChangeEPSDataAddress 0
3700 #endif // CC_ChangeEPS
3701
3702 #if CC_Clear
3703 #include "Clear_fp.h"
3704
3705 typedef TPM_RC (Clear_Entry) (
3706     Clear_In*                in
3707 );
3708
3709
3710 typedef const struct
3711 {
3712     Clear_Entry              *entry;
3713     UINT16                   inSize;
3714     UINT16                   outSize;
3715     UINT16                   offsetOfTypes;
3716     BYTE                     types[3];
3717 } Clear_COMMAND_DESCRIPTOR_t;
3718
3719 Clear_COMMAND_DESCRIPTOR_t _ClearData = {
3720     /* entry */                &TPM2_Clear,
3721     /* inSize */               (UINT16) (sizeof(Clear_In)),
3722     /* outSize */              0,
3723     /* offsetOfTypes */        offsetof(Clear_COMMAND_DESCRIPTOR_t, types),
3724     /* offsets */              // No parameter offsets
3725     /* types */                {TPMI_RH_CLEAR_H_UNMARSHAL,
3726                                END_OF_LIST,

```

```

3727                                     END_OF_LIST}
3728 };
3729
3730 #define _ClearDataAddress (&_ClearData)
3731 #else
3732 #define _ClearDataAddress 0
3733 #endif // CC_Clear
3734
3735 #if CC_ClearControl
3736 #include "ClearControl_fp.h"
3737
3738 typedef TPM_RC (ClearControl_Entry) (
3739     ClearControl_In*      in
3740 );
3741
3742
3743 typedef const struct
3744 {
3745     ClearControl_Entry      *entry;
3746     UINT16                  inSize;
3747     UINT16                  outSize;
3748     UINT16                  offsetOfTypes;
3749     UINT16                  paramOffsets[1];
3750     BYTE                    types[4];
3751 } ClearControl_COMMAND_DESCRIPTOR_t;
3752
3753 ClearControl_COMMAND_DESCRIPTOR_t _ClearControlData = {
3754     /* entry */                &TPM2_ClearControl,
3755     /* inSize */               (UINT16) (sizeof(ClearControl_In)),
3756     /* outSize */              0,
3757     /* offsetOfTypes */        offsetof(ClearControl_COMMAND_DESCRIPTOR_t, types),
3758     /* offsets */              {(UINT16) (offsetof(ClearControl_In, disable))},
3759     /* types */                {TPMI_RH_CLEAR_H_UNMARSHAL,
3760                                TPMI_YES_NO_P_UNMARSHAL,
3761                                END_OF_LIST,
3762                                END_OF_LIST}
3763 };
3764
3765 #define _ClearControlDataAddress (&_ClearControlData)
3766 #else
3767 #define _ClearControlDataAddress 0
3768 #endif // CC_ClearControl
3769
3770 #if CC_HierarchyChangeAuth
3771 #include "HierarchyChangeAuth_fp.h"
3772
3773 typedef TPM_RC (HierarchyChangeAuth_Entry) (
3774     HierarchyChangeAuth_In*  in
3775 );
3776
3777
3778 typedef const struct
3779 {
3780     HierarchyChangeAuth_Entry *entry;
3781     UINT16                    inSize;
3782     UINT16                    outSize;
3783     UINT16                    offsetOfTypes;
3784     UINT16                    paramOffsets[1];
3785     BYTE                      types[4];
3786 } HierarchyChangeAuth_COMMAND_DESCRIPTOR_t;
3787
3788 HierarchyChangeAuth_COMMAND_DESCRIPTOR_t _HierarchyChangeAuthData = {
3789     /* entry */                &TPM2_HierarchyChangeAuth,
3790     /* inSize */               (UINT16) (sizeof(HierarchyChangeAuth_In)),
3791     /* outSize */              0,

```

```

3792     /* offsetOfTypes */           offsetof(HierarchyChangeAuth_COMMAND_DESCRIPTOR_t,
types),
3793     /* offsets */                 {(UINT16) (offsetof(HierarchyChangeAuth_In, newAuth))},
3794     /* types */                   {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
3795                                 TPM2B_AUTH_P_UNMARSHAL,
3796                                 END_OF_LIST,
3797                                 END_OF_LIST}
3798 };
3799
3800 #define _HierarchyChangeAuthDataAddress (&_HierarchyChangeAuthData)
3801 #else
3802 #define _HierarchyChangeAuthDataAddress 0
3803 #endif // CC_HierarchyChangeAuth
3804
3805 #if CC_DictionaryAttackLockReset
3806 #include "DictionaryAttackLockReset_fp.h"
3807
3808 typedef TPM_RC (DictionaryAttackLockReset_Entry) (
3809     DictionaryAttackLockReset_In* in
3810 );
3811
3812 typedef const struct
3813 {
3814     DictionaryAttackLockReset_Entry *entry;
3815     UINT16 inSize;
3816     UINT16 outSize;
3817     UINT16 offsetOfTypes;
3818     BYTE types[3];
3819 } DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t;
3820
3821 DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t _DictionaryAttackLockResetData = {
3822     /* entry */ &TPM2_DictionaryAttackLockReset,
3823     /* inSize */ (UINT16) (sizeof(DictionaryAttackLockReset_In)),
3824     /* outSize */ 0,
3825     /* offsetOfTypes */
3826     offsetof(DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t, types),
3827     /* offsets */ // No parameter offsets
3828     /* types */ {TPMI_RH_LOCKOUT_H_UNMARSHAL,
3829                 END_OF_LIST,
3830                 END_OF_LIST}
3831 };
3832
3833 #define _DictionaryAttackLockResetDataAddress (&_DictionaryAttackLockResetData)
3834 #else
3835 #define _DictionaryAttackLockResetDataAddress 0
3836 #endif // CC_DictionaryAttackLockReset
3837
3838 #if CC_DictionaryAttackParameters
3839 #include "DictionaryAttackParameters_fp.h"
3840
3841 typedef TPM_RC (DictionaryAttackParameters_Entry) (
3842     DictionaryAttackParameters_In* in
3843 );
3844
3845 typedef const struct
3846 {
3847     DictionaryAttackParameters_Entry *entry;
3848     UINT16 inSize;
3849     UINT16 outSize;
3850     UINT16 offsetOfTypes;
3851     UINT16 paramOffsets[3];
3852     BYTE types[6];
3853 } DictionaryAttackParameters_COMMAND_DESCRIPTOR_t;
3854
3855

```

```

3856 DictionaryAttackParameters_COMMAND_DESCRIPTOR_t DictionaryAttackParametersData = {
3857     /* entry */ &TPM2_DictionaryAttackParameters,
3858     /* inSize */ (UINT16) (sizeof(DictionaryAttackParameters_In)),
3859     /* outSize */ 0,
3860     /* offsetOfTypes */
offsetof(DictionaryAttackParameters_COMMAND_DESCRIPTOR_t, types),
3861     /* offsets */ { (UINT16) (offsetof(DictionaryAttackParameters_In,
newMaxTries)),
3862
(UINT16) (offsetof(DictionaryAttackParameters_In,
newRecoveryTime)),
3863
(UINT16) (offsetof(DictionaryAttackParameters_In,
lockoutRecovery))},
3864     /* types */ {TPMI_RH_LOCKOUT_H_UNMARSHAL,
3865
UINT32_P_UNMARSHAL,
3866
UINT32_P_UNMARSHAL,
3867
UINT32_P_UNMARSHAL,
3868
END_OF_LIST,
3869
END_OF_LIST}
3870 };
3871
3872 #define DictionaryAttackParametersDataAddress (&DictionaryAttackParametersData)
3873 #else
3874 #define DictionaryAttackParametersDataAddress 0
3875 #endif // CC_DictionaryAttackParameters
3876
3877 #if CC_PP_Commands
3878 #include "PP_Commands_fp.h"
3879
3880 typedef TPM_RC (PP_Commands_Entry) (
3881     PP_Commands_In* in
3882 );
3883
3884
3885 typedef const struct
3886 {
3887     PP_Commands_Entry *entry;
3888     UINT16 inSize;
3889     UINT16 outSize;
3890     UINT16 offsetOfTypes;
3891     UINT16 paramOffsets[2];
3892     BYTE types[5];
3893 } PP_Commands_COMMAND_DESCRIPTOR_t;
3894
3895 PP_Commands_COMMAND_DESCRIPTOR_t PP_CommandsData = {
3896     /* entry */ &TPM2_PP_Commands,
3897     /* inSize */ (UINT16) (sizeof(PP_Commands_In)),
3898     /* outSize */ 0,
3899     /* offsetOfTypes */ offsetof(PP_Commands_COMMAND_DESCRIPTOR_t, types),
3900     /* offsets */ { (UINT16) (offsetof(PP_Commands_In, setList)),
3901
(UINT16) (offsetof(PP_Commands_In, clearList))},
3902     /* types */ {TPMI_RH_PLATFORM_H_UNMARSHAL,
3903
TPML_CC_P_UNMARSHAL,
3904
TPML_CC_P_UNMARSHAL,
3905
END_OF_LIST,
3906
END_OF_LIST}
3907 };
3908
3909 #define PP_CommandsDataAddress (&PP_CommandsData)
3910 #else
3911 #define PP_CommandsDataAddress 0
3912 #endif // CC_PP_Commands
3913
3914 #if CC_SetAlgorithmSet
3915 #include "SetAlgorithmSet_fp.h"
3916
3917 typedef TPM_RC (SetAlgorithmSet_Entry) (

```



```

3918     SetAlgorithmSet_In*      in
3919 );
3920
3921
3922 typedef const struct
3923 {
3924     SetAlgorithmSet_Entry    *entry;
3925     UINT16                   inSize;
3926     UINT16                   outSize;
3927     UINT16                   offsetOfTypes;
3928     UINT16                   paramOffsets[1];
3929     BYTE                     types[4];
3930 } SetAlgorithmSet_COMMAND_DESCRIPTOR_t;
3931
3932 SetAlgorithmSet_COMMAND_DESCRIPTOR_t _SetAlgorithmSetData = {
3933     /* entry */                &TPM2_SetAlgorithmSet,
3934     /* inSize */               (UINT16) (sizeof(SetAlgorithmSet_In)),
3935     /* outSize */              0,
3936     /* offsetOfTypes */        offsetof(SetAlgorithmSet_COMMAND_DESCRIPTOR_t, types),
3937     /* offsets */              {(UINT16) (offsetof(SetAlgorithmSet_In,
3938 algorithmSet))},
3939     /* types */                {TPMI_RH_PLATFORM_H_UNMARSHAL,
3940                                UINT32_P_UNMARSHAL,
3941                                END_OF_LIST,
3942                                END_OF_LIST}
3943 };
3944
3945 #define _SetAlgorithmSetDataAddress (&_SetAlgorithmSetData)
3946 #else
3947 #define _SetAlgorithmSetDataAddress 0
3948 #endif // CC_SetAlgorithmSet
3949
3950 #if CC_FieldUpgradeStart
3951 #include "FieldUpgradeStart_fp.h"
3952
3953 typedef TPM_RC (FieldUpgradeStart_Entry) (
3954     FieldUpgradeStart_In*      in
3955 );
3956
3957 typedef const struct
3958 {
3959     FieldUpgradeStart_Entry    *entry;
3960     UINT16                   inSize;
3961     UINT16                   outSize;
3962     UINT16                   offsetOfTypes;
3963     UINT16                   paramOffsets[3];
3964     BYTE                     types[6];
3965 } FieldUpgradeStart_COMMAND_DESCRIPTOR_t;
3966
3967 FieldUpgradeStart_COMMAND_DESCRIPTOR_t _FieldUpgradeStartData = {
3968     /* entry */                &TPM2_FieldUpgradeStart,
3969     /* inSize */               (UINT16) (sizeof(FieldUpgradeStart_In)),
3970     /* outSize */              0,
3971     /* offsetOfTypes */        offsetof(FieldUpgradeStart_COMMAND_DESCRIPTOR_t,
3972 types),
3973     /* offsets */              {(UINT16) (offsetof(FieldUpgradeStart_In, keyHandle)),
3974                                (UINT16) (offsetof(FieldUpgradeStart_In, fuDigest)),
3975                                (UINT16) (offsetof(FieldUpgradeStart_In,
3976 manifestSignature))},
3977     /* types */                {TPMI_RH_PLATFORM_H_UNMARSHAL,
3978                                TPMI_DH_OBJECT_H_UNMARSHAL,
3979                                TPM2B_DIGEST_P_UNMARSHAL,
3980                                TPMT_SIGNATURE_P_UNMARSHAL,
3981                                END_OF_LIST,
3982                                END_OF_LIST}

```

```

3981 };
3982
3983 #define _FieldUpgradeStartDataAddress (&_FieldUpgradeStartData)
3984 #else
3985 #define _FieldUpgradeStartDataAddress 0
3986 #endif // CC_FieldUpgradeStart
3987
3988 #if CC_FieldUpgradeData
3989 #include "FieldUpgradeData_fp.h"
3990
3991 typedef TPM_RC (FieldUpgradeData_Entry) (
3992     FieldUpgradeData_In*      in,
3993     FieldUpgradeData_Out*     out
3994 );
3995
3996
3997 typedef const struct
3998 {
3999     FieldUpgradeData_Entry      *entry;
4000     UINT16                      inSize;
4001     UINT16                      outSize;
4002     UINT16                      offsetOfTypes;
4003     UINT16                      paramOffsets[1];
4004     BYTE                        types[5];
4005 } FieldUpgradeData_COMMAND_DESCRIPTOR_t;
4006
4007 FieldUpgradeData_COMMAND_DESCRIPTOR_t _FieldUpgradeDataData = {
4008     /* entry */ &TPM2_FieldUpgradeData,
4009     /* inSize */ (UINT16) (sizeof(FieldUpgradeData_In)),
4010     /* outSize */ (UINT16) (sizeof(FieldUpgradeData_Out)),
4011     /* offsetOfTypes */ offsetof(FieldUpgradeData_COMMAND_DESCRIPTOR_t,
4012 types),
4013     /* offsets */ { (UINT16) (offsetof(FieldUpgradeData_Out,
4014 firstDigest))},
4015     /* types */ {TPM2B_MAX_BUFFER_P_UNMARSHAL,
4016 END_OF_LIST,
4017 TPMT_HA_P_MARSHAL,
4018 TPMT_HA_P_MARSHAL,
4019 END_OF_LIST}
4020 };
4021
4022 #define _FieldUpgradeDataDataAddress (&_FieldUpgradeDataData)
4023 #else
4024 #define _FieldUpgradeDataDataAddress 0
4025 #endif // CC_FieldUpgradeData
4026
4027 #if CC_FirmwareRead
4028 #include "FirmwareRead_fp.h"
4029
4030 typedef TPM_RC (FirmwareRead_Entry) (
4031     FirmwareRead_In*      in,
4032     FirmwareRead_Out*     out
4033 );
4034
4035
4036 typedef const struct
4037 {
4038     FirmwareRead_Entry      *entry;
4039     UINT16                      inSize;
4040     UINT16                      outSize;
4041     UINT16                      offsetOfTypes;
4042     BYTE                        types[4];
4043 } FirmwareRead_COMMAND_DESCRIPTOR_t;
4044
4045 FirmwareRead_COMMAND_DESCRIPTOR_t _FirmwareReadData = {
4046     /* entry */ &TPM2_FirmwareRead,

```

```

4045     /* inSize      */      (UINT16) (sizeof(FirmwareRead_In)),
4046     /* outSize     */      (UINT16) (sizeof(FirmwareRead_Out)),
4047     /* offsetOfTypes */      offsetof(FirmwareRead_COMMAND_DESCRIPTOR_t, types),
4048     /* offsets      */      // No parameter offsets
4049     /* types        */      {UINT32_P_UNMARSHAL,
4050                             END_OF_LIST,
4051                             TPM2B_MAX_BUFFER_P_MARSHAL,
4052                             END_OF_LIST}
4053 };
4054
4055 #define _FirmwareReadDataAddress (&_FirmwareReadData)
4056 #else
4057 #define _FirmwareReadDataAddress 0
4058 #endif // CC_FirmwareRead
4059
4060 #if CC_ContextSave
4061 #include "ContextSave_fp.h"
4062
4063 typedef TPM_RC (ContextSave_Entry) (
4064     ContextSave_In*      in,
4065     ContextSave_Out*     out
4066 );
4067
4068
4069 typedef const struct
4070 {
4071     ContextSave_Entry      *entry;
4072     UINT16                 inSize;
4073     UINT16                 outSize;
4074     UINT16                 offsetOfTypes;
4075     BYTE                   types[4];
4076 } ContextSave_COMMAND_DESCRIPTOR_t;
4077
4078 ContextSave_COMMAND_DESCRIPTOR_t _ContextSaveData = {
4079     /* entry      */      &TPM2_ContextSave,
4080     /* inSize     */      (UINT16) (sizeof(ContextSave_In)),
4081     /* outSize    */      (UINT16) (sizeof(ContextSave_Out)),
4082     /* offsetOfTypes */      offsetof(ContextSave_COMMAND_DESCRIPTOR_t, types),
4083     /* offsets     */      // No parameter offsets
4084     /* types      */      {TPMI_DH_CONTEXT_H_UNMARSHAL,
4085                             END_OF_LIST,
4086                             TPMS_CONTEXT_P_MARSHAL,
4087                             END_OF_LIST}
4088 };
4089
4090 #define _ContextSaveDataAddress (&_ContextSaveData)
4091 #else
4092 #define _ContextSaveDataAddress 0
4093 #endif // CC_ContextSave
4094
4095 #if CC_ContextLoad
4096 #include "ContextLoad_fp.h"
4097
4098 typedef TPM_RC (ContextLoad_Entry) (
4099     ContextLoad_In*      in,
4100     ContextLoad_Out*     out
4101 );
4102
4103
4104 typedef const struct
4105 {
4106     ContextLoad_Entry      *entry;
4107     UINT16                 inSize;
4108     UINT16                 outSize;
4109     UINT16                 offsetOfTypes;
4110     BYTE                   types[4];

```

```

4111 } ContextLoad_COMMAND_DESCRIPTOR_t;
4112
4113 ContextLoad_COMMAND_DESCRIPTOR_t _ContextLoadData = {
4114     /* entry */ &TPM2_ContextLoad,
4115     /* inSize */ (UINT16) (sizeof(ContextLoad_In)),
4116     /* outSize */ (UINT16) (sizeof(ContextLoad_Out)),
4117     /* offsetOfTypes */ offsetof(ContextLoad_COMMAND_DESCRIPTOR_t, types),
4118     /* offsets */ // No parameter offsets
4119     /* types */ {TPMS_CONTEXT_P_UNMARSHAL,
4120                 END_OF_LIST,
4121                 TPMI_DH_CONTEXT_H_MARSHAL,
4122                 END_OF_LIST};
4123 };
4124
4125 #define _ContextLoadDataAddress (&_ContextLoadData)
4126 #else
4127 #define _ContextLoadDataAddress 0
4128 #endif // CC_ContextLoad
4129
4130 #if CC_FlushContext
4131 #include "FlushContext_fp.h"
4132
4133 typedef TPM_RC (FlushContext_Entry) (
4134     FlushContext_In* in
4135 );
4136
4137 typedef const struct
4138 {
4139     FlushContext_Entry *entry;
4140     UINT16 inSize;
4141     UINT16 outSize;
4142     UINT16 offsetOfTypes;
4143     BYTE types[3];
4144 } FlushContext_COMMAND_DESCRIPTOR_t;
4145
4146 FlushContext_COMMAND_DESCRIPTOR_t _FlushContextData = {
4147     /* entry */ &TPM2_FlushContext,
4148     /* inSize */ (UINT16) (sizeof(FlushContext_In)),
4149     /* outSize */ 0,
4150     /* offsetOfTypes */ offsetof(FlushContext_COMMAND_DESCRIPTOR_t, types),
4151     /* offsets */ // No parameter offsets
4152     /* types */ {TPMI_DH_CONTEXT_P_UNMARSHAL,
4153                 END_OF_LIST,
4154                 END_OF_LIST};
4155 };
4156
4157 #define _FlushContextDataAddress (&_FlushContextData)
4158 #else
4159 #define _FlushContextDataAddress 0
4160 #endif // CC_FlushContext
4161
4162 #if CC_EvictControl
4163 #include "EvictControl_fp.h"
4164
4165 typedef TPM_RC (EvictControl_Entry) (
4166     EvictControl_In* in
4167 );
4168
4169 typedef const struct
4170 {
4171     EvictControl_Entry *entry;
4172     UINT16 inSize;
4173     UINT16 outSize;
4174     UINT16 offsetOfTypes;

```

```

4177     UINT16                                paramOffsets[2];
4178     BYTE                                  types[5];
4179 } EvictControl_COMMAND_DESCRIPTOR_t;
4180
4181 EvictControl_COMMAND_DESCRIPTOR_t _EvictControlData = {
4182     /* entry */                            &TPM2_EvictControl,
4183     /* inSize */                          (UINT16) (sizeof(EvictControl_In)),
4184     /* outSize */                          0,
4185     /* offsetOfTypes */                   offsetof(EvictControl_COMMAND_DESCRIPTOR_t, types),
4186     /* offsets */                         {(UINT16) (offsetof(EvictControl_In, objectHandle)),
4187                                           (UINT16) (offsetof(EvictControl_In,
4188 persistentHandle)))},
4189     /* types */                           {TPMI_RH_PROVISION_H_UNMARSHAL,
4190                                           TPMI_DH_OBJECT_H_UNMARSHAL,
4191                                           TPMI_DH_PERSISTENT_P_UNMARSHAL,
4192                                           END_OF_LIST,
4193                                           END_OF_LIST}
4194 };
4195
4196 #define _EvictControlDataAddress (&_EvictControlData)
4197 #else
4198 #define _EvictControlDataAddress 0
4199 #endif // CC_EvictControl
4200
4201 #if CC_ReadClock
4202 #include "ReadClock_fp.h"
4203
4204 typedef TPM_RC (ReadClock_Entry) (
4205     ReadClock_Out* out
4206 );
4207
4208 typedef const struct
4209 {
4210     ReadClock_Entry *entry;
4211     UINT16 inSize;
4212     UINT16 outSize;
4213     UINT16 offsetOfTypes;
4214     BYTE types[3];
4215 } ReadClock_COMMAND_DESCRIPTOR_t;
4216
4217 ReadClock_COMMAND_DESCRIPTOR_t _ReadClockData = {
4218     /* entry */                            &TPM2_ReadClock,
4219     /* inSize */                          0,
4220     /* outSize */                         (UINT16) (sizeof(ReadClock_Out)),
4221     /* offsetOfTypes */                   offsetof(ReadClock_COMMAND_DESCRIPTOR_t, types),
4222     /* offsets */                         // No parameter offsets
4223     /* types */                           {END_OF_LIST,
4224                                           TPMS_TIME_INFO_P_MARSHAL,
4225                                           END_OF_LIST}
4226 };
4227
4228 #define _ReadClockDataAddress (&_ReadClockData)
4229 #else
4230 #define _ReadClockDataAddress 0
4231 #endif // CC_ReadClock
4232
4233 #if CC_ClockSet
4234 #include "ClockSet_fp.h"
4235
4236 typedef TPM_RC (ClockSet_Entry) (
4237     ClockSet_In* in
4238 );
4239
4240
4241 typedef const struct

```

```

4242 {
4243     ClockSet_Entry          *entry;
4244     UINT16                  inSize;
4245     UINT16                  outSize;
4246     UINT16                  offsetOfTypes;
4247     UINT16                  paramOffsets[1];
4248     BYTE                    types[4];
4249 } ClockSet_COMMAND_DESCRIPTOR_t;
4250
4251 ClockSet_COMMAND_DESCRIPTOR_t _ClockSetData = {
4252     /* entry          */      &TPM2_ClockSet,
4253     /* inSize         */      (UINT16) (sizeof(ClockSet_In)),
4254     /* outSize        */      0,
4255     /* offsetOfTypes  */      offsetof(ClockSet_COMMAND_DESCRIPTOR_t, types),
4256     /* offsets        */      {(UINT16) (offsetof(ClockSet_In, newTime))},
4257     /* types          */      {TPMI_RH_PROVISION_H_UNMARSHAL,
4258                             UINT64_P_UNMARSHAL,
4259                             END_OF_LIST,
4260                             END_OF_LIST};
4261 };
4262
4263 #define _ClockSetDataAddress (&_ClockSetData)
4264 #else
4265 #define _ClockSetDataAddress 0
4266 #endif // CC_ClockSet
4267
4268 #if CC_ClockRateAdjust
4269 #include "ClockRateAdjust_fp.h"
4270
4271 typedef TPM_RC (ClockRateAdjust_Entry) (
4272     ClockRateAdjust_In*      in
4273 );
4274
4275 typedef const struct
4276 {
4277     ClockRateAdjust_Entry    *entry;
4278     UINT16                   inSize;
4279     UINT16                   outSize;
4280     UINT16                   offsetOfTypes;
4281     UINT16                   paramOffsets[1];
4282     BYTE                     types[4];
4283 } ClockRateAdjust_COMMAND_DESCRIPTOR_t;
4284
4285 ClockRateAdjust_COMMAND_DESCRIPTOR_t _ClockRateAdjustData = {
4286     /* entry          */      &TPM2_ClockRateAdjust,
4287     /* inSize         */      (UINT16) (sizeof(ClockRateAdjust_In)),
4288     /* outSize        */      0,
4289     /* offsetOfTypes  */      offsetof(ClockRateAdjust_COMMAND_DESCRIPTOR_t, types),
4290     /* offsets        */      {(UINT16) (offsetof(ClockRateAdjust_In, rateAdjust))},
4291     /* types          */      {TPMI_RH_PROVISION_H_UNMARSHAL,
4292                             TPM_CLOCK_ADJUST_P_UNMARSHAL,
4293                             END_OF_LIST,
4294                             END_OF_LIST};
4295 };
4296
4297 #define _ClockRateAdjustDataAddress (&_ClockRateAdjustData)
4298 #else
4299 #define _ClockRateAdjustDataAddress 0
4300 #endif // CC_ClockRateAdjust
4301
4302 #if CC_GetCapability
4303 #include "GetCapability_fp.h"
4304
4305 typedef TPM_RC (GetCapability_Entry) (
4306     GetCapability_In*        in,

```



```

4308     GetCapability_Out*      out
4309 );
4310
4311 typedef const struct
4312 {
4313     GetCapability_Entry      *entry;
4314     UINT16                   inSize;
4315     UINT16                   outSize;
4316     UINT16                   offsetOfTypes;
4317     UINT16                   paramOffsets[3];
4318     BYTE                     types[7];
4319 } GetCapability_COMMAND_DESCRIPTOR_t;
4320
4321 GetCapability_COMMAND_DESCRIPTOR_t _GetCapabilityData = {
4322     /* entry */                &TPM2_GetCapability,
4323     /* inSize */               (UINT16) (sizeof(GetCapability_In)),
4324     /* outSize */              (UINT16) (sizeof(GetCapability_Out)),
4325     /* offsetOfTypes */        offsetof(GetCapability_COMMAND_DESCRIPTOR_t, types),
4326     /* offsets */              { (UINT16) (offsetof(GetCapability_In, property)),
4327                                (UINT16) (offsetof(GetCapability_In, propertyCount)),
4328                                (UINT16) (offsetof(GetCapability_Out,
4329 capabilityData)) },
4330     /* types */                { TPM_CAP_P_UNMARSHAL,
4331                                UINT32_P_UNMARSHAL,
4332                                UINT32_P_UNMARSHAL,
4333                                END_OF_LIST,
4334                                TPMI_YES_NO_P_MARSHAL,
4335                                TPMS_CAPABILITY_DATA_P_MARSHAL,
4336                                END_OF_LIST }
4337 };
4338
4339 #define _GetCapabilityDataAddress (&_GetCapabilityData)
4340 #else
4341 #define _GetCapabilityDataAddress 0
4342 #endif // CC_GetCapability
4343
4344 #if CC_TestParms
4345 #include "TestParms_fp.h"
4346
4347 typedef TPM_RC (TestParms_Entry) (
4348     TestParms_In*            in
4349 );
4350
4351 typedef const struct
4352 {
4353     TestParms_Entry          *entry;
4354     UINT16                   inSize;
4355     UINT16                   outSize;
4356     UINT16                   offsetOfTypes;
4357     BYTE                     types[3];
4358 } TestParms_COMMAND_DESCRIPTOR_t;
4359
4360 TestParms_COMMAND_DESCRIPTOR_t _TestParmsData = {
4361     /* entry */                &TPM2_TestParms,
4362     /* inSize */               (UINT16) (sizeof(TestParms_In)),
4363     /* outSize */              0,
4364     /* offsetOfTypes */        offsetof(TestParms_COMMAND_DESCRIPTOR_t, types),
4365     /* offsets */              // No parameter offsets
4366     /* types */                { TPMT_PUBLIC_PARMS_P_UNMARSHAL,
4367                                END_OF_LIST,
4368                                END_OF_LIST }
4369 };
4370
4371 #define _TestParmsDataAddress (&_TestParmsData)
4372

```

```

4373 #else
4374 #define _TestParmsDataAddress 0
4375 #endif // CC_TestParms
4376
4377 #if CC_NV_DefineSpace
4378 #include "NV_DefineSpace_fp.h"
4379
4380 typedef TPM_RC (NV_DefineSpace_Entry) (
4381     NV_DefineSpace_In* in
4382 );
4383
4384
4385 typedef const struct
4386 {
4387     NV_DefineSpace_Entry *entry;
4388     UINT16 inSize;
4389     UINT16 outSize;
4390     UINT16 offsetOfTypes;
4391     UINT16 paramOffsets[2];
4392     BYTE types[5];
4393 } NV_DefineSpace_COMMAND_DESCRIPTOR_t;
4394
4395 NV_DefineSpace_COMMAND_DESCRIPTOR_t _NV_DefineSpaceData = {
4396     /* entry */ &TPM2_NV_DefineSpace,
4397     /* inSize */ (UINT16) (sizeof(NV_DefineSpace_In)),
4398     /* outSize */ 0,
4399     /* offsetOfTypes */ offsetof(NV_DefineSpace_COMMAND_DESCRIPTOR_t, types),
4400     /* offsets */ { (UINT16) (offsetof(NV_DefineSpace_In, auth)),
4401                   (UINT16) (offsetof(NV_DefineSpace_In, publicInfo)) },
4402     /* types */ { TPMI_RH_PROVISION_H_UNMARSHAL,
4403                 TPM2B_AUTH_P_UNMARSHAL,
4404                 TPM2B_NV_PUBLIC_P_UNMARSHAL,
4405                 END_OF_LIST,
4406                 END_OF_LIST }
4407 };
4408
4409 #define _NV_DefineSpaceDataAddress (&_NV_DefineSpaceData)
4410 #else
4411 #define _NV_DefineSpaceDataAddress 0
4412 #endif // CC_NV_DefineSpace
4413
4414 #if CC_NV_UndefineSpace
4415 #include "NV_UndefineSpace_fp.h"
4416
4417 typedef TPM_RC (NV_UndefineSpace_Entry) (
4418     NV_UndefineSpace_In* in
4419 );
4420
4421
4422 typedef const struct
4423 {
4424     NV_UndefineSpace_Entry *entry;
4425     UINT16 inSize;
4426     UINT16 outSize;
4427     UINT16 offsetOfTypes;
4428     UINT16 paramOffsets[1];
4429     BYTE types[4];
4430 } NV_UndefineSpace_COMMAND_DESCRIPTOR_t;
4431
4432 NV_UndefineSpace_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceData = {
4433     /* entry */ &TPM2_NV_UndefineSpace,
4434     /* inSize */ (UINT16) (sizeof(NV_UndefineSpace_In)),
4435     /* outSize */ 0,
4436     /* offsetOfTypes */ offsetof(NV_UndefineSpace_COMMAND_DESCRIPTOR_t,
4437     types),
4437     /* offsets */ { (UINT16) (offsetof(NV_UndefineSpace_In, nvIndex)) },

```

```

4438     /* types */ {TPMI_RH_PROVISION_H_UNMARSHAL,
4439                  TPMI_RH_NV_DEFINED_INDEX_H_UNMARSHAL,
4440                  END_OF_LIST,
4441                  END_OF_LIST}
4442 };
4443
4444 #define _NV_UndefineSpaceDataAddress (&_NV_UndefineSpaceData)
4445 #else
4446 #define _NV_UndefineSpaceDataAddress 0
4447 #endif // CC_NV_UndefineSpace
4448
4449 #if CC_NV_UndefineSpaceSpecial
4450 #include "NV_UndefineSpaceSpecial_fp.h"
4451
4452 typedef TPM_RC (NV_UndefineSpaceSpecial_Entry) (
4453     NV_UndefineSpaceSpecial_In* in
4454 );
4455
4456 typedef const struct
4457 {
4458     NV_UndefineSpaceSpecial_Entry *entry;
4459     UINT16 inSize;
4460     UINT16 outSize;
4461     UINT16 offsetOfTypes;
4462     UINT16 paramOffsets[1];
4463     BYTE types[4];
4464 } NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t;
4465
4466 NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceSpecialData = {
4467     /* entry */ &TPM2_NV_UndefineSpaceSpecial,
4468     /* inSize */ (UINT16)(sizeof(NV_UndefineSpaceSpecial_In)),
4469     /* outSize */ 0,
4470     /* offsetOfTypes */ offsetof(NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t,
4471     types),
4472     /* offsets */ { (UINT16)(offsetof(NV_UndefineSpaceSpecial_In,
4473     platform))},
4474     /* types */ {TPMI_RH_NV_DEFINED_INDEX_H_UNMARSHAL,
4475                  TPMI_RH_PLATFORM_H_UNMARSHAL,
4476                  END_OF_LIST,
4477                  END_OF_LIST}
4478 };
4479
4480 #define _NV_UndefineSpaceSpecialDataAddress (&_NV_UndefineSpaceSpecialData)
4481 #else
4482 #define _NV_UndefineSpaceSpecialDataAddress 0
4483 #endif // CC_NV_UndefineSpaceSpecial
4484
4485 #if CC_NV_ReadPublic
4486 #include "NV_ReadPublic_fp.h"
4487
4488 typedef TPM_RC (NV_ReadPublic_Entry) (
4489     NV_ReadPublic_In* in,
4490     NV_ReadPublic_Out* out
4491 );
4492
4493 typedef const struct
4494 {
4495     NV_ReadPublic_Entry *entry;
4496     UINT16 inSize;
4497     UINT16 outSize;
4498     UINT16 offsetOfTypes;
4499     UINT16 paramOffsets[1];
4500     BYTE types[5];
4501 } NV_ReadPublic_COMMAND_DESCRIPTOR_t;

```

```

4502
4503 NV_ReadPublic_COMMAND_DESCRIPTOR_t NV_ReadPublicData = {
4504     /* entry */ &TPM2_NV_ReadPublic,
4505     /* inSize */ (UINT16) (sizeof(NV_ReadPublic_In)),
4506     /* outSize */ (UINT16) (sizeof(NV_ReadPublic_Out)),
4507     /* offsetOfTypes */ offsetof(NV_ReadPublic_COMMAND_DESCRIPTOR_t, types),
4508     /* offsets */ { (UINT16) (offsetof(NV_ReadPublic_Out, nvName)) },
4509     /* types */ { TPMI_RH_NV_INDEX_H_UNMARSHAL,
4510                 END_OF_LIST,
4511                 TPM2B_NV_PUBLIC_P_MARSHAL,
4512                 TPM2B_NAME_P_MARSHAL,
4513                 END_OF_LIST };
4514 };
4515
4516 #define NV_ReadPublicDataAddress (&NV_ReadPublicData)
4517 #else
4518 #define NV_ReadPublicDataAddress 0
4519 #endif // CC_NV_ReadPublic
4520
4521 #if CC_NV_Write
4522 #include "NV_Write_fp.h"
4523
4524 typedef TPM_RC (NV_Write_Entry) (
4525     NV_Write_In* in
4526 );
4527
4528 typedef const struct
4529 {
4530     NV_Write_Entry *entry;
4531     UINT16 inSize;
4532     UINT16 outSize;
4533     UINT16 offsetOfTypes;
4534     UINT16 paramOffsets[3];
4535     BYTE types[6];
4536 } NV_Write_COMMAND_DESCRIPTOR_t;
4537
4538 NV_Write_COMMAND_DESCRIPTOR_t NV_WriteData = {
4539     /* entry */ &TPM2_NV_Write,
4540     /* inSize */ (UINT16) (sizeof(NV_Write_In)),
4541     /* outSize */ 0,
4542     /* offsetOfTypes */ offsetof(NV_Write_COMMAND_DESCRIPTOR_t, types),
4543     /* offsets */ { (UINT16) (offsetof(NV_Write_In, nvIndex)),
4544                   (UINT16) (offsetof(NV_Write_In, data)),
4545                   (UINT16) (offsetof(NV_Write_In, offset)) },
4546     /* types */ { TPMI_RH_NV_AUTH_H_UNMARSHAL,
4547                  TPMI_RH_NV_INDEX_H_UNMARSHAL,
4548                  TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
4549                  UINT16_P_UNMARSHAL,
4550                  END_OF_LIST,
4551                  END_OF_LIST };
4552 };
4553
4554 #define NV_WriteDataAddress (&NV_WriteData)
4555 #else
4556 #define NV_WriteDataAddress 0
4557 #endif // CC_NV_Write
4558
4559 #if CC_NV_Increment
4560 #include "NV_Increment_fp.h"
4561
4562 typedef TPM_RC (NV_Increment_Entry) (
4563     NV_Increment_In* in
4564 );
4565
4566
4567

```

```

4568 typedef const struct
4569 {
4570     NV_Increment_Entry      *entry;
4571     UINT16                   inSize;
4572     UINT16                   outSize;
4573     UINT16                   offsetOfTypes;
4574     UINT16                   paramOffsets[1];
4575     BYTE                     types[4];
4576 } NV_Increment_COMMAND_DESCRIPTOR_t;
4577
4578 NV_Increment_COMMAND_DESCRIPTOR_t NV_IncrementData = {
4579     /* entry */           &TPM2_NV_Increment,
4580     /* inSize */          (UINT16) (sizeof(NV_Increment_In)),
4581     /* outSize */         0,
4582     /* offsetOfTypes */   offsetof(NV_Increment_COMMAND_DESCRIPTOR_t, types),
4583     /* offsets */          {(UINT16) (offsetof(NV_Increment_In, nvIndex))},
4584     /* types */           {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4585                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
4586                           END_OF_LIST,
4587                           END_OF_LIST};
4588 };
4589
4590 #define _NV_IncrementDataAddress (&_NV_IncrementData)
4591 #else
4592 #define _NV_IncrementDataAddress 0
4593 #endif // CC_NV_Increment
4594
4595 #if CC_NV_Extend
4596 #include "NV_Extend_fp.h"
4597
4598 typedef TPM_RC (NV_Extend_Entry) (
4599     NV_Extend_In*          in
4600 );
4601
4602 typedef const struct
4603 {
4604     NV_Extend_Entry      *entry;
4605     UINT16               inSize;
4606     UINT16               outSize;
4607     UINT16               offsetOfTypes;
4608     UINT16               paramOffsets[2];
4609     BYTE                 types[5];
4610 } NV_Extend_COMMAND_DESCRIPTOR_t;
4611
4612 NV_Extend_COMMAND_DESCRIPTOR_t NV_ExtendData = {
4613     /* entry */           &TPM2_NV_Extend,
4614     /* inSize */          (UINT16) (sizeof(NV_Extend_In)),
4615     /* outSize */         0,
4616     /* offsetOfTypes */   offsetof(NV_Extend_COMMAND_DESCRIPTOR_t, types),
4617     /* offsets */          {(UINT16) (offsetof(NV_Extend_In, nvIndex))},
4618                          {(UINT16) (offsetof(NV_Extend_In, data))},
4619     /* types */           {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4620                           TPMI_RH_NV_INDEX_H_UNMARSHAL,
4621                           TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
4622                           END_OF_LIST,
4623                           END_OF_LIST};
4624 };
4625
4626 #define _NV_ExtendDataAddress (&_NV_ExtendData)
4627 #else
4628 #define _NV_ExtendDataAddress 0
4629 #endif // CC_NV_Extend
4630
4631 #if CC_NV_SetBits
4632 #include "NV_SetBits_fp.h"

```

```

4634
4635 typedef TPM_RC (NV_SetBits_Entry) (
4636     NV_SetBits_In*          in
4637 );
4638
4639
4640 typedef const struct
4641 {
4642     NV_SetBits_Entry          *entry;
4643     UINT16                    inSize;
4644     UINT16                    outSize;
4645     UINT16                    offsetOfTypes;
4646     UINT16                    paramOffsets[2];
4647     BYTE                      types[5];
4648 } NV_SetBits_COMMAND_DESCRIPTOR_t;
4649
4650 NV_SetBits_COMMAND_DESCRIPTOR_t _NV_SetBitsData = {
4651     /* entry          */      &TPM2_NV_SetBits,
4652     /* inSize         */      (UINT16) (sizeof(NV_SetBits_In)),
4653     /* outSize        */      0,
4654     /* offsetOfTypes  */      offsetof(NV_SetBits_COMMAND_DESCRIPTOR_t, types),
4655     /* offsets        */      {(UINT16) (offsetof(NV_SetBits_In, nvIndex)),
4656                               (UINT16) (offsetof(NV_SetBits_In, bits))},
4657     /* types          */      {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4658                               TPMI_RH_NV_INDEX_H_UNMARSHAL,
4659                               UINT64_P_UNMARSHAL,
4660                               END_OF_LIST,
4661                               END_OF_LIST}
4662 };
4663
4664 #define _NV_SetBitsDataAddress (&_NV_SetBitsData)
4665 #else
4666 #define _NV_SetBitsDataAddress 0
4667 #endif // CC_NV_SetBits
4668
4669 #if          CC_NV_WriteLock
4670 #include     "NV_WriteLock_fp.h"
4671
4672 typedef TPM_RC (NV_WriteLock_Entry) (
4673     NV_WriteLock_In*          in
4674 );
4675
4676
4677 typedef const struct
4678 {
4679     NV_WriteLock_Entry          *entry;
4680     UINT16                    inSize;
4681     UINT16                    outSize;
4682     UINT16                    offsetOfTypes;
4683     UINT16                    paramOffsets[1];
4684     BYTE                      types[4];
4685 } NV_WriteLock_COMMAND_DESCRIPTOR_t;
4686
4687 NV_WriteLock_COMMAND_DESCRIPTOR_t _NV_WriteLockData = {
4688     /* entry          */      &TPM2_NV_WriteLock,
4689     /* inSize         */      (UINT16) (sizeof(NV_WriteLock_In)),
4690     /* outSize        */      0,
4691     /* offsetOfTypes  */      offsetof(NV_WriteLock_COMMAND_DESCRIPTOR_t, types),
4692     /* offsets        */      {(UINT16) (offsetof(NV_WriteLock_In, nvIndex))},
4693     /* types          */      {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4694                               TPMI_RH_NV_INDEX_H_UNMARSHAL,
4695                               END_OF_LIST,
4696                               END_OF_LIST}
4697 };
4698
4699 #define _NV_WriteLockDataAddress (&_NV_WriteLockData)

```



```

4700 #else
4701 #define _NV_WriteLockDataAddress 0
4702 #endif // CC_NV_WriteLock
4703
4704 #if CC_NV_GlobalWriteLock
4705 #include "NV_GlobalWriteLock_fp.h"
4706
4707 typedef TPM_RC (NV_GlobalWriteLock_Entry) (
4708     NV_GlobalWriteLock_In* in
4709 );
4710
4711
4712 typedef const struct
4713 {
4714     NV_GlobalWriteLock_Entry *entry;
4715     UINT16 inSize;
4716     UINT16 outSize;
4717     UINT16 offsetOfTypes;
4718     BYTE types[3];
4719 } NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t;
4720
4721 NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t _NV_GlobalWriteLockData = {
4722     /* entry */ &TPM2_NV_GlobalWriteLock,
4723     /* inSize */ (UINT16) (sizeof(NV_GlobalWriteLock_In)),
4724     /* outSize */ 0,
4725     /* offsetOfTypes */ offsetof(NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t,
4726     types),
4727     /* offsets */ // No parameter offsets
4728     /* types */ {TPMI_RH_PROVISION_H_UNMARSHAL,
4729     END_OF_LIST,
4730     END_OF_LIST}
4731 };
4732
4733 #define _NV_GlobalWriteLockDataAddress (&_NV_GlobalWriteLockData)
4734 #else
4735 #define _NV_GlobalWriteLockDataAddress 0
4736 #endif // CC_NV_GlobalWriteLock
4737
4738 #if CC_NV_Read
4739 #include "NV_Read_fp.h"
4740
4741 typedef TPM_RC (NV_Read_Entry) (
4742     NV_Read_In* in,
4743     NV_Read_Out* out
4744 );
4745
4746
4747 typedef const struct
4748 {
4749     NV_Read_Entry *entry;
4750     UINT16 inSize;
4751     UINT16 outSize;
4752     UINT16 offsetOfTypes;
4753     UINT16 paramOffsets[3];
4754     BYTE types[7];
4755 } NV_Read_COMMAND_DESCRIPTOR_t;
4756
4757 NV_Read_COMMAND_DESCRIPTOR_t _NV_ReadData = {
4758     /* entry */ &TPM2_NV_Read,
4759     /* inSize */ (UINT16) (sizeof(NV_Read_In)),
4760     /* outSize */ (UINT16) (sizeof(NV_Read_Out)),
4761     /* offsetOfTypes */ offsetof(NV_Read_COMMAND_DESCRIPTOR_t, types),
4762     /* offsets */ { (UINT16) (offsetof(NV_Read_In, nvIndex)),
4763     (UINT16) (offsetof(NV_Read_In, size)),
4764     (UINT16) (offsetof(NV_Read_In, offset))},
4765     /* types */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,

```

```

4765             TPMI_RH_NV_INDEX_H_UNMARSHAL,
4766             UINT16_P_UNMARSHAL,
4767             UINT16_P_UNMARSHAL,
4768             END_OF_LIST,
4769             TPM2B_MAX_NV_BUFFER_P_MARSHAL,
4770             END_OF_LIST};
4771 };
4772
4773 #define _NV_ReadDataAddress (&_NV_ReadData)
4774 #else
4775 #define _NV_ReadDataAddress 0
4776 #endif // CC_NV_Read
4777
4778 #if CC_NV_ReadLock
4779 #include "NV_ReadLock_fp.h"
4780
4781 typedef TPM_RC (NV_ReadLock_Entry) (
4782     NV_ReadLock_In* in
4783 );
4784
4785 typedef const struct
4786 {
4787     NV_ReadLock_Entry *entry;
4788     UINT16 inSize;
4789     UINT16 outSize;
4790     UINT16 offsetOfTypes;
4791     UINT16 paramOffsets[1];
4792     BYTE types[4];
4793 } NV_ReadLock_COMMAND_DESCRIPTOR_t;
4794
4795 NV_ReadLock_COMMAND_DESCRIPTOR_t _NV_ReadLockData = {
4796     /* entry */ &TPM2_NV_ReadLock,
4797     /* inSize */ (UINT16)(sizeof(NV_ReadLock_In)),
4798     /* outSize */ 0,
4799     /* offsetOfTypes */ offsetof(NV_ReadLock_COMMAND_DESCRIPTOR_t, types),
4800     /* offsets */ {(UINT16)(offsetof(NV_ReadLock_In, nvIndex))},
4801     /* types */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4802                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
4803                 END_OF_LIST,
4804                 END_OF_LIST};
4805 };
4806
4807 #define _NV_ReadLockDataAddress (&_NV_ReadLockData)
4808 #else
4809 #define _NV_ReadLockDataAddress 0
4810 #endif // CC_NV_ReadLock
4811
4812 #if CC_NV_ChangeAuth
4813 #include "NV_ChangeAuth_fp.h"
4814
4815 typedef TPM_RC (NV_ChangeAuth_Entry) (
4816     NV_ChangeAuth_In* in
4817 );
4818
4819 typedef const struct
4820 {
4821     NV_ChangeAuth_Entry *entry;
4822     UINT16 inSize;
4823     UINT16 outSize;
4824     UINT16 offsetOfTypes;
4825     UINT16 paramOffsets[1];
4826     BYTE types[4];
4827 } NV_ChangeAuth_COMMAND_DESCRIPTOR_t;
4828
4829
4830

```

```

4831 NV_ChangeAuth_COMMAND_DESCRIPTOR_t _NV_ChangeAuthData = {
4832     /* entry */ &TPM2_NV_ChangeAuth,
4833     /* inSize */ (UINT16) (sizeof(NV_ChangeAuth_In)),
4834     /* outSize */ 0,
4835     /* offsetOfTypes */ offsetof(NV_ChangeAuth_COMMAND_DESCRIPTOR_t, types),
4836     /* offsets */ { (UINT16) (offsetof(NV_ChangeAuth_In, newAuth)) },
4837     /* types */ { TPMI_RH_NV_INDEX_H_UNMARSHAL,
4838                  TPM2B_AUTH_P_UNMARSHAL,
4839                  END_OF_LIST,
4840                  END_OF_LIST };
4841 };
4842
4843 #define _NV_ChangeAuthDataAddress (&_NV_ChangeAuthData)
4844 #else
4845 #define _NV_ChangeAuthDataAddress 0
4846 #endif // CC_NV_ChangeAuth
4847
4848 #if CC_NV_Certify
4849 #include "NV_Certify_fp.h"
4850
4851 typedef TPM_RC (NV_Certify_Entry) (
4852     NV_Certify_In* in,
4853     NV_Certify_Out* out
4854 );
4855
4856 typedef const struct
4857 {
4858     NV_Certify_Entry *entry;
4859     UINT16 inSize;
4860     UINT16 outSize;
4861     UINT16 offsetOfTypes;
4862     UINT16 paramOffsets[7];
4863     BYTE types[11];
4864 } NV_Certify_COMMAND_DESCRIPTOR_t;
4865
4866 NV_Certify_COMMAND_DESCRIPTOR_t _NV_CertifyData = {
4867     /* entry */ &TPM2_NV_Certify,
4868     /* inSize */ (UINT16) (sizeof(NV_Certify_In)),
4869     /* outSize */ (UINT16) (sizeof(NV_Certify_Out)),
4870     /* offsetOfTypes */ offsetof(NV_Certify_COMMAND_DESCRIPTOR_t, types),
4871     /* offsets */ { (UINT16) (offsetof(NV_Certify_In, authHandle)),
4872                   (UINT16) (offsetof(NV_Certify_In, nvIndex)),
4873                   (UINT16) (offsetof(NV_Certify_In, qualifyingData)),
4874                   (UINT16) (offsetof(NV_Certify_In, inScheme)),
4875                   (UINT16) (offsetof(NV_Certify_In, size)),
4876                   (UINT16) (offsetof(NV_Certify_In, offset)),
4877                   (UINT16) (offsetof(NV_Certify_Out, signature)) },
4878     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
4879                  TPMI_RH_NV_AUTH_H_UNMARSHAL,
4880                  TPMI_RH_NV_INDEX_H_UNMARSHAL,
4881                  TPM2B_DATA_P_UNMARSHAL,
4882                  TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
4883                  UINT16_P_UNMARSHAL,
4884                  UINT16_P_UNMARSHAL,
4885                  END_OF_LIST,
4886                  TPM2B_ATTEST_P_UNMARSHAL,
4887                  TPMT_SIGNATURE_P_UNMARSHAL,
4888                  END_OF_LIST };
4889 };
4890
4891 #define _NV_CertifyDataAddress (&_NV_CertifyData)
4892 #else
4893 #define _NV_CertifyDataAddress 0
4894 #endif // CC_NV_Certify
4895
4896

```

```

4897 #if CC_NV_DefineSpace2
4898 #include "NV_DefineSpace2_fp.h"
4899
4900 typedef TPM_RC (NV_DefineSpace2_Entry) (
4901     NV_DefineSpace2_In* in
4902 );
4903
4904
4905 typedef const struct
4906 {
4907     NV_DefineSpace2_Entry *entry;
4908     UINT16 inSize;
4909     UINT16 outSize;
4910     UINT16 offsetOfTypes;
4911     UINT16 paramOffsets[2];
4912     BYTE types[5];
4913 } NV_DefineSpace2_COMMAND_DESCRIPTOR_t;
4914
4915 NV_DefineSpace2_COMMAND_DESCRIPTOR_t _NV_DefineSpace2Data = {
4916     /* entry */ &TPM2_NV_DefineSpace2,
4917     /* inSize */ (UINT16) (sizeof(NV_DefineSpace2_In)),
4918     /* outSize */ 0,
4919     /* offsetOfTypes */ offsetof(NV_DefineSpace2_COMMAND_DESCRIPTOR_t, types),
4920     /* offsets */ { (UINT16) (offsetof(NV_DefineSpace2_In, auth)),
4921                    (UINT16) (offsetof(NV_DefineSpace2_In, publicInfo)) },
4922     /* types */ { TPMI_RH_PROVISION_H_UNMARSHAL,
4923                  TPM2B_AUTH_P_UNMARSHAL,
4924                  TPM2B_NV_PUBLIC_2_P_UNMARSHAL,
4925                  END_OF_LIST,
4926                  END_OF_LIST }
4927 };
4928
4929 #define _NV_DefineSpace2DataAddress (&_NV_DefineSpace2Data)
4930 #else
4931 #define NV_DefineSpace2DataAddress 0
4932 #endif // CC_NV_DefineSpace2
4933
4934 #if CC_NV_ReadPublic2
4935 #include "NV_ReadPublic2_fp.h"
4936
4937 typedef TPM_RC (NV_ReadPublic2_Entry) (
4938     NV_ReadPublic2_In* in,
4939     NV_ReadPublic2_Out* out
4940 );
4941
4942
4943 typedef const struct
4944 {
4945     NV_ReadPublic2_Entry *entry;
4946     UINT16 inSize;
4947     UINT16 outSize;
4948     UINT16 offsetOfTypes;
4949     UINT16 paramOffsets[1];
4950     BYTE types[5];
4951 } NV_ReadPublic2_COMMAND_DESCRIPTOR_t;
4952
4953 NV_ReadPublic2_COMMAND_DESCRIPTOR_t _NV_ReadPublic2Data = {
4954     /* entry */ &TPM2_NV_ReadPublic2,
4955     /* inSize */ (UINT16) (sizeof(NV_ReadPublic2_In)),
4956     /* outSize */ (UINT16) (sizeof(NV_ReadPublic2_Out)),
4957     /* offsetOfTypes */ offsetof(NV_ReadPublic2_COMMAND_DESCRIPTOR_t, types),
4958     /* offsets */ { (UINT16) (offsetof(NV_ReadPublic2_Out, nvName)) },
4959     /* types */ { TPMI_RH_NV_INDEX_H_UNMARSHAL,
4960                  END_OF_LIST,
4961                  TPM2B_NV_PUBLIC_2_P_MARSHAL,
4962                  TPM2B_NAME_P_MARSHAL,

```

```

4963                                     END_OF_LIST}
4964 };
4965
4966 #define _NV_ReadPublic2DataAddress (&_NV_ReadPublic2Data)
4967 #else
4968 #define _NV_ReadPublic2DataAddress 0
4969 #endif // CC_NV_ReadPublic2
4970
4971 #if CC_SetCapability
4972 #include "SetCapability_fp.h"
4973
4974 typedef TPM_RC (SetCapability_Entry) (
4975     SetCapability_In*      in
4976 );
4977
4978 typedef const struct
4979 {
4980     SetCapability_Entry      *entry;
4981     UINT16                   inSize;
4982     UINT16                   outSize;
4983     UINT16                   offsetOfTypes;
4984     UINT16                   paramOffsets[1];
4985     BYTE                     types[4];
4986 } SetCapability_COMMAND_DESCRIPTOR_t;
4987
4988 SetCapability_COMMAND_DESCRIPTOR_t _SetCapabilityData = {
4989     /* entry */                &TPM2_SetCapability,
4990     /* inSize */              (UINT16) (sizeof(SetCapability_In)),
4991     /* outSize */              0,
4992     /* offsetOfTypes */        offsetof(SetCapability_COMMAND_DESCRIPTOR_t, types),
4993     /* offsets */              {(UINT16) (offsetof(SetCapability_In,
4994 setCapabilityData))},
4995     /* types */                {TPMI_RH_HIERARCHY_H_UNMARSHAL,
4996 TPM2B_SET_CAPABILITY_DATA_P_UNMARSHAL,
4997 END_OF_LIST,
4998 END_OF_LIST}
4999 };
5000
5001 #define _SetCapabilityDataAddress (&_SetCapabilityData)
5002 #else
5003 #define _SetCapabilityDataAddress 0
5004 #endif // CC_SetCapability
5005
5006 #if CC_AC_Send
5007 #include "AC_Send_fp.h"
5008
5009 typedef TPM_RC (AC_Send_Entry) (
5010     AC_Send_In*              in,
5011     AC_Send_Out*             out
5012 );
5013
5014 typedef const struct
5015 {
5016     AC_Send_Entry            *entry;
5017     UINT16                   inSize;
5018     UINT16                   outSize;
5019     UINT16                   offsetOfTypes;
5020     UINT16                   paramOffsets[3];
5021     BYTE                     types[7];
5022 } AC_Send_COMMAND_DESCRIPTOR_t;
5023
5024 AC_Send_COMMAND_DESCRIPTOR_t _AC_SendData = {
5025     /* entry */                &TPM2_AC_Send,
5026     /* inSize */              (UINT16) (sizeof(AC_Send_In)),
5027     /* outSize */              (UINT16) (sizeof(AC_Send_Out)),

```

```

5028     /* offsetOfTypes */           offsetof(AC_Send_COMMAND_DESCRIPTOR t, types),
5029     /* offsets */                 {(UINT16) (offsetof(AC_Send_In, authHandle)),
5030                                   (UINT16) (offsetof(AC_Send_In, ac)),
5031                                   (UINT16) (offsetof(AC_Send_In, acDataIn))},
5032     /* types */                   {TPMI_DH_OBJECT_H_UNMARSHAL,
5033                                   TPMI_RH_NV_AUTH_H_UNMARSHAL,
5034                                   TPMI_RH_AC_H_UNMARSHAL,
5035                                   TPM2B_MAX_BUFFER_P_UNMARSHAL,
5036                                   END_OF_LIST,
5037                                   TPMS_AC_OUTPUT_P_MARSHAL,
5038                                   END_OF_LIST};
5039 };
5040
5041 #define _AC_SendDataAddress (&_AC_SendData)
5042 #else
5043 #define _AC_SendDataAddress 0
5044 #endif // CC_AC_Send
5045
5046 #if CC_Policy_AC_SendSelect
5047 #include "Policy_AC_SendSelect_fp.h"
5048
5049 typedef TPM_RC (Policy_AC_SendSelect_Entry) (
5050     Policy_AC_SendSelect_In* in
5051 );
5052
5053
5054 typedef const struct
5055 {
5056     Policy_AC_SendSelect_Entry *entry;
5057     UINT16 inSize;
5058     UINT16 outSize;
5059     UINT16 offsetOfTypes;
5060     UINT16 paramOffsets[4];
5061     BYTE types[7];
5062 } Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t;
5063
5064 Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t _Policy_AC_SendSelectData = {
5065     /* entry */ &TPM2_Policy_AC_SendSelect,
5066     /* inSize */ (UINT16) (sizeof(Policy_AC_SendSelect_In)),
5067     /* outSize */ 0,
5068     /* offsetOfTypes */ offsetof(Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t,
5069     types),
5070     /* offsets */ {(UINT16) (offsetof(Policy_AC_SendSelect_In,
5071     objectName)),
5072                   (UINT16) (offsetof(Policy_AC_SendSelect_In,
5073     authHandleName)),
5074                   (UINT16) (offsetof(Policy_AC_SendSelect_In, acName)),
5075                   (UINT16) (offsetof(Policy_AC_SendSelect_In,
5076     includeObject))},
5077     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
5078                 TPM2B_NAME_P_UNMARSHAL,
5079                 TPM2B_NAME_P_UNMARSHAL,
5080                 TPM2B_NAME_P_UNMARSHAL,
5081                 TPMI_YES_NO_P_UNMARSHAL,
5082                 END_OF_LIST,
5083                 END_OF_LIST};
5084 };
5085
5086 #define _Policy_AC_SendSelectDataAddress (&_Policy_AC_SendSelectData)
5087 #else
5088 #define _Policy_AC_SendSelectDataAddress 0
5089 #endif // CC_Policy_AC_SendSelect
5090
5091 #if CC_ACT_SetTimeout
5092 #include "ACT_SetTimeout_fp.h"
5093

```



```

5090 typedef TPM_RC (ACT_SetTimeout_Entry) (
5091     ACT_SetTimeout_In*      in
5092 );
5093
5094
5095 typedef const struct
5096 {
5097     ACT_SetTimeout_Entry      *entry;
5098     UINT16                    inSize;
5099     UINT16                    outSize;
5100     UINT16                    offsetOfTypes;
5101     UINT16                    paramOffsets[1];
5102     BYTE                      types[4];
5103 } ACT_SetTimeout_COMMAND_DESCRIPTOR_t;
5104
5105 ACT_SetTimeout_COMMAND_DESCRIPTOR_t _ACT_SetTimeoutData = {
5106     /* entry */          &TPM2_ACT_SetTimeout,
5107     /* inSize */         (UINT16) (sizeof(ACT_SetTimeout_In)),
5108     /* outSize */        0,
5109     /* offsetOfTypes */  offsetof(ACT_SetTimeout_COMMAND_DESCRIPTOR_t, types),
5110     /* offsets */        {(UINT16) (offsetof(ACT_SetTimeout_In, startTimeout))},
5111     /* types */          {TPMI_RH_ACT_H_UNMARSHAL,
5112                          UINT32_P_UNMARSHAL,
5113                          END_OF_LIST,
5114                          END_OF_LIST};
5115 };
5116
5117 #define _ACT_SetTimeoutDataAddress (&_ACT_SetTimeoutData)
5118 #else
5119 #define _ACT_SetTimeoutDataAddress 0
5120 #endif // CC_ACT_SetTimeout
5121
5122 #if CC_Vendor_TCG_Test
5123 #include "Vendor_TCG_Test_fp.h"
5124
5125 typedef TPM_RC (Vendor_TCG_Test_Entry) (
5126     Vendor_TCG_Test_In*      in,
5127     Vendor_TCG_Test_Out*     out
5128 );
5129
5130
5131 typedef const struct
5132 {
5133     Vendor_TCG_Test_Entry      *entry;
5134     UINT16                    inSize;
5135     UINT16                    outSize;
5136     UINT16                    offsetOfTypes;
5137     BYTE                      types[4];
5138 } Vendor_TCG_Test_COMMAND_DESCRIPTOR_t;
5139
5140 Vendor_TCG_Test_COMMAND_DESCRIPTOR_t _Vendor_TCG_TestData = {
5141     /* entry */          &TPM2_Vendor_TCG_Test,
5142     /* inSize */         (UINT16) (sizeof(Vendor_TCG_Test_In)),
5143     /* outSize */        (UINT16) (sizeof(Vendor_TCG_Test_Out)),
5144     /* offsetOfTypes */  offsetof(Vendor_TCG_Test_COMMAND_DESCRIPTOR_t, types),
5145     /* offsets */        // No parameter offsets
5146     /* types */          {TPM2B_DATA_P_UNMARSHAL,
5147                          END_OF_LIST,
5148                          TPM2B_DATA_P_MARSHAL,
5149                          END_OF_LIST};
5150 };
5151
5152 #define _Vendor_TCG_TestDataAddress (&_Vendor_TCG_TestData)
5153 #else
5154 #define _Vendor_TCG_TestDataAddress 0
5155 #endif // CC_Vendor_TCG_Test

```

```

5156
5157
5158 // Lookup table to access the per-command tables above
5159
5160 COMMAND_DESCRIPTOR_t* s_CommandDataArray[] = {
5161 #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
5162     (COMMAND_DESCRIPTOR_t*)_NV_UndefineSpaceSpecialDataAddress,
5163 #endif // CC_NV_UndefineSpaceSpecial
5164 #if (PAD_LIST || CC_EvictControl)
5165     (COMMAND_DESCRIPTOR_t*)_EvictControlDataAddress,
5166 #endif // CC_EvictControl
5167 #if (PAD_LIST || CC_HierarchyControl)
5168     (COMMAND_DESCRIPTOR_t*)_HierarchyControlDataAddress,
5169 #endif // CC_HierarchyControl
5170 #if (PAD_LIST || CC_NV_UndefineSpace)
5171     (COMMAND_DESCRIPTOR_t*)_NV_UndefineSpaceDataAddress,
5172 #endif // CC_NV_UndefineSpace
5173 #if (PAD_LIST)
5174     (COMMAND_DESCRIPTOR_t*)0,
5175 #endif //
5176 #if (PAD_LIST || CC_ChangeEPS)
5177     (COMMAND_DESCRIPTOR_t*)_ChangeEPSDataAddress,
5178 #endif // CC_ChangeEPS
5179 #if (PAD_LIST || CC_ChangePPS)
5180     (COMMAND_DESCRIPTOR_t*)_ChangePPSDataAddress,
5181 #endif // CC_ChangePPS
5182 #if (PAD_LIST || CC_Clear)
5183     (COMMAND_DESCRIPTOR_t*)_ClearDataAddress,
5184 #endif // CC_Clear
5185 #if (PAD_LIST || CC_ClearControl)
5186     (COMMAND_DESCRIPTOR_t*)_ClearControlDataAddress,
5187 #endif // CC_ClearControl
5188 #if (PAD_LIST || CC_ClockSet)
5189     (COMMAND_DESCRIPTOR_t*)_ClockSetDataAddress,
5190 #endif // CC_ClockSet
5191 #if (PAD_LIST || CC_HierarchyChangeAuth)
5192     (COMMAND_DESCRIPTOR_t*)_HierarchyChangeAuthDataAddress,
5193 #endif // CC_HierarchyChangeAuth
5194 #if (PAD_LIST || CC_NV_DefineSpace)
5195     (COMMAND_DESCRIPTOR_t*)_NV_DefineSpaceDataAddress,
5196 #endif // CC_NV_DefineSpace
5197 #if (PAD_LIST || CC_PCR_Allocate)
5198     (COMMAND_DESCRIPTOR_t*)_PCR_AllocateDataAddress,
5199 #endif // CC_PCR_Allocate
5200 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
5201     (COMMAND_DESCRIPTOR_t*)_PCR_SetAuthPolicyDataAddress,
5202 #endif // CC_PCR_SetAuthPolicy
5203 #if (PAD_LIST || CC_PP_Commands)
5204     (COMMAND_DESCRIPTOR_t*)_PP_CommandsDataAddress,
5205 #endif // CC_PP_Commands
5206 #if (PAD_LIST || CC_SetPrimaryPolicy)
5207     (COMMAND_DESCRIPTOR_t*)_SetPrimaryPolicyDataAddress,
5208 #endif // CC_SetPrimaryPolicy
5209 #if (PAD_LIST || CC_FieldUpgradeStart)
5210     (COMMAND_DESCRIPTOR_t*)_FieldUpgradeStartDataAddress,
5211 #endif // CC_FieldUpgradeStart
5212 #if (PAD_LIST || CC_ClockRateAdjust)
5213     (COMMAND_DESCRIPTOR_t*)_ClockRateAdjustDataAddress,
5214 #endif // CC_ClockRateAdjust
5215 #if (PAD_LIST || CC_CreatePrimary)
5216     (COMMAND_DESCRIPTOR_t*)_CreatePrimaryDataAddress,
5217 #endif // CC_CreatePrimary
5218 #if (PAD_LIST || CC_NV_GlobalWriteLock)
5219     (COMMAND_DESCRIPTOR_t*)_NV_GlobalWriteLockDataAddress,
5220 #endif // CC_NV_GlobalWriteLock
5221 #if (PAD_LIST || CC_GetCommandAuditDigest)

```

```

5222         (COMMAND_DESCRIPTOR_t*)_GetCommandAuditDigestDataAddress,
5223 #endif // CC_GetCommandAuditDigest
5224 #if (PAD_LIST || CC_NV_Increment)
5225         (COMMAND_DESCRIPTOR_t*)_NV_IncrementDataAddress,
5226 #endif // CC_NV_Increment
5227 #if (PAD_LIST || CC_NV_SetBits)
5228         (COMMAND_DESCRIPTOR_t*)_NV_SetBitsDataAddress,
5229 #endif // CC_NV_SetBits
5230 #if (PAD_LIST || CC_NV_Extend)
5231         (COMMAND_DESCRIPTOR_t*)_NV_ExtendDataAddress,
5232 #endif // CC_NV_Extend
5233 #if (PAD_LIST || CC_NV_Write)
5234         (COMMAND_DESCRIPTOR_t*)_NV_WriteDataAddress,
5235 #endif // CC_NV_Write
5236 #if (PAD_LIST || CC_NV_WriteLock)
5237         (COMMAND_DESCRIPTOR_t*)_NV_WriteLockDataAddress,
5238 #endif // CC_NV_WriteLock
5239 #if (PAD_LIST || CC_DictionaryAttackLockReset)
5240         (COMMAND_DESCRIPTOR_t*)_DictionaryAttackLockResetDataAddress,
5241 #endif // CC_DictionaryAttackLockReset
5242 #if (PAD_LIST || CC_DictionaryAttackParameters)
5243         (COMMAND_DESCRIPTOR_t*)_DictionaryAttackParametersDataAddress,
5244 #endif // CC_DictionaryAttackParameters
5245 #if (PAD_LIST || CC_NV_ChangeAuth)
5246         (COMMAND_DESCRIPTOR_t*)_NV_ChangeAuthDataAddress,
5247 #endif // CC_NV_ChangeAuth
5248 #if (PAD_LIST || CC_PCR_Event)
5249         (COMMAND_DESCRIPTOR_t*)_PCR_EventDataAddress,
5250 #endif // CC_PCR_Event
5251 #if (PAD_LIST || CC_PCR_Reset)
5252         (COMMAND_DESCRIPTOR_t*)_PCR_ResetDataAddress,
5253 #endif // CC_PCR_Reset
5254 #if (PAD_LIST || CC_SequenceComplete)
5255         (COMMAND_DESCRIPTOR_t*)_SequenceCompleteDataAddress,
5256 #endif // CC_SequenceComplete
5257 #if (PAD_LIST || CC_SetAlgorithmSet)
5258         (COMMAND_DESCRIPTOR_t*)_SetAlgorithmSetDataAddress,
5259 #endif // CC_SetAlgorithmSet
5260 #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
5261         (COMMAND_DESCRIPTOR_t*)_SetCommandCodeAuditStatusDataAddress,
5262 #endif // CC_SetCommandCodeAuditStatus
5263 #if (PAD_LIST || CC_FieldUpgradeData)
5264         (COMMAND_DESCRIPTOR_t*)_FieldUpgradeDataDataAddress,
5265 #endif // CC_FieldUpgradeData
5266 #if (PAD_LIST || CC_IncrementalSelfTest)
5267         (COMMAND_DESCRIPTOR_t*)_IncrementalSelfTestDataAddress,
5268 #endif // CC_IncrementalSelfTest
5269 #if (PAD_LIST || CC_SelfTest)
5270         (COMMAND_DESCRIPTOR_t*)_SelfTestDataAddress,
5271 #endif // CC_SelfTest
5272 #if (PAD_LIST || CC_Startup)
5273         (COMMAND_DESCRIPTOR_t*)_StartupDataAddress,
5274 #endif // CC_Startup
5275 #if (PAD_LIST || CC_Shutdown)
5276         (COMMAND_DESCRIPTOR_t*)_ShutdownDataAddress,
5277 #endif // CC_Shutdown
5278 #if (PAD_LIST || CC_StirRandom)
5279         (COMMAND_DESCRIPTOR_t*)_StirRandomDataAddress,
5280 #endif // CC_StirRandom
5281 #if (PAD_LIST || CC_ActivateCredential)
5282         (COMMAND_DESCRIPTOR_t*)_ActivateCredentialDataAddress,
5283 #endif // CC_ActivateCredential
5284 #if (PAD_LIST || CC_Certify)
5285         (COMMAND_DESCRIPTOR_t*)_CertifyDataAddress,
5286 #endif // CC_Certify
5287 #if (PAD_LIST || CC_PolicyNV)

```

```

5288         (COMMAND_DESCRIPTOR_t*)_PolicyNVDataAddress,
5289 #endif // CC_PolicyNV
5290 #if (PAD_LIST || CC_CertifyCreation)
5291         (COMMAND_DESCRIPTOR_t*)_CertifyCreationDataAddress,
5292 #endif // CC_CertifyCreation
5293 #if (PAD_LIST || CC_Duplicate)
5294         (COMMAND_DESCRIPTOR_t*)_DuplicateDataAddress,
5295 #endif // CC_Duplicate
5296 #if (PAD_LIST || CC_GetTime)
5297         (COMMAND_DESCRIPTOR_t*)_GetTimeDataAddress,
5298 #endif // CC_GetTime
5299 #if (PAD_LIST || CC_GetSessionAuditDigest)
5300         (COMMAND_DESCRIPTOR_t*)_GetSessionAuditDigestDataAddress,
5301 #endif // CC_GetSessionAuditDigest
5302 #if (PAD_LIST || CC_NV_Read)
5303         (COMMAND_DESCRIPTOR_t*)_NV_ReadDataAddress,
5304 #endif // CC_NV_Read
5305 #if (PAD_LIST || CC_NV_ReadLock)
5306         (COMMAND_DESCRIPTOR_t*)_NV_ReadLockDataAddress,
5307 #endif // CC_NV_ReadLock
5308 #if (PAD_LIST || CC_ObjectChangeAuth)
5309         (COMMAND_DESCRIPTOR_t*)_ObjectChangeAuthDataAddress,
5310 #endif // CC_ObjectChangeAuth
5311 #if (PAD_LIST || CC_PolicySecret)
5312         (COMMAND_DESCRIPTOR_t*)_PolicySecretDataAddress,
5313 #endif // CC_PolicySecret
5314 #if (PAD_LIST || CC_Rewrap)
5315         (COMMAND_DESCRIPTOR_t*)_RewrapDataAddress,
5316 #endif // CC_Rewrap
5317 #if (PAD_LIST || CC_Create)
5318         (COMMAND_DESCRIPTOR_t*)_CreateDataAddress,
5319 #endif // CC_Create
5320 #if (PAD_LIST || CC_ECDH_ZGen)
5321         (COMMAND_DESCRIPTOR_t*)_ECDH_ZGenDataAddress,
5322 #endif // CC_ECDH_ZGen
5323 #if (PAD_LIST || (CC_HMAC || CC_MAC))
5324 #   if CC_HMAC
5325         (COMMAND_DESCRIPTOR_t*)_HMACDataAddress,
5326 #   endif
5327 #   if CC_MAC
5328         (COMMAND_DESCRIPTOR_t*)_MACDataAddress,
5329 #   endif
5330 #endif // (CC_HMAC || CC_MAC)
5331 #if (PAD_LIST || CC_Import)
5332         (COMMAND_DESCRIPTOR_t*)_ImportDataAddress,
5333 #endif // CC_Import
5334 #if (PAD_LIST || CC_Load)
5335         (COMMAND_DESCRIPTOR_t*)_LoadDataAddress,
5336 #endif // CC_Load
5337 #if (PAD_LIST || CC_Quote)
5338         (COMMAND_DESCRIPTOR_t*)_QuoteDataAddress,
5339 #endif // CC_Quote
5340 #if (PAD_LIST || CC_RSA_Decrypt)
5341         (COMMAND_DESCRIPTOR_t*)_RSA_DecryptDataAddress,
5342 #endif // CC_RSA_Decrypt
5343 #if (PAD_LIST)
5344         (COMMAND_DESCRIPTOR_t*)0,
5345 #endif //
5346 #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
5347 #   if CC_HMAC_Start
5348         (COMMAND_DESCRIPTOR_t*)_HMAC_StartDataAddress,
5349 #   endif
5350 #   if CC_MAC_Start
5351         (COMMAND_DESCRIPTOR_t*)_MAC_StartDataAddress,
5352 #   endif
5353 #endif // (CC_HMAC_Start || CC_MAC_Start)

```

```
5354 #if (PAD_LIST || CC_SequenceUpdate)
5355     (COMMAND_DESCRIPTOR_t*)_SequenceUpdateDataAddress,
5356 #endif // CC_SequenceUpdate
5357 #if (PAD_LIST || CC_Sign)
5358     (COMMAND_DESCRIPTOR_t*)_SignDataAddress,
5359 #endif // CC_Sign
5360 #if (PAD_LIST || CC_Unseal)
5361     (COMMAND_DESCRIPTOR_t*)_UnsealDataAddress,
5362 #endif // CC_Unseal
5363 #if (PAD_LIST)
5364     (COMMAND_DESCRIPTOR_t*)0,
5365 #endif //
5366 #if (PAD_LIST || CC_PolicySigned)
5367     (COMMAND_DESCRIPTOR_t*)_PolicySignedDataAddress,
5368 #endif // CC_PolicySigned
5369 #if (PAD_LIST || CC_ContextLoad)
5370     (COMMAND_DESCRIPTOR_t*)_ContextLoadDataAddress,
5371 #endif // CC_ContextLoad
5372 #if (PAD_LIST || CC_ContextSave)
5373     (COMMAND_DESCRIPTOR_t*)_ContextSaveDataAddress,
5374 #endif // CC_ContextSave
5375 #if (PAD_LIST || CC_ECDH_KeyGen)
5376     (COMMAND_DESCRIPTOR_t*)_ECDH_KeyGenDataAddress,
5377 #endif // CC_ECDH_KeyGen
5378 #if (PAD_LIST || CC_EncryptDecrypt)
5379     (COMMAND_DESCRIPTOR_t*)_EncryptDecryptDataAddress,
5380 #endif // CC_EncryptDecrypt
5381 #if (PAD_LIST || CC_FlushContext)
5382     (COMMAND_DESCRIPTOR_t*)_FlushContextDataAddress,
5383 #endif // CC_FlushContext
5384 #if (PAD_LIST)
5385     (COMMAND_DESCRIPTOR_t*)0,
5386 #endif //
5387 #if (PAD_LIST || CC_LoadExternal)
5388     (COMMAND_DESCRIPTOR_t*)_LoadExternalDataAddress,
5389 #endif // CC_LoadExternal
5390 #if (PAD_LIST || CC_MakeCredential)
5391     (COMMAND_DESCRIPTOR_t*)_MakeCredentialDataAddress,
5392 #endif // CC_MakeCredential
5393 #if (PAD_LIST || CC_NV_ReadPublic)
5394     (COMMAND_DESCRIPTOR_t*)_NV_ReadPublicDataAddress,
5395 #endif // CC_NV_ReadPublic
5396 #if (PAD_LIST || CC_PolicyAuthorize)
5397     (COMMAND_DESCRIPTOR_t*)_PolicyAuthorizeDataAddress,
5398 #endif // CC_PolicyAuthorize
5399 #if (PAD_LIST || CC_PolicyAuthValue)
5400     (COMMAND_DESCRIPTOR_t*)_PolicyAuthValueDataAddress,
5401 #endif // CC_PolicyAuthValue
5402 #if (PAD_LIST || CC_PolicyCommandCode)
5403     (COMMAND_DESCRIPTOR_t*)_PolicyCommandCodeDataAddress,
5404 #endif // CC_PolicyCommandCode
5405 #if (PAD_LIST || CC_PolicyCounterTimer)
5406     (COMMAND_DESCRIPTOR_t*)_PolicyCounterTimerDataAddress,
5407 #endif // CC_PolicyCounterTimer
5408 #if (PAD_LIST || CC_PolicyCpHash)
5409     (COMMAND_DESCRIPTOR_t*)_PolicyCpHashDataAddress,
5410 #endif // CC_PolicyCpHash
5411 #if (PAD_LIST || CC_PolicyLocality)
5412     (COMMAND_DESCRIPTOR_t*)_PolicyLocalityDataAddress,
5413 #endif // CC_PolicyLocality
5414 #if (PAD_LIST || CC_PolicyNameHash)
5415     (COMMAND_DESCRIPTOR_t*)_PolicyNameHashDataAddress,
5416 #endif // CC_PolicyNameHash
5417 #if (PAD_LIST || CC_PolicyOR)
5418     (COMMAND_DESCRIPTOR_t*)_PolicyORDataAddress,
5419 #endif // CC_PolicyOR
```



```

5420 #if (PAD_LIST || CC_PolicyTicket)
5421     (COMMAND_DESCRIPTOR_t*)_PolicyTicketDataAddress,
5422 #endif // CC_PolicyTicket
5423 #if (PAD_LIST || CC_ReadPublic)
5424     (COMMAND_DESCRIPTOR_t*)_ReadPublicDataAddress,
5425 #endif // CC_ReadPublic
5426 #if (PAD_LIST || CC_RSA_Encrypt)
5427     (COMMAND_DESCRIPTOR_t*)_RSA_EncryptDataAddress,
5428 #endif // CC_RSA_Encrypt
5429 #if (PAD_LIST)
5430     (COMMAND_DESCRIPTOR_t*)0,
5431 #endif //
5432 #if (PAD_LIST || CC_StartAuthSession)
5433     (COMMAND_DESCRIPTOR_t*)_StartAuthSessionDataAddress,
5434 #endif // CC_StartAuthSession
5435 #if (PAD_LIST || CC_VerifySignature)
5436     (COMMAND_DESCRIPTOR_t*)_VerifySignatureDataAddress,
5437 #endif // CC_VerifySignature
5438 #if (PAD_LIST || CC_ECC_Parameters)
5439     (COMMAND_DESCRIPTOR_t*)_ECC_ParametersDataAddress,
5440 #endif // CC_ECC_Parameters
5441 #if (PAD_LIST || CC_FirmwareRead)
5442     (COMMAND_DESCRIPTOR_t*)_FirmwareReadDataAddress,
5443 #endif // CC_FirmwareRead
5444 #if (PAD_LIST || CC_GetCapability)
5445     (COMMAND_DESCRIPTOR_t*)_GetCapabilityDataAddress,
5446 #endif // CC_GetCapability
5447 #if (PAD_LIST || CC_GetRandom)
5448     (COMMAND_DESCRIPTOR_t*)_GetRandomDataAddress,
5449 #endif // CC_GetRandom
5450 #if (PAD_LIST || CC_GetTestResult)
5451     (COMMAND_DESCRIPTOR_t*)_GetTestResultDataAddress,
5452 #endif // CC_GetTestResult
5453 #if (PAD_LIST || CC_Hash)
5454     (COMMAND_DESCRIPTOR_t*)_HashDataAddress,
5455 #endif // CC_Hash
5456 #if (PAD_LIST || CC_PCR_Read)
5457     (COMMAND_DESCRIPTOR_t*)_PCR_ReadDataAddress,
5458 #endif // CC_PCR_Read
5459 #if (PAD_LIST || CC_PolicyPCR)
5460     (COMMAND_DESCRIPTOR_t*)_PolicyPCRDataAddress,
5461 #endif // CC_PolicyPCR
5462 #if (PAD_LIST || CC_PolicyRestart)
5463     (COMMAND_DESCRIPTOR_t*)_PolicyRestartDataAddress,
5464 #endif // CC_PolicyRestart
5465 #if (PAD_LIST || CC_ReadClock)
5466     (COMMAND_DESCRIPTOR_t*)_ReadClockDataAddress,
5467 #endif // CC_ReadClock
5468 #if (PAD_LIST || CC_PCR_Extend)
5469     (COMMAND_DESCRIPTOR_t*)_PCR_ExtendDataAddress,
5470 #endif // CC_PCR_Extend
5471 #if (PAD_LIST || CC_PCR_SetAuthValue)
5472     (COMMAND_DESCRIPTOR_t*)_PCR_SetAuthValueDataAddress,
5473 #endif // CC_PCR_SetAuthValue
5474 #if (PAD_LIST || CC_NV_Certify)
5475     (COMMAND_DESCRIPTOR_t*)_NV_CertifyDataAddress,
5476 #endif // CC_NV_Certify
5477 #if (PAD_LIST || CC_EventSequenceComplete)
5478     (COMMAND_DESCRIPTOR_t*)_EventSequenceCompleteDataAddress,
5479 #endif // CC_EventSequenceComplete
5480 #if (PAD_LIST || CC_HashSequenceStart)
5481     (COMMAND_DESCRIPTOR_t*)_HashSequenceStartDataAddress,
5482 #endif // CC_HashSequenceStart
5483 #if (PAD_LIST || CC_PolicyPhysicalPresence)
5484     (COMMAND_DESCRIPTOR_t*)_PolicyPhysicalPresenceDataAddress,
5485 #endif // CC_PolicyPhysicalPresence

```



```

5486 #if (PAD_LIST || CC_PolicyDuplicationSelect)
5487     (COMMAND_DESCRIPTOR_t*)_PolicyDuplicationSelectDataAddress,
5488 #endif // CC_PolicyDuplicationSelect
5489 #if (PAD_LIST || CC_PolicyGetDigest)
5490     (COMMAND_DESCRIPTOR_t*)_PolicyGetDigestDataAddress,
5491 #endif // CC_PolicyGetDigest
5492 #if (PAD_LIST || CC_TestParms)
5493     (COMMAND_DESCRIPTOR_t*)_TestParmsDataAddress,
5494 #endif // CC_TestParms
5495 #if (PAD_LIST || CC_Commit)
5496     (COMMAND_DESCRIPTOR_t*)_CommitDataAddress,
5497 #endif // CC_Commit
5498 #if (PAD_LIST || CC_PolicyPassword)
5499     (COMMAND_DESCRIPTOR_t*)_PolicyPasswordDataAddress,
5500 #endif // CC_PolicyPassword
5501 #if (PAD_LIST || CC_ZGen_2Phase)
5502     (COMMAND_DESCRIPTOR_t*)_ZGen_2PhaseDataAddress,
5503 #endif // CC_ZGen_2Phase
5504 #if (PAD_LIST || CC_EC_Ephemeral)
5505     (COMMAND_DESCRIPTOR_t*)_EC_EphemeralDataAddress,
5506 #endif // CC_EC_Ephemeral
5507 #if (PAD_LIST || CC_PolicyNvWritten)
5508     (COMMAND_DESCRIPTOR_t*)_PolicyNvWrittenDataAddress,
5509 #endif // CC_PolicyNvWritten
5510 #if (PAD_LIST || CC_PolicyTemplate)
5511     (COMMAND_DESCRIPTOR_t*)_PolicyTemplateDataAddress,
5512 #endif // CC_PolicyTemplate
5513 #if (PAD_LIST || CC_CreateLoaded)
5514     (COMMAND_DESCRIPTOR_t*)_CreateLoadedDataAddress,
5515 #endif // CC_CreateLoaded
5516 #if (PAD_LIST || CC_PolicyAuthorizeNV)
5517     (COMMAND_DESCRIPTOR_t*)_PolicyAuthorizeNVDataAddress,
5518 #endif // CC_PolicyAuthorizeNV
5519 #if (PAD_LIST || CC_EncryptDecrypt2)
5520     (COMMAND_DESCRIPTOR_t*)_EncryptDecrypt2DataAddress,
5521 #endif // CC_EncryptDecrypt2
5522 #if (PAD_LIST || CC_AC_GetCapability)
5523     (COMMAND_DESCRIPTOR_t*)_GetCapabilityDataAddress,
5524 #endif // CC_AC_GetCapability
5525 #if (PAD_LIST || CC_AC_Send)
5526     (COMMAND_DESCRIPTOR_t*)_AC_SendDataAddress,
5527 #endif // CC_AC_Send
5528 #if (PAD_LIST || CC_Policy_AC_SendSelect)
5529     (COMMAND_DESCRIPTOR_t*)_Policy_AC_SendSelectDataAddress,
5530 #endif // CC_Policy_AC_SendSelect
5531 #if (PAD_LIST || CC_CertifyX509)
5532     (COMMAND_DESCRIPTOR_t*)_CertifyX509DataAddress,
5533 #endif // CC_CertifyX509
5534 #if (PAD_LIST || CC_ACT_SetTimeout)
5535     (COMMAND_DESCRIPTOR_t*)_ACT_SetTimeoutDataAddress,
5536 #endif // CC_ACT_SetTimeout
5537 #if (PAD_LIST || CC_ECC_Encrypt)
5538     (COMMAND_DESCRIPTOR_t*)_ECC_EncryptDataAddress,
5539 #endif // CC_ECC_Encrypt
5540 #if (PAD_LIST || CC_ECC_Decrypt)
5541     (COMMAND_DESCRIPTOR_t*)_ECC_DecryptDataAddress,
5542 #endif // CC_ECC_Decrypt
5543 #if (PAD_LIST || CC_PolicyCapability)
5544     (COMMAND_DESCRIPTOR_t*)_PolicyCapabilityDataAddress,
5545 #endif // CC_PolicyCapability
5546 #if (PAD_LIST || CC_PolicyParameters)
5547     (COMMAND_DESCRIPTOR_t*)_PolicyParametersDataAddress,
5548 #endif // CC_PolicyParameters
5549 #if (PAD_LIST || CC_NV_DefineSpace2)
5550     (COMMAND_DESCRIPTOR_t*)_NV_DefineSpace2DataAddress,
5551 #endif // CC_NV_DefineSpace2

```

```

5552 #if (PAD_LIST || CC_NV_ReadPublic2)
5553     (COMMAND_DESCRIPTOR_t*)_NV_ReadPublic2DataAddress,
5554 #endif // CC_NV_ReadPublic2
5555 #if (PAD_LIST || CC_SetCapability)
5556     (COMMAND_DESCRIPTOR_t*)_SetCapabilityDataAddress,
5557 #endif // CC_SetCapability
5558 #if (PAD_LIST || CC_Vendor_TCG_Test)
5559     (COMMAND_DESCRIPTOR_t*)_Vendor_TCG_TestDataAddress,
5560 #endif // CC_Vendor_TCG_Test
5561
5562     0
5563 };
5564
5565 #endif // _COMMAND_TABLE_DISPATCH_

```

6.14 /tpm/include/private/CommandDispatcher.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  // This macro is added just so that the code is only excessively long.
4  #define EXIT_IF_ERROR_PLUS(x) \
5      if(TPM_RC_SUCCESS != result) \
6      { \
7          result += (x); \
8          goto Exit; \
9      }
10 #if CC_Startup
11 case TPM_CC_Startup:
12 {
13     Startup_In* in = (Startup_In*)MemoryGetInBuffer(sizeof(Startup_In));
14     result = TPM_SU_Unmarshal(&in->startupType, paramBuffer, paramBufferSize);
15     EXIT_IF_ERROR_PLUS(RC_Startup_startupType);
16     if(*paramBufferSize != 0)
17     {
18         result = TPM_RC_SIZE;
19         goto Exit;
20     }
21     result = TPM2_Startup(in);
22     break;
23 }
24 #endif // CC_Startup
25 #if CC_Shutdown
26 case TPM_CC_Shutdown:
27 {
28     Shutdown_In* in = (Shutdown_In*)MemoryGetInBuffer(sizeof(Shutdown_In));
29     result = TPM_SU_Unmarshal(&in->shutdownType, paramBuffer, paramBufferSize);
30     EXIT_IF_ERROR_PLUS(RC_Shutdown_shutdownType);
31     if(*paramBufferSize != 0)
32     {
33         result = TPM_RC_SIZE;
34         goto Exit;
35     }
36     result = TPM2_Shutdown(in);
37     break;
38 }
39 #endif // CC_Shutdown
40 #if CC_SelfTest
41 case TPM_CC_SelfTest:
42 {
43     SelfTest_In* in = (SelfTest_In*)MemoryGetInBuffer(sizeof(SelfTest_In));
44     result = TPMI_YES_NO_Unmarshal(&in->fullTest, paramBuffer, paramBufferSize);
45     EXIT_IF_ERROR_PLUS(RC_SelfTest_fullTest);
46     if(*paramBufferSize != 0)
47     {
48         result = TPM_RC_SIZE;

```

```

49         goto Exit;
50     }
51     result = TPM2_SelfTest(in);
52     break;
53 }
54 #endif // CC_SelfTest
55 #if CC_IncrementalSelfTest
56 case TPM_CC_IncrementalSelfTest:
57 {
58     IncrementalSelfTest_In* in =
59         (IncrementalSelfTest_In*)MemoryGetInBuffer(sizeof(IncrementalSelfTest_In));
60     IncrementalSelfTest_Out* out =
61         (IncrementalSelfTest_Out*)MemoryGetOutBuffer(sizeof(IncrementalSelfTest_Out));
62     result = TPML_ALG_Unmarshal(&in->toTest, paramBuffer, paramBufferSize);
63     EXIT_IF_ERROR_PLUS(RC_IncrementalSelfTest_toTest);
64     if(*paramBufferSize != 0)
65     {
66         result = TPM_RC_SIZE;
67         goto Exit;
68     }
69     result = TPM2_IncrementalSelfTest(in, out);
70     rSize = sizeof(IncrementalSelfTest_Out);
71     *respParamSize += TPML_ALG_Marshal(&out->toDoList, responseBuffer, &rSize);
72     break;
73 }
74 #endif // CC_IncrementalSelfTest
75 #if CC_GetTestResult
76 case TPM_CC_GetTestResult:
77 {
78     GetTestResult_Out* out =
79         (GetTestResult_Out*)MemoryGetOutBuffer(sizeof(GetTestResult_Out));
80     if(*paramBufferSize != 0)
81     {
82         result = TPM_RC_SIZE;
83         goto Exit;
84     }
85     result = TPM2_GetTestResult(out);
86     rSize = sizeof(GetTestResult_Out);
87     *respParamSize += TPM2B_MAX_BUFFER_Marshal(&out->outData, responseBuffer, &rSize);
88     *respParamSize += TPM_RC_Marshal(&out->testResult, responseBuffer, &rSize);
89     break;
90 }
91 #endif // CC_GetTestResult
92 #if CC_StartAuthSession
93 case TPM_CC_StartAuthSession:
94 {
95     StartAuthSession_In* in =
96         (StartAuthSession_In*)MemoryGetInBuffer(sizeof(StartAuthSession_In));
97     StartAuthSession_Out* out =
98         (StartAuthSession_Out*)MemoryGetOutBuffer(sizeof(StartAuthSession_Out));
99     in->tpmKey = handles[0];
100     in->bind = handles[1];
101     result = TPM2B_NONCE_Unmarshal(&in->nonceCaller, paramBuffer, paramBufferSize);
102     EXIT_IF_ERROR_PLUS(RC_StartAuthSession_nonceCaller);
103     result = TPM2B_ENCRYPTED_SECRET_Unmarshal(
104         &in->encryptedSalt, paramBuffer, paramBufferSize);
105     EXIT_IF_ERROR_PLUS(RC_StartAuthSession_encryptedSalt);
106     result = TPM_SE_Unmarshal(&in->sessionType, paramBuffer, paramBufferSize);
107     EXIT_IF_ERROR_PLUS(RC_StartAuthSession_sessionType);
108     result =
109         TPMT_SYM_DEF_Unmarshal(&in->symmetric, paramBuffer, paramBufferSize, TRUE);
110     EXIT_IF_ERROR_PLUS(RC_StartAuthSession_symmetric);
111     result =
112         TPMT_ALG_HASH_Unmarshal(&in->authHash, paramBuffer, paramBufferSize, FALSE);
113     EXIT_IF_ERROR_PLUS(RC_StartAuthSession_authHash);
114     if(*paramBufferSize != 0)

```

```

115     {
116         result = TPM_RC_SIZE;
117         goto Exit;
118     }
119     result = TPM2_StartAuthSession(in, out);
120     rSize = sizeof(StartAuthSession_Out);
121     if(TPM_RC_SUCCESS != result)
122         goto Exit;
123     command->handles[command->handleNum++] = out->sessionHandle;
124     *respParmSize += TPM2B_NONCE_Marshal(&out->nonceTPM, responseBuffer, &rSize);
125     break;
126 }
127 #endif // CC_StartAuthSession
128 #if CC_PolicyRestart
129 case TPM_CC_PolicyRestart:
130 {
131     PolicyRestart_In* in =
132         (PolicyRestart_In*)MemoryGetInBuffer(sizeof(PolicyRestart_In));
133     in->sessionHandle = handles[0];
134     if(*paramBufferSize != 0)
135     {
136         result = TPM_RC_SIZE;
137         goto Exit;
138     }
139     result = TPM2_PolicyRestart(in);
140     break;
141 }
142 #endif // CC_PolicyRestart
143 #if CC_Create
144 case TPM_CC_Create:
145 {
146     Create_In* in = (Create_In*)MemoryGetInBuffer(sizeof(Create_In));
147     Create_Out* out = (Create_Out*)MemoryGetOutBuffer(sizeof(Create_Out));
148     in->parentHandle = handles[0];
149     result = TPM2B_SENSITIVE_CREATE_Unmarshal(
150         &in->inSensitive, paramBuffer, paramBufferSize);
151     EXIT_IF_ERROR_PLUS(RC_Create_inSensitive);
152     result =
153         TPM2B_PUBLIC_Unmarshal(&in->inPublic, paramBuffer, paramBufferSize, FALSE);
154     EXIT_IF_ERROR_PLUS(RC_Create_inPublic);
155     result = TPM2B_DATA_Unmarshal(&in->outsideInfo, paramBuffer, paramBufferSize);
156     EXIT_IF_ERROR_PLUS(RC_Create_outsideInfo);
157     result =
158         TPML_PCR_SELECTION_Unmarshal(&in->creationPCR, paramBuffer, paramBufferSize);
159     EXIT_IF_ERROR_PLUS(RC_Create_creationPCR);
160     if(*paramBufferSize != 0)
161     {
162         result = TPM_RC_SIZE;
163         goto Exit;
164     }
165     result = TPM2_Create(in, out);
166     rSize = sizeof(Create_Out);
167     *respParmSize += TPM2B_PRIVATE_Marshal(&out->outPrivate, responseBuffer, &rSize);
168     *respParmSize += TPM2B_PUBLIC_Marshal(&out->outPublic, responseBuffer, &rSize);
169     *respParmSize +=
170         TPM2B_CREATION_DATA_Marshal(&out->creationData, responseBuffer, &rSize);
171     *respParmSize += TPM2B_DIGEST_Marshal(&out->creationHash, responseBuffer, &rSize);
172     *respParmSize +=
173         TPMT_TK_CREATION_Marshal(&out->creationTicket, responseBuffer, &rSize);
174     break;
175 }
176 #endif // CC_Create
177 #if CC_Load
178 case TPM_CC_Load:
179 {
180     Load_In* in = (Load_In*)MemoryGetInBuffer(sizeof(Load_In));

```

```

181     Load_Out* out      = (Load_Out*)MemoryGetOutBuffer(sizeof(Load_Out));
182     in->parentHandle = handles[0];
183     result = TPM2B_PRIVATE_Unmarshal(&in->inPrivate, paramBuffer, paramBufferSize);
184     EXIT_IF_ERROR_PLUS(RC_Load_inPrivate);
185     result =
186         TPM2B_PUBLIC_Unmarshal(&in->inPublic, paramBuffer, paramBufferSize, FALSE);
187     EXIT_IF_ERROR_PLUS(RC_Load_inPublic);
188     if(*paramBufferSize != 0)
189     {
190         result = TPM_RC_SIZE;
191         goto Exit;
192     }
193     result = TPM2_Load(in, out);
194     rSize = sizeof(Load_Out);
195     if(TPM_RC_SUCCESS != result)
196         goto Exit;
197     command->handles[command->handleNum++] = out->objectHandle;
198     *respParmSize += TPM2B_NAME_Marshal(&out->name, responseBuffer, &rSize);
199     break;
200 }
201 #endif // CC_Load
202 #if CC_LoadExternal
203 case TPM_CC_LoadExternal:
204 {
205     LoadExternal_In* in =
206         (LoadExternal_In*)MemoryGetInBuffer(sizeof(LoadExternal_In));
207     LoadExternal_Out* out =
208         (LoadExternal_Out*)MemoryGetOutBuffer(sizeof(LoadExternal_Out));
209     result = TPM2B_SENSITIVE_Unmarshal(&in->inPrivate, paramBuffer, paramBufferSize);
210     EXIT_IF_ERROR_PLUS(RC_LoadExternal_inPrivate);
211     result =
212         TPM2B_PUBLIC_Unmarshal(&in->inPublic, paramBuffer, paramBufferSize, TRUE);
213     EXIT_IF_ERROR_PLUS(RC_LoadExternal_inPublic);
214     result = TPMI_RH_HIERARCHY_Unmarshal(
215         &in->hierarchy, paramBuffer, paramBufferSize, TRUE);
216     EXIT_IF_ERROR_PLUS(RC_LoadExternal_hierarchy);
217     if(*paramBufferSize != 0)
218     {
219         result = TPM_RC_SIZE;
220         goto Exit;
221     }
222     result = TPM2_LoadExternal(in, out);
223     rSize = sizeof(LoadExternal_Out);
224     if(TPM_RC_SUCCESS != result)
225         goto Exit;
226     command->handles[command->handleNum++] = out->objectHandle;
227     *respParmSize += TPM2B_NAME_Marshal(&out->name, responseBuffer, &rSize);
228     break;
229 }
230 #endif // CC_LoadExternal
231 #if CC_ReadPublic
232 case TPM_CC_ReadPublic:
233 {
234     ReadPublic_In* in = (ReadPublic_In*)MemoryGetInBuffer(sizeof(ReadPublic_In));
235     ReadPublic_Out* out = (ReadPublic_Out*)MemoryGetOutBuffer(sizeof(ReadPublic_Out));
236     in->objectHandle = handles[0];
237     if(*paramBufferSize != 0)
238     {
239         result = TPM_RC_SIZE;
240         goto Exit;
241     }
242     result = TPM2_ReadPublic(in, out);
243     rSize = sizeof(ReadPublic_Out);
244     *respParmSize += TPM2B_PUBLIC_Marshal(&out->outPublic, responseBuffer, &rSize);
245     *respParmSize += TPM2B_NAME_Marshal(&out->name, responseBuffer, &rSize);
246     *respParmSize += TPM2B_NAME_Marshal(&out->qualifiedName, responseBuffer, &rSize);

```



```

247     break;
248 }
249 #endif // CC_ReadPublic
250 #if CC_ActivateCredential
251 case TPM_CC_ActivateCredential:
252 {
253     ActivateCredential_In* in =
254         (ActivateCredential_In*)MemoryGetInBuffer(sizeof(ActivateCredential_In));
255     ActivateCredential_Out* out =
256         (ActivateCredential_Out*)MemoryGetOutBuffer(sizeof(ActivateCredential_Out));
257     in->activateHandle = handles[0];
258     in->keyHandle      = handles[1];
259     result =
260         TPM2B_ID_OBJECT_Unmarshal(&in->credentialBlob, paramBuffer, paramBufferSize);
261     EXIT_IF_ERROR_PLUS(RC_ActivateCredential_credentialBlob);
262     result =
263         TPM2B_ENCRYPTED_SECRET_Unmarshal(&in->secret, paramBuffer, paramBufferSize);
264     EXIT_IF_ERROR_PLUS(RC_ActivateCredential_secret);
265     if(*paramBufferSize != 0)
266     {
267         result = TPM_RC_SIZE;
268         goto Exit;
269     }
270     result = TPM2_ActivateCredential(in, out);
271     rSize = sizeof(ActivateCredential_Out);
272     *respParmSize += TPM2B_DIGEST_Marshal(&out->certInfo, responseBuffer, &rSize);
273     break;
274 }
275 #endif // CC_ActivateCredential
276 #if CC_MakeCredential
277 case TPM_CC_MakeCredential:
278 {
279     MakeCredential_In* in =
280         (MakeCredential_In*)MemoryGetInBuffer(sizeof(MakeCredential_In));
281     MakeCredential_Out* out =
282         (MakeCredential_Out*)MemoryGetOutBuffer(sizeof(MakeCredential_Out));
283     in->handle = handles[0];
284     result = TPM2B_DIGEST_Unmarshal(&in->credential, paramBuffer, paramBufferSize);
285     EXIT_IF_ERROR_PLUS(RC_MakeCredential_credential);
286     result = TPM2B_NAME_Unmarshal(&in->objectName, paramBuffer, paramBufferSize);
287     EXIT_IF_ERROR_PLUS(RC_MakeCredential_objectName);
288     if(*paramBufferSize != 0)
289     {
290         result = TPM_RC_SIZE;
291         goto Exit;
292     }
293     result = TPM2_MakeCredential(in, out);
294     rSize = sizeof(MakeCredential_Out);
295     *respParmSize +=
296         TPM2B_ID_OBJECT_Marshal(&out->credentialBlob, responseBuffer, &rSize);
297     *respParmSize +=
298         TPM2B_ENCRYPTED_SECRET_Marshal(&out->secret, responseBuffer, &rSize);
299     break;
300 }
301 #endif // CC_MakeCredential
302 #if CC_Unseal
303 case TPM_CC_Unseal:
304 {
305     Unseal_In* in = (Unseal_In*)MemoryGetInBuffer(sizeof(Unseal_In));
306     Unseal_Out* out = (Unseal_Out*)MemoryGetOutBuffer(sizeof(Unseal_Out));
307     in->itemHandle = handles[0];
308     if(*paramBufferSize != 0)
309     {
310         result = TPM_RC_SIZE;
311         goto Exit;
312     }

```



```

313     result = TPM2_Unseal(in, out);
314     rSize = sizeof(Unseal_Out);
315     *respParmSize +=
316         TPM2B_SENSITIVE_DATA_Marshal(&out->outData, responseBuffer, &rSize);
317     break;
318 }
319 #endif // CC_Unseal
320 #if CC_ObjectChangeAuth
321 case TPM_CC_ObjectChangeAuth:
322 {
323     ObjectChangeAuth_In* in =
324         (ObjectChangeAuth_In*)MemoryGetInBuffer(sizeof(ObjectChangeAuth_In));
325     ObjectChangeAuth_Out* out =
326         (ObjectChangeAuth_Out*)MemoryGetOutBuffer(sizeof(ObjectChangeAuth_Out));
327     in->objectHandle = handles[0];
328     in->parentHandle = handles[1];
329     result = TPM2B_AUTH_Unmarshal(&in->newAuth, paramBuffer, paramBufferSize);
330     EXIT_IF_ERROR_PLUS(RC_ObjectChangeAuth_newAuth);
331     if(*paramBufferSize != 0)
332     {
333         result = TPM_RC_SIZE;
334         goto Exit;
335     }
336     result = TPM2_ObjectChangeAuth(in, out);
337     rSize = sizeof(ObjectChangeAuth_Out);
338     *respParmSize += TPM2B_PRIVATE_Marshal(&out->outPrivate, responseBuffer, &rSize);
339     break;
340 }
341 #endif // CC_ObjectChangeAuth
342 #if CC_CreateLoaded
343 case TPM_CC_CreateLoaded:
344 {
345     CreateLoaded_In* in =
346         (CreateLoaded_In*)MemoryGetInBuffer(sizeof(CreateLoaded_In));
347     CreateLoaded_Out* out =
348         (CreateLoaded_Out*)MemoryGetOutBuffer(sizeof(CreateLoaded_Out));
349     in->parentHandle = handles[0];
350     result = TPM2B_SENSITIVE_CREATE_Unmarshal(
351         &in->inSensitive, paramBuffer, paramBufferSize);
352     EXIT_IF_ERROR_PLUS(RC_CreateLoaded_inSensitive);
353     result = TPM2B_TEMPLATE_Unmarshal(&in->inPublic, paramBuffer, paramBufferSize);
354     EXIT_IF_ERROR_PLUS(RC_CreateLoaded_inPublic);
355     if(*paramBufferSize != 0)
356     {
357         result = TPM_RC_SIZE;
358         goto Exit;
359     }
360     result = TPM2_CreateLoaded(in, out);
361     rSize = sizeof(CreateLoaded_Out);
362     if(TPM_RC_SUCCESS != result)
363         goto Exit;
364     command->handles[command->handleNum++] = out->objectHandle;
365     *respParmSize += TPM2B_PRIVATE_Marshal(&out->outPrivate, responseBuffer, &rSize);
366     *respParmSize += TPM2B_PUBLIC_Marshal(&out->outPublic, responseBuffer, &rSize);
367     *respParmSize += TPM2B_NAME_Marshal(&out->name, responseBuffer, &rSize);
368     break;
369 }
370 #endif // CC_CreateLoaded
371 #if CC_Duplicate
372 case TPM_CC_Duplicate:
373 {
374     Duplicate_In* in = (Duplicate_In*)MemoryGetInBuffer(sizeof(Duplicate_In));
375     Duplicate_Out* out = (Duplicate_Out*)MemoryGetOutBuffer(sizeof(Duplicate_Out));
376     in->objectHandle = handles[0];
377     in->newParentHandle = handles[1];
378     result = TPM2B_DATA_Unmarshal(&in->encryptionKeyIn, paramBuffer, paramBufferSize);

```

```

379     EXIT_IF_ERROR_PLUS(RC_Duplicate_encryptionKeyIn);
380     result = TPMT_SYM_DEF_OBJECT_Unmarshal(
381         &in->symmetricAlg, paramBuffer, paramBufferSize, TRUE);
382     EXIT_IF_ERROR_PLUS(RC_Duplicate_symmetricAlg);
383     if(*paramBufferSize != 0)
384     {
385         result = TPM_RC_SIZE;
386         goto Exit;
387     }
388     result = TPM2_Duplicate(in, out);
389     rSize = sizeof(Duplicate_Out);
390     *respParmSize +=
391         TPM2B_DATA_Marshal(&out->encryptionKeyOut, responseBuffer, &rSize);
392     *respParmSize += TPM2B_PRIVATE_Marshal(&out->duplicate, responseBuffer, &rSize);
393     *respParmSize +=
394         TPM2B_ENCRYPTED_SECRET_Marshal(&out->outSymSeed, responseBuffer, &rSize);
395     break;
396 }
397 #endif // CC_Duplicate
398 #if CC_Rewrap
399 case TPM_CC_Rewrap:
400 {
401     Rewrap_In* in = (Rewrap_In*)MemoryGetInBuffer(sizeof(Rewrap_In));
402     Rewrap_Out* out = (Rewrap_Out*)MemoryGetOutBuffer(sizeof(Rewrap_Out));
403     in->oldParent = handles[0];
404     in->newParent = handles[1];
405     result = TPM2B_PRIVATE_Unmarshal(&in->inDuplicate, paramBuffer, paramBufferSize);
406     EXIT_IF_ERROR_PLUS(RC_Rewrap_inDuplicate);
407     result = TPM2B_NAME_Unmarshal(&in->name, paramBuffer, paramBufferSize);
408     EXIT_IF_ERROR_PLUS(RC_Rewrap_name);
409     result = TPM2B_ENCRYPTED_SECRET_Unmarshal(
410         &in->inSymSeed, paramBuffer, paramBufferSize);
411     EXIT_IF_ERROR_PLUS(RC_Rewrap_inSymSeed);
412     if(*paramBufferSize != 0)
413     {
414         result = TPM_RC_SIZE;
415         goto Exit;
416     }
417     result = TPM2_Rewrap(in, out);
418     rSize = sizeof(Rewrap_Out);
419     *respParmSize +=
420         TPM2B_PRIVATE_Marshal(&out->outDuplicate, responseBuffer, &rSize);
421     *respParmSize +=
422         TPM2B_ENCRYPTED_SECRET_Marshal(&out->outSymSeed, responseBuffer, &rSize);
423     break;
424 }
425 #endif // CC_Rewrap
426 #if CC_Import
427 case TPM_CC_Import:
428 {
429     Import_In* in = (Import_In*)MemoryGetInBuffer(sizeof(Import_In));
430     Import_Out* out = (Import_Out*)MemoryGetOutBuffer(sizeof(Import_Out));
431     in->parentHandle = handles[0];
432     result = TPM2B_DATA_Unmarshal(&in->encryptionKey, paramBuffer, paramBufferSize);
433     EXIT_IF_ERROR_PLUS(RC_Import_encryptionKey);
434     result = TPM2B_PUBLIC_Unmarshal(
435         &in->objectPublic, paramBuffer, paramBufferSize, FALSE);
436     EXIT_IF_ERROR_PLUS(RC_Import_objectPublic);
437     result = TPM2B_PRIVATE_Unmarshal(&in->duplicate, paramBuffer, paramBufferSize);
438     EXIT_IF_ERROR_PLUS(RC_Import_duplicate);
439     result = TPM2B_ENCRYPTED_SECRET_Unmarshal(
440         &in->inSymSeed, paramBuffer, paramBufferSize);
441     EXIT_IF_ERROR_PLUS(RC_Import_inSymSeed);
442     result = TPMT_SYM_DEF_OBJECT_Unmarshal(
443         &in->symmetricAlg, paramBuffer, paramBufferSize, TRUE);
444     EXIT_IF_ERROR_PLUS(RC_Import_symmetricAlg);

```

```

445     if(*paramBufferSize != 0)
446     {
447         result = TPM_RC_SIZE;
448         goto Exit;
449     }
450     result = TPM2_Import(in, out);
451     rSize = sizeof(Import_Out);
452     *respParmSize += TPM2B_PRIVATE_Marshal(&out->outPrivate, responseBuffer, &rSize);
453     break;
454 }
455 #endif // CC_Import
456 #if CC_RSA_Encrypt
457 case TPM_CC_RSA_Encrypt:
458 {
459     RSA_Encrypt_In* in = (RSA_Encrypt_In*)MemoryGetInBuffer(sizeof(RSA_Encrypt_In));
460     RSA_Encrypt_Out* out =
461         (RSA_Encrypt_Out*)MemoryGetOutBuffer(sizeof(RSA_Encrypt_Out));
462     in->keyHandle = handles[0];
463     result =
464         TPM2B_PUBLIC_KEY_RSA_Unmarshal(&in->message, paramBuffer, paramBufferSize);
465     EXIT_IF_ERROR_PLUS(RC_RSA_Encrypt_message);
466     result =
467         TPMT_RSA_DECRYPT_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
468     EXIT_IF_ERROR_PLUS(RC_RSA_Encrypt_inScheme);
469     result = TPM2B_DATA_Unmarshal(&in->label, paramBuffer, paramBufferSize);
470     EXIT_IF_ERROR_PLUS(RC_RSA_Encrypt_label);
471     if(*paramBufferSize != 0)
472     {
473         result = TPM_RC_SIZE;
474         goto Exit;
475     }
476     result = TPM2_RSA_Encrypt(in, out);
477     rSize = sizeof(RSA_Encrypt_Out);
478     *respParmSize +=
479         TPM2B_PUBLIC_KEY_RSA_Marshal(&out->outData, responseBuffer, &rSize);
480     break;
481 }
482 #endif // CC_RSA_Encrypt
483 #if CC_RSA_Decrypt
484 case TPM_CC_RSA_Decrypt:
485 {
486     RSA_Decrypt_In* in = (RSA_Decrypt_In*)MemoryGetInBuffer(sizeof(RSA_Decrypt_In));
487     RSA_Decrypt_Out* out =
488         (RSA_Decrypt_Out*)MemoryGetOutBuffer(sizeof(RSA_Decrypt_Out));
489     in->keyHandle = handles[0];
490     result =
491         TPM2B_PUBLIC_KEY_RSA_Unmarshal(&in->cipherText, paramBuffer, paramBufferSize);
492     EXIT_IF_ERROR_PLUS(RC_RSA_Decrypt_cipherText);
493     result =
494         TPMT_RSA_DECRYPT_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
495     EXIT_IF_ERROR_PLUS(RC_RSA_Decrypt_inScheme);
496     result = TPM2B_DATA_Unmarshal(&in->label, paramBuffer, paramBufferSize);
497     EXIT_IF_ERROR_PLUS(RC_RSA_Decrypt_label);
498     if(*paramBufferSize != 0)
499     {
500         result = TPM_RC_SIZE;
501         goto Exit;
502     }
503     result = TPM2_RSA_Decrypt(in, out);
504     rSize = sizeof(RSA_Decrypt_Out);
505     *respParmSize +=
506         TPM2B_PUBLIC_KEY_RSA_Marshal(&out->message, responseBuffer, &rSize);
507     break;
508 }
509 #endif // CC_RSA_Decrypt
510 #if CC_ECDH_KeyGen

```

```

511 case TPM_CC_ECDH_KeyGen:
512 {
513     ECDH_KeyGen_In* in = (ECDH_KeyGen_In*)MemoryGetInBuffer(sizeof(ECDH_KeyGen_In));
514     ECDH_KeyGen_Out* out =
515         (ECDH_KeyGen_Out*)MemoryGetOutBuffer(sizeof(ECDH_KeyGen_Out));
516     in->keyHandle = handles[0];
517     if(*paramBufferSize != 0)
518     {
519         result = TPM_RC_SIZE;
520         goto Exit;
521     }
522     result = TPM2_ECDH_KeyGen(in, out);
523     rSize = sizeof(ECDH_KeyGen_Out);
524     *respParmSize += TPM2B_ECC_POINT_Marshal(&out->zPoint, responseBuffer, &rSize);
525     *respParmSize += TPM2B_ECC_POINT_Marshal(&out->pubPoint, responseBuffer, &rSize);
526     break;
527 }
528 #endif // CC_ECDH_KeyGen
529 #if CC_ECDH_ZGen
530 case TPM_CC_ECDH_ZGen:
531 {
532     ECDH_ZGen_In* in = (ECDH_ZGen_In*)MemoryGetInBuffer(sizeof(ECDH_ZGen_In));
533     ECDH_ZGen_Out* out = (ECDH_ZGen_Out*)MemoryGetOutBuffer(sizeof(ECDH_ZGen_Out));
534     in->keyHandle = handles[0];
535     result = TPM2B_ECC_POINT_Unmarshal(&in->inPoint, paramBuffer, paramBufferSize);
536     EXIT_IF_ERROR_PLUS(RC_ECDH_ZGen_inPoint);
537     if(*paramBufferSize != 0)
538     {
539         result = TPM_RC_SIZE;
540         goto Exit;
541     }
542     result = TPM2_ECDH_ZGen(in, out);
543     rSize = sizeof(ECDH_ZGen_Out);
544     *respParmSize += TPM2B_ECC_POINT_Marshal(&out->outPoint, responseBuffer, &rSize);
545     break;
546 }
547 #endif // CC_ECDH_ZGen
548 #if CC_ECC_Parameters
549 case TPM_CC_ECC_Parameters:
550 {
551     ECC_Parameters_In* in =
552         (ECC_Parameters_In*)MemoryGetInBuffer(sizeof(ECC_Parameters_In));
553     ECC_Parameters_Out* out =
554         (ECC_Parameters_Out*)MemoryGetOutBuffer(sizeof(ECC_Parameters_Out));
555     result =
556         TPMI_ECC_CURVE_Unmarshal(&in->curveID, paramBuffer, paramBufferSize, FALSE);
557     EXIT_IF_ERROR_PLUS(RC_ECC_Parameters_curveID);
558     if(*paramBufferSize != 0)
559     {
560         result = TPM_RC_SIZE;
561         goto Exit;
562     }
563     result = TPM2_ECC_Parameters(in, out);
564     rSize = sizeof(ECC_Parameters_Out);
565     *respParmSize +=
566         TPMS_ALGORITHM_DETAIL_ECC_Marshal(&out->parameters, responseBuffer, &rSize);
567     break;
568 }
569 #endif // CC_ECC_Parameters
570 #if CC_ZGen_2Phase
571 case TPM_CC_ZGen_2Phase:
572 {
573     ZGen_2Phase_In* in = (ZGen_2Phase_In*)MemoryGetInBuffer(sizeof(ZGen_2Phase_In));
574     ZGen_2Phase_Out* out =
575         (ZGen_2Phase_Out*)MemoryGetOutBuffer(sizeof(ZGen_2Phase_Out));
576     in->keyA = handles[0];

```

```

577     result = TPM2B_ECC_POINT_Unmarshal(&in->inQsB, paramBuffer, paramBufferSize);
578     EXIT_IF_ERROR_PLUS(RC_ZGen_2Phase_inQsB);
579     result = TPM2B_ECC_POINT_Unmarshal(&in->inQeB, paramBuffer, paramBufferSize);
580     EXIT_IF_ERROR_PLUS(RC_ZGen_2Phase_inQeB);
581     result = TPMI_ECC_KEY_EXCHANGE_Unmarshal(
582         &in->inScheme, paramBuffer, paramBufferSize, FALSE);
583     EXIT_IF_ERROR_PLUS(RC_ZGen_2Phase_inScheme);
584     result = UINT16_Unmarshal(&in->counter, paramBuffer, paramBufferSize);
585     EXIT_IF_ERROR_PLUS(RC_ZGen_2Phase_counter);
586     if(*paramBufferSize != 0)
587     {
588         result = TPM_RC_SIZE;
589         goto Exit;
590     }
591     result = TPM2_ZGen_2Phase(in, out);
592     rSize = sizeof(ZGen_2Phase_Out);
593     *respParmSize += TPM2B_ECC_POINT_Marshal(&out->outZ1, responseBuffer, &rSize);
594     *respParmSize += TPM2B_ECC_POINT_Marshal(&out->outZ2, responseBuffer, &rSize);
595     break;
596 }
597 #endif // CC_ZGen_2Phase
598 #if CC_ECC_Encrypt
599 case TPM_CC_ECC_Encrypt:
600 {
601     ECC_Encrypt_In* in = (ECC_Encrypt_In*)MemoryGetInBuffer(sizeof(ECC_Encrypt_In));
602     ECC_Encrypt_Out* out =
603         (ECC_Encrypt_Out*)MemoryGetOutBuffer(sizeof(ECC_Encrypt_Out));
604     in->keyHandle = handles[0];
605     result = TPM2B_MAX_BUFFER_Unmarshal(&in->plainText, paramBuffer, paramBufferSize);
606     EXIT_IF_ERROR_PLUS(RC_ECC_Encrypt_plainText);
607     result =
608         TPMT_KDF_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
609     EXIT_IF_ERROR_PLUS(RC_ECC_Encrypt_inScheme);
610     if(*paramBufferSize != 0)
611     {
612         result = TPM_RC_SIZE;
613         goto Exit;
614     }
615     result = TPM2_ECC_Encrypt(in, out);
616     rSize = sizeof(ECC_Encrypt_Out);
617     *respParmSize += TPM2B_ECC_POINT_Marshal(&out->C1, responseBuffer, &rSize);
618     *respParmSize += TPM2B_MAX_BUFFER_Marshal(&out->C2, responseBuffer, &rSize);
619     *respParmSize += TPM2B_DIGEST_Marshal(&out->C3, responseBuffer, &rSize);
620     break;
621 }
622 #endif // CC_ECC_Encrypt
623 #if CC_ECC_Decrypt
624 case TPM_CC_ECC_Decrypt:
625 {
626     ECC_Decrypt_In* in = (ECC_Decrypt_In*)MemoryGetInBuffer(sizeof(ECC_Decrypt_In));
627     ECC_Decrypt_Out* out =
628         (ECC_Decrypt_Out*)MemoryGetOutBuffer(sizeof(ECC_Decrypt_Out));
629     in->keyHandle = handles[0];
630     result = TPM2B_ECC_POINT_Unmarshal(&in->C1, paramBuffer, paramBufferSize);
631     EXIT_IF_ERROR_PLUS(RC_ECC_Decrypt_C1);
632     result = TPM2B_MAX_BUFFER_Unmarshal(&in->C2, paramBuffer, paramBufferSize);
633     EXIT_IF_ERROR_PLUS(RC_ECC_Decrypt_C2);
634     result = TPM2B_DIGEST_Unmarshal(&in->C3, paramBuffer, paramBufferSize);
635     EXIT_IF_ERROR_PLUS(RC_ECC_Decrypt_C3);
636     result =
637         TPMT_KDF_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
638     EXIT_IF_ERROR_PLUS(RC_ECC_Decrypt_inScheme);
639     if(*paramBufferSize != 0)
640     {
641         result = TPM_RC_SIZE;
642         goto Exit;

```



```

643     }
644     result = TPM2_ECC_Decrypt(in, out);
645     rSize = sizeof(ECC_Decrypt_Out);
646     *respParmSize +=
647         TPM2B_MAX_BUFFER_Marshal(&out->plainText, responseBuffer, &rSize);
648     break;
649 }
650 #endif // CC_ECC_Decrypt
651 #if CC_EncryptDecrypt
652 case TPM_CC_EncryptDecrypt:
653 {
654     EncryptDecrypt_In* in =
655         (EncryptDecrypt_In*)MemoryGetInBuffer(sizeof(EncryptDecrypt_In));
656     EncryptDecrypt_Out* out =
657         (EncryptDecrypt_Out*)MemoryGetOutBuffer(sizeof(EncryptDecrypt_Out));
658     in->keyHandle = handles[0];
659     result = TPMI_YES_NO_Unmarshal(&in->decrypt, paramBuffer, paramBufferSize);
660     EXIT_IF_ERROR_PLUS(RC_EncryptDecrypt_decrypt);
661     result =
662         TPMI_ALG_CIPHER_MODE_Unmarshal(&in->mode, paramBuffer, paramBufferSize, TRUE);
663     EXIT_IF_ERROR_PLUS(RC_EncryptDecrypt_mode);
664     result = TPM2B_IV_Unmarshal(&in->ivIn, paramBuffer, paramBufferSize);
665     EXIT_IF_ERROR_PLUS(RC_EncryptDecrypt_ivIn);
666     result = TPM2B_MAX_BUFFER_Unmarshal(&in->inData, paramBuffer, paramBufferSize);
667     EXIT_IF_ERROR_PLUS(RC_EncryptDecrypt_inData);
668     if(*paramBufferSize != 0)
669     {
670         result = TPM_RC_SIZE;
671         goto Exit;
672     }
673     result = TPM2_EncryptDecrypt(in, out);
674     rSize = sizeof(EncryptDecrypt_Out);
675     *respParmSize += TPM2B_MAX_BUFFER_Marshal(&out->outData, responseBuffer, &rSize);
676     *respParmSize += TPM2B_IV_Marshal(&out->ivOut, responseBuffer, &rSize);
677     break;
678 }
679 #endif // CC_EncryptDecrypt
680 #if CC_EncryptDecrypt2
681 case TPM_CC_EncryptDecrypt2:
682 {
683     EncryptDecrypt2_In* in =
684         (EncryptDecrypt2_In*)MemoryGetInBuffer(sizeof(EncryptDecrypt2_In));
685     EncryptDecrypt2_Out* out =
686         (EncryptDecrypt2_Out*)MemoryGetOutBuffer(sizeof(EncryptDecrypt2_Out));
687     in->keyHandle = handles[0];
688     result = TPM2B_MAX_BUFFER_Unmarshal(&in->inData, paramBuffer, paramBufferSize);
689     EXIT_IF_ERROR_PLUS(RC_EncryptDecrypt2_inData);
690     result = TPMI_YES_NO_Unmarshal(&in->decrypt, paramBuffer, paramBufferSize);
691     EXIT_IF_ERROR_PLUS(RC_EncryptDecrypt2_decrypt);
692     result =
693         TPMI_ALG_CIPHER_MODE_Unmarshal(&in->mode, paramBuffer, paramBufferSize, TRUE);
694     EXIT_IF_ERROR_PLUS(RC_EncryptDecrypt2_mode);
695     result = TPM2B_IV_Unmarshal(&in->ivIn, paramBuffer, paramBufferSize);
696     EXIT_IF_ERROR_PLUS(RC_EncryptDecrypt2_ivIn);
697     if(*paramBufferSize != 0)
698     {
699         result = TPM_RC_SIZE;
700         goto Exit;
701     }
702     result = TPM2_EncryptDecrypt2(in, out);
703     rSize = sizeof(EncryptDecrypt2_Out);
704     *respParmSize += TPM2B_MAX_BUFFER_Marshal(&out->outData, responseBuffer, &rSize);
705     *respParmSize += TPM2B_IV_Marshal(&out->ivOut, responseBuffer, &rSize);
706     break;
707 }
708 #endif // CC_EncryptDecrypt2

```



```

709 #if CC_Hash
710 case TPM_CC_Hash:
711 {
712     Hash_In* in = (Hash_In*)MemoryGetInBuffer(sizeof(Hash_In));
713     Hash_Out* out = (Hash_Out*)MemoryGetOutBuffer(sizeof(Hash_Out));
714     result = TPM2B_MAX_BUFFER_Unmarshal(&in->data, paramBuffer, paramBufferSize);
715     EXIT_IF_ERROR_PLUS(RC_Hash_data);
716     result =
717         TPMI_ALG_HASH_Unmarshal(&in->hashAlg, paramBuffer, paramBufferSize, FALSE);
718     EXIT_IF_ERROR_PLUS(RC_Hash_hashAlg);
719     result = TPMI_RH_HIERARCHY_Unmarshal(
720         &in->hierarchy, paramBuffer, paramBufferSize, TRUE);
721     EXIT_IF_ERROR_PLUS(RC_Hash_hierarchy);
722     if(*paramBufferSize != 0)
723     {
724         result = TPM_RC_SIZE;
725         goto Exit;
726     }
727     result = TPM2_Hash(in, out);
728     rSize = sizeof(Hash_Out);
729     *respParmSize += TPM2B_DIGEST_Marshal(&out->outHash, responseBuffer, &rSize);
730     *respParmSize +=
731         TPMT_TK_HASHCHECK_Marshal(&out->validation, responseBuffer, &rSize);
732     break;
733 }
734 #endif // CC_Hash
735 #if CC_HMAC
736 case TPM_CC_HMAC:
737 {
738     HMAC_In* in = (HMAC_In*)MemoryGetInBuffer(sizeof(HMAC_In));
739     HMAC_Out* out = (HMAC_Out*)MemoryGetOutBuffer(sizeof(HMAC_Out));
740     in->handle = handles[0];
741     result = TPM2B_MAX_BUFFER_Unmarshal(&in->buffer, paramBuffer, paramBufferSize);
742     EXIT_IF_ERROR_PLUS(RC_HMAC_buffer);
743     result =
744         TPMI_ALG_HASH_Unmarshal(&in->hashAlg, paramBuffer, paramBufferSize, TRUE);
745     EXIT_IF_ERROR_PLUS(RC_HMAC_hashAlg);
746     if(*paramBufferSize != 0)
747     {
748         result = TPM_RC_SIZE;
749         goto Exit;
750     }
751     result = TPM2_HMAC(in, out);
752     rSize = sizeof(HMAC_Out);
753     *respParmSize += TPM2B_DIGEST_Marshal(&out->outHMAC, responseBuffer, &rSize);
754     break;
755 }
756 #endif // CC_HMAC
757 #if CC_MAC
758 case TPM_CC_MAC:
759 {
760     MAC_In* in = (MAC_In*)MemoryGetInBuffer(sizeof(MAC_In));
761     MAC_Out* out = (MAC_Out*)MemoryGetOutBuffer(sizeof(MAC_Out));
762     in->handle = handles[0];
763     result = TPM2B_MAX_BUFFER_Unmarshal(&in->buffer, paramBuffer, paramBufferSize);
764     EXIT_IF_ERROR_PLUS(RC_MAC_buffer);
765     result = TPMI_ALG_MAC_SCHEME_Unmarshal(
766         &in->inScheme, paramBuffer, paramBufferSize, TRUE);
767     EXIT_IF_ERROR_PLUS(RC_MAC_inScheme);
768     if(*paramBufferSize != 0)
769     {
770         result = TPM_RC_SIZE;
771         goto Exit;
772     }
773     result = TPM2_MAC(in, out);
774     rSize = sizeof(MAC_Out);

```

```

775     *respParmSize += TPM2B_DIGEST_Marshal(&out->outMAC, responseBuffer, &rSize);
776     break;
777 }
778 #endif // CC_MAC
779 #if CC_GetRandom
780 case TPM_CC_GetRandom:
781 {
782     GetRandom_In* in = (GetRandom_In*)MemoryGetInBuffer(sizeof(GetRandom_In));
783     GetRandom_Out* out = (GetRandom_Out*)MemoryGetOutBuffer(sizeof(GetRandom_Out));
784     result = UINT16_Unmarshal(&in->bytesRequested, paramBuffer, paramBufferSize);
785     EXIT_IF_ERROR_PLUS(RC_GetRandom_bytesRequested);
786     if(*paramBufferSize != 0)
787     {
788         result = TPM_RC_SIZE;
789         goto Exit;
790     }
791     result = TPM2_GetRandom(in, out);
792     rSize = sizeof(GetRandom_Out);
793     *respParmSize += TPM2B_DIGEST_Marshal(&out->randomBytes, responseBuffer, &rSize);
794     break;
795 }
796 #endif // CC_GetRandom
797 #if CC_StirRandom
798 case TPM_CC_StirRandom:
799 {
800     StirRandom_In* in = (StirRandom_In*)MemoryGetInBuffer(sizeof(StirRandom_In));
801     result =
802         TPM2B_SENSITIVE_DATA_Unmarshal(&in->inData, paramBuffer, paramBufferSize);
803     EXIT_IF_ERROR_PLUS(RC_StirRandom_inData);
804     if(*paramBufferSize != 0)
805     {
806         result = TPM_RC_SIZE;
807         goto Exit;
808     }
809     result = TPM2_StirRandom(in);
810     break;
811 }
812 #endif // CC_StirRandom
813 #if CC_HMAC_Start
814 case TPM_CC_HMAC_Start:
815 {
816     HMAC_Start_In* in = (HMAC_Start_In*)MemoryGetInBuffer(sizeof(HMAC_Start_In));
817     HMAC_Start_Out* out = (HMAC_Start_Out*)MemoryGetOutBuffer(sizeof(HMAC_Start_Out));
818     in->handle = handles[0];
819     result = TPM2B_AUTH_Unmarshal(&in->auth, paramBuffer, paramBufferSize);
820     EXIT_IF_ERROR_PLUS(RC_HMAC_Start_auth);
821     result =
822         TPMI_ALG_HASH_Unmarshal(&in->hashAlg, paramBuffer, paramBufferSize, TRUE);
823     EXIT_IF_ERROR_PLUS(RC_HMAC_Start_hashAlg);
824     if(*paramBufferSize != 0)
825     {
826         result = TPM_RC_SIZE;
827         goto Exit;
828     }
829     result = TPM2_HMAC_Start(in, out);
830     rSize = sizeof(HMAC_Start_Out);
831     if(TPM_RC_SUCCESS != result)
832         goto Exit;
833     command->handles[command->handleNum++] = out->sequenceHandle;
834     break;
835 }
836 #endif // CC_HMAC_Start
837 #if CC_MAC_Start
838 case TPM_CC_MAC_Start:
839 {
840     MAC_Start_In* in = (MAC_Start_In*)MemoryGetInBuffer(sizeof(MAC_Start_In));

```

```

841     MAC_Start_Out* out = (MAC_Start_Out*)MemoryGetOutBuffer(sizeof(MAC_Start_Out));
842     in->handle = handles[0];
843     result = TPM2B_AUTH_Unmarshal(&in->auth, paramBuffer, paramBufferSize);
844     EXIT_IF_ERROR_PLUS(RC_MAC_Start_auth);
845     result = TPMI_ALG_MAC_SCHEME_Unmarshal(
846         &in->inScheme, paramBuffer, paramBufferSize, TRUE);
847     EXIT_IF_ERROR_PLUS(RC_MAC_Start_inScheme);
848     if(*paramBufferSize != 0)
849     {
850         result = TPM_RC_SIZE;
851         goto Exit;
852     }
853     result = TPM2_MAC_Start(in, out);
854     rSize = sizeof(MAC_Start_Out);
855     if(TPM_RC_SUCCESS != result)
856         goto Exit;
857     command->handles[command->handleNum++] = out->sequenceHandle;
858     break;
859 }
860 #endif // CC_MAC_Start
861 #if CC_HashSequenceStart
862 case TPM_CC_HashSequenceStart:
863 {
864     HashSequenceStart_In* in =
865         (HashSequenceStart_In*)MemoryGetInBuffer(sizeof(HashSequenceStart_In));
866     HashSequenceStart_Out* out =
867         (HashSequenceStart_Out*)MemoryGetOutBuffer(sizeof(HashSequenceStart_Out));
868     result = TPM2B_AUTH_Unmarshal(&in->auth, paramBuffer, paramBufferSize);
869     EXIT_IF_ERROR_PLUS(RC_HashSequenceStart_auth);
870     result =
871         TPMI_ALG_HASH_Unmarshal(&in->hashAlg, paramBuffer, paramBufferSize, TRUE);
872     EXIT_IF_ERROR_PLUS(RC_HashSequenceStart_hashAlg);
873     if(*paramBufferSize != 0)
874     {
875         result = TPM_RC_SIZE;
876         goto Exit;
877     }
878     result = TPM2_HashSequenceStart(in, out);
879     rSize = sizeof(HashSequenceStart_Out);
880     if(TPM_RC_SUCCESS != result)
881         goto Exit;
882     command->handles[command->handleNum++] = out->sequenceHandle;
883     break;
884 }
885 #endif // CC_HashSequenceStart
886 #if CC_SequenceUpdate
887 case TPM_CC_SequenceUpdate:
888 {
889     SequenceUpdate_In* in =
890         (SequenceUpdate_In*)MemoryGetInBuffer(sizeof(SequenceUpdate_In));
891     in->sequenceHandle = handles[0];
892     result = TPM2B_MAX_BUFFER_Unmarshal(&in->buffer, paramBuffer, paramBufferSize);
893     EXIT_IF_ERROR_PLUS(RC_SequenceUpdate_buffer);
894     if(*paramBufferSize != 0)
895     {
896         result = TPM_RC_SIZE;
897         goto Exit;
898     }
899     result = TPM2_SequenceUpdate(in);
900     break;
901 }
902 #endif // CC_SequenceUpdate
903 #if CC_SequenceComplete
904 case TPM_CC_SequenceComplete:
905 {
906     SequenceComplete_In* in =

```

```

907     (SequenceComplete_In*)MemoryGetInBuffer(sizeof(SequenceComplete_In));
908     SequenceComplete_Out* out =
909         (SequenceComplete_Out*)MemoryGetOutBuffer(sizeof(SequenceComplete_Out));
910     in->sequenceHandle = handles[0];
911     result = TPM2B_MAX_BUFFER_Unmarshal(&in->buffer, paramBuffer, paramBufferSize);
912     EXIT_IF_ERROR_PLUS(RC_SequenceComplete_buffer);
913     result = TPMI_RH_HIERARCHY_Unmarshal(
914         &in->hierarchy, paramBuffer, paramBufferSize, TRUE);
915     EXIT_IF_ERROR_PLUS(RC_SequenceComplete_hierarchy);
916     if(*paramBufferSize != 0)
917     {
918         result = TPM_RC_SIZE;
919         goto Exit;
920     }
921     result = TPM2_SequenceComplete(in, out);
922     rSize = sizeof(SequenceComplete_Out);
923     *respParmSize += TPM2B_DIGEST_Marshal(&out->result, responseBuffer, &rSize);
924     *respParmSize +=
925         TPMT_TK_HASHCHECK_Marshal(&out->validation, responseBuffer, &rSize);
926     break;
927 }
928 #endif // CC_SequenceComplete
929 #if CC_EventSequenceComplete
930 case TPM_CC_EventSequenceComplete:
931 {
932     EventSequenceComplete_In* in = (EventSequenceComplete_In*)MemoryGetInBuffer(
933         sizeof(EventSequenceComplete_In));
934     EventSequenceComplete_Out* out = (EventSequenceComplete_Out*)MemoryGetOutBuffer(
935         sizeof(EventSequenceComplete_Out));
936     in->pcrHandle = handles[0];
937     in->sequenceHandle = handles[1];
938     result = TPM2B_MAX_BUFFER_Unmarshal(&in->buffer, paramBuffer, paramBufferSize);
939     EXIT_IF_ERROR_PLUS(RC_EventSequenceComplete_buffer);
940     if(*paramBufferSize != 0)
941     {
942         result = TPM_RC_SIZE;
943         goto Exit;
944     }
945     result = TPM2_EventSequenceComplete(in, out);
946     rSize = sizeof(EventSequenceComplete_Out);
947     *respParmSize +=
948         TPML_DIGEST_VALUES_Marshal(&out->results, responseBuffer, &rSize);
949     break;
950 }
951 #endif // CC_EventSequenceComplete
952 #if CC_Certify
953 case TPM_CC_Certify:
954 {
955     Certify_In* in = (Certify_In*)MemoryGetInBuffer(sizeof(Certify_In));
956     Certify_Out* out = (Certify_Out*)MemoryGetOutBuffer(sizeof(Certify_Out));
957     in->objectHandle = handles[0];
958     in->signHandle = handles[1];
959     result = TPM2B_DATA_Unmarshal(&in->qualifyingData, paramBuffer, paramBufferSize);
960     EXIT_IF_ERROR_PLUS(RC_Certify_qualifyingData);
961     result =
962         TPMT_SIG_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
963     EXIT_IF_ERROR_PLUS(RC_Certify_inScheme);
964     if(*paramBufferSize != 0)
965     {
966         result = TPM_RC_SIZE;
967         goto Exit;
968     }
969     result = TPM2_Certify(in, out);
970     rSize = sizeof(Certify_Out);
971     *respParmSize += TPM2B_ATTEST_Marshal(&out->certifyInfo, responseBuffer, &rSize);
972     *respParmSize += TPMT_SIGNATURE_Marshal(&out->signature, responseBuffer, &rSize);

```

```

973     break;
974 }
975 #endif // CC_Certify
976 #if CC_CertifyCreation
977 case TPM_CC_CertifyCreation:
978 {
979     CertifyCreation_In* in =
980         (CertifyCreation_In*)MemoryGetInBuffer(sizeof(CertifyCreation_In));
981     CertifyCreation_Out* out =
982         (CertifyCreation_Out*)MemoryGetOutBuffer(sizeof(CertifyCreation_Out));
983     in->signHandle = handles[0];
984     in->objectHandle = handles[1];
985     result = TPM2B_DATA_Unmarshal(&in->qualifyingData, paramBuffer, paramBufferSize);
986     EXIT_IF_ERROR_PLUS(RC_CertifyCreation_qualifyingData);
987     result = TPM2B_DIGEST_Unmarshal(&in->creationHash, paramBuffer, paramBufferSize);
988     EXIT_IF_ERROR_PLUS(RC_CertifyCreation_creationHash);
989     result =
990         TPMT_SIG_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
991     EXIT_IF_ERROR_PLUS(RC_CertifyCreation_inScheme);
992     result =
993         TPMT_TK_CREATION_Unmarshal(&in->creationTicket, paramBuffer, paramBufferSize);
994     EXIT_IF_ERROR_PLUS(RC_CertifyCreation_creationTicket);
995     if(*paramBufferSize != 0)
996     {
997         result = TPM_RC_SIZE;
998         goto Exit;
999     }
1000     result = TPM2_CertifyCreation(in, out);
1001     rSize = sizeof(CertifyCreation_Out);
1002     *respParmSize += TPM2B_ATTEST_Marshal(&out->certifyInfo, responseBuffer, &rSize);
1003     *respParmSize += TPMT_SIGNATURE_Marshal(&out->signature, responseBuffer, &rSize);
1004     break;
1005 }
1006 #endif // CC_CertifyCreation
1007 #if CC_Quote
1008 case TPM_CC_Quote:
1009 {
1010     Quote_In* in = (Quote_In*)MemoryGetInBuffer(sizeof(Quote_In));
1011     Quote_Out* out = (Quote_Out*)MemoryGetOutBuffer(sizeof(Quote_Out));
1012     in->signHandle = handles[0];
1013     result = TPM2B_DATA_Unmarshal(&in->qualifyingData, paramBuffer, paramBufferSize);
1014     EXIT_IF_ERROR_PLUS(RC_Quote_qualifyingData);
1015     result =
1016         TPMT_SIG_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
1017     EXIT_IF_ERROR_PLUS(RC_Quote_inScheme);
1018     result =
1019         TPML_PCR_SELECTION_Unmarshal(&in->PCRselect, paramBuffer, paramBufferSize);
1020     EXIT_IF_ERROR_PLUS(RC_Quote_PCRselect);
1021     if(*paramBufferSize != 0)
1022     {
1023         result = TPM_RC_SIZE;
1024         goto Exit;
1025     }
1026     result = TPM2_Quote(in, out);
1027     rSize = sizeof(Quote_Out);
1028     *respParmSize += TPM2B_ATTEST_Marshal(&out->quoted, responseBuffer, &rSize);
1029     *respParmSize += TPMT_SIGNATURE_Marshal(&out->signature, responseBuffer, &rSize);
1030     break;
1031 }
1032 #endif // CC_Quote
1033 #if CC_GetSessionAuditDigest
1034 case TPM_CC_GetSessionAuditDigest:
1035 {
1036     GetSessionAuditDigest_In* in = (GetSessionAuditDigest_In*)MemoryGetInBuffer(
1037         sizeof(GetSessionAuditDigest_In));
1038     GetSessionAuditDigest_Out* out = (GetSessionAuditDigest_Out*)MemoryGetOutBuffer(

```



```

1039     sizeof(GetSessionAuditDigest_Out));
1040     in->privacyAdminHandle = handles[0];
1041     in->signHandle         = handles[1];
1042     in->sessionHandle       = handles[2];
1043     result = TPM2B_DATA_Unmarshal(&in->qualifyingData, paramBuffer, paramBufferSize);
1044     EXIT_IF_ERROR_PLUS(RC_GetSessionAuditDigest_qualifyingData);
1045     result =
1046         TPMT_SIG_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
1047     EXIT_IF_ERROR_PLUS(RC_GetSessionAuditDigest_inScheme);
1048     if(*paramBufferSize != 0)
1049     {
1050         result = TPM_RC_SIZE;
1051         goto Exit;
1052     }
1053     result = TPM2_GetSessionAuditDigest(in, out);
1054     rSize = sizeof(GetSessionAuditDigest_Out);
1055     *respParmSize += TPM2B_ATTEST_Marshal(&out->auditInfo, responseBuffer, &rSize);
1056     *respParmSize += TPMT_SIGNATURE_Marshal(&out->signature, responseBuffer, &rSize);
1057     break;
1058 }
1059 #endif // CC_GetSessionAuditDigest
1060 #if CC_GetCommandAuditDigest
1061 case TPM_CC_GetCommandAuditDigest:
1062 {
1063     GetCommandAuditDigest_In* in = (GetCommandAuditDigest_In*)MemoryGetInBuffer(
1064         sizeof(GetCommandAuditDigest_In));
1065     GetCommandAuditDigest_Out* out = (GetCommandAuditDigest_Out*)MemoryGetOutBuffer(
1066         sizeof(GetCommandAuditDigest_Out));
1067     in->privacyHandle = handles[0];
1068     in->signHandle    = handles[1];
1069     result = TPM2B_DATA_Unmarshal(&in->qualifyingData, paramBuffer, paramBufferSize);
1070     EXIT_IF_ERROR_PLUS(RC_GetCommandAuditDigest_qualifyingData);
1071     result =
1072         TPMT_SIG_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
1073     EXIT_IF_ERROR_PLUS(RC_GetCommandAuditDigest_inScheme);
1074     if(*paramBufferSize != 0)
1075     {
1076         result = TPM_RC_SIZE;
1077         goto Exit;
1078     }
1079     result = TPM2_GetCommandAuditDigest(in, out);
1080     rSize = sizeof(GetCommandAuditDigest_Out);
1081     *respParmSize += TPM2B_ATTEST_Marshal(&out->auditInfo, responseBuffer, &rSize);
1082     *respParmSize += TPMT_SIGNATURE_Marshal(&out->signature, responseBuffer, &rSize);
1083     break;
1084 }
1085 #endif // CC_GetCommandAuditDigest
1086 #if CC_GetTime
1087 case TPM_CC_GetTime:
1088 {
1089     GetTime_In* in = (GetTime_In*)MemoryGetInBuffer(sizeof(GetTime_In));
1090     GetTime_Out* out = (GetTime_Out*)MemoryGetOutBuffer(sizeof(GetTime_Out));
1091     in->privacyAdminHandle = handles[0];
1092     in->signHandle         = handles[1];
1093     result = TPM2B_DATA_Unmarshal(&in->qualifyingData, paramBuffer, paramBufferSize);
1094     EXIT_IF_ERROR_PLUS(RC_GetTime_qualifyingData);
1095     result =
1096         TPMT_SIG_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
1097     EXIT_IF_ERROR_PLUS(RC_GetTime_inScheme);
1098     if(*paramBufferSize != 0)
1099     {
1100         result = TPM_RC_SIZE;
1101         goto Exit;
1102     }
1103     result = TPM2_GetTime(in, out);
1104     rSize = sizeof(GetTime_Out);

```



```

1105     *respParamSize += TPM2B_ATTEST_Marshal(&out->timeInfo, responseBuffer, &rSize);
1106     *respParamSize += TPMT_SIGNATURE_Marshal(&out->signature, responseBuffer, &rSize);
1107     break;
1108 }
1109 #endif // CC_GetTime
1110 #if CC_CertifyX509
1111 case TPM_CC_CertifyX509:
1112 {
1113     CertifyX509_In* in = (CertifyX509_In*)MemoryGetInBuffer(sizeof(CertifyX509_In));
1114     CertifyX509_Out* out =
1115         (CertifyX509_Out*)MemoryGetOutBuffer(sizeof(CertifyX509_Out));
1116     in->objectHandle = handles[0];
1117     in->signHandle = handles[1];
1118     result = TPM2B_DATA_Unmarshal(&in->reserved, paramBuffer, paramBufferSize);
1119     EXIT_IF_ERROR_PLUS(RC_CertifyX509_reserved);
1120     result =
1121         TPMT_SIG_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
1122     EXIT_IF_ERROR_PLUS(RC_CertifyX509_inScheme);
1123     result = TPM2B_MAX_BUFFER_Unmarshal(
1124         &in->partialCertificate, paramBuffer, paramBufferSize);
1125     EXIT_IF_ERROR_PLUS(RC_CertifyX509_partialCertificate);
1126     if(*paramBufferSize != 0)
1127     {
1128         result = TPM_RC_SIZE;
1129         goto Exit;
1130     }
1131     result = TPM2_CertifyX509(in, out);
1132     rSize = sizeof(CertifyX509_Out);
1133     *respParamSize +=
1134         TPM2B_MAX_BUFFER_Marshal(&out->addedToCertificate, responseBuffer, &rSize);
1135     *respParamSize += TPM2B_DIGEST_Marshal(&out->tbsDigest, responseBuffer, &rSize);
1136     *respParamSize += TPMT_SIGNATURE_Marshal(&out->signature, responseBuffer, &rSize);
1137     break;
1138 }
1139 #endif // CC_CertifyX509
1140 #if CC_Commit
1141 case TPM_CC_Commit:
1142 {
1143     Commit_In* in = (Commit_In*)MemoryGetInBuffer(sizeof(Commit_In));
1144     Commit_Out* out = (Commit_Out*)MemoryGetOutBuffer(sizeof(Commit_Out));
1145     in->signHandle = handles[0];
1146     result = TPM2B_ECC_POINT_Unmarshal(&in->P1, paramBuffer, paramBufferSize);
1147     EXIT_IF_ERROR_PLUS(RC_Commit_P1);
1148     result = TPM2B_SENSITIVE_DATA_Unmarshal(&in->s2, paramBuffer, paramBufferSize);
1149     EXIT_IF_ERROR_PLUS(RC_Commit_s2);
1150     result = TPM2B_ECC_PARAMETER_Unmarshal(&in->y2, paramBuffer, paramBufferSize);
1151     EXIT_IF_ERROR_PLUS(RC_Commit_y2);
1152     if(*paramBufferSize != 0)
1153     {
1154         result = TPM_RC_SIZE;
1155         goto Exit;
1156     }
1157     result = TPM2_Commit(in, out);
1158     rSize = sizeof(Commit_Out);
1159     *respParamSize += TPM2B_ECC_POINT_Marshal(&out->K, responseBuffer, &rSize);
1160     *respParamSize += TPM2B_ECC_POINT_Marshal(&out->L, responseBuffer, &rSize);
1161     *respParamSize += TPM2B_ECC_POINT_Marshal(&out->E, responseBuffer, &rSize);
1162     *respParamSize += UINT16_Marshal(&out->counter, responseBuffer, &rSize);
1163     break;
1164 }
1165 #endif // CC_Commit
1166 #if CC_EC_Ephemeral
1167 case TPM_CC_EC_Ephemeral:
1168 {
1169     EC_Ephemeral_In* in =
1170         (EC_Ephemeral_In*)MemoryGetInBuffer(sizeof(EC_Ephemeral_In));

```

```

1171     EC_Ephemeral_Out* out =
1172         (EC_Ephemeral_Out*)MemoryGetOutBuffer(sizeof(EC_Ephemeral_Out));
1173     result =
1174         TPMI_ECC_CURVE_Unmarshal(&in->curveID, paramBuffer, paramBufferSize, FALSE);
1175     EXIT_IF_ERROR_PLUS(RC_EC_Ephemeral_curveID);
1176     if(*paramBufferSize != 0)
1177     {
1178         result = TPM_RC_SIZE;
1179         goto Exit;
1180     }
1181     result = TPM2_EC_Ephemeral(in, out);
1182     rSize = sizeof(EC_Ephemeral_Out);
1183     *respParmSize += TPM2B_ECC_POINT_Marshal(&out->Q, responseBuffer, &rSize);
1184     *respParmSize += UINT16_Marshal(&out->counter, responseBuffer, &rSize);
1185     break;
1186 }
1187 #endif // CC_EC_Ephemeral
1188 #if CC_VerifySignature
1189 case TPM_CC_VerifySignature:
1190 {
1191     VerifySignature_In* in =
1192         (VerifySignature_In*)MemoryGetInBuffer(sizeof(VerifySignature_In));
1193     VerifySignature_Out* out =
1194         (VerifySignature_Out*)MemoryGetOutBuffer(sizeof(VerifySignature_Out));
1195     in->keyHandle = handles[0];
1196     result = TPM2B_DIGEST_Unmarshal(&in->digest, paramBuffer, paramBufferSize);
1197     EXIT_IF_ERROR_PLUS(RC_VerifySignature_digest);
1198     result =
1199         TPMT_SIGNATURE_Unmarshal(&in->signature, paramBuffer, paramBufferSize, FALSE);
1200     EXIT_IF_ERROR_PLUS(RC_VerifySignature_signature);
1201     if(*paramBufferSize != 0)
1202     {
1203         result = TPM_RC_SIZE;
1204         goto Exit;
1205     }
1206     result = TPM2_VerifySignature(in, out);
1207     rSize = sizeof(VerifySignature_Out);
1208     *respParmSize +=
1209         TPMT_TK_VERIFIED_Marshal(&out->validation, responseBuffer, &rSize);
1210     break;
1211 }
1212 #endif // CC_VerifySignature
1213 #if CC_Sign
1214 case TPM_CC_Sign:
1215 {
1216     Sign_In* in = (Sign_In*)MemoryGetInBuffer(sizeof(Sign_In));
1217     Sign_Out* out = (Sign_Out*)MemoryGetOutBuffer(sizeof(Sign_Out));
1218     in->keyHandle = handles[0];
1219     result = TPM2B_DIGEST_Unmarshal(&in->digest, paramBuffer, paramBufferSize);
1220     EXIT_IF_ERROR_PLUS(RC_Sign_digest);
1221     result =
1222         TPMT_SIG_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
1223     EXIT_IF_ERROR_PLUS(RC_Sign_inScheme);
1224     result =
1225         TPMT_TK_HASHCHECK_Unmarshal(&in->validation, paramBuffer, paramBufferSize);
1226     EXIT_IF_ERROR_PLUS(RC_Sign_validation);
1227     if(*paramBufferSize != 0)
1228     {
1229         result = TPM_RC_SIZE;
1230         goto Exit;
1231     }
1232     result = TPM2_Sign(in, out);
1233     rSize = sizeof(Sign_Out);
1234     *respParmSize += TPMT_SIGNATURE_Marshal(&out->signature, responseBuffer, &rSize);
1235     break;
1236 }

```

```

1237 #endif // CC_Sign
1238 #if CC_SetCommandCodeAuditStatus
1239 case TPM_CC_SetCommandCodeAuditStatus:
1240 {
1241     SetCommandCodeAuditStatus_In* in =
1242         (SetCommandCodeAuditStatus_In*)MemoryGetInBuffer(
1243             sizeof(SetCommandCodeAuditStatus_In));
1244     in->auth = handles[0];
1245     result =
1246         TPMI_ALG_HASH_Unmarshal(&in->auditAlg, paramBuffer, paramBufferSize, TRUE);
1247     EXIT_IF_ERROR_PLUS(RC_SetCommandCodeAuditStatus_auditAlg);
1248     result = TPML_CC_Unmarshal(&in->setList, paramBuffer, paramBufferSize);
1249     EXIT_IF_ERROR_PLUS(RC_SetCommandCodeAuditStatus_setList);
1250     result = TPML_CC_Unmarshal(&in->clearList, paramBuffer, paramBufferSize);
1251     EXIT_IF_ERROR_PLUS(RC_SetCommandCodeAuditStatus_clearList);
1252     if(*paramBufferSize != 0)
1253     {
1254         result = TPM_RC_SIZE;
1255         goto Exit;
1256     }
1257     result = TPM2_SetCommandCodeAuditStatus(in);
1258     break;
1259 }
1260 #endif // CC_SetCommandCodeAuditStatus
1261 #if CC_PCR_Extend
1262 case TPM_CC_PCR_Extend:
1263 {
1264     PCR_Extend_In* in = (PCR_Extend_In*)MemoryGetInBuffer(sizeof(PCR_Extend_In));
1265     in->pcrHandle = handles[0];
1266     result = TPML_DIGEST_VALUES_Unmarshal(&in->digests, paramBuffer, paramBufferSize);
1267     EXIT_IF_ERROR_PLUS(RC_PCR_Extend_digests);
1268     if(*paramBufferSize != 0)
1269     {
1270         result = TPM_RC_SIZE;
1271         goto Exit;
1272     }
1273     result = TPM2_PCR_Extend(in);
1274     break;
1275 }
1276 #endif // CC_PCR_Extend
1277 #if CC_PCR_Event
1278 case TPM_CC_PCR_Event:
1279 {
1280     PCR_Event_In* in = (PCR_Event_In*)MemoryGetInBuffer(sizeof(PCR_Event_In));
1281     PCR_Event_Out* out = (PCR_Event_Out*)MemoryGetOutBuffer(sizeof(PCR_Event_Out));
1282     in->pcrHandle = handles[0];
1283     result = TPM2B_EVENT_Unmarshal(&in->eventData, paramBuffer, paramBufferSize);
1284     EXIT_IF_ERROR_PLUS(RC_PCR_Event_eventData);
1285     if(*paramBufferSize != 0)
1286     {
1287         result = TPM_RC_SIZE;
1288         goto Exit;
1289     }
1290     result = TPM2_PCR_Event(in, out);
1291     rSize = sizeof(PCR_Event_Out);
1292     *respParmSize +=
1293         TPML_DIGEST_VALUES_Marshal(&out->digests, responseBuffer, &rSize);
1294     break;
1295 }
1296 #endif // CC_PCR_Event
1297 #if CC_PCR_Read
1298 case TPM_CC_PCR_Read:
1299 {
1300     PCR_Read_In* in = (PCR_Read_In*)MemoryGetInBuffer(sizeof(PCR_Read_In));
1301     PCR_Read_Out* out = (PCR_Read_Out*)MemoryGetOutBuffer(sizeof(PCR_Read_Out));
1302     result = TPML_PCR_SELECTION_Unmarshal(

```

```

1303     &in->pcrSelectionIn, paramBuffer, paramBufferSize);
1304 EXIT_IF_ERROR_PLUS(RC_PCR_Read_pcrSelectionIn);
1305 if(*paramBufferSize != 0)
1306 {
1307     result = TPM_RC_SIZE;
1308     goto Exit;
1309 }
1310 result = TPM2_PCR_Read(in, out);
1311 rSize = sizeof(PCR_Read_Out);
1312 *respParmSize += UINT32_Marshal(&out->pcrUpdateCounter, responseBuffer, &rSize);
1313 *respParmSize +=
1314     TPML_PCR_SELECTION_Marshal(&out->pcrSelectionOut, responseBuffer, &rSize);
1315 *respParmSize += TPML_DIGEST_Marshal(&out->pcrValues, responseBuffer, &rSize);
1316 break;
1317 }
1318 #endif // CC_PCR_Read
1319 #if CC_PCR_Allocate
1320 case TPM_CC_PCR_Allocate:
1321 {
1322     PCR_Allocate_In* in =
1323         (PCR_Allocate_In*)MemoryGetInBuffer(sizeof(PCR_Allocate_In));
1324     PCR_Allocate_Out* out =
1325         (PCR_Allocate_Out*)MemoryGetOutBuffer(sizeof(PCR_Allocate_Out));
1326     in->authHandle = handles[0];
1327     result = TPML_PCR_SELECTION_Unmarshal(
1328         &in->pcrAllocation, paramBuffer, paramBufferSize);
1329     EXIT_IF_ERROR_PLUS(RC_PCR_Allocate_pcrAllocation);
1330     if(*paramBufferSize != 0)
1331     {
1332         result = TPM_RC_SIZE;
1333         goto Exit;
1334     }
1335     result = TPM2_PCR_Allocate(in, out);
1336     rSize = sizeof(PCR_Allocate_Out);
1337     *respParmSize +=
1338         TPMI_YES_NO_Marshal(&out->allocationSuccess, responseBuffer, &rSize);
1339     *respParmSize += UINT32_Marshal(&out->maxPCR, responseBuffer, &rSize);
1340     *respParmSize += UINT32_Marshal(&out->sizeNeeded, responseBuffer, &rSize);
1341     *respParmSize += UINT32_Marshal(&out->sizeAvailable, responseBuffer, &rSize);
1342     break;
1343 }
1344 #endif // CC_PCR_Allocate
1345 #if CC_PCR_SetAuthPolicy
1346 case TPM_CC_PCR_SetAuthPolicy:
1347 {
1348     PCR_SetAuthPolicy_In* in =
1349         (PCR_SetAuthPolicy_In*)MemoryGetInBuffer(sizeof(PCR_SetAuthPolicy_In));
1350     in->authHandle = handles[0];
1351     result = TPM2B_DIGEST_Unmarshal(&in->authPolicy, paramBuffer, paramBufferSize);
1352     EXIT_IF_ERROR_PLUS(RC_PCR_SetAuthPolicy_authPolicy);
1353     result =
1354         TPMI_ALG_HASH_Unmarshal(&in->hashAlg, paramBuffer, paramBufferSize, TRUE);
1355     EXIT_IF_ERROR_PLUS(RC_PCR_SetAuthPolicy_hashAlg);
1356     result = TPMI_DH_PCR_Unmarshal(&in->pcrNum, paramBuffer, paramBufferSize, FALSE);
1357     EXIT_IF_ERROR_PLUS(RC_PCR_SetAuthPolicy_pcrNum);
1358     if(*paramBufferSize != 0)
1359     {
1360         result = TPM_RC_SIZE;
1361         goto Exit;
1362     }
1363     result = TPM2_PCR_SetAuthPolicy(in);
1364     break;
1365 }
1366 #endif // CC_PCR_SetAuthPolicy
1367 #if CC_PCR_SetAuthValue
1368 case TPM_CC_PCR_SetAuthValue:

```

```

1369 {
1370     PCR_SetAuthValue_In* in =
1371         (PCR_SetAuthValue_In*)MemoryGetInBuffer(sizeof(PCR_SetAuthValue_In));
1372     in->pcrHandle = handles[0];
1373     result = TPM2B_DIGEST_Unmarshal(&in->auth, paramBuffer, paramBufferSize);
1374     EXIT_IF_ERROR_PLUS(RC_PCR_SetAuthValue_auth);
1375     if(*paramBufferSize != 0)
1376     {
1377         result = TPM_RC_SIZE;
1378         goto Exit;
1379     }
1380     result = TPM2_PCR_SetAuthValue(in);
1381     break;
1382 }
1383 #endif // CC_PCR_SetAuthValue
1384 #if CC_PCR_Reset
1385 case TPM_CC_PCR_Reset:
1386 {
1387     PCR_Reset_In* in = (PCR_Reset_In*)MemoryGetInBuffer(sizeof(PCR_Reset_In));
1388     in->pcrHandle = handles[0];
1389     if(*paramBufferSize != 0)
1390     {
1391         result = TPM_RC_SIZE;
1392         goto Exit;
1393     }
1394     result = TPM2_PCR_Reset(in);
1395     break;
1396 }
1397 #endif // CC_PCR_Reset
1398 #if CC_PolicySigned
1399 case TPM_CC_PolicySigned:
1400 {
1401     PolicySigned_In* in =
1402         (PolicySigned_In*)MemoryGetInBuffer(sizeof(PolicySigned_In));
1403     PolicySigned_Out* out =
1404         (PolicySigned_Out*)MemoryGetOutBuffer(sizeof(PolicySigned_Out));
1405     in->authObject = handles[0];
1406     in->policySession = handles[1];
1407     result = TPM2B_NONCE_Unmarshal(&in->nonceTPM, paramBuffer, paramBufferSize);
1408     EXIT_IF_ERROR_PLUS(RC_PolicySigned_nonceTPM);
1409     result = TPM2B_DIGEST_Unmarshal(&in->cpHashA, paramBuffer, paramBufferSize);
1410     EXIT_IF_ERROR_PLUS(RC_PolicySigned_cpHashA);
1411     result = TPM2B_NONCE_Unmarshal(&in->policyRef, paramBuffer, paramBufferSize);
1412     EXIT_IF_ERROR_PLUS(RC_PolicySigned_policyRef);
1413     result = INT32_Unmarshal(&in->expiration, paramBuffer, paramBufferSize);
1414     EXIT_IF_ERROR_PLUS(RC_PolicySigned_expiration);
1415     result = TPMT_SIGNATURE_Unmarshal(&in->auth, paramBuffer, paramBufferSize, FALSE);
1416     EXIT_IF_ERROR_PLUS(RC_PolicySigned_auth);
1417     if(*paramBufferSize != 0)
1418     {
1419         result = TPM_RC_SIZE;
1420         goto Exit;
1421     }
1422     result = TPM2_PolicySigned(in, out);
1423     rSize = sizeof(PolicySigned_Out);
1424     *respParmSize += TPM2B_TIMEOUT_Marshal(&out->timeout, responseBuffer, &rSize);
1425     *respParmSize += TPMT_TK_AUTH_Marshal(&out->policyTicket, responseBuffer, &rSize);
1426     break;
1427 }
1428 #endif // CC_PolicySigned
1429 #if CC_PolicySecret
1430 case TPM_CC_PolicySecret:
1431 {
1432     PolicySecret_In* in =
1433         (PolicySecret_In*)MemoryGetInBuffer(sizeof(PolicySecret_In));
1434     PolicySecret_Out* out =

```



```

1435     (PolicySecret_Out*)MemoryGetOutBuffer(sizeof(PolicySecret_Out));
1436     in->authHandle = handles[0];
1437     in->policySession = handles[1];
1438     result = TPM2B_NONCE_Unmarshal(&in->nonceTPM, paramBuffer, paramBufferSize);
1439     EXIT_IF_ERROR_PLUS(RC_PolicySecret_nonceTPM);
1440     result = TPM2B_DIGEST_Unmarshal(&in->cpHashA, paramBuffer, paramBufferSize);
1441     EXIT_IF_ERROR_PLUS(RC_PolicySecret_cpHashA);
1442     result = TPM2B_NONCE_Unmarshal(&in->policyRef, paramBuffer, paramBufferSize);
1443     EXIT_IF_ERROR_PLUS(RC_PolicySecret_policyRef);
1444     result = INT32_Unmarshal(&in->expiration, paramBuffer, paramBufferSize);
1445     EXIT_IF_ERROR_PLUS(RC_PolicySecret_expiration);
1446     if(*paramBufferSize != 0)
1447     {
1448         result = TPM_RC_SIZE;
1449         goto Exit;
1450     }
1451     result = TPM2_PolicySecret(in, out);
1452     rSize = sizeof(PolicySecret_Out);
1453     *respParmSize += TPM2B_TIMEOUT_Marshal(&out->timeout, responseBuffer, &rSize);
1454     *respParmSize += TPMT_TK_AUTH_Marshal(&out->policyTicket, responseBuffer, &rSize);
1455     break;
1456 }
1457 #endif // CC_PolicySecret
1458 #if CC_PolicyTicket
1459 case TPM_CC_PolicyTicket:
1460 {
1461     PolicyTicket_In* in =
1462         (PolicyTicket_In*)MemoryGetInBuffer(sizeof(PolicyTicket_In));
1463     in->policySession = handles[0];
1464     result = TPM2B_TIMEOUT_Unmarshal(&in->timeout, paramBuffer, paramBufferSize);
1465     EXIT_IF_ERROR_PLUS(RC_PolicyTicket_timeout);
1466     result = TPM2B_DIGEST_Unmarshal(&in->cpHashA, paramBuffer, paramBufferSize);
1467     EXIT_IF_ERROR_PLUS(RC_PolicyTicket_cpHashA);
1468     result = TPM2B_NONCE_Unmarshal(&in->policyRef, paramBuffer, paramBufferSize);
1469     EXIT_IF_ERROR_PLUS(RC_PolicyTicket_policyRef);
1470     result = TPM2B_NAME_Unmarshal(&in->authName, paramBuffer, paramBufferSize);
1471     EXIT_IF_ERROR_PLUS(RC_PolicyTicket_authName);
1472     result = TPMT_TK_AUTH_Unmarshal(&in->ticket, paramBuffer, paramBufferSize);
1473     EXIT_IF_ERROR_PLUS(RC_PolicyTicket_ticket);
1474     if(*paramBufferSize != 0)
1475     {
1476         result = TPM_RC_SIZE;
1477         goto Exit;
1478     }
1479     result = TPM2_PolicyTicket(in);
1480     break;
1481 }
1482 #endif // CC_PolicyTicket
1483 #if CC_PolicyOR
1484 case TPM_CC_PolicyOR:
1485 {
1486     PolicyOR_In* in = (PolicyOR_In*)MemoryGetInBuffer(sizeof(PolicyOR_In));
1487     in->policySession = handles[0];
1488     result = TPML_DIGEST_Unmarshal(&in->pHashList, paramBuffer, paramBufferSize);
1489     EXIT_IF_ERROR_PLUS(RC_PolicyOR_pHashList);
1490     if(*paramBufferSize != 0)
1491     {
1492         result = TPM_RC_SIZE;
1493         goto Exit;
1494     }
1495     result = TPM2_PolicyOR(in);
1496     break;
1497 }
1498 #endif // CC_PolicyOR
1499 #if CC_PolicyPCR
1500 case TPM_CC_PolicyPCR:

```



```

1501 {
1502     PolicyPCR_In* in = (PolicyPCR_In*)MemoryGetInBuffer(sizeof(PolicyPCR_In));
1503     in->policySession = handles[0];
1504     result = TPM2B_DIGEST_Unmarshal(&in->pcrDigest, paramBuffer, paramBufferSize);
1505     EXIT_IF_ERROR_PLUS(RC_PolicyPCR_pcrDigest);
1506     result = TPML_PCR_SELECTION_Unmarshal(&in->pcrs, paramBuffer, paramBufferSize);
1507     EXIT_IF_ERROR_PLUS(RC_PolicyPCR_pcrs);
1508     if(*paramBufferSize != 0)
1509     {
1510         result = TPM_RC_SIZE;
1511         goto Exit;
1512     }
1513     result = TPM2_PolicyPCR(in);
1514     break;
1515 }
1516 #endif // CC_PolicyPCR
1517 #if CC_PolicyLocality
1518 case TPM_CC_PolicyLocality:
1519 {
1520     PolicyLocality_In* in =
1521         (PolicyLocality_In*)MemoryGetInBuffer(sizeof(PolicyLocality_In));
1522     in->policySession = handles[0];
1523     result = TPMA_LOCALITY_Unmarshal(&in->locality, paramBuffer, paramBufferSize);
1524     EXIT_IF_ERROR_PLUS(RC_PolicyLocality_locality);
1525     if(*paramBufferSize != 0)
1526     {
1527         result = TPM_RC_SIZE;
1528         goto Exit;
1529     }
1530     result = TPM2_PolicyLocality(in);
1531     break;
1532 }
1533 #endif // CC_PolicyLocality
1534 #if CC_PolicyNV
1535 case TPM_CC_PolicyNV:
1536 {
1537     PolicyNV_In* in = (PolicyNV_In*)MemoryGetInBuffer(sizeof(PolicyNV_In));
1538     in->authHandle = handles[0];
1539     in->nvIndex = handles[1];
1540     in->policySession = handles[2];
1541     result = TPM2B_OPERAND_Unmarshal(&in->operandB, paramBuffer, paramBufferSize);
1542     EXIT_IF_ERROR_PLUS(RC_PolicyNV_operandB);
1543     result = UINT16_Unmarshal(&in->offset, paramBuffer, paramBufferSize);
1544     EXIT_IF_ERROR_PLUS(RC_PolicyNV_offset);
1545     result = TPM_EO_Unmarshal(&in->operation, paramBuffer, paramBufferSize);
1546     EXIT_IF_ERROR_PLUS(RC_PolicyNV_operation);
1547     if(*paramBufferSize != 0)
1548     {
1549         result = TPM_RC_SIZE;
1550         goto Exit;
1551     }
1552     result = TPM2_PolicyNV(in);
1553     break;
1554 }
1555 #endif // CC_PolicyNV
1556 #if CC_PolicyCounterTimer
1557 case TPM_CC_PolicyCounterTimer:
1558 {
1559     PolicyCounterTimer_In* in =
1560         (PolicyCounterTimer_In*)MemoryGetInBuffer(sizeof(PolicyCounterTimer_In));
1561     in->policySession = handles[0];
1562     result = TPM2B_OPERAND_Unmarshal(&in->operandB, paramBuffer, paramBufferSize);
1563     EXIT_IF_ERROR_PLUS(RC_PolicyCounterTimer_operandB);
1564     result = UINT16_Unmarshal(&in->offset, paramBuffer, paramBufferSize);
1565     EXIT_IF_ERROR_PLUS(RC_PolicyCounterTimer_offset);
1566     result = TPM_EO_Unmarshal(&in->operation, paramBuffer, paramBufferSize);

```

```

1567     EXIT_IF_ERROR_PLUS(RC_PolicyCounterTimer_operation);
1568     if(*paramBufferSize != 0)
1569     {
1570         result = TPM_RC_SIZE;
1571         goto Exit;
1572     }
1573     result = TPM2_PolicyCounterTimer(in);
1574     break;
1575 }
1576 #endif // CC_PolicyCounterTimer
1577 #if CC_PolicyCommandCode
1578 case TPM_CC_PolicyCommandCode:
1579 {
1580     PolicyCommandCode_In* in =
1581         (PolicyCommandCode_In*)MemoryGetInBuffer(sizeof(PolicyCommandCode_In));
1582     in->policySession = handles[0];
1583     result = TPM_CC_Unmarshal(&in->code, paramBuffer, paramBufferSize);
1584     EXIT_IF_ERROR_PLUS(RC_PolicyCommandCode_code);
1585     if(*paramBufferSize != 0)
1586     {
1587         result = TPM_RC_SIZE;
1588         goto Exit;
1589     }
1590     result = TPM2_PolicyCommandCode(in);
1591     break;
1592 }
1593 #endif // CC_PolicyCommandCode
1594 #if CC_PolicyPhysicalPresence
1595 case TPM_CC_PolicyPhysicalPresence:
1596 {
1597     PolicyPhysicalPresence_In* in = (PolicyPhysicalPresence_In*)MemoryGetInBuffer(
1598         sizeof(PolicyPhysicalPresence_In));
1599     in->policySession = handles[0];
1600     if(*paramBufferSize != 0)
1601     {
1602         result = TPM_RC_SIZE;
1603         goto Exit;
1604     }
1605     result = TPM2_PolicyPhysicalPresence(in);
1606     break;
1607 }
1608 #endif // CC_PolicyPhysicalPresence
1609 #if CC_PolicyCpHash
1610 case TPM_CC_PolicyCpHash:
1611 {
1612     PolicyCpHash_In* in =
1613         (PolicyCpHash_In*)MemoryGetInBuffer(sizeof(PolicyCpHash_In));
1614     in->policySession = handles[0];
1615     result = TPM2B_DIGEST_Unmarshal(&in->cpHashA, paramBuffer, paramBufferSize);
1616     EXIT_IF_ERROR_PLUS(RC_PolicyCpHash_cpHashA);
1617     if(*paramBufferSize != 0)
1618     {
1619         result = TPM_RC_SIZE;
1620         goto Exit;
1621     }
1622     result = TPM2_PolicyCpHash(in);
1623     break;
1624 }
1625 #endif // CC_PolicyCpHash
1626 #if CC_PolicyNameHash
1627 case TPM_CC_PolicyNameHash:
1628 {
1629     PolicyNameHash_In* in =
1630         (PolicyNameHash_In*)MemoryGetInBuffer(sizeof(PolicyNameHash_In));
1631     in->policySession = handles[0];
1632     result = TPM2B_DIGEST_Unmarshal(&in->nameHash, paramBuffer, paramBufferSize);

```

```

1633     EXIT_IF_ERROR_PLUS(RC_PolicyNameHash_nameHash);
1634     if(*paramBufferSize != 0)
1635     {
1636         result = TPM_RC_SIZE;
1637         goto Exit;
1638     }
1639     result = TPM2_PolicyNameHash(in);
1640     break;
1641 }
1642 #endif // CC_PolicyNameHash
1643 #if CC_PolicyDuplicationSelect
1644 case TPM_CC_PolicyDuplicationSelect:
1645 {
1646     PolicyDuplicationSelect_In* in = (PolicyDuplicationSelect_In*)MemoryGetInBuffer(
1647         sizeof(PolicyDuplicationSelect_In));
1648     in->policySession = handles[0];
1649     result = TPM2B_NAME_Unmarshal(&in->objectName, paramBuffer, paramBufferSize);
1650     EXIT_IF_ERROR_PLUS(RC_PolicyDuplicationSelect_objectName);
1651     result = TPM2B_NAME_Unmarshal(&in->newParentName, paramBuffer, paramBufferSize);
1652     EXIT_IF_ERROR_PLUS(RC_PolicyDuplicationSelect_newParentName);
1653     result = TPM1_YES_NO_Unmarshal(&in->includeObject, paramBuffer, paramBufferSize);
1654     EXIT_IF_ERROR_PLUS(RC_PolicyDuplicationSelect_includeObject);
1655     if(*paramBufferSize != 0)
1656     {
1657         result = TPM_RC_SIZE;
1658         goto Exit;
1659     }
1660     result = TPM2_PolicyDuplicationSelect(in);
1661     break;
1662 }
1663 #endif // CC_PolicyDuplicationSelect
1664 #if CC_PolicyAuthorize
1665 case TPM_CC_PolicyAuthorize:
1666 {
1667     PolicyAuthorize_In* in =
1668         (PolicyAuthorize_In*)MemoryGetInBuffer(sizeof(PolicyAuthorize_In));
1669     in->policySession = handles[0];
1670     result =
1671         TPM2B_DIGEST_Unmarshal(&in->approvedPolicy, paramBuffer, paramBufferSize);
1672     EXIT_IF_ERROR_PLUS(RC_PolicyAuthorize_approvedPolicy);
1673     result = TPM2B_NONCE_Unmarshal(&in->policyRef, paramBuffer, paramBufferSize);
1674     EXIT_IF_ERROR_PLUS(RC_PolicyAuthorize_policyRef);
1675     result = TPM2B_NAME_Unmarshal(&in->keySign, paramBuffer, paramBufferSize);
1676     EXIT_IF_ERROR_PLUS(RC_PolicyAuthorize_keySign);
1677     result =
1678         TPMT_TK_VERIFIED_Unmarshal(&in->checkTicket, paramBuffer, paramBufferSize);
1679     EXIT_IF_ERROR_PLUS(RC_PolicyAuthorize_checkTicket);
1680     if(*paramBufferSize != 0)
1681     {
1682         result = TPM_RC_SIZE;
1683         goto Exit;
1684     }
1685     result = TPM2_PolicyAuthorize(in);
1686     break;
1687 }
1688 #endif // CC_PolicyAuthorize
1689 #if CC_PolicyAuthValue
1690 case TPM_CC_PolicyAuthValue:
1691 {
1692     PolicyAuthValue_In* in =
1693         (PolicyAuthValue_In*)MemoryGetInBuffer(sizeof(PolicyAuthValue_In));
1694     in->policySession = handles[0];
1695     if(*paramBufferSize != 0)
1696     {
1697         result = TPM_RC_SIZE;
1698         goto Exit;

```

```

1699     }
1700     result = TPM2_PolicyAuthValue(in);
1701     break;
1702 }
1703 #endif // CC_PolicyAuthValue
1704 #if CC_PolicyPassword
1705 case TPM_CC_PolicyPassword:
1706 {
1707     PolicyPassword_In* in =
1708         (PolicyPassword_In*)MemoryGetInBuffer(sizeof(PolicyPassword_In));
1709     in->policySession = handles[0];
1710     if(*paramBufferSize != 0)
1711     {
1712         result = TPM_RC_SIZE;
1713         goto Exit;
1714     }
1715     result = TPM2_PolicyPassword(in);
1716     break;
1717 }
1718 #endif // CC_PolicyPassword
1719 #if CC_PolicyGetDigest
1720 case TPM_CC_PolicyGetDigest:
1721 {
1722     PolicyGetDigest_In* in =
1723         (PolicyGetDigest_In*)MemoryGetInBuffer(sizeof(PolicyGetDigest_In));
1724     PolicyGetDigest_Out* out =
1725         (PolicyGetDigest_Out*)MemoryGetOutBuffer(sizeof(PolicyGetDigest_Out));
1726     in->policySession = handles[0];
1727     if(*paramBufferSize != 0)
1728     {
1729         result = TPM_RC_SIZE;
1730         goto Exit;
1731     }
1732     result = TPM2_PolicyGetDigest(in, out);
1733     rSize = sizeof(PolicyGetDigest_Out);
1734     *respParmSize += TPM2B_DIGEST_Marshal(&out->policyDigest, responseBuffer, &rSize);
1735     break;
1736 }
1737 #endif // CC_PolicyGetDigest
1738 #if CC_PolicyNvWritten
1739 case TPM_CC_PolicyNvWritten:
1740 {
1741     PolicyNvWritten_In* in =
1742         (PolicyNvWritten_In*)MemoryGetInBuffer(sizeof(PolicyNvWritten_In));
1743     in->policySession = handles[0];
1744     result = TPMI_YES_NO_Unmarshal(&in->writtenSet, paramBuffer, paramBufferSize);
1745     EXIT_IF_ERROR_PLUS(RC_PolicyNvWritten_writtenSet);
1746     if(*paramBufferSize != 0)
1747     {
1748         result = TPM_RC_SIZE;
1749         goto Exit;
1750     }
1751     result = TPM2_PolicyNvWritten(in);
1752     break;
1753 }
1754 #endif // CC_PolicyNvWritten
1755 #if CC_PolicyTemplate
1756 case TPM_CC_PolicyTemplate:
1757 {
1758     PolicyTemplate_In* in =
1759         (PolicyTemplate_In*)MemoryGetInBuffer(sizeof(PolicyTemplate_In));
1760     in->policySession = handles[0];
1761     result = TPM2B_DIGEST_Unmarshal(&in->templateHash, paramBuffer, paramBufferSize);
1762     EXIT_IF_ERROR_PLUS(RC_PolicyTemplate_templateHash);
1763     if(*paramBufferSize != 0)
1764     {

```

```

1765         result = TPM_RC_SIZE;
1766         goto Exit;
1767     }
1768     result = TPM2_PolicyTemplate(in);
1769     break;
1770 }
1771 #endif // CC_PolicyTemplate
1772 #if CC_PolicyAuthorizeNV
1773 case TPM_CC_PolicyAuthorizeNV:
1774 {
1775     PolicyAuthorizeNV_In* in =
1776         (PolicyAuthorizeNV_In*)MemoryGetInBuffer(sizeof(PolicyAuthorizeNV_In));
1777     in->authHandle = handles[0];
1778     in->nvIndex = handles[1];
1779     in->policySession = handles[2];
1780     if(*paramBufferSize != 0)
1781     {
1782         result = TPM_RC_SIZE;
1783         goto Exit;
1784     }
1785     result = TPM2_PolicyAuthorizeNV(in);
1786     break;
1787 }
1788 #endif // CC_PolicyAuthorizeNV
1789 #if CC_PolicyCapability
1790 case TPM_CC_PolicyCapability:
1791 {
1792     PolicyCapability_In* in =
1793         (PolicyCapability_In*)MemoryGetInBuffer(sizeof(PolicyCapability_In));
1794     in->policySession = handles[0];
1795     result = TPM2B_OPERAND_Unmarshal(&in->operandB, paramBuffer, paramBufferSize);
1796     EXIT_IF_ERROR_PLUS(RC_PolicyCapability_operandB);
1797     result = UINT16_Unmarshal(&in->offset, paramBuffer, paramBufferSize);
1798     EXIT_IF_ERROR_PLUS(RC_PolicyCapability_offset);
1799     result = TPM_EO_Unmarshal(&in->operation, paramBuffer, paramBufferSize);
1800     EXIT_IF_ERROR_PLUS(RC_PolicyCapability_operation);
1801     result = TPM_CAP_Unmarshal(&in->capability, paramBuffer, paramBufferSize);
1802     EXIT_IF_ERROR_PLUS(RC_PolicyCapability_capability);
1803     result = UINT32_Unmarshal(&in->property, paramBuffer, paramBufferSize);
1804     EXIT_IF_ERROR_PLUS(RC_PolicyCapability_property);
1805     if(*paramBufferSize != 0)
1806     {
1807         result = TPM_RC_SIZE;
1808         goto Exit;
1809     }
1810     result = TPM2_PolicyCapability(in);
1811     break;
1812 }
1813 #endif // CC_PolicyCapability
1814 #if CC_PolicyParameters
1815 case TPM_CC_PolicyParameters:
1816 {
1817     PolicyParameters_In* in =
1818         (PolicyParameters_In*)MemoryGetInBuffer(sizeof(PolicyParameters_In));
1819     in->policySession = handles[0];
1820     result = TPM2B_DIGEST_Unmarshal(&in->pHash, paramBuffer, paramBufferSize);
1821     EXIT_IF_ERROR_PLUS(RC_PolicyParameters_pHash);
1822     if(*paramBufferSize != 0)
1823     {
1824         result = TPM_RC_SIZE;
1825         goto Exit;
1826     }
1827     result = TPM2_PolicyParameters(in);
1828     break;
1829 }
1830 #endif // CC_PolicyParameters

```

```

1831 #if CC_CreatePrimary
1832 case TPM_CC_CreatePrimary:
1833 {
1834     CreatePrimary_In* in =
1835         (CreatePrimary_In*)MemoryGetInBuffer(sizeof(CreatePrimary_In));
1836     CreatePrimary_Out* out =
1837         (CreatePrimary_Out*)MemoryGetOutBuffer(sizeof(CreatePrimary_Out));
1838     in->primaryHandle = handles[0];
1839     result = TPM2B_SENSITIVE_CREATE_Unmarshal(
1840         &in->inSensitive, paramBuffer, paramBufferSize);
1841     EXIT_IF_ERROR_PLUS(RC_CreatePrimary_inSensitive);
1842     result =
1843         TPM2B_PUBLIC_Unmarshal(&in->inPublic, paramBuffer, paramBufferSize, FALSE);
1844     EXIT_IF_ERROR_PLUS(RC_CreatePrimary_inPublic);
1845     result = TPM2B_DATA_Unmarshal(&in->outsideInfo, paramBuffer, paramBufferSize);
1846     EXIT_IF_ERROR_PLUS(RC_CreatePrimary_outsideInfo);
1847     result =
1848         TPML_PCR_SELECTION_Unmarshal(&in->creationPCR, paramBuffer, paramBufferSize);
1849     EXIT_IF_ERROR_PLUS(RC_CreatePrimary_creationPCR);
1850     if(*paramBufferSize != 0)
1851     {
1852         result = TPM_RC_SIZE;
1853         goto Exit;
1854     }
1855     result = TPM2_CreatePrimary(in, out);
1856     rSize = sizeof(CreatePrimary_Out);
1857     if(TPM_RC_SUCCESS != result)
1858         goto Exit;
1859     command->handles[command->handleNum++] = out->objectHandle;
1860     *respParamSize += TPM2B_PUBLIC_Marshal(&out->outPublic, responseBuffer, &rSize);
1861     *respParamSize +=
1862         TPM2B_CREATION_DATA_Marshal(&out->creationData, responseBuffer, &rSize);
1863     *respParamSize += TPM2B_DIGEST_Marshal(&out->creationHash, responseBuffer, &rSize);
1864     *respParamSize +=
1865         TPMT_TK_CREATION_Marshal(&out->creationTicket, responseBuffer, &rSize);
1866     *respParamSize += TPM2B_NAME_Marshal(&out->name, responseBuffer, &rSize);
1867     break;
1868 }
1869 #endif // CC_CreatePrimary
1870 #if CC_HierarchyControl
1871 case TPM_CC_HierarchyControl:
1872 {
1873     HierarchyControl_In* in =
1874         (HierarchyControl_In*)MemoryGetInBuffer(sizeof(HierarchyControl_In));
1875     in->authHandle = handles[0];
1876     result =
1877         TPMI_RH_ENABLES_Unmarshal(&in->enable, paramBuffer, paramBufferSize, FALSE);
1878     EXIT_IF_ERROR_PLUS(RC_HierarchyControl_enable);
1879     result = TPMI_YES_NO_Unmarshal(&in->state, paramBuffer, paramBufferSize);
1880     EXIT_IF_ERROR_PLUS(RC_HierarchyControl_state);
1881     if(*paramBufferSize != 0)
1882     {
1883         result = TPM_RC_SIZE;
1884         goto Exit;
1885     }
1886     result = TPM2_HierarchyControl(in);
1887     break;
1888 }
1889 #endif // CC_HierarchyControl
1890 #if CC_SetPrimaryPolicy
1891 case TPM_CC_SetPrimaryPolicy:
1892 {
1893     SetPrimaryPolicy_In* in =
1894         (SetPrimaryPolicy_In*)MemoryGetInBuffer(sizeof(SetPrimaryPolicy_In));
1895     in->authHandle = handles[0];
1896     result = TPM2B_DIGEST_Unmarshal(&in->authPolicy, paramBuffer, paramBufferSize);

```



```

1897     EXIT_IF_ERROR_PLUS(RC_SetPrimaryPolicy_authPolicy);
1898     result =
1899         TPMI_ALG_HASH_Unmarshal(&in->hashAlg, paramBuffer, paramBufferSize, TRUE);
1900     EXIT_IF_ERROR_PLUS(RC_SetPrimaryPolicy_hashAlg);
1901     if(*paramBufferSize != 0)
1902     {
1903         result = TPM_RC_SIZE;
1904         goto Exit;
1905     }
1906     result = TPM2_SetPrimaryPolicy(in);
1907     break;
1908 }
1909 #endif // CC_SetPrimaryPolicy
1910 #if CC_ChangePPS
1911 case TPM_CC_ChangePPS:
1912 {
1913     ChangePPS_In* in = (ChangePPS_In*)MemoryGetInBuffer(sizeof(ChangePPS_In));
1914     in->authHandle = handles[0];
1915     if(*paramBufferSize != 0)
1916     {
1917         result = TPM_RC_SIZE;
1918         goto Exit;
1919     }
1920     result = TPM2_ChangePPS(in);
1921     break;
1922 }
1923 #endif // CC_ChangePPS
1924 #if CC_ChangeEPS
1925 case TPM_CC_ChangeEPS:
1926 {
1927     ChangeEPS_In* in = (ChangeEPS_In*)MemoryGetInBuffer(sizeof(ChangeEPS_In));
1928     in->authHandle = handles[0];
1929     if(*paramBufferSize != 0)
1930     {
1931         result = TPM_RC_SIZE;
1932         goto Exit;
1933     }
1934     result = TPM2_ChangeEPS(in);
1935     break;
1936 }
1937 #endif // CC_ChangeEPS
1938 #if CC_Clear
1939 case TPM_CC_Clear:
1940 {
1941     Clear_In* in = (Clear_In*)MemoryGetInBuffer(sizeof(Clear_In));
1942     in->authHandle = handles[0];
1943     if(*paramBufferSize != 0)
1944     {
1945         result = TPM_RC_SIZE;
1946         goto Exit;
1947     }
1948     result = TPM2_Clear(in);
1949     break;
1950 }
1951 #endif // CC_Clear
1952 #if CC_ClearControl
1953 case TPM_CC_ClearControl:
1954 {
1955     ClearControl_In* in =
1956         (ClearControl_In*)MemoryGetInBuffer(sizeof(ClearControl_In));
1957     in->auth = handles[0];
1958     result = TPMI_YES_NO_Unmarshal(&in->disable, paramBuffer, paramBufferSize);
1959     EXIT_IF_ERROR_PLUS(RC_ClearControl_disable);
1960     if(*paramBufferSize != 0)
1961     {
1962         result = TPM_RC_SIZE;

```

```

1963         goto Exit;
1964     }
1965     result = TPM2_ClearControl(in);
1966     break;
1967 }
1968 #endif // CC_ClearControl
1969 #if CC_HierarchyChangeAuth
1970 case TPM_CC_HierarchyChangeAuth:
1971 {
1972     HierarchyChangeAuth_In* in =
1973         (HierarchyChangeAuth_In*)MemoryGetInBuffer(sizeof(HierarchyChangeAuth_In));
1974     in->authHandle = handles[0];
1975     result = TPM2B_AUTH_Unmarshal(&in->newAuth, paramBuffer, paramBufferSize);
1976     EXIT_IF_ERROR_PLUS(RC_HierarchyChangeAuth_newAuth);
1977     if(*paramBufferSize != 0)
1978     {
1979         result = TPM_RC_SIZE;
1980         goto Exit;
1981     }
1982     result = TPM2_HierarchyChangeAuth(in);
1983     break;
1984 }
1985 #endif // CC_HierarchyChangeAuth
1986 #if CC_DictionaryAttackLockReset
1987 case TPM_CC_DictionaryAttackLockReset:
1988 {
1989     DictionaryAttackLockReset_In* in =
1990         (DictionaryAttackLockReset_In*)MemoryGetInBuffer(
1991             sizeof(DictionaryAttackLockReset_In));
1992     in->lockHandle = handles[0];
1993     if(*paramBufferSize != 0)
1994     {
1995         result = TPM_RC_SIZE;
1996         goto Exit;
1997     }
1998     result = TPM2_DictionaryAttackLockReset(in);
1999     break;
2000 }
2001 #endif // CC_DictionaryAttackLockReset
2002 #if CC_DictionaryAttackParameters
2003 case TPM_CC_DictionaryAttackParameters:
2004 {
2005     DictionaryAttackParameters_In* in =
2006         (DictionaryAttackParameters_In*)MemoryGetInBuffer(
2007             sizeof(DictionaryAttackParameters_In));
2008     in->lockHandle = handles[0];
2009     result = UINT32_Unmarshal(&in->newMaxTries, paramBuffer, paramBufferSize);
2010     EXIT_IF_ERROR_PLUS(RC_DictionaryAttackParameters_newMaxTries);
2011     result = UINT32_Unmarshal(&in->newRecoveryTime, paramBuffer, paramBufferSize);
2012     EXIT_IF_ERROR_PLUS(RC_DictionaryAttackParameters_newRecoveryTime);
2013     result = UINT32_Unmarshal(&in->lockoutRecovery, paramBuffer, paramBufferSize);
2014     EXIT_IF_ERROR_PLUS(RC_DictionaryAttackParameters_lockoutRecovery);
2015     if(*paramBufferSize != 0)
2016     {
2017         result = TPM_RC_SIZE;
2018         goto Exit;
2019     }
2020     result = TPM2_DictionaryAttackParameters(in);
2021     break;
2022 }
2023 #endif // CC_DictionaryAttackParameters
2024 #if CC_PP_Commands
2025 case TPM_CC_PP_Commands:
2026 {
2027     PP_Commands_In* in = (PP_Commands_In*)MemoryGetInBuffer(sizeof(PP_Commands_In));
2028     in->auth = handles[0];

```

```

2029     result = TPML_CC_Unmarshal(&in->setList, paramBuffer, paramBufferSize);
2030     EXIT_IF_ERROR_PLUS(RC_PP_Commands_setList);
2031     result = TPML_CC_Unmarshal(&in->clearList, paramBuffer, paramBufferSize);
2032     EXIT_IF_ERROR_PLUS(RC_PP_Commands_clearList);
2033     if(*paramBufferSize != 0)
2034     {
2035         result = TPM_RC_SIZE;
2036         goto Exit;
2037     }
2038     result = TPM2_PP_Commands(in);
2039     break;
2040 }
2041 #endif // CC_PP_Commands
2042 #if CC_SetAlgorithmSet
2043 case TPM_CC_SetAlgorithmSet:
2044 {
2045     SetAlgorithmSet_In* in =
2046         (SetAlgorithmSet_In*)MemoryGetInBuffer(sizeof(SetAlgorithmSet_In));
2047     in->authHandle = handles[0];
2048     result = UINT32_Unmarshal(&in->algorithmSet, paramBuffer, paramBufferSize);
2049     EXIT_IF_ERROR_PLUS(RC_SetAlgorithmSet_algorithmSet);
2050     if(*paramBufferSize != 0)
2051     {
2052         result = TPM_RC_SIZE;
2053         goto Exit;
2054     }
2055     result = TPM2_SetAlgorithmSet(in);
2056     break;
2057 }
2058 #endif // CC_SetAlgorithmSet
2059 #if CC_FieldUpgradeStart
2060 case TPM_CC_FieldUpgradeStart:
2061 {
2062     FieldUpgradeStart_In* in =
2063         (FieldUpgradeStart_In*)MemoryGetInBuffer(sizeof(FieldUpgradeStart_In));
2064     in->authorization = handles[0];
2065     in->keyHandle = handles[1];
2066     result = TPM2B_DIGEST_Unmarshal(&in->fuDigest, paramBuffer, paramBufferSize);
2067     EXIT_IF_ERROR_PLUS(RC_FieldUpgradeStart_fuDigest);
2068     result = TPMT_SIGNATURE_Unmarshal(
2069         &in->manifestSignature, paramBuffer, paramBufferSize, FALSE);
2070     EXIT_IF_ERROR_PLUS(RC_FieldUpgradeStart_manifestSignature);
2071     if(*paramBufferSize != 0)
2072     {
2073         result = TPM_RC_SIZE;
2074         goto Exit;
2075     }
2076     result = TPM2_FieldUpgradeStart(in);
2077     break;
2078 }
2079 #endif // CC_FieldUpgradeStart
2080 #if CC_FieldUpgradeData
2081 case TPM_CC_FieldUpgradeData:
2082 {
2083     FieldUpgradeData_In* in =
2084         (FieldUpgradeData_In*)MemoryGetInBuffer(sizeof(FieldUpgradeData_In));
2085     FieldUpgradeData_Out* out =
2086         (FieldUpgradeData_Out*)MemoryGetOutBuffer(sizeof(FieldUpgradeData_Out));
2087     result = TPM2B_MAX_BUFFER_Unmarshal(&in->fuData, paramBuffer, paramBufferSize);
2088     EXIT_IF_ERROR_PLUS(RC_FieldUpgradeData_fuData);
2089     if(*paramBufferSize != 0)
2090     {
2091         result = TPM_RC_SIZE;
2092         goto Exit;
2093     }
2094     result = TPM2_FieldUpgradeData(in, out);

```

```

2095     rSize = sizeof(FieldUpgradeData_Out);
2096     *respParmSize += TPMT_HA_Marshal(&out->nextDigest, responseBuffer, &rSize);
2097     *respParmSize += TPMT_HA_Marshal(&out->firstDigest, responseBuffer, &rSize);
2098     break;
2099 }
2100 #endif // CC_FieldUpgradeData
2101 #if CC_FirmwareRead
2102 case TPM_CC_FirmwareRead:
2103 {
2104     FirmwareRead_In* in =
2105         (FirmwareRead_In*)MemoryGetInBuffer(sizeof(FirmwareRead_In));
2106     FirmwareRead_Out* out =
2107         (FirmwareRead_Out*)MemoryGetOutBuffer(sizeof(FirmwareRead_Out));
2108     result = UINT32_Unmarshal(&in->sequenceNumber, paramBuffer, paramBufferSize);
2109     EXIT_IF_ERROR_PLUS(RC_FirmwareRead_sequenceNumber);
2110     if(*paramBufferSize != 0)
2111     {
2112         result = TPM_RC_SIZE;
2113         goto Exit;
2114     }
2115     result = TPM2_FirmwareRead(in, out);
2116     rSize = sizeof(FirmwareRead_Out);
2117     *respParmSize += TPM2B_MAX_BUFFER_Marshal(&out->fuData, responseBuffer, &rSize);
2118     break;
2119 }
2120 #endif // CC_FirmwareRead
2121 #if CC_ContextSave
2122 case TPM_CC_ContextSave:
2123 {
2124     ContextSave_In* in = (ContextSave_In*)MemoryGetInBuffer(sizeof(ContextSave_In));
2125     ContextSave_Out* out =
2126         (ContextSave_Out*)MemoryGetOutBuffer(sizeof(ContextSave_Out));
2127     in->saveHandle = handles[0];
2128     if(*paramBufferSize != 0)
2129     {
2130         result = TPM_RC_SIZE;
2131         goto Exit;
2132     }
2133     result = TPM2_ContextSave(in, out);
2134     rSize = sizeof(ContextSave_Out);
2135     *respParmSize += TPMS_CONTEXT_Marshal(&out->context, responseBuffer, &rSize);
2136     break;
2137 }
2138 #endif // CC_ContextSave
2139 #if CC_ContextLoad
2140 case TPM_CC_ContextLoad:
2141 {
2142     ContextLoad_In* in = (ContextLoad_In*)MemoryGetInBuffer(sizeof(ContextLoad_In));
2143     ContextLoad_Out* out =
2144         (ContextLoad_Out*)MemoryGetOutBuffer(sizeof(ContextLoad_Out));
2145     result = TPMS_CONTEXT_Unmarshal(&in->context, paramBuffer, paramBufferSize);
2146     EXIT_IF_ERROR_PLUS(RC_ContextLoad_context);
2147     if(*paramBufferSize != 0)
2148     {
2149         result = TPM_RC_SIZE;
2150         goto Exit;
2151     }
2152     result = TPM2_ContextLoad(in, out);
2153     rSize = sizeof(ContextLoad_Out);
2154     if(TPM_RC_SUCCESS != result)
2155         goto Exit;
2156     command->handles[command->handleNum++] = out->loadedHandle;
2157     break;
2158 }
2159 #endif // CC_ContextLoad
2160 #if CC_FlushContext

```

```

2161 case TPM_CC_FlushContext:
2162 {
2163     FlushContext_In* in =
2164         (FlushContext_In*)MemoryGetInBuffer(sizeof(FlushContext_In));
2165     result =
2166         TPMI_DH_CONTEXT_Unmarshal(&in->flushHandle, paramBuffer, paramBufferSize);
2167     EXIT_IF_ERROR_PLUS(RC_FlushContext_flushHandle);
2168     if(*paramBufferSize != 0)
2169     {
2170         result = TPM_RC_SIZE;
2171         goto Exit;
2172     }
2173     result = TPM2_FlushContext(in);
2174     break;
2175 }
2176 #endif // CC_FlushContext
2177 #if CC_EvictControl
2178 case TPM_CC_EvictControl:
2179 {
2180     EvictControl_In* in =
2181         (EvictControl_In*)MemoryGetInBuffer(sizeof(EvictControl_In));
2182     in->auth = handles[0];
2183     in->objectHandle = handles[1];
2184     result = TPMI_DH_PERSISTENT_Unmarshal(
2185         &in->persistentHandle, paramBuffer, paramBufferSize);
2186     EXIT_IF_ERROR_PLUS(RC_EvictControl_persistentHandle);
2187     if(*paramBufferSize != 0)
2188     {
2189         result = TPM_RC_SIZE;
2190         goto Exit;
2191     }
2192     result = TPM2_EvictControl(in);
2193     break;
2194 }
2195 #endif // CC_EvictControl
2196 #if CC_ReadClock
2197 case TPM_CC_ReadClock:
2198 {
2199     ReadClock_Out* out = (ReadClock_Out*)MemoryGetOutBuffer(sizeof(ReadClock_Out));
2200     if(*paramBufferSize != 0)
2201     {
2202         result = TPM_RC_SIZE;
2203         goto Exit;
2204     }
2205     result = TPM2_ReadClock(out);
2206     rSize = sizeof(ReadClock_Out);
2207     *respParmSize +=
2208         TPMS_TIME_INFO_Marshal(&out->currentTime, responseBuffer, &rSize);
2209     break;
2210 }
2211 #endif // CC_ReadClock
2212 #if CC_ClockSet
2213 case TPM_CC_ClockSet:
2214 {
2215     ClockSet_In* in = (ClockSet_In*)MemoryGetInBuffer(sizeof(ClockSet_In));
2216     in->auth = handles[0];
2217     result = UINT64_Unmarshal(&in->newTime, paramBuffer, paramBufferSize);
2218     EXIT_IF_ERROR_PLUS(RC_ClockSet_newTime);
2219     if(*paramBufferSize != 0)
2220     {
2221         result = TPM_RC_SIZE;
2222         goto Exit;
2223     }
2224     result = TPM2_ClockSet(in);
2225     break;
2226 }

```

```

2227 #endif // CC_ClockSet
2228 #if CC_ClockRateAdjust
2229 case TPM_CC_ClockRateAdjust:
2230 {
2231     ClockRateAdjust_In* in =
2232         (ClockRateAdjust_In*)MemoryGetInBuffer(sizeof(ClockRateAdjust_In));
2233     in->auth = handles[0];
2234     result =
2235         TPM_CLOCK_ADJUST_Unmarshal(&in->rateAdjust, paramBuffer, paramBufferSize);
2236     EXIT_IF_ERROR_PLUS(RC_ClockRateAdjust_rateAdjust);
2237     if(*paramBufferSize != 0)
2238     {
2239         result = TPM_RC_SIZE;
2240         goto Exit;
2241     }
2242     result = TPM2_ClockRateAdjust(in);
2243     break;
2244 }
2245 #endif // CC_ClockRateAdjust
2246 #if CC_GetCapability
2247 case TPM_CC_GetCapability:
2248 {
2249     GetCapability_In* in =
2250         (GetCapability_In*)MemoryGetInBuffer(sizeof(GetCapability_In));
2251     GetCapability_Out* out =
2252         (GetCapability_Out*)MemoryGetOutBuffer(sizeof(GetCapability_Out));
2253     result = TPM_CAP_Unmarshal(&in->capability, paramBuffer, paramBufferSize);
2254     EXIT_IF_ERROR_PLUS(RC_GetCapability_capability);
2255     result = UINT32_Unmarshal(&in->property, paramBuffer, paramBufferSize);
2256     EXIT_IF_ERROR_PLUS(RC_GetCapability_property);
2257     result = UINT32_Unmarshal(&in->propertyCount, paramBuffer, paramBufferSize);
2258     EXIT_IF_ERROR_PLUS(RC_GetCapability_propertyCount);
2259     if(*paramBufferSize != 0)
2260     {
2261         result = TPM_RC_SIZE;
2262         goto Exit;
2263     }
2264     result = TPM2_GetCapability(in, out);
2265     rSize = sizeof(GetCapability_Out);
2266     *respParamSize += TPMT_YES_NO_Marshal(&out->moreData, responseBuffer, &rSize);
2267     *respParamSize +=
2268         TPMS_CAPABILITY_DATA_Marshal(&out->capabilityData, responseBuffer, &rSize);
2269     break;
2270 }
2271 #endif // CC_GetCapability
2272 #if CC_TestParms
2273 case TPM_CC_TestParms:
2274 {
2275     TestParms_In* in = (TestParms_In*)MemoryGetInBuffer(sizeof(TestParms_In));
2276     result =
2277         TPMT_PUBLIC_PARMS_Unmarshal(&in->parameters, paramBuffer, paramBufferSize);
2278     EXIT_IF_ERROR_PLUS(RC_TestParms_parameters);
2279     if(*paramBufferSize != 0)
2280     {
2281         result = TPM_RC_SIZE;
2282         goto Exit;
2283     }
2284     result = TPM2_TestParms(in);
2285     break;
2286 }
2287 #endif // CC_TestParms
2288 #if CC_NV_DefineSpace
2289 case TPM_CC_NV_DefineSpace:
2290 {
2291     NV_DefineSpace_In* in =
2292         (NV_DefineSpace_In*)MemoryGetInBuffer(sizeof(NV_DefineSpace_In));

```



```

2293     in->authHandle = handles[0];
2294     result         = TPM2B_AUTH_Unmarshal(&in->auth, paramBuffer, paramBufferSize);
2295     EXIT_IF_ERROR_PLUS(RC_NV_DefineSpace_auth);
2296     result = TPM2B_NV_PUBLIC_Unmarshal(&in->publicInfo, paramBuffer, paramBufferSize);
2297     EXIT_IF_ERROR_PLUS(RC_NV_DefineSpace_publicInfo);
2298     if(*paramBufferSize != 0)
2299     {
2300         result = TPM_RC_SIZE;
2301         goto Exit;
2302     }
2303     result = TPM2_NV_DefineSpace(in);
2304     break;
2305 }
2306 #endif // CC_NV_DefineSpace
2307 #if CC_NV_UndefineSpace
2308 case TPM_CC_NV_UndefineSpace:
2309 {
2310     NV_UndefineSpace_In* in =
2311         (NV_UndefineSpace_In*)MemoryGetInBuffer(sizeof(NV_UndefineSpace_In));
2312     in->authHandle = handles[0];
2313     in->nvIndex     = handles[1];
2314     if(*paramBufferSize != 0)
2315     {
2316         result = TPM_RC_SIZE;
2317         goto Exit;
2318     }
2319     result = TPM2_NV_UndefineSpace(in);
2320     break;
2321 }
2322 #endif // CC_NV_UndefineSpace
2323 #if CC_NV_UndefineSpaceSpecial
2324 case TPM_CC_NV_UndefineSpaceSpecial:
2325 {
2326     NV_UndefineSpaceSpecial_In* in = (NV_UndefineSpaceSpecial_In*)MemoryGetInBuffer(
2327         sizeof(NV_UndefineSpaceSpecial_In));
2328     in->nvIndex = handles[0];
2329     in->platform = handles[1];
2330     if(*paramBufferSize != 0)
2331     {
2332         result = TPM_RC_SIZE;
2333         goto Exit;
2334     }
2335     result = TPM2_NV_UndefineSpaceSpecial(in);
2336     break;
2337 }
2338 #endif // CC_NV_UndefineSpaceSpecial
2339 #if CC_NV_ReadPublic
2340 case TPM_CC_NV_ReadPublic:
2341 {
2342     NV_ReadPublic_In* in =
2343         (NV_ReadPublic_In*)MemoryGetInBuffer(sizeof(NV_ReadPublic_In));
2344     NV_ReadPublic_Out* out =
2345         (NV_ReadPublic_Out*)MemoryGetOutBuffer(sizeof(NV_ReadPublic_Out));
2346     in->nvIndex = handles[0];
2347     if(*paramBufferSize != 0)
2348     {
2349         result = TPM_RC_SIZE;
2350         goto Exit;
2351     }
2352     result = TPM2_NV_ReadPublic(in, out);
2353     rSize = sizeof(NV_ReadPublic_Out);
2354     *respParamSize += TPM2B_NV_PUBLIC_Marshal(&out->nvPublic, responseBuffer, &rSize);
2355     *respParamSize += TPM2B_NAME_Marshal(&out->nvName, responseBuffer, &rSize);
2356     break;
2357 }
2358 #endif // CC_NV_ReadPublic

```

```

2359 #if CC_NV_Write
2360 case TPM_CC_NV_Write:
2361 {
2362     NV_Write_In* in = (NV_Write_In*)MemoryGetInBuffer(sizeof(NV_Write_In));
2363     in->authHandle = handles[0];
2364     in->nvIndex = handles[1];
2365     result = TPM2B_MAX_NV_BUFFER_Unmarshal(&in->data, paramBuffer, paramBufferSize);
2366     EXIT_IF_ERROR_PLUS(RC_NV_Write_data);
2367     result = UINT16_Unmarshal(&in->offset, paramBuffer, paramBufferSize);
2368     EXIT_IF_ERROR_PLUS(RC_NV_Write_offset);
2369     if(*paramBufferSize != 0)
2370     {
2371         result = TPM_RC_SIZE;
2372         goto Exit;
2373     }
2374     result = TPM2_NV_Write(in);
2375     break;
2376 }
2377 #endif // CC_NV_Write
2378 #if CC_NV_Increment
2379 case TPM_CC_NV_Increment:
2380 {
2381     NV_Increment_In* in =
2382         (NV_Increment_In*)MemoryGetInBuffer(sizeof(NV_Increment_In));
2383     in->authHandle = handles[0];
2384     in->nvIndex = handles[1];
2385     if(*paramBufferSize != 0)
2386     {
2387         result = TPM_RC_SIZE;
2388         goto Exit;
2389     }
2390     result = TPM2_NV_Increment(in);
2391     break;
2392 }
2393 #endif // CC_NV_Increment
2394 #if CC_NV_Extend
2395 case TPM_CC_NV_Extend:
2396 {
2397     NV_Extend_In* in = (NV_Extend_In*)MemoryGetInBuffer(sizeof(NV_Extend_In));
2398     in->authHandle = handles[0];
2399     in->nvIndex = handles[1];
2400     result = TPM2B_MAX_NV_BUFFER_Unmarshal(&in->data, paramBuffer, paramBufferSize);
2401     EXIT_IF_ERROR_PLUS(RC_NV_Extend_data);
2402     if(*paramBufferSize != 0)
2403     {
2404         result = TPM_RC_SIZE;
2405         goto Exit;
2406     }
2407     result = TPM2_NV_Extend(in);
2408     break;
2409 }
2410 #endif // CC_NV_Extend
2411 #if CC_NV_SetBits
2412 case TPM_CC_NV_SetBits:
2413 {
2414     NV_SetBits_In* in = (NV_SetBits_In*)MemoryGetInBuffer(sizeof(NV_SetBits_In));
2415     in->authHandle = handles[0];
2416     in->nvIndex = handles[1];
2417     result = UINT64_Unmarshal(&in->bits, paramBuffer, paramBufferSize);
2418     EXIT_IF_ERROR_PLUS(RC_NV_SetBits_bits);
2419     if(*paramBufferSize != 0)
2420     {
2421         result = TPM_RC_SIZE;
2422         goto Exit;
2423     }
2424     result = TPM2_NV_SetBits(in);

```

```

2425     break;
2426 }
2427 #endif // CC_NV_SetBits
2428 #if CC_NV_WriteLock
2429 case TPM_CC_NV_WriteLock:
2430 {
2431     NV_WriteLock_In* in =
2432         (NV_WriteLock_In*)MemoryGetInBuffer(sizeof(NV_WriteLock_In));
2433     in->authHandle = handles[0];
2434     in->nvIndex     = handles[1];
2435     if(*paramBufferSize != 0)
2436     {
2437         result = TPM_RC_SIZE;
2438         goto Exit;
2439     }
2440     result = TPM2_NV_WriteLock(in);
2441     break;
2442 }
2443 #endif // CC_NV_WriteLock
2444 #if CC_NV_GlobalWriteLock
2445 case TPM_CC_NV_GlobalWriteLock:
2446 {
2447     NV_GlobalWriteLock_In* in =
2448         (NV_GlobalWriteLock_In*)MemoryGetInBuffer(sizeof(NV_GlobalWriteLock_In));
2449     in->authHandle = handles[0];
2450     if(*paramBufferSize != 0)
2451     {
2452         result = TPM_RC_SIZE;
2453         goto Exit;
2454     }
2455     result = TPM2_NV_GlobalWriteLock(in);
2456     break;
2457 }
2458 #endif // CC_NV_GlobalWriteLock
2459 #if CC_NV_Read
2460 case TPM_CC_NV_Read:
2461 {
2462     NV_Read_In* in = (NV_Read_In*)MemoryGetInBuffer(sizeof(NV_Read_In));
2463     NV_Read_Out* out = (NV_Read_Out*)MemoryGetOutBuffer(sizeof(NV_Read_Out));
2464     in->authHandle = handles[0];
2465     in->nvIndex     = handles[1];
2466     result = UINT16_Unmarshal(&in->size, paramBuffer, paramBufferSize);
2467     EXIT_IF_ERROR_PLUS(RC_NV_Read_size);
2468     result = UINT16_Unmarshal(&in->offset, paramBuffer, paramBufferSize);
2469     EXIT_IF_ERROR_PLUS(RC_NV_Read_offset);
2470     if(*paramBufferSize != 0)
2471     {
2472         result = TPM_RC_SIZE;
2473         goto Exit;
2474     }
2475     result = TPM2_NV_Read(in, out);
2476     rSize = sizeof(NV_Read_Out);
2477     *respParamSize += TPM2B_MAX_NV_BUFFER_Marshal(&out->data, responseBuffer, &rSize);
2478     break;
2479 }
2480 #endif // CC_NV_Read
2481 #if CC_NV_ReadLock
2482 case TPM_CC_NV_ReadLock:
2483 {
2484     NV_ReadLock_In* in = (NV_ReadLock_In*)MemoryGetInBuffer(sizeof(NV_ReadLock_In));
2485     in->authHandle = handles[0];
2486     in->nvIndex     = handles[1];
2487     if(*paramBufferSize != 0)
2488     {
2489         result = TPM_RC_SIZE;
2490         goto Exit;

```

```

2491     }
2492     result = TPM2_NV_ReadLock(in);
2493     break;
2494 }
2495 #endif // CC_NV_ReadLock
2496 #if CC_NV_ChangeAuth
2497 case TPM_CC_NV_ChangeAuth:
2498 {
2499     NV_ChangeAuth_In* in =
2500         (NV_ChangeAuth_In*)MemoryGetInBuffer(sizeof(NV_ChangeAuth_In));
2501     in->nvIndex = handles[0];
2502     result = TPM2B_AUTH_Unmarshal(&in->newAuth, paramBuffer, paramBufferSize);
2503     EXIT_IF_ERROR_PLUS(RC_NV_ChangeAuth_newAuth);
2504     if(*paramBufferSize != 0)
2505     {
2506         result = TPM_RC_SIZE;
2507         goto Exit;
2508     }
2509     result = TPM2_NV_ChangeAuth(in);
2510     break;
2511 }
2512 #endif // CC_NV_ChangeAuth
2513 #if CC_NV_Certify
2514 case TPM_CC_NV_Certify:
2515 {
2516     NV_Certify_In* in = (NV_Certify_In*)MemoryGetInBuffer(sizeof(NV_Certify_In));
2517     NV_Certify_Out* out = (NV_Certify_Out*)MemoryGetOutBuffer(sizeof(NV_Certify_Out));
2518     in->signHandle = handles[0];
2519     in->authHandle = handles[1];
2520     in->nvIndex = handles[2];
2521     result = TPM2B_DATA_Unmarshal(&in->qualifyingData, paramBuffer, paramBufferSize);
2522     EXIT_IF_ERROR_PLUS(RC_NV_Certify_qualifyingData);
2523     result =
2524         TPMT_SIG_SCHEME_Unmarshal(&in->inScheme, paramBuffer, paramBufferSize, TRUE);
2525     EXIT_IF_ERROR_PLUS(RC_NV_Certify_inScheme);
2526     result = UINT16_Unmarshal(&in->size, paramBuffer, paramBufferSize);
2527     EXIT_IF_ERROR_PLUS(RC_NV_Certify_size);
2528     result = UINT16_Unmarshal(&in->offset, paramBuffer, paramBufferSize);
2529     EXIT_IF_ERROR_PLUS(RC_NV_Certify_offset);
2530     if(*paramBufferSize != 0)
2531     {
2532         result = TPM_RC_SIZE;
2533         goto Exit;
2534     }
2535     result = TPM2_NV_Certify(in, out);
2536     rSize = sizeof(NV_Certify_Out);
2537     *respParmSize += TPM2B_ATTEST_Marshal(&out->certifyInfo, responseBuffer, &rSize);
2538     *respParmSize += TPMT_SIGNATURE_Marshal(&out->signature, responseBuffer, &rSize);
2539     break;
2540 }
2541 #endif // CC_NV_Certify
2542 #if CC_NV_DefineSpace2
2543 case TPM_CC_NV_DefineSpace2:
2544 {
2545     NV_DefineSpace2_In* in =
2546         (NV_DefineSpace2_In*)MemoryGetInBuffer(sizeof(NV_DefineSpace2_In));
2547     in->authHandle = handles[0];
2548     result = TPM2B_AUTH_Unmarshal(&in->auth, paramBuffer, paramBufferSize);
2549     EXIT_IF_ERROR_PLUS(RC_NV_DefineSpace2_auth);
2550     result =
2551         TPM2B_NV_PUBLIC_2_Unmarshal(&in->publicInfo, paramBuffer, paramBufferSize);
2552     EXIT_IF_ERROR_PLUS(RC_NV_DefineSpace2_publicInfo);
2553     if(*paramBufferSize != 0)
2554     {
2555         result = TPM_RC_SIZE;
2556         goto Exit;

```

```

2557     }
2558     result = TPM2_NV_DefineSpace2(in);
2559     break;
2560 }
2561 #endif // CC_NV_DefineSpace2
2562 #if CC_NV_ReadPublic2
2563 case TPM_CC_NV_ReadPublic2:
2564 {
2565     NV_ReadPublic2_In* in =
2566         (NV_ReadPublic2_In*)MemoryGetInBuffer(sizeof(NV_ReadPublic2_In));
2567     NV_ReadPublic2_Out* out =
2568         (NV_ReadPublic2_Out*)MemoryGetOutBuffer(sizeof(NV_ReadPublic2_Out));
2569     in->nvIndex = handles[0];
2570     if(*paramBufferSize != 0)
2571     {
2572         result = TPM_RC_SIZE;
2573         goto Exit;
2574     }
2575     result = TPM2_NV_ReadPublic2(in, out);
2576     rSize = sizeof(NV_ReadPublic2_Out);
2577     *respParmSize +=
2578         TPM2B_NV_PUBLIC_2_Marshal(&out->nvPublic, responseBuffer, &rSize);
2579     *respParmSize += TPM2B_NAME_Marshal(&out->nvName, responseBuffer, &rSize);
2580     break;
2581 }
2582 #endif // CC_NV_ReadPublic2
2583 #if CC_SetCapability
2584 case TPM_CC_SetCapability:
2585 {
2586     SetCapability_In* in =
2587         (SetCapability_In*)MemoryGetInBuffer(sizeof(SetCapability_In));
2588     SetCapability_Out* out =
2589         (SetCapability_Out*)MemoryGetOutBuffer(sizeof(SetCapability_Out));
2590     in->authHandle = handles[0];
2591     result = TPM2B_SET_CAPABILITY_DATA_Unmarshal(
2592         &in->setCapabilityData, paramBuffer, paramBufferSize);
2593     EXIT_IF_ERROR_PLUS(RC_SetCapability_setCapabilityData);
2594     if(*paramBufferSize != 0)
2595     {
2596         result = TPM_RC_SIZE;
2597         goto Exit;
2598     }
2599     result = TPM2_SetCapability(in);
2600     break;
2601 }
2602 #endif // CC_SetCapability
2603 #if CC_AC_GetCapability
2604 case TPM_CC_AC_GetCapability:
2605 {
2606     AC_GetCapability_In* in =
2607         (AC_GetCapability_In*)MemoryGetInBuffer(sizeof(AC_GetCapability_In));
2608     AC_GetCapability_Out* out =
2609         (AC_GetCapability_Out*)MemoryGetOutBuffer(sizeof(AC_GetCapability_Out));
2610     in->ac = handles[0];
2611     result = TPM_AT_Unmarshal(&in->capability, paramBuffer, paramBufferSize);
2612     EXIT_IF_ERROR_PLUS(RC_AC_GetCapability_capability);
2613     result = UINT32_Unmarshal(&in->count, paramBuffer, paramBufferSize);
2614     EXIT_IF_ERROR_PLUS(RC_AC_GetCapability_count);
2615     if(*paramBufferSize != 0)
2616     {
2617         result = TPM_RC_SIZE;
2618         goto Exit;
2619     }
2620     result = TPM2_AC_GetCapability(in, out);
2621     rSize = sizeof(AC_GetCapability_Out);
2622     *respParmSize += TPMT_YES_NO_Marshal(&out->moreData, responseBuffer, &rSize);

```

```

2623     *respParamSize +=
2624         TPML_AC_CAPABILITIES_Marshal(&out->capabilitiesData, responseBuffer, &rSize);
2625     break;
2626 }
2627 #endif // CC_AC_GetCapability
2628 #if CC_AC_Send
2629 case TPM_CC_AC_Send:
2630 {
2631     AC_Send_In* in = (AC_Send_In*)MemoryGetInBuffer(sizeof(AC_Send_In));
2632     AC_Send_Out* out = (AC_Send_Out*)MemoryGetOutBuffer(sizeof(AC_Send_Out));
2633     in->sendObject = handles[0];
2634     in->authHandle = handles[1];
2635     in->ac = handles[2];
2636     result = TPM2B_MAX_BUFFER_Unmarshal(&in->acDataIn, paramBuffer, paramBufferSize);
2637     EXIT_IF_ERROR_PLUS(RC_AC_Send_acDataIn);
2638     if(*paramBufferSize != 0)
2639     {
2640         result = TPM_RC_SIZE;
2641         goto Exit;
2642     }
2643     result = TPM2_AC_Send(in, out);
2644     rSize = sizeof(AC_Send_Out);
2645     *respParamSize += TPMS_AC_OUTPUT_Marshal(&out->acDataOut, responseBuffer, &rSize);
2646     break;
2647 }
2648 #endif // CC_AC_Send
2649 #if CC_Policy_AC_SendSelect
2650 case TPM_CC_Policy_AC_SendSelect:
2651 {
2652     Policy_AC_SendSelect_In* in =
2653         (Policy_AC_SendSelect_In*)MemoryGetInBuffer(sizeof(Policy_AC_SendSelect_In));
2654     in->policySession = handles[0];
2655     result = TPM2B_NAME_Unmarshal(&in->objectName, paramBuffer, paramBufferSize);
2656     EXIT_IF_ERROR_PLUS(RC_Policy_AC_SendSelect_objectName);
2657     result = TPM2B_NAME_Unmarshal(&in->authHandleName, paramBuffer, paramBufferSize);
2658     EXIT_IF_ERROR_PLUS(RC_Policy_AC_SendSelect_authHandleName);
2659     result = TPM2B_NAME_Unmarshal(&in->acName, paramBuffer, paramBufferSize);
2660     EXIT_IF_ERROR_PLUS(RC_Policy_AC_SendSelect_acName);
2661     result = TPMI_YES_NO_Unmarshal(&in->includeObject, paramBuffer, paramBufferSize);
2662     EXIT_IF_ERROR_PLUS(RC_Policy_AC_SendSelect_includeObject);
2663     if(*paramBufferSize != 0)
2664     {
2665         result = TPM_RC_SIZE;
2666         goto Exit;
2667     }
2668     result = TPM2_Policy_AC_SendSelect(in);
2669     break;
2670 }
2671 #endif // CC_Policy_AC_SendSelect
2672 #if CC_ACT_SetTimeout
2673 case TPM_CC_ACT_SetTimeout:
2674 {
2675     ACT_SetTimeout_In* in =
2676         (ACT_SetTimeout_In*)MemoryGetInBuffer(sizeof(ACT_SetTimeout_In));
2677     in->actHandle = handles[0];
2678     result = UINT32_Unmarshal(&in->startTimeout, paramBuffer, paramBufferSize);
2679     EXIT_IF_ERROR_PLUS(RC_ACT_SetTimeout_startTimeout);
2680     if(*paramBufferSize != 0)
2681     {
2682         result = TPM_RC_SIZE;
2683         goto Exit;
2684     }
2685     result = TPM2_ACT_SetTimeout(in);
2686     break;
2687 }
2688 #endif // CC_ACT_SetTimeout

```



```

2689 #if CC_Vendor_TCG_Test
2690 case TPM_CC_Vendor_TCG_Test:
2691 {
2692     Vendor_TCG_Test_In* in =
2693         (Vendor_TCG_Test_In*)MemoryGetInBuffer(sizeof(Vendor_TCG_Test_In));
2694     Vendor_TCG_Test_Out* out =
2695         (Vendor_TCG_Test_Out*)MemoryGetOutBuffer(sizeof(Vendor_TCG_Test_Out));
2696     result = TPM2B_DATA_Unmarshal(&in->inputData, paramBuffer, paramBufferSize);
2697     EXIT_IF_ERROR_PLUS(RC_Vendor_TCG_Test_inputData);
2698     if(*paramBufferSize != 0)
2699     {
2700         result = TPM_RC_SIZE;
2701         goto Exit;
2702     }
2703     result = TPM2_Vendor_TCG_Test(in, out);
2704     rSize = sizeof(Vendor_TCG_Test_Out);
2705     *respParamSize += TPM2B_DATA_Marshal(&out->outputData, responseBuffer, &rSize);
2706     break;
2707 }
2708 #endif // CC_Vendor_TCG_Test

```

6.15 /tpm/include/private/Commands.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #ifndef _COMMANDS_H_
4 #define _COMMANDS_H_
5
6 #if CC_Startup
7 # include "Startup_fp.h"
8 #endif
9 #if CC_Shutdown
10 # include "Shutdown_fp.h"
11 #endif
12 #if CC_SelfTest
13 # include "SelfTest_fp.h"
14 #endif
15 #if CC_IncrementalSelfTest
16 # include "IncrementalSelfTest_fp.h"
17 #endif
18 #if CC_GetTestResult
19 # include "GetTestResult_fp.h"
20 #endif
21 #if CC_StartAuthSession
22 # include "StartAuthSession_fp.h"
23 #endif
24 #if CC_PolicyRestart
25 # include "PolicyRestart_fp.h"
26 #endif
27 #if CC_Create
28 # include "Create_fp.h"
29 #endif
30 #if CC_Load
31 # include "Load_fp.h"
32 #endif
33 #if CC_LoadExternal
34 # include "LoadExternal_fp.h"
35 #endif
36 #if CC_ReadPublic
37 # include "ReadPublic_fp.h"
38 #endif
39 #if CC_ActivateCredential
40 # include "ActivateCredential_fp.h"
41 #endif
42 #if CC_MakeCredential

```

```
43 # include "MakeCredential_fp.h"
44 #endif
45 #if CC_Unseal
46 # include "Unseal_fp.h"
47 #endif
48 #if CC_ObjectChangeAuth
49 # include "ObjectChangeAuth_fp.h"
50 #endif
51 #if CC_CreateLoaded
52 # include "CreateLoaded_fp.h"
53 #endif
54 #if CC_Duplicate
55 # include "Duplicate_fp.h"
56 #endif
57 #if CC_Rewrap
58 # include "Rewrap_fp.h"
59 #endif
60 #if CC_Import
61 # include "Import_fp.h"
62 #endif
63 #if CC_RSA_Encrypt
64 # include "RSA_Encrypt_fp.h"
65 #endif
66 #if CC_RSA_Decrypt
67 # include "RSA_Decrypt_fp.h"
68 #endif
69 #if CC_ECDH_KeyGen
70 # include "ECDH_KeyGen_fp.h"
71 #endif
72 #if CC_ECDH_ZGen
73 # include "ECDH_ZGen_fp.h"
74 #endif
75 #if CC_ECC_Parameters
76 # include "ECC_Parameters_fp.h"
77 #endif
78 #if CC_ZGen_2Phase
79 # include "ZGen_2Phase_fp.h"
80 #endif
81 #if CC_ECC_Encrypt
82 # include "ECC_Encrypt_fp.h"
83 #endif
84 #if CC_ECC_Decrypt
85 # include "ECC_Decrypt_fp.h"
86 #endif
87 #if CC_EncryptDecrypt
88 # include "EncryptDecrypt_fp.h"
89 #endif
90 #if CC_EncryptDecrypt2
91 # include "EncryptDecrypt2_fp.h"
92 #endif
93 #if CC_Hash
94 # include "Hash_fp.h"
95 #endif
96 #if CC_HMAC
97 # include "HMAC_fp.h"
98 #endif
99 #if CC_MAC
100 # include "MAC_fp.h"
101 #endif
102 #if CC_GetRandom
103 # include "GetRandom_fp.h"
104 #endif
105 #if CC_StirRandom
106 # include "StirRandom_fp.h"
107 #endif
108 #if CC_HMAC_Start
```

```
109 # include "HMAC_Start_fp.h"
110 #endif
111 #if CC_MAC_Start
112 # include "MAC_Start_fp.h"
113 #endif
114 #if CC_HashSequenceStart
115 # include "HashSequenceStart_fp.h"
116 #endif
117 #if CC_SequenceUpdate
118 # include "SequenceUpdate_fp.h"
119 #endif
120 #if CC_SequenceComplete
121 # include "SequenceComplete_fp.h"
122 #endif
123 #if CC_EventSequenceComplete
124 # include "EventSequenceComplete_fp.h"
125 #endif
126 #if CC_Certify
127 # include "Certify_fp.h"
128 #endif
129 #if CC_CertifyCreation
130 # include "CertifyCreation_fp.h"
131 #endif
132 #if CC_Quote
133 # include "Quote_fp.h"
134 #endif
135 #if CC_GetSessionAuditDigest
136 # include "GetSessionAuditDigest_fp.h"
137 #endif
138 #if CC_GetCommandAuditDigest
139 # include "GetCommandAuditDigest_fp.h"
140 #endif
141 #if CC_GetTime
142 # include "GetTime_fp.h"
143 #endif
144 #if CC_CertifyX509
145 # include "CertifyX509_fp.h"
146 #endif
147 #if CC_Commit
148 # include "Commit_fp.h"
149 #endif
150 #if CC_EC_Ephemeral
151 # include "EC_Ephemeral_fp.h"
152 #endif
153 #if CC_VerifySignature
154 # include "VerifySignature_fp.h"
155 #endif
156 #if CC_Sign
157 # include "Sign_fp.h"
158 #endif
159 #if CC_SetCommandCodeAuditStatus
160 # include "SetCommandCodeAuditStatus_fp.h"
161 #endif
162 #if CC_PCR_Extend
163 # include "PCR_Extend_fp.h"
164 #endif
165 #if CC_PCR_Event
166 # include "PCR_Event_fp.h"
167 #endif
168 #if CC_PCR_Read
169 # include "PCR_Read_fp.h"
170 #endif
171 #if CC_PCR_Allocate
172 # include "PCR_Allocate_fp.h"
173 #endif
174 #if CC_PCR_SetAuthPolicy
```

```
175 # include "PCR_SetAuthPolicy_fp.h"
176 #endif
177 #if CC_PCR_SetAuthValue
178 # include "PCR_SetAuthValue_fp.h"
179 #endif
180 #if CC_PCR_Reset
181 # include "PCR_Reset_fp.h"
182 #endif
183 #if CC_PolicySigned
184 # include "PolicySigned_fp.h"
185 #endif
186 #if CC_PolicySecret
187 # include "PolicySecret_fp.h"
188 #endif
189 #if CC_PolicyTicket
190 # include "PolicyTicket_fp.h"
191 #endif
192 #if CC_PolicyOR
193 # include "PolicyOR_fp.h"
194 #endif
195 #if CC_PolicyPCR
196 # include "PolicyPCR_fp.h"
197 #endif
198 #if CC_PolicyLocality
199 # include "PolicyLocality_fp.h"
200 #endif
201 #if CC_PolicyNV
202 # include "PolicyNV_fp.h"
203 #endif
204 #if CC_PolicyCounterTimer
205 # include "PolicyCounterTimer_fp.h"
206 #endif
207 #if CC_PolicyCommandCode
208 # include "PolicyCommandCode_fp.h"
209 #endif
210 #if CC_PolicyPhysicalPresence
211 # include "PolicyPhysicalPresence_fp.h"
212 #endif
213 #if CC_PolicyCpHash
214 # include "PolicyCpHash_fp.h"
215 #endif
216 #if CC_PolicyNameHash
217 # include "PolicyNameHash_fp.h"
218 #endif
219 #if CC_PolicyDuplicationSelect
220 # include "PolicyDuplicationSelect_fp.h"
221 #endif
222 #if CC_PolicyAuthorize
223 # include "PolicyAuthorize_fp.h"
224 #endif
225 #if CC_PolicyAuthValue
226 # include "PolicyAuthValue_fp.h"
227 #endif
228 #if CC_PolicyPassword
229 # include "PolicyPassword_fp.h"
230 #endif
231 #if CC_PolicyGetDigest
232 # include "PolicyGetDigest_fp.h"
233 #endif
234 #if CC_PolicyNvWritten
235 # include "PolicyNvWritten_fp.h"
236 #endif
237 #if CC_PolicyTemplate
238 # include "PolicyTemplate_fp.h"
239 #endif
240 #if CC_PolicyAuthorizeNV
```

```
241 # include "PolicyAuthorizeNV_fp.h"
242 #endif
243 #if CC_PolicyCapability
244 # include "PolicyCapability_fp.h"
245 #endif
246 #if CC_PolicyParameters
247 # include "PolicyParameters_fp.h"
248 #endif
249 #if CC_CreatePrimary
250 # include "CreatePrimary_fp.h"
251 #endif
252 #if CC_HierarchyControl
253 # include "HierarchyControl_fp.h"
254 #endif
255 #if CC_SetPrimaryPolicy
256 # include "SetPrimaryPolicy_fp.h"
257 #endif
258 #if CC_ChangePPS
259 # include "ChangePPS_fp.h"
260 #endif
261 #if CC_ChangeEPS
262 # include "ChangeEPS_fp.h"
263 #endif
264 #if CC_Clear
265 # include "Clear_fp.h"
266 #endif
267 #if CC_ClearControl
268 # include "ClearControl_fp.h"
269 #endif
270 #if CC_HierarchyChangeAuth
271 # include "HierarchyChangeAuth_fp.h"
272 #endif
273 #if CC_DictionaryAttackLockReset
274 # include "DictionaryAttackLockReset_fp.h"
275 #endif
276 #if CC_DictionaryAttackParameters
277 # include "DictionaryAttackParameters_fp.h"
278 #endif
279 #if CC_PP_Commands
280 # include "PP_Commands_fp.h"
281 #endif
282 #if CC_SetAlgorithmSet
283 # include "SetAlgorithmSet_fp.h"
284 #endif
285 #if CC_FieldUpgradeStart
286 # include "FieldUpgradeStart_fp.h"
287 #endif
288 #if CC_FieldUpgradeData
289 # include "FieldUpgradeData_fp.h"
290 #endif
291 #if CC_FirmwareRead
292 # include "FirmwareRead_fp.h"
293 #endif
294 #if CC_ContextSave
295 # include "ContextSave_fp.h"
296 #endif
297 #if CC_ContextLoad
298 # include "ContextLoad_fp.h"
299 #endif
300 #if CC_FlushContext
301 # include "FlushContext_fp.h"
302 #endif
303 #if CC_EvictControl
304 # include "EvictControl_fp.h"
305 #endif
306 #if CC_ReadClock
```

```
307 # include "ReadClock_fp.h"
308 #endif
309 #if CC_ClockSet
310 # include "ClockSet_fp.h"
311 #endif
312 #if CC_ClockRateAdjust
313 # include "ClockRateAdjust_fp.h"
314 #endif
315 #if CC_GetCapability
316 # include "GetCapability_fp.h"
317 #endif
318 #if CC_TestParms
319 # include "TestParms_fp.h"
320 #endif
321 #if CC_NV_DefineSpace
322 # include "NV_DefineSpace_fp.h"
323 #endif
324 #if CC_NV_UndefineSpace
325 # include "NV_UndefineSpace_fp.h"
326 #endif
327 #if CC_NV_UndefineSpaceSpecial
328 # include "NV_UndefineSpaceSpecial_fp.h"
329 #endif
330 #if CC_NV_ReadPublic
331 # include "NV_ReadPublic_fp.h"
332 #endif
333 #if CC_NV_Write
334 # include "NV_Write_fp.h"
335 #endif
336 #if CC_NV_Increment
337 # include "NV_Increment_fp.h"
338 #endif
339 #if CC_NV_Extend
340 # include "NV_Extend_fp.h"
341 #endif
342 #if CC_NV_SetBits
343 # include "NV_SetBits_fp.h"
344 #endif
345 #if CC_NV_WriteLock
346 # include "NV_WriteLock_fp.h"
347 #endif
348 #if CC_NV_GlobalWriteLock
349 # include "NV_GlobalWriteLock_fp.h"
350 #endif
351 #if CC_NV_Read
352 # include "NV_Read_fp.h"
353 #endif
354 #if CC_NV_ReadLock
355 # include "NV_ReadLock_fp.h"
356 #endif
357 #if CC_NV_ChangeAuth
358 # include "NV_ChangeAuth_fp.h"
359 #endif
360 #if CC_NV_Certify
361 # include "NV_Certify_fp.h"
362 #endif
363 #if CC_NV_DefineSpace2
364 # include "NV_DefineSpace2_fp.h"
365 #endif
366 #if CC_NV_ReadPublic2
367 # include "NV_ReadPublic2_fp.h"
368 #endif
369 #if CC_SetCapability
370 # include "SetCapability_fp.h"
371 #endif
372 #if CC_AC_Send
```



```

373 # include "AC_Send_fp.h"
374 #endif
375 #if CC_Policy_AC_SendSelect
376 # include "Policy_AC_SendSelect_fp.h"
377 #endif
378 #if CC_ACT_SetTimeout
379 # include "ACT_SetTimeout_fp.h"
380 #endif
381 #if CC_Vendor_TCG_Test
382 # include "Vendor_TCG_Test_fp.h"
383 #endif
384
385 #endif // _COMMANDS_H_

```

6.16 /tpm/include/private/CryptEcc.h

```

1  /** Introduction
2  //
3  // This file contains structure definitions used for ECC. The structures in this
4  // file are only used internally. The ECC-related structures that cross the
5  // public TPM interface are defined in TpmTypes.h
6  //
7
8  // ECC Curve data type decoder ring
9  // =====
10 // | Name | Old Name* | Comments
11 // | ----- | ----- | -----
12 // | TPM_ECC_CURVE | | 16-bit Curve ID from Part 2 of TCG
13 // | TPM_ECC_CURVE_METADATA | ECC_CURVE | See description below
14 // | | |
15 // * - if different
16
17 // TPM_ECC_CURVE_METADATA
18 // =====
19 // TPM-specific metadata for a particular curve, such as OIDs and signing/kdf
20 // schemes associated with the curve.
21 //
22 // TODO_ECC: Need to remove the curve constants from this structure and replace
23 // them with a reference to math-lib provided calls. <Once done, add this
24 // revised comment to the above description> Note: this structure does *NOT*
25 // include the actual curve constants. The curve constants are no longer in this
26 // structure because the constants need to be in a format compatible with the
27 // math library and are retrieved by the `ExtEcc_CurveGet*` family of functions.
28 //
29 // Using the math library's constant structure here is not necessary and breaks
30 // encapsulation. Using a tpm-specific format means either redundancy (the same
31 // values exist here and in a math-specific format), or forces the math library
32 // to adopt a particular format determined by this structure. Neither outcome
33 // is as clean as simply leaving the actual constants out of this structure.
34
35 #ifndef _CRYPT_ECC_H
36 #define _CRYPT_ECC_H
37
38 /** Structures
39
40 #define ECC_BITS (MAX_ECC_KEY_BYTES * 8)
41 CRYPT_INT_TYPE(ecc, ECC_BITS);
42
43 #define CRYPT_ECC_NUM(name) CRYPT_INT_VAR(name, ECC_BITS)
44

```

```

45 #define CRYPT_ECC_INITIALIZED(name, initializer) \
46     CRYPT_INT_INITIALIZED(name, ECC_BITS, initializer)
47
48 typedef struct TPM_ECC_CURVE_METADATA
49 {
50     const TPM_ECC_CURVE    curveId;
51     const UINT16           keySizeBits;
52     const TPMT_KDF_SCHEME kdf;
53     const TPMT_ECC_SCHEME sign;
54     const BYTE*            OID;
55 } TPM_ECC_CURVE_METADATA;
56
57 /*** Macros
58 extern const TPM_ECC_CURVE_METADATA eccCurves[ECC_CURVE_COUNT];
59
60 #endif

```

6.17 /tpm/include/private/CryptHash.h

```

1  /*** Introduction
2  // This header contains the hash structure definitions used in the TPM code
3  // to define the amount of space to be reserved for the hash state. This allows
4  // the TPM code to not have to import all of the symbols used by the hash
5  // computations. This lets the build environment of the TPM code not to have
6  // include the header files associated with the CryptoEngine code.
7
8  #ifndef _CRYPT_HASH_H
9  #define _CRYPT_HASH_H
10
11 /*** Hash-related Structures
12
13 union SMAC_STATES;
14
15 // These definitions add the high-level methods for processing state that may be
16 // an SMAC
17 typedef void (*SMAC_DATA_METHOD)(
18     union SMAC_STATES* state, UINT32 size, const BYTE* buffer);
19
20 typedef UINT16 (*SMAC_END_METHOD)(
21     union SMAC_STATES* state, UINT32 size, BYTE* buffer);
22
23 typedef struct sequenceMethods
24 {
25     SMAC_DATA_METHOD data;
26     SMAC_END_METHOD end;
27 } SMAC_METHODS;
28
29 #define SMAC_IMPLEMENTED (CC_MAC || CC_MAC_Start)
30
31 // These definitions are here because the SMAC state is in the union of hash states.
32 typedef struct tpmCmacState
33 {
34     TPM_ALG_ID    symAlg;
35     UINT16        keySizeBits;
36     INT16         bcount; // current count of bytes accumulated in IV
37     TPM2B_IV      iv;     // IV buffer
38     TPM2B_SYM_KEY symKey;
39 } tpmCmacState_t;
40
41 typedef union SMAC_STATES
42 {
43     #if ALG_CMACH
44         tpmCmacState_t cmac;
45     #endif
46     UINT64 pad;

```

```

47 } SMAC_STATES;
48
49 typedef struct SMAC_STATE
50 {
51     SMAC_METHODS smacMethods;
52     SMAC_STATES state;
53 } SMAC_STATE;
54
55 #if ALG_SHA1
56 # define IF_IMPLEMENTED_SHA1(op) op(SHA1, Sha1)
57 #else
58 # define IF_IMPLEMENTED_SHA1(op)
59 #endif
60 #if ALG_SHA256
61 # define IF_IMPLEMENTED_SHA256(op) op(SHA256, Sha256)
62 #else
63 # define IF_IMPLEMENTED_SHA256(op)
64 #endif
65 #if ALG_SHA384
66 # define IF_IMPLEMENTED_SHA384(op) op(SHA384, Sha384)
67 #else
68 # define IF_IMPLEMENTED_SHA384(op)
69 #endif
70 #if ALG_SHA512
71 # define IF_IMPLEMENTED_SHA512(op) op(SHA512, Sha512)
72 #else
73 # define IF_IMPLEMENTED_SHA512(op)
74 #endif
75 #if ALG_SM3_256
76 # define IF_IMPLEMENTED_SM3_256(op) op(SM3_256, Sm3_256)
77 #else
78 # define IF_IMPLEMENTED_SM3_256(op)
79 #endif
80 #if ALG_SHA3_256
81 # define IF_IMPLEMENTED_SHA3_256(op) op(SHA3_256, Sha3_256)
82 #else
83 # define IF_IMPLEMENTED_SHA3_256(op)
84 #endif
85 #if ALG_SHA3_384
86 # define IF_IMPLEMENTED_SHA3_384(op) op(SHA3_384, Sha3_384)
87 #else
88 # define IF_IMPLEMENTED_SHA3_384(op)
89 #endif
90 #if ALG_SHA3_512
91 # define IF_IMPLEMENTED_SHA3_512(op) op(SHA3_512, Sha3_512)
92 #else
93 # define IF_IMPLEMENTED_SHA3_512(op)
94 #endif
95
96 #define FOR_EACH_HASH(op) \
97     IF_IMPLEMENTED_SHA1(op) \
98     IF_IMPLEMENTED_SHA256(op) \
99     IF_IMPLEMENTED_SHA384(op) \
100     IF_IMPLEMENTED_SHA512(op) \
101     IF_IMPLEMENTED_SM3_256(op) \
102     IF_IMPLEMENTED_SHA3_256(op) \
103     IF_IMPLEMENTED_SHA3_384(op) \
104     IF_IMPLEMENTED_SHA3_512(op)
105
106 #define HASH_TYPE(HASH, Hash) tpmHashState##HASH##_t Hash;
107 typedef union
108 {
109     FOR_EACH_HASH(HASH_TYPE)
110 // Additions for symmetric block cipher MAC
111 #if SMAC_IMPLEMENTED
112     SMAC_STATE smac;

```

```

113 #endif
114 // to force structure alignment to be no worse than HASH_ALIGNMENT
115 #if HASH_ALIGNMENT == 8
116     uint64_t align;
117 #else
118     uint32_t align;
119 #endif
120 } ANY_HASH_STATE;
121
122 typedef ANY_HASH_STATE*      PANY_HASH_STATE;
123 typedef const ANY_HASH_STATE* PCANY_HASH_STATE;
124
125 #define ALIGNED_SIZE(x, b) (((x) + (b)-1) / (b)) * (b)
126 // MAX_HASH_STATE_SIZE will change with each implementation. It is assumed that
127 // a hash state will not be larger than twice the block size plus some
128 // overhead (in this case, 16 bytes). The overall size needs to be as
129 // large as any of the hash contexts. The structure needs to start on an
130 // alignment boundary and be an even multiple of the alignment
131 #define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
132 #define MAX_HASH_STATE_SIZE_ALIGNED ALIGNED_SIZE(MAX_HASH_STATE_SIZE, HASH_ALIGNMENT)
133
134 // This is an aligned byte array that will hold any of the hash contexts.
135 typedef ANY_HASH_STATE ALIGNED_HASH_STATE;
136
137 // The header associated with the hash library is expected to define the methods
138 // which include the calling sequence. When not compiling CryptHash.c, the methods
139 // are not defined so we need placeholder functions for the structures
140
141 #ifndef HASH_START_METHOD_DEF
142 # define HASH_START_METHOD_DEF void(HASH_START_METHOD) (void)
143 #endif
144 #ifndef HASH_DATA_METHOD_DEF
145 # define HASH_DATA_METHOD_DEF void(HASH_DATA_METHOD) (void)
146 #endif
147 #ifndef HASH_END_METHOD_DEF
148 # define HASH_END_METHOD_DEF void(HASH_END_METHOD) (void)
149 #endif
150 #ifndef HASH_STATE_COPY_METHOD_DEF
151 # define HASH_STATE_COPY_METHOD_DEF void(HASH_STATE_COPY_METHOD) (void)
152 #endif
153 #ifndef HASH_STATE_EXPORT_METHOD_DEF
154 # define HASH_STATE_EXPORT_METHOD_DEF void(HASH_STATE_EXPORT_METHOD) (void)
155 #endif
156 #ifndef HASH_STATE_IMPORT_METHOD_DEF
157 # define HASH_STATE_IMPORT_METHOD_DEF void(HASH_STATE_IMPORT_METHOD) (void)
158 #endif
159
160 // Define the prototypical function call for each of the methods. This defines the
161 // order in which the parameters are passed to the underlying function.
162 typedef HASH_START_METHOD_DEF;
163 typedef HASH_DATA_METHOD_DEF;
164 typedef HASH_END_METHOD_DEF;
165 typedef HASH_STATE_COPY_METHOD_DEF;
166 typedef HASH_STATE_EXPORT_METHOD_DEF;
167 typedef HASH_STATE_IMPORT_METHOD_DEF;
168
169 typedef struct _HASH_METHODS
170 {
171     HASH_START_METHOD*      start;
172     HASH_DATA_METHOD*       data;
173     HASH_END_METHOD*        end;
174     HASH_STATE_COPY_METHOD* copy;    // Copy a hash block
175     HASH_STATE_EXPORT_METHOD* copyOut; // Copy a hash block from a hash
176                                     // context
177     HASH_STATE_IMPORT_METHOD* copyIn;  // Copy a hash block to a proper hash
178                                     // context

```

```

179 } HASH_METHODS, *PHASH_METHODS;
180
181 #define HASH_TPM2B(HASH, Hash) TPM2B_TYPE(HASH##_DIGEST, HASH##_DIGEST_SIZE);
182
183 FOR_EACH_HASH(HASH_TPM2B)
184
185 // When the TPM implements RSA, the hash-dependent OID pointers are part of the
186 // HASH_DEF. These macros conditionally add the OID reference to the HASH_DEF and the
187 // HASH_DEF_TEMPLATE.
188 #if ALG_RSA
189 # define PKCS1_HASH_REF const BYTE* PKCS1;
190 # define PKCS1_OID(NAME) , OID_PKCS1_##NAME
191 #else
192 # define PKCS1_HASH_REF
193 # define PKCS1_OID(NAME)
194 #endif
195
196 // When the TPM implements ECC, the hash-dependent OID pointers are part of the
197 // HASH_DEF. These macros conditionally add the OID reference to the HASH_DEF and the
198 // HASH_DEF_TEMPLATE.
199 #if ALG_ECDSA
200 # define ECDSA_HASH_REF const BYTE* ECDSA;
201 # define ECDSA_OID(NAME) , OID_ECDSA_##NAME
202 #else
203 # define ECDSA_HASH_REF
204 # define ECDSA_OID(NAME)
205 #endif
206
207 typedef const struct HASH_DEF_STRUCT
208 {
209     HASH_METHODS method;
210     uint16_t blockSize;
211     uint16_t digestSize;
212     uint16_t contextSize;
213     uint16_t hashAlg;
214     const BYTE* OID;
215     PKCS1_HASH_REF // PKCS1 OID
216     ECDSA_HASH_REF // ECDSA OID
217 } HASH_DEF, *PHASH_DEF;
218
219 // Macro to fill in the HASH_DEF for an algorithm. For SHA1, the instance would be:
220 // HASH_DEF_TEMPLATE(Sha1, SHA1)
221 // This handles the difference in capitalization for the various pieces.
222 #define HASH_DEF_TEMPLATE(HASH, Hash)
223     HASH_DEF Hash##_Def = {
224         (HASH_START_METHOD*) &tpmHashStart_##HASH,
225         (HASH_DATA_METHOD*) &tpmHashData_##HASH,
226         (HASH_END_METHOD*) &tpmHashEnd_##HASH,
227         (HASH_STATE_COPY_METHOD*) &tpmHashStateCopy_##HASH,
228         (HASH_STATE_EXPORT_METHOD*) &tpmHashStateExport_##HASH,
229         (HASH_STATE_IMPORT_METHOD*) &tpmHashStateImport_##HASH,
230     },
231     HASH##_BLOCK_SIZE, /*block size */
232     HASH##_DIGEST_SIZE, /*data size */
233     sizeof(tpmHashState_##HASH##_t),
234     TPM_ALG_##HASH,
235     OID_##HASH PKCS1_OID(HASH) ECDSA_OID(HASH) };
236
237 // These definitions are for the types that can be in a hash state structure.
238 // These types are used in the cryptographic utilities. This is a define rather than
239 // an enum so that the size of this field can be explicit.
240 typedef BYTE HASH_STATE_TYPE;
241 #define HASH_STATE_EMPTY ((HASH_STATE_TYPE)0)
242 #define HASH_STATE_HASH ((HASH_STATE_TYPE)1)
243 #define HASH_STATE_HMAC ((HASH_STATE_TYPE)2)
244 #if CC_MAC || CC_MAC_Start

```

```

245 # define HASH_STATE_SMAC ((HASH_STATE_TYPE)3)
246 #endif
247
248 // This is the structure that is used for passing a context into the hashing
249 // functions. It should be the same size as the function context used within
250 // the hashing functions. This is checked when the hash function is initialized.
251 // This version uses a new layout for the contexts and a different definition. The
252 // state buffer is an array of HASH_UNIT values so that a decent compiler will put
253 // the structure on a HASH_UNIT boundary. If the structure is not properly aligned,
254 // the code that manipulates the structure will copy to a properly aligned
255 // structure before it is used and copy the result back. This just makes things
256 // slower.
257 // NOTE: This version of the state had the pointer to the update method in the
258 // state. This is to allow the SMAC functions to use the same structure without
259 // having to replicate the entire HASH_DEF structure.
260 typedef struct _HASH_STATE
261 {
262     HASH_STATE_TYPE type; // type of the context
263     TPM_ALG_ID      hashAlg;
264     PHASH_DEF       def;
265     ANY_HASH_STATE  state;
266 } HASH_STATE, *PHASH_STATE;
267 typedef const HASH_STATE* PCHASH_STATE;
268
269 /** HMAC State Structures
270
271 // An HMAC_STATE structure contains an opaque HMAC stack state. A caller would
272 // use this structure when performing incremental HMAC operations. This structure
273 // contains a hash state and an HMAC key and allows slightly better stack
274 // optimization than adding an HMAC key to each hash state.
275 typedef struct hmacState
276 {
277     HASH_STATE      hashState; // the hash state
278     TPM2B_HASH_BLOCK hmacKey;  // the HMAC key
279 } HMAC_STATE, *PHMAC_STATE;
280
281 // This is for the external hash state. This implementation assumes that the size
282 // of the exported hash state is no larger than the internal hash state.
283 typedef struct
284 {
285     BYTE buffer[sizeof(HASH_STATE)];
286 } EXPORT_HASH_STATE, *PEXPORT_HASH_STATE;
287
288 typedef const EXPORT_HASH_STATE* PEXPORT_HASH_STATE;
289
290 #endif // _CRYPT_HASH_H

```

6.18 /tpm/include/private/CryptRand.h

```

1  /** Introduction
2  // This file contains constant definition shared by CryptUtil and the parts
3  // of the Crypto Engine.
4  //
5
6  #ifndef _CRYPT_RAND_H
7  #define _CRYPT_RAND_H
8
9  /** DRBG Structures and Defines
10
11 // Values and structures for the random number generator. These values are defined
12 // in this header file so that the size of the RNG state can be known to TPM.lib.
13 // This allows the allocation of some space in NV memory for the state to
14 // be stored on an orderly shutdown.
15
16 // The DRBG based on a symmetric block cipher is defined by three values,

```



```

17 // 1) the key size
18 // 2) the block size (the IV size)
19 // 3) the symmetric algorithm
20
21 #define DRBG_KEY_SIZE_BITS AES_MAX_KEY_SIZE_BITS
22 #define DRBG_IV_SIZE_BITS (AES_MAX_BLOCK_SIZE * 8)
23 #define DRBG_ALGORITHM TPM_ALG_AES
24
25 #define DRBG_ENCRYPT_SETUP(key, keySizeInBits, schedule) \
26     TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule)
27 #define DRBG_ENCRYPT(keySchedule, in, out) \
28     TpmCryptEncryptAES(SWIZZLE(keySchedule, in, out))
29
30 #if((DRBG_KEY_SIZE_BITS % RADIX_BITS) != 0) || ((DRBG_IV_SIZE_BITS % RADIX_BITS) != 0)
31 # error "Key size and IV for DRBG must be even multiples of the radix"
32 #endif
33 #if(DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
34 # error "Key size for DRBG must be even multiple of the cypher block size"
35 #endif
36
37 // Derived values
38 #define DRBG_MAX_REQUESTS_PER_RESEED (1 << 48)
39 #define DRBG_MAX_REQUEST_SIZE (1 << 32)
40
41 #define pDRBG_KEY(seed) ((DRBG_KEY*)&((BYTE*)(seed))[0]))
42 #define pDRBG_IV(seed) ((DRBG_IV*)&((BYTE*)(seed))[DRBG_KEY_SIZE_BYTES]))
43
44 #define DRBG_KEY_SIZE_WORDS (BITS_TO_CRYPT_WORDS(DRBG_KEY_SIZE_BITS))
45 #define DRBG_KEY_SIZE_BYTES (DRBG_KEY_SIZE_WORDS * RADIX_BYTES)
46
47 #define DRBG_IV_SIZE_WORDS (BITS_TO_CRYPT_WORDS(DRBG_IV_SIZE_BITS))
48 #define DRBG_IV_SIZE_BYTES (DRBG_IV_SIZE_WORDS * RADIX_BYTES)
49
50 #define DRBG_SEED_SIZE_WORDS (DRBG_KEY_SIZE_WORDS + DRBG_IV_SIZE_WORDS)
51 #define DRBG_SEED_SIZE_BYTES (DRBG_KEY_SIZE_BYTES + DRBG_IV_SIZE_BYTES)
52
53 typedef union
54 {
55     BYTE bytes[DRBG_KEY_SIZE_BYTES];
56     crypt_ushort_t words[DRBG_KEY_SIZE_WORDS];
57 } DRBG_KEY;
58
59 typedef union
60 {
61     BYTE bytes[DRBG_IV_SIZE_BYTES];
62     crypt_ushort_t words[DRBG_IV_SIZE_WORDS];
63 } DRBG_IV;
64
65 typedef union
66 {
67     BYTE bytes[DRBG_SEED_SIZE_BYTES];
68     crypt_ushort_t words[DRBG_SEED_SIZE_WORDS];
69 } DRBG_SEED;
70
71 #define CTR_DRBG_MAX_REQUESTS_PER_RESEED ((UINT64)1 << 20)
72 #define CTR_DRBG_MAX_BYTES_PER_REQUEST (1 << 16)
73
74 #define CTR_DRBG_MIN_ENTROPY_INPUT_LENGTH DRBG_SEED_SIZE_BYTES
75 #define CTR_DRBG_MAX_ENTROPY_INPUT_LENGTH DRBG_SEED_SIZE_BYTES
76 #define CTR_DRBG_MAX_ADDITIONAL_INPUT_LENGTH DRBG_SEED_SIZE_BYTES
77
78 #define TESTING (1 << 0)
79 #define ENTROPY (1 << 1)
80 #define TESTED (1 << 2)
81
82 #define IsTestStateSet(BIT) ((g_cryptoSelfTestState.rng & BIT) != 0)

```

```

83 #define SetTestStateBit(BIT)    (g_cryptoSelfTestState.rng |= BIT)
84 #define ClearTestStateBit(BIT) (g_cryptoSelfTestState.rng &= ~BIT)
85
86 #define IsSelfTest()    IsTestStateSet(TESTING)
87 #define SetSelfTest()   SetTestStateBit(TESTING)
88 #define ClearSelfTest() ClearTestStateBit(TESTING)
89
90 #define IsEntropyBad()    IsTestStateSet(ENTROPY)
91 #define SetEntropyBad()   SetTestStateBit(ENTROPY)
92 #define ClearEntropyBad() ClearTestStateBit(ENTROPY)
93
94 #define IsDrbgTested()    IsTestStateSet(TESTED)
95 #define SetDrbgTested()   SetTestStateBit(TESTED)
96 #define ClearDrbgTested() ClearTestStateBit(TESTED)
97
98 typedef struct
99 {
100     UINT64    reseedCounter;
101     UINT32    magic;
102     DRBG_SEED seed;           // contains the key and IV for the counter mode DRBG
103     UINT32    lastValue[4];  // used when the TPM does continuous self-test
104                               // for FIPS compliance of DRBG
105 } DRBG_STATE, *pDRBG_STATE;
106 #define DRBG_MAGIC ((UINT32)0x47425244) // "DRBG" backwards so that it displays
107
108 typedef struct KDF_STATE
109 {
110     UINT64    counter;
111     UINT32    magic;
112     UINT32    limit;
113     TPM2B*    seed;
114     const TPM2B* label;
115     TPM2B*    context;
116     TPM_ALG_ID hash;
117     TPM_ALG_ID kdf;
118     UINT16    digestSize;
119     TPM2B_DIGEST residual;
120 } KDF_STATE, *pKDR_STATE;
121 #define KDF_MAGIC ((UINT32)0x4048444a) // "KDF " backwards
122
123 // Make sure that any other structures added to this union start with a 64-bit
124 // counter and a 32-bit magic number
125 typedef union
126 {
127     DRBG_STATE drbg;
128     KDF_STATE  kdf;
129 } RAND_STATE;
130
131 // This is the state used when the library uses a random number generator.
132 // A special function is installed for the library to call. That function
133 // picks up the state from this location and uses it for the generation
134 // of the random number.
135 extern RAND_STATE* s_random;
136
137 // When instrumenting RSA key sieve
138 #if RSA_INSTRUMENT
139 # define PRIME_INDEX(x)    ((x) == 512 ? 0 : (x) == 1024 ? 1 : 2)
140 # define INSTRUMENT_SET(a, b) ((a) = (b))
141 # define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
142 # define INSTRUMENT_INC(a)    (a) = (a) + 1
143
144 extern UINT32 PrimeIndex;
145 extern UINT32 failedAtIteration[10];
146 extern UINT32 PrimeCounts[3];
147 extern UINT32 MillerRabinTrials[3];
148 extern UINT32 totalFieldsSieved[3];

```

```

149 extern UINT32 bitsInFieldAfterSieve[3];
150 extern UINT32 emptyFieldsSieved[3];
151 extern UINT32 noPrimeFields[3];
152 extern UINT32 primesChecked[3];
153 extern UINT16 lastSievePrime;
154 #else
155 # define INSTRUMENT_SET(a, b)
156 # define INSTRUMENT_ADD(a, b)
157 # define INSTRUMENT_INC(a)
158 #endif
159
160 #endif // _CRYPT_RAND_H

```

6.19 /tpm/include/private/CryptRsa.h

```

1 // This file contains the RSA-related structures and defines.
2
3 #ifndef _CRYPT_RSA_H
4 #define _CRYPT_RSA_H
5
6 // These values are used in the Crypt_Int* representation of various RSA values.
7 // define ci_rsa_t as buffer containing a CRYPT_INT object with space for
8 // (MAX_RSA_KEY_BITS) of actual data.
9 CRYPT_INT_TYPE(rsa, MAX_RSA_KEY_BITS);
10 #define CRYPT_RSA_VAR(name) CRYPT_INT_VAR(name, MAX_RSA_KEY_BITS)
11 #define CRYPT_RSA_INITIALIZED(name, initializer) \
12     CRYPT_INT_INITIALIZED(name, MAX_RSA_KEY_BITS, initializer)
13
14 #define CRYPT_PRIME_VAR(name) CRYPT_INT_VAR(name, (MAX_RSA_KEY_BITS / 2))
15 // define ci_prime_t as buffer containing a CRYPT_INT object with space for
16 // (MAX_RSA_KEY_BITS/2) of actual data.
17 CRYPT_INT_TYPE(prime, (MAX_RSA_KEY_BITS / 2));
18 #define CRYPT_PRIME_INITIALIZED(name, initializer) \
19     CRYPT_INT_INITIALIZED(name, MAX_RSA_KEY_BITS / 2, initializer)
20
21 #if !CRT_FORMAT_RSA
22 # error This version only works with CRT formatted data
23 #endif // !CRT_FORMAT_RSA
24
25 typedef struct privateExponent
26 {
27     Crypt_Int* P;
28     Crypt_Int* Q;
29     Crypt_Int* dP;
30     Crypt_Int* dQ;
31     Crypt_Int* qInv;
32     ci_prime_t entries[5];
33 } privateExponent;
34
35 #define NEW_PRIVATE_EXPONENT(X) \
36     privateExponent _##X; \
37     privateExponent* X = RsaInitializeExponent(&(_##X))
38
39 #endif // _CRYPT_RSA_H

```

6.20 /tpm/include/private/CryptSym.h

```

1 /** Introduction
2 //
3 // This file contains the implementation of the symmetric block cipher modes
4 // allowed for a TPM. These functions only use the single block encryption functions
5 // of the selected symmetric cryptographic library.
6
7 /** Includes, Defines, and Typedefs

```

```

8  #ifndef CRYPT_SYM_H
9  #define CRYPT_SYM_H
10
11 #if ALG_AES
12 # define IF_IMPLEMENTED_AES(op) op(AES, aes)
13 #else
14 # define IF_IMPLEMENTED_AES(op)
15 #endif
16 #if ALG_SM4
17 # define IF_IMPLEMENTED_SM4(op) op(SM4, sm4)
18 #else
19 # define IF_IMPLEMENTED_SM4(op)
20 #endif
21 #if ALG_CAMELLIA
22 # define IF_IMPLEMENTED_CAMELLIA(op) op(CAMELLIA, camellia)
23 #else
24 # define IF_IMPLEMENTED_CAMELLIA(op)
25 #endif
26
27 #define FOR_EACH_SYM(op) \
28     IF_IMPLEMENTED_AES(op) \
29     IF_IMPLEMENTED_SM4(op) \
30     IF_IMPLEMENTED_CAMELLIA(op)
31
32 // Macros for creating the key schedule union
33 #define KEY_SCHEDULE(SYM, sym) tpmKeySchedule##SYM sym;
34 typedef union tpmCryptKeySchedule_t
35 {
36     FOR_EACH_SYM(KEY_SCHEDULE)
37
38     #if SYMMETRIC_ALIGNMENT == 8
39         uint64_t alignment;
40     #else
41         uint32_t alignment;
42     #endif
43 } tpmCryptKeySchedule_t;
44
45 // Each block cipher within a library is expected to conform to the same calling
46 // conventions with three parameters ('keySchedule', 'in', and 'out') in the same
47 // order. That means that all algorithms would use the same order of the same
48 // parameters. The code is written assuming the ('keySchedule', 'in', and 'out')
49 // order. However, if the library uses a different order, the order can be changed
50 // with a SWIZZLE macro that puts the parameters in the correct order.
51 // Note that all algorithms have to use the same order and number of parameters
52 // because the code to build the calling list is common for each call to encrypt
53 // or decrypt with the algorithm chosen by setting a function pointer to select
54 // the algorithm that is used.
55
56 #define ENCRYPT(keySchedule, in, out) encrypt(SWIZZLE(keySchedule, in, out))
57
58 #define DECRYPT(keySchedule, in, out) decrypt(SWIZZLE(keySchedule, in, out))
59
60 // Note that the macros rely on 'encrypt' as local values in the
61 // functions that use these macros. Those parameters are set by the macro that
62 // set the key schedule to be used for the call.
63
64 #define ENCRYPT_CASE(ALG, alg) \
65     case TPM_ALG_##ALG: \
66         TpmCryptSetEncryptKey##ALG(key, keySizeInBits, &keySchedule.alg); \
67         encrypt = (TpmCryptSetSymKeyCall_t)TpmCryptEncrypt##ALG; \
68         break;
69 #define DECRYPT_CASE(ALG, alg) \
70     case TPM_ALG_##ALG: \
71         TpmCryptSetDecryptKey##ALG(key, keySizeInBits, &keySchedule.alg); \
72         decrypt = (TpmCryptSetSymKeyCall_t)TpmCryptDecrypt##ALG; \
73         break;

```

```

74
75 #endif // CRYPT_SYM_H

```

6.21 /tpm/include/private/CryptTest.h

```

1 // This file contains constant definitions used for self-test.
2
3 #ifndef _CRYPT_TEST_H
4 #define _CRYPT_TEST_H
5
6 // This is the definition of a bit array with one bit per algorithm.
7 // NOTE: Since bit numbering starts at zero, when TPM_ALG_LAST is a multiple of 8,
8 // ALGORITHM_VECTOR will need to have byte for the single bit in the last byte. So,
9 // for example, when TPM_ALG_LAST is 8, ALGORITHM_VECTOR will need 2 bytes.
10 #define ALGORITHM_VECTOR_BYTES ((TPM_ALG_LAST + 8) / 8)
11 typedef BYTE ALGORITHM_VECTOR[ALGORITHM_VECTOR_BYTES];
12
13 #ifdef TEST_SELF_TEST
14 LIB_EXPORT extern ALGORITHM_VECTOR LibToTest;
15 #endif
16
17 // This structure is used to contain self-test tracking information for the
18 // cryptographic modules. Each of the major modules is given a 32-bit value in
19 // which it may maintain its own self test information. The convention for this
20 // state is that when all of the bits in this structure are 0, all functions need
21 // to be tested.
22 typedef struct
23 {
24     UINT32 rng;
25     UINT32 hash;
26     UINT32 sym;
27 #if ALG_RSA
28     UINT32 rsa;
29 #endif
30 #if ALG_ECC
31     UINT32 ecc;
32 #endif
33 } CRYPTO_SELF_TEST_STATE;
34
35 #endif // _CRYPT_TEST_H

```

6.22 /tpm/include/private/EccTestData.h

```

1 // This file contains the parameter data for ECC testing.
2
3 #ifndef SELF_TEST_DATA
4
5 TPM2B_TYPE(EC_TEST, 32);
6 const TPM_ECC_CURVE c_testCurve = 00003;
7
8 // The "static" key
9
10 const TPM2B_EC_TEST c_ecTestKey_ds = {
11     {32, {0xdf, 0x8d, 0xa4, 0xa3, 0x88, 0xf6, 0x76, 0x96, 0x89, 0xfc, 0x2f,
12           0x2d, 0xa1, 0xb4, 0x39, 0x7a, 0x78, 0xc4, 0x7f, 0x71, 0x8c, 0xa6,
13           0x91, 0x85, 0xc0, 0xbf, 0xf3, 0x54, 0x20, 0x91, 0x2f, 0x73}}};
14
15 const TPM2B_EC_TEST c_ecTestKey_QsX = {
16     {32, {0x17, 0xad, 0x2f, 0xcb, 0x18, 0xd4, 0xdb, 0x3f, 0x2c, 0x53, 0x13,
17           0x82, 0x42, 0x97, 0xff, 0x8d, 0x99, 0x50, 0x16, 0x02, 0x35, 0xa7,
18           0x06, 0xae, 0x1f, 0xda, 0xe2, 0x9c, 0x12, 0x77, 0xc0, 0xf9}}};
19
20 const TPM2B_EC_TEST c_ecTestKey_QsY = {
21     {32, {0xa6, 0xca, 0xf2, 0x18, 0x45, 0x96, 0x6e, 0x58, 0xe6, 0x72, 0x34,

```

```

22         0x12, 0x89, 0xcd, 0xaa, 0xad, 0xcb, 0x68, 0xb2, 0x51, 0xdc, 0x5e,
23         0xd1, 0x6d, 0x38, 0x20, 0x35, 0x57, 0xb2, 0xfd, 0xc7, 0x52}}};
24
25 // The "ephemeral" key
26
27 const TPM2B_EC_TEST c_ecTestKey_de = {
28     {32, {0xb6, 0xb5, 0x33, 0x5c, 0xd1, 0xee, 0x52, 0x07, 0x99, 0xea, 0x2e,
29           0x8f, 0x8b, 0x19, 0x18, 0x07, 0xc1, 0xf8, 0xdf, 0xdd, 0xb8, 0x77,
30           0x00, 0xc7, 0xd6, 0x53, 0x21, 0xed, 0x02, 0x53, 0xee, 0xac}}};
31
32 const TPM2B_EC_TEST c_ecTestKey_QeX = {
33     {32, {0xa5, 0x1e, 0x80, 0xd1, 0x76, 0x3e, 0x8b, 0x96, 0xce, 0xcc, 0x21,
34           0x82, 0xc9, 0xa2, 0xa2, 0xed, 0x47, 0x21, 0x89, 0x53, 0x44, 0xe9,
35           0xc7, 0x92, 0xe7, 0x31, 0x48, 0x38, 0xe6, 0xea, 0x93, 0x47}}};
36
37 const TPM2B_EC_TEST c_ecTestKey_QeY = {
38     {32, {0x30, 0xe6, 0x4f, 0x97, 0x03, 0xa1, 0xcb, 0x3b, 0x32, 0x2a, 0x70,
39           0x39, 0x94, 0xeb, 0x4e, 0xea, 0x55, 0x88, 0x81, 0x3f, 0xb5, 0x00,
40           0xb8, 0x54, 0x25, 0xab, 0xd4, 0xda, 0xfd, 0x53, 0x7a, 0x18}}};
41
42 // ECDH test results
43 const TPM2B_EC_TEST c_ecTestEcdh_X = {
44     {32, {0x64, 0x02, 0x68, 0x92, 0x78, 0xdb, 0x33, 0x52, 0xed, 0x3b, 0xfa,
45           0x3b, 0x74, 0xa3, 0x3d, 0x2c, 0x2f, 0x9c, 0x59, 0x03, 0x07, 0xf8,
46           0x22, 0x90, 0xed, 0xe3, 0x45, 0xf8, 0x2a, 0x0a, 0xd8, 0x1d}}};
47
48 const TPM2B_EC_TEST c_ecTestEcdh_Y = {
49     {32, {0x58, 0x94, 0x05, 0x82, 0xbe, 0x5f, 0x33, 0x02, 0x25, 0x90, 0x3a,
50           0x33, 0x90, 0x89, 0xe3, 0xe5, 0x10, 0x4a, 0xbc, 0x78, 0xa5, 0xc5,
51           0x07, 0x64, 0xaf, 0x91, 0xbc, 0xe6, 0xff, 0x85, 0x11, 0x40}}};
52
53 TPM2B_TYPE(TEST_VALUE, 64);
54 const TPM2B_TEST_VALUE c_ecTestValue = {
55     {64,
56      {0x78, 0xd5, 0xd4, 0x56, 0x43, 0x61, 0xdb, 0x97, 0xa4, 0x32, 0xc4, 0x0b, 0x06,
57        0xa9, 0xa8, 0xa0, 0xf4, 0x45, 0x7f, 0x13, 0xd8, 0x13, 0x81, 0x0b, 0xe5, 0x76,
58        0xbe, 0xaa, 0xb6, 0x3f, 0x8d, 0x4d, 0x23, 0x65, 0xcc, 0xa7, 0xc9, 0x19, 0x10,
59        0xce, 0x69, 0xcb, 0x0c, 0xc7, 0x11, 0x8d, 0xc3, 0xff, 0x62, 0x69, 0xa2, 0xbe,
60        0x46, 0x90, 0xe7, 0x7d, 0x81, 0x77, 0x94, 0x65, 0x1c, 0x3e, 0xc1, 0x3e}}};
61
62 # if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
63
64 const TPM2B_EC_TEST c_TestEcDsa_r = {
65     {32, {0x57, 0xf3, 0x36, 0xb7, 0xec, 0xc2, 0xdd, 0x76, 0x0e, 0xe2, 0x81,
66           0x21, 0x49, 0xc5, 0x66, 0x11, 0x4b, 0x8a, 0x4f, 0x17, 0x62, 0x82,
67           0xcc, 0x06, 0xf6, 0x64, 0x78, 0xef, 0x6b, 0x7c, 0xf2, 0x6c}}};
68 const TPM2B_EC_TEST c_TestEcDsa_s = {
69     {32, {0x1b, 0xed, 0x23, 0x72, 0x8f, 0x17, 0x5f, 0x47, 0x2e, 0xa7, 0x97,
70           0x2c, 0x51, 0x57, 0x20, 0x70, 0x6f, 0x89, 0x74, 0x8a, 0xa8, 0xf4,
71           0x26, 0xf4, 0x96, 0xa1, 0xb8, 0x3e, 0xe5, 0x35, 0xc5, 0x94}}};
72
73 const TPM2B_EC_TEST c_TestEcSchnorr_r = {
74     {32, {0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
75           0x00, 0x1b, 0x08, 0x9f, 0xde, 0xef, 0x62, 0xe3, 0xf1, 0x14, 0xcb,
76           0x54, 0x28, 0x13, 0x76, 0xfc, 0x6d, 0x69, 0x22, 0xb5, 0x3e}}};
77 const TPM2B_EC_TEST c_TestEcSchnorr_s = {
78     {32, {0xd9, 0xd3, 0x20, 0xfb, 0x4d, 0x16, 0xf2, 0xe6, 0xe2, 0x45, 0x07,
79           0x45, 0x1c, 0x92, 0x92, 0x92, 0xa9, 0x6b, 0x48, 0xf8, 0xd1, 0x98,
80           0x29, 0x4d, 0xd3, 0x8f, 0x56, 0xf2, 0xbb, 0x2e, 0x22, 0x3b}}};
81
82 # endif // SHA1
83
84 # if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
85
86 const TPM2B_EC_TEST c_TestEcDsa_r = {
87     {32, {0x04, 0x7d, 0x54, 0xeb, 0x04, 0x6f, 0x56, 0xec, 0xa2, 0x6c, 0x38,

```



```

88         0x8c, 0xeb, 0x43, 0x0b, 0x71, 0xf8, 0xf2, 0xf4, 0xa5, 0xe0, 0x1d,
89         0x3c, 0xa2, 0x39, 0x31, 0xe4, 0xe7, 0x36, 0x3b, 0xb5, 0x5f}}};
90 const TPM2B_EC_TEST c_TestEcDsa_s = {
91     {32, {0x8f, 0xd0, 0x12, 0xd9, 0x24, 0x75, 0xf6, 0xc4, 0x3b, 0xb5, 0x46,
92           0x75, 0x3a, 0x41, 0x8d, 0x80, 0x23, 0x99, 0x38, 0xd7, 0xe2, 0x40,
93           0xca, 0x9a, 0x19, 0x2a, 0xfc, 0x54, 0x75, 0xd3, 0x4a, 0x6e}}}};
94
95 const TPM2B_EC_TEST c_TestEcSchnorr_r = {
96     {32, {0xf7, 0xb9, 0x15, 0x4c, 0x34, 0xf6, 0x41, 0x19, 0xa3, 0xd2, 0xf1,
97           0xbd, 0xf4, 0x13, 0x6a, 0x4f, 0x63, 0xb8, 0x4d, 0xb5, 0xc8, 0xcd,
98           0xde, 0x85, 0x95, 0xa5, 0x39, 0x0a, 0x14, 0x49, 0x3d, 0x2f}}}};
99 const TPM2B_EC_TEST c_TestEcSchnorr_s = {
100     {32, {0xfe, 0xbe, 0x17, 0xaa, 0x31, 0x22, 0x9f, 0xd0, 0xd2, 0xf5, 0x25,
101           0x04, 0x92, 0xb0, 0xaa, 0x4e, 0xcc, 0x1c, 0xb6, 0x79, 0xd6, 0x42,
102           0xb3, 0x4e, 0x3f, 0xbb, 0xfe, 0x5f, 0xd0, 0xd0, 0x8b, 0xc3}}}};
103
104 # endif // SHA256
105
106 # if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
107
108 const TPM2B_EC_TEST c_TestEcDsa_r = {
109     {32, {0xf5, 0x74, 0x6d, 0xd6, 0xc6, 0x56, 0x86, 0xbb, 0xba, 0x1c, 0xba,
110           0x75, 0x65, 0xee, 0x64, 0x31, 0xce, 0x04, 0xe3, 0x9f, 0x24, 0x3f,
111           0xbd, 0xfe, 0x04, 0xcd, 0xab, 0x7e, 0xfe, 0xad, 0xcb, 0x82}}}};
112 const TPM2B_EC_TEST c_TestEcDsa_s = {
113     {32, {0xc2, 0x4f, 0x32, 0xa1, 0x06, 0xc0, 0x85, 0x4f, 0xc6, 0xd8, 0x31,
114           0x66, 0x91, 0x9f, 0x79, 0xcd, 0x5b, 0xe5, 0x7b, 0x94, 0xa1, 0x91,
115           0x38, 0xac, 0xd4, 0x20, 0xa2, 0x10, 0xf0, 0xd5, 0x9d, 0xbf}}}};
116
117 const TPM2B_EC_TEST c_TestEcSchnorr_r = {
118     {32, {0x1e, 0xb8, 0xe1, 0xbf, 0xa1, 0x9e, 0x39, 0x1e, 0x58, 0xa2, 0xe6,
119           0x59, 0xd0, 0x1a, 0x6a, 0x03, 0x6a, 0x1f, 0x1c, 0x4f, 0x36, 0x19,
120           0xc1, 0xec, 0x30, 0xa4, 0x85, 0x1b, 0xe9, 0x74, 0x35, 0x66}}}};
121 const TPM2B_EC_TEST c_TestEcSchnorr_s = {
122     {32, {0xb9, 0xe6, 0xe3, 0x7e, 0xcb, 0xb9, 0xea, 0xf1, 0xcc, 0xf4, 0x48,
123           0x44, 0x4a, 0xda, 0xc8, 0xd7, 0x87, 0xb4, 0xba, 0x40, 0xfe, 0x5b,
124           0x68, 0x11, 0x14, 0xcf, 0xa0, 0x0e, 0x85, 0x46, 0x99, 0x01}}}};
125
126 # endif // SHA384
127
128 # if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
129
130 const TPM2B_EC_TEST c_TestEcDsa_r = {
131     {32, {0xc9, 0x71, 0xa6, 0xb4, 0xaf, 0x46, 0x26, 0x8c, 0x27, 0x00, 0x06,
132           0x3b, 0x00, 0x0f, 0xa3, 0x17, 0x72, 0x48, 0x40, 0x49, 0x4d, 0x51,
133           0x4f, 0xa4, 0xcb, 0x7e, 0x86, 0xe9, 0xe7, 0xb4, 0x79, 0xb2}}}};
134 const TPM2B_EC_TEST c_TestEcDsa_s = {
135     {32, {0x87, 0xbc, 0xc0, 0xed, 0x74, 0x60, 0x9e, 0xfa, 0x4e, 0xe8, 0x16,
136           0xf3, 0xf9, 0x6b, 0x26, 0x07, 0x3c, 0x74, 0x31, 0x7e, 0xf0, 0x62,
137           0x46, 0xdc, 0xd6, 0x45, 0x22, 0x47, 0x3e, 0x0c, 0xa0, 0x02}}}};
138
139 const TPM2B_EC_TEST c_TestEcSchnorr_r = {
140     {32, {0xcc, 0x07, 0xad, 0x65, 0x91, 0xdd, 0xa0, 0x10, 0x23, 0xae, 0x53,
141           0xec, 0xdf, 0xf1, 0x50, 0x90, 0x16, 0x96, 0xf4, 0x45, 0x09, 0x73,
142           0x9c, 0x84, 0xb5, 0x5c, 0x5f, 0x08, 0x51, 0xcb, 0x60, 0x01}}}};
143 const TPM2B_EC_TEST c_TestEcSchnorr_s = {
144     {32, {0x55, 0x20, 0x21, 0x54, 0xe2, 0x49, 0x07, 0x47, 0x71, 0xf4, 0x99,
145           0x15, 0x54, 0xf3, 0xab, 0x14, 0xdb, 0x8e, 0xda, 0x79, 0xb6, 0x02,
146           0x0e, 0xe3, 0x5e, 0x6f, 0x2c, 0xb6, 0x05, 0xbd, 0x14, 0x10}}}};
147
148 # endif // SHA512
149
150 #endif // SELF_TEST_DATA

```

6.23 /tpm/include/private/Global.h

```

1  /** Description
2
3  // This file contains internal global type definitions and data declarations that
4  // are need between subsystems. The instantiation of global data is in Global.c.
5  // The initialization of global data is in the subsystem that is the primary owner
6  // of the data.
7  //
8  // The first part of this file has the 'typedefs' for structures and other defines
9  // used in many portions of the code. After the 'typedef' section, is a section that
10 // defines global values that are only present in RAM. The next three sections
11 // define the structures for the NV data areas: persistent, orderly, and state
12 // save. Additional sections define the data that is used in specific modules. That
13 // data is private to the module but is collected here to simplify the management
14 // of the instance data.
15 //
16 // All the data is instanced in Global.c.
17 #if !defined _TPM_H_
18 # error "Should only be instanced in TPM.h"
19 #endif
20
21 /** Includes
22
23 #ifndef GLOBAL_H
24 # define GLOBAL_H
25
26 _REDUCE_WARNING_LEVEL_(2)
27 # include <string.h>
28 # include <stddef.h>
29 _NORMAL_WARNING_LEVEL_
30
31 # include "GpMacros.h"
32 # include "Capabilities.h"
33 # include "TpmTypes.h" // requires GpMacros & Capabilities
34 # include "CommandAttributes.h"
35 # include "CryptTest.h"
36
37 # ifndef MATH_LIB
38 # error MATH_LIB required
39 # endif
40 # include LIB_INCLUDE(TpmTo, MATH_LIB, Math)
41
42 # include "CryptHash.h"
43 # include "CryptSym.h"
44 # include "CryptRand.h"
45 # include "CryptEcc.h"
46 # include "CryptRsa.h"
47 # include "CryptTest.h"
48 # include "NV.h"
49 # include "ACT.h"
50
51 /** Defines and Types
52
53 /**** Other Types
54 // An AUTH_VALUE is a BYTE array containing a digest (TPMU_HA)
55 typedef BYTE AUTH_VALUE[sizeof(TPMU_HA)];
56
57 // A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO
58 typedef BYTE TIME_INFO[sizeof(TPMS_TIME_INFO)];
59
60 // A NAME is a BYTE array that can contain a TPMU_NAME
61 typedef BYTE NAME[sizeof(TPMU_NAME)];
62
63 // Definition for a PROOF value
64 TPM2B_TYPE(PROOF, PROOF_SIZE);

```

```

65
66 // Definition for a Primary Seed value
67 TPM2B_TYPE(SEED, PRIMARY_SEED_SIZE);
68
69 // A CLOCK_NONCE is used to tag the time value in the authorization session and
70 // in the ticket computation so that the ticket expires when there is a time
71 // discontinuity. When the clock stops during normal operation, the nonce is
72 // 64-bit value kept in RAM but it is a 32-bit counter when the clock only stops
73 // during power events.
74 # if CLOCK_STOPS
75 typedef UINT64 CLOCK_NONCE;
76 # else
77 typedef UINT32 CLOCK_NONCE;
78 # endif
79
80 /** Loaded Object Structures
81 **** Description
82 // The structures in this section define the object layout as it exists in TPM
83 // memory.
84 //
85 // Two types of objects are defined: an ordinary object such as a key, and a
86 // sequence object that may be a hash, HMAC, or event.
87 //
88 **** OBJECT_ATTRIBUTES
89 // An OBJECT_ATTRIBUTES structure contains the variable attributes of an object.
90 // These properties are not part of the public properties but are used by the
91 // TPM in managing the object. An OBJECT_ATTRIBUTES is used in the definition of
92 // the OBJECT data type.
93
94 typedef struct
95 {
96     unsigned publicOnly : 1; //0) SET if only the public portion of
97                               // an object is loaded
98     unsigned epsHierarchy : 1; //1) SET if the object belongs to EPS
99                               // Hierarchy
100    unsigned ppsHierarchy : 1; //2) SET if the object belongs to PPS
101                               // Hierarchy
102    unsigned spsHierarchy : 1; //3) SET if the object belongs to SPS
103                               // Hierarchy
104    unsigned evict : 1; //4) SET if the object is a platform or
105                        // owner evict object. Platform-
106                        // evict object belongs to PPS
107                        // hierarchy, owner-evict object
108                        // belongs to SPS or EPS hierarchy.
109                        // This bit is also used to mark a
110                        // completed sequence object so it
111                        // will be flush when the
112                        // SequenceComplete command succeeds.
113    unsigned primary : 1; //5) SET for a primary object
114    unsigned temporary : 1; //6) SET for a temporary object
115    unsigned stClear : 1; //7) SET for an stClear object
116    unsigned hmacSeq : 1; //8) SET for an HMAC or MAC sequence
117                        // object
118    unsigned hashSeq : 1; //9) SET for a hash sequence object
119    unsigned eventSeq : 1; //10) SET for an event sequence object
120    unsigned ticketSafe : 1; //11) SET if a ticket is safe to create
121                        // for hash sequence object
122    unsigned firstBlock : 1; //12) SET if the first block of hash
123                        // data has been received. It
124                        // works with ticketSafe bit
125    unsigned isParent : 1; //13) SET if the key has the proper
126                        // attributes to be a parent key
127    // unsigned privateExp : 1; //14) SET when the private exponent
128    // of an RSA key has been
129    validated.
130    unsigned not_used_14 : 1;

```

```

130     unsigned occupied      : 1;  //15) SET when the slot is occupied.
131     unsigned derivation    : 1;  //16) SET when the key is a derivation
132                                     // parent
133     unsigned external : 1;      //17) SET when the object is loaded with
134                                     // TPM2_LoadExternal();
135 } OBJECT_ATTRIBUTES;
136
137 # if ALG_RSA
138 // There is an overload of the sensitive.rsa.t.size field of a TPMT_SENSITIVE when an
139 // RSA key is loaded. When the sensitive->sensitive contains an RSA key with all of
140 // the CRT values, then the MSB of the size field will be set to indicate that the
141 // buffer contains all 5 of the CRT private key values.
142 # define RSA_prime_flag 0x8000
143 # endif
144
145 /*** OBJECT Structure
146 // An OBJECT structure holds the object public, sensitive, and meta-data
147 // associated. This structure is implementation dependent. For this
148 // implementation, the structure is not optimized for space but rather
149 // for clarity of the reference implementation. Other implementations
150 // may choose to overlap portions of the structure that are not used
151 // simultaneously. These changes would necessitate changes to the source
152 // code but those changes would be compatible with the reference
153 // implementation.
154
155 typedef struct OBJECT
156 {
157     // The attributes field is required to be first followed by the publicArea.
158     // This allows the overlay of the object structure and a sequence structure
159     OBJECT_ATTRIBUTES attributes; // object attributes
160     TPMT_PUBLIC publicArea; // public area of an object
161     TPMT_SENSITIVE sensitive; // sensitive area of an object
162     TPM2B_NAME qualifiedName; // object qualified name
163     TPMI_DH_OBJECT evictHandle; // if the object is an evict object,
164                                     // the original handle is kept here.
165                                     // The 'working' handle will be the
166                                     // handle of an object slot.
167     TPM2B_NAME name; // Name of the object name. Kept here
168                                     // to avoid repeatedly computing it.
169     TPMI_RH_HIERARCHY hierarchy; // Hierarchy for the object. While the
170                                     // base hierarchy can be deduced from
171                                     // 'attributes', if the hierarchy is
172                                     // firmware-bound or SVN-bound then
173                                     // this field carries additional metadata
174                                     // needed to derive the proof value for
175                                     // the object.
176 } OBJECT;
177
178 /*** HASH_OBJECT Structure
179 // This structure holds a hash sequence object or an event sequence object.
180 //
181 // The first four components of this structure are manually set to be the same as
182 // the first four components of the object structure. This prevents the object
183 // from being inadvertently misused as sequence objects occupy the same memory as
184 // a regular object. A debug check is present to make sure that the offsets are
185 // what they are supposed to be.
186 // NOTE: In a future version, this will probably be renamed as SEQUENCE_OBJECT
187 typedef struct HASH_OBJECT
188 {
189     OBJECT_ATTRIBUTES attributes; // The attributes of the HASH object
190     TPMI_ALG_PUBLIC type; // algorithm
191     TPMI_ALG_HASH nameAlg; // name algorithm
192     TPMA_OBJECT objectAttributes; // object attributes
193
194     // The data below is unique to a sequence object
195     TPM2B_AUTH auth; // authorization for use of sequence

```

```

196     union
197     {
198         HASH_STATE hashState[HASH_COUNT];
199         HMAC_STATE hmacState;
200     } state;
201 } HASH_OBJECT;
202
203 typedef BYTE HASH_OBJECT_BUFFER[sizeof(HASH_OBJECT)];
204
205 /*** ANY_OBJECT
206 // This is the union for holding either a sequence object or a regular object
207 // for ContextSave and ContextLoad.
208 typedef union ANY_OBJECT
209 {
210     OBJECT      entity;
211     HASH_OBJECT hash;
212 } ANY_OBJECT;
213
214 typedef BYTE ANY_OBJECT_BUFFER[sizeof(ANY_OBJECT)];
215
216 /***AUTH_DUP Types
217 // These values are used in the authorization processing.
218
219 typedef UINT32 AUTH_ROLE;
220 # define AUTH_NONE ((AUTH_ROLE)(0))
221 # define AUTH_USER ((AUTH_ROLE)(1))
222 # define AUTH_ADMIN ((AUTH_ROLE)(2))
223 # define AUTH_DUP ((AUTH_ROLE)(3))
224
225 /*** Active Session Context
226 /*** Description
227 // The structures in this section define the internal structure of a session
228 // context.
229 //
230 /*** SESSION ATTRIBUTES
231 // The attributes in the SESSION_ATTRIBUTES structure track the various properties
232 // of the session. It maintains most of the tracking state information for the
233 // policy session. It is used within the SESSION structure.
234
235 typedef struct SESSION_ATTRIBUTES
236 {
237     // SET if the session may only be used for policy
238     unsigned isPolicy : 1;
239     // SET if the session is used for audit
240     unsigned isAudit : 1;
241     // SET if the session is bound to an entity. This attribute will be CLEAR if
242     // either isPolicy or isAudit is SET.
243     unsigned isBound : 1;
244     // SET if the cpHash has been defined. This attribute is not SET unless
245     // 'isPolicy' is SET.
246     unsigned isCpHashDefined : 1;
247     // SET if the nameHash has been defined. This attribute is not SET unless
248     // 'isPolicy' is SET.
249     unsigned isNameHashDefined : 1;
250     // SET if the pHash has been defined. This attribute is not SET unless
251     // 'isPolicy' is SET.
252     unsigned isParametersHashDefined : 1;
253     // SET if the templateHash needs to be checked for Create, CreatePrimary, or
254     // CreateLoaded.
255     unsigned isTemplateHashDefined : 1;
256     // SET if the authValue is required for computing the session HMAC. This
257     // attribute is not SET unless 'isPolicy' is SET.
258     unsigned isAuthValueNeeded : 1;
259     // SET if a password authValue is required for authorization This attribute
260     // is not SET unless 'isPolicy' is SET.
261     unsigned isPasswordNeeded : 1;

```



```

262 // SET if physical presence is required to be asserted when the
263 // authorization is checked. This attribute is not SET unless 'isPolicy' is
264 // SET.
265 unsigned isPPRequired : 1;
266 // SET if the policy session is created for trial of the policy's policyHash
267 // generation. This attribute is not SET unless 'isPolicy' is SET.
268 unsigned isTrialPolicy : 1;
269 // SET if the bind entity had noDA CLEAR. If this is SET, then an
270 // authorization failure using this session will count against lockout even
271 // if the object being authorized is exempt from DA.
272 unsigned isDaBound : 1;
273 // SET if the session is bound to lockoutAuth.
274 unsigned isLockoutBound : 1;
275 // This attribute is SET when the authValue of an object is to be included
276 // in the computation of the HMAC key for the command and response
277 // computations. (was 'requestWasBound')
278 unsigned includeAuth : 1;
279 // SET if the TPMA_NV_WRITTEN attribute needs to be checked when the policy
280 // is used for authorization for NV access. If this is SET for any other
281 // type, the policy will fail.
282 unsigned checkNvWritten : 1;
283 // SET if TPMA_NV_WRITTEN is required to be SET. Used when 'checkNvWritten'
284 // is SET
285 unsigned nvWrittenState : 1;
286 } SESSION_ATTRIBUTES;
287
288 /*** IsCpHashUnionOccupied()
289 // This function indicates whether the session attributes indicate that one of
290 // the members of the union containing `cpHash` are set.
291 BOOL IsCpHashUnionOccupied(SESSION_ATTRIBUTES attrs);
292
293 /*** SESSION Structure
294 // The SESSION structure contains all the context of a session except for the
295 // associated contextID.
296 //
297 // Note: The contextID of a session is only relevant when the session context
298 // is stored off the TPM.
299
300 typedef struct SESSION
301 {
302     SESSION_ATTRIBUTES attributes; // session attributes
303     UINT32 pcrCounter; // PCR counter value when PCR is
304                        // included (policy session)
305                        // If no PCR is included, this
306                        // value is 0.
307     UINT64 startTime; // The value in g_time when the session
308                       // was started (policy session)
309     UINT64 timeout; // The timeout relative to g_time
310                    // There is no timeout if this value
311                    // is 0.
312     CLOCK_NONCE epoch; // The g_clockEpoch value when the
313                        // session was started. If g_clockEpoch
314                        // does not match this value when the
315                        // timeout is used, then
316                        // then the command will fail.
317     TPM_CC commandCode; // command code (policy session)
318     TPM_ALG_ID authHashAlg; // session hash algorithm
319     TPMA_LOCALITY commandLocality; // command locality (policy session)
320     TPMT_SYM_DEF symmetric; // session symmetric algorithm (if any)
321     TPM2B_AUTH sessionKey; // session secret value used for
322                             // this session
323     TPM2B_NONCE nonceTPM; // last TPM-generated nonce for
324                           // generating HMAC and encryption keys
325     union
326     {
327         TPM2B_NAME boundEntity; // value used to track the entity to

```



```

328                                     // which the session is bound
329
330     TPM2B_DIGEST cpHash;             // the required cpHash value for the
331                                     // command being authorized
332     TPM2B_DIGEST nameHash;           // the required nameHash
333     TPM2B_DIGEST templateHash;       // the required template for creation
334     TPM2B_DIGEST pHash;              // the required parameter hash value for the
335                                     // command being authorized
336 } u1;
337
338 union
339 {
340     TPM2B_DIGEST auditDigest;        // audit session digest
341     TPM2B_DIGEST policyDigest;       // policyHash
342 } u2;                                // audit log and policyHash may
343                                     // share space to save memory
344 } SESSION;
345
346 # define EXPIRES_ON_RESET    INT32_MIN
347 # define TIMEOUT_ON_RESET    UINT64_MAX
348 # define EXPIRES_ON_RESTART  (INT32_MIN + 1)
349 # define TIMEOUT_ON_RESTART  (UINT64_MAX - 1)
350
351 typedef BYTE SESSION_BUF[sizeof(SESSION)];
352
353 /*******
354 /** PCR
355 /*******
356 /**PCR_SAVE Structure
357 /** The PCR_SAVE structure type contains the PCR data that are saved across power
358 /** cycles. Only the static PCR are required to be saved across power cycles. The
359 /** DRTM and resettable PCR are not saved. The number of static and resettable PCR
360 /** is determined by the platform-specific specification to which the TPM is built.
361
362 # define PCR_SAVE_SPACE(HASH, Hash) BYTE Hash[NUM_STATIC_PCR][HASH##_DIGEST_SIZE];
363
364 typedef struct PCR_SAVE
365 {
366     FOR_EACH_HASH(PCR_SAVE_SPACE)
367
368     // This counter increments whenever the PCR are updated.
369     // NOTE: A platform-specific specification may designate
370     // certain PCR changes as not causing this counter
371     // to increment.
372     UINT32 pcrCounter;
373 } PCR_SAVE;
374
375 /**PCR_POLICY
376 # if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
377 // This structure holds the PCR policies, one for each group of PCR controlled
378 // by policy.
379 typedef struct PCR_POLICY
380 {
381     TPMI_ALG_HASH hashAlg[NUM_POLICY_PCR_GROUP];
382     TPM2B_DIGEST a;
383     TPM2B_DIGEST policy[NUM_POLICY_PCR_GROUP];
384 } PCR_POLICY;
385 # endif
386
387 /**PCR_AUTHVALUE
388 // This structure holds the PCR policies, one for each group of PCR controlled
389 // by policy.
390 typedef struct PCR_AUTH_VALUE
391 {
392     TPM2B_DIGEST auth[NUM_AUTHVALUE_PCR_GROUP];
393 } PCR_AUTHVALUE;

```

```

394
395 /***STARTUP_TYPE
396 // This enumeration is the possible startup types. The type is determined
397 // by the combination of TPM2_ShutDown and TPM2_Startup.
398 typedef enum
399 {
400     SU_RESET,
401     SU_RESTART,
402     SU_RESUME
403 } STARTUP_TYPE;
404
405 /***NV
406
407 /***NV_INDEX
408 // The NV_INDEX structure defines the internal format for an NV index.
409 // The 'indexData' size varies according to the type of the index.
410 // In this implementation, all of the index is manipulated as a unit.
411 // NOTE: In this implementation of the TPM, the extended bits are always 0.
412 // Therefore, they are stored in the NV subsystem as legacy structures,
413 // even when the handle type indicates that the index can have extended
414 // attributes.
415 typedef struct NV_INDEX
416 {
417     TPMS_NV_PUBLIC publicArea;
418     TPM2B_AUTH      authValue;
419 } NV_INDEX;
420
421 /*** NV_REF
422 // An NV_REF is an opaque value returned by the NV subsystem. It is used to
423 // reference and NV Index in a relatively efficient way. Rather than having to
424 // continually search for an Index, its reference value may be used. In this
425 // implementation, an NV_REF is a byte pointer that points to the copy of the
426 // NV memory that is kept in RAM.
427 typedef UINT32 NV_REF;
428
429 typedef BYTE*   NV_RAM_REF;
430 /***NV_PIN
431 // This structure deals with the possible endianness differences between the
432 // canonical form of the TPMS_NV_PIN_COUNTER_PARAMETERS structure and the internal
433 // value. The structures allow the data in a PIN index to be read as an 8-octet
434 // value using NvReadUINT64Data(). That function will byte swap all the values on a
435 // little endian system. This will put the bytes with the 4-octet values in the
436 // correct order but will swap the pinLimit and pinCount values. When written, the
437 // PIN index is simply handled as a normal index with the octets in canonical order.
438 # if BIG_ENDIAN_TPM
439 typedef struct
440 {
441     UINT32 pinCount;
442     UINT32 pinLimit;
443 } PIN_DATA;
444 # else
445 typedef struct
446 {
447     UINT32 pinLimit;
448     UINT32 pinCount;
449 } PIN_DATA;
450 # endif
451
452 typedef union
453 {
454     UINT64   intVal;
455     PIN_DATA pin;
456 } NV_PIN;
457
458 /***COMMIT_INDEX_MASK
459 // This is the define for the mask value that is used when manipulating

```

```

460 // the bits in the commit bit array. The commit counter is a 64-bit
461 // value and the low order bits are used to index the commitArray.
462 // This mask value is applied to the commit counter to extract the
463 // bit number in the array.
464 # if ALG_ECC
465
466 #   define COMMIT_INDEX_MASK ((UINT16)((sizeof(gr.commitArray) * 8) - 1))
467
468 # endif
469
470 /*******
471 /*******
472 /** RAM Global Values
473 /*******
474 /*******
475 /** Description
476 // The values in this section are only extant in RAM or ROM as constant values.
477
478 /** Crypto Self-Test Values
479 EXTERN ALGORITHM_VECTOR g_implementedAlgorithms;
480 EXTERN ALGORITHM_VECTOR g_toTest;
481
482 /** g_rcIndex[]
483 // This array is used to contain the array of values that are added to a return
484 // code when it is a parameter-, handle-, or session-related error.
485 // This is an implementation choice and the same result can be achieved by using
486 // a macro.
487 extern const UINT16 g_rcIndex[15];
488
489 /** g_exclusiveAuditSession
490 // This location holds the session handle for the current exclusive audit
491 // session. If there is no exclusive audit session, the location is set to
492 // TPM_RH_UNASSIGNED.
493 EXTERN TPM_HANDLE g_exclusiveAuditSession;
494
495 /** g_time
496 // This is the value in which we keep the current command time. This is initialized
497 // at the start of each command. The time is the accumulated time since the last
498 // time that the TPM's timer was last powered up. Clock is the accumulated time
499 // since the last time that the TPM was cleared. g_time is in mS.
500 EXTERN UINT64 g_time;
501
502 /** g_timeEpoch
503 // This value contains the current clock Epoch. It changes when there is a clock
504 // discontinuity. It may be necessary to place this in NV should the timer be able
505 // to run across a power down of the TPM but not in all cases (e.g. dead battery).
506 // If the nonce is placed in NV, it should go in gp because it should be changing
507 // slowly.
508 # if CLOCK_STOPS
509 EXTERN CLOCK_NONCE g_timeEpoch;
510 # else
511 #   define g_timeEpoch gp.timeEpoch
512 # endif
513
514 /** g_phEnable
515 // This is the platform hierarchy control and determines if the platform hierarchy
516 // is available. This value is SET on each TPM2_Startup(). The default value is
517 // SET.
518 EXTERN BOOL g_phEnable;
519
520 /** g_pcrReConfig
521 // This value is SET if a TPM2_PCR_Allocate command successfully executed since
522 // the last TPM2_Startup(). If so, then the next shutdown is required to be
523 // Shutdown(CLEAR).
524 EXTERN BOOL g_pcrReConfig;
525

```

```

526  /*** g_DRTMHandle
527  // This location indicates the sequence object handle that holds the DRTM
528  // sequence data. When not used, it is set to TPM_RH_UNASSIGNED. A sequence
529  // DRTM sequence is started on either _TPM_Init or _TPM_Hash_Start.
530  EXTERN TPMI_DH_OBJECT g_DRTMHandle;
531
532  /*** g_DrtmPreStartup
533  // This value indicates that an H-CRTM occurred after _TPM_Init but before
534  // TPM2_Startup(). The define for PRE_STARTUP_FLAG is used to add the
535  // g_DrtmPreStartup value to gp_orderlyState at shutdown. This hack is to avoid
536  // adding another NV variable.
537  EXTERN BOOL g_DrtmPreStartup;
538
539  /*** g_StartupLocality3
540  // This value indicates that a TPM2_Startup() occurred at locality 3. Otherwise, it
541  // at locality 0. The define for STARTUP_LOCALITY_3 is to
542  // indicate that the startup was not at locality 0. This hack is to avoid
543  // adding another NV variable.
544  EXTERN BOOL g_StartupLocality3;
545
546  /***TPM_SU_NONE
547  // Part 2 defines the two shutdown/startup types that may be used in
548  // TPM2_Shutdown() and TPM2_Startup(). This additional define is
549  // used by the TPM to indicate that no shutdown was received.
550  // NOTE: This is a reserved value.
551  # define SU_NONE_VALUE (0xFFFF)
552  # define TPM_SU_NONE (TPM_SU) (SU_NONE_VALUE)
553
554  /*** TPM_SU_DA_USED
555  // As with TPM_SU_NONE, this value is added to allow indication that the shutdown
556  // was not orderly and that a DA=protected object was reference during the previous
557  // cycle.
558  # define SU_DA_USED_VALUE (SU_NONE_VALUE - 1)
559  # define TPM_SU_DA_USED (TPM_SU) (SU_DA_USED_VALUE)
560
561  /*** Startup Flags
562  // These flags are included in gp_orderlyState. These are hacks and are being
563  // used to avoid having to change the layout of gp. The PRE_STARTUP_FLAG indicates
564  // that a _TPM_Hash_Start/_Data/_End sequence was received after _TPM_Init but
565  // before TPM2_Startup(). STARTUP_LOCALITY_3 indicates that the last TPM2_Startup()
566  // was received at locality 3. These flags are only relevant if after a
567  // TPM2_Shutdown(STATE).
568  # define PRE_STARTUP_FLAG 0x8000
569  # define STARTUP_LOCALITY_3 0x4000
570
571  # if USE_DA_USED
572  /*** g_daUsed
573  // This location indicates if a DA-protected value is accessed during a boot
574  // cycle. If none has, then there is no need to increment 'failedTries' on the
575  // next non-orderly startup. This bit is merged with gp_orderlyState when
576  // gp_orderly is set to SU_NONE_VALUE
577  EXTERN BOOL g_daUsed;
578  # endif
579
580  /*** g_updateNV
581  // This flag indicates if NV should be updated at the end of a command.
582  // This flag is set to UT_NONE at the beginning of each command in ExecuteCommand().
583  // This flag is checked in ExecuteCommand() after the detailed actions of a command
584  // complete. If the command execution was successful and this flag is not UT_NONE,
585  // any pending NV writes will be committed to NV.
586  // UT_ORDERLY causes any RAM data to be written to the orderly space for staging
587  // the write to NV.
588  typedef BYTE UPDATE_TYPE;
589  # define UT_NONE (UPDATE_TYPE) 0
590  # define UT_NV (UPDATE_TYPE) 1
591  # define UT_ORDERLY (UPDATE_TYPE) (UT_NV + 2)

```

```

592  EXTERN UPDATE_TYPE g_updateNV;
593
594  /*** g_powerWasLost
595  // This flag is used to indicate if the power was lost. It is SET in _TPM_Init.
596  // This flag is cleared by TPM2_Startup() after all power-lost activities are
597  // completed.
598  // Note: When power is applied, this value can come up as anything. However,
599  // _plat_WasPowerLost() will provide the proper indication in that case. So, when
600  // power is actually lost, we get the correct answer. When power was not lost, but
601  // the power-lost processing has not been completed before the next _TPM_Init(),
602  // then the TPM still does the correct thing.
603  EXTERN BOOL g_powerWasLost;
604
605  /*** g_clearOrderly
606  // This flag indicates if the execution of a command should cause the orderly
607  // state to be cleared. This flag is set to FALSE at the beginning of each
608  // command in ExecuteCommand() and is checked in ExecuteCommand() after the
609  // detailed actions of a command complete but before the check of
610  // 'g_updateNV'. If this flag is TRUE, and the orderly state is not
611  // SU_NONE_VALUE, then the orderly state in NV memory will be changed to
612  // SU_NONE_VALUE or SU_DA_USED_VALUE.
613  EXTERN BOOL g_clearOrderly;
614
615  /*** g_prevOrderlyState
616  // This location indicates how the TPM was shut down before the most recent
617  // TPM2_Startup(). This value, along with the startup type, determines if
618  // the TPM should do a TPM Reset, TPM Restart, or TPM Resume.
619  EXTERN TPM_SU g_prevOrderlyState;
620
621  /*** g_nvOk
622  // This value indicates if the NV integrity check was successful or not. If not and
623  // the failure was severe, then the TPM would have been put into failure mode after
624  // it had been re-manufactured. If the NV failure was in the area where the state-save
625  // data is kept, then this variable will have a value of FALSE indicating that
626  // a TPM2_Startup(CLEAR) is required.
627  EXTERN BOOL g_nvOk;
628  // NV availability is sampled as the start of each command and stored here
629  // so that its value remains consistent during the command execution
630  EXTERN TPM_RC g_NvStatus;
631
632  /*** g_platformUnique
633
634  // This location contains unique value(s) used by the TPM Platform vendor.
635  // These are loaded on every _TPM2_Startup() using the _plat_GetUnique function.
636  // The "which" parameter to _plat_GetUnique indicates the value to return.
637  // If used, the TPM vendor is expected to use these values for authentication.
638  # if VENDOR_PERMANENT_AUTH_ENABLED == YES
639  // which = 1, the authorization value for VENDOR_PERMANENT_AUTH_HANDLE
640  EXTERN TPM2B_AUTH g_platformUniqueAuth;
641  # endif
642
643  /*******
644  /*******
645  /*** Persistent Global Values
646  /*******
647  /*******
648  /*** Description
649  // The values in this section are global values that are persistent across power
650  // events. The lifetime of the values determines the structure in which the value
651  // is placed.
652
653  /*******
654  /*** PERSISTENT_DATA
655  /*******
656  // This structure holds the persistent values that only change as a consequence
657  // of a specific Protected Capability and are not affected by TPM power events

```



```

658 // (TPM2_Startup() or TPM2_Shutdown()).
659 typedef struct
660 {
661     // data provided by the platform library during manufacturing.
662     // Opaque to the TPM Core library, but may be used by the platform library.
663     BYTE platformReserved[PERSISTENT_DATA_PLATFORM_SPACE];
664
665     /**
666      * // Hierarchy
667      */
668     // The values in this section are related to the hierarchies.
669
670     BOOL disableClear; // TRUE if TPM2_Clear() using
671                       // lockoutAuth is disabled
672
673     // Hierarchy authPolicies
674     TPMI_ALG_HASH ownerAlg;
675     TPMI_ALG_HASH endorsementAlg;
676     TPMI_ALG_HASH lockoutAlg;
677     TPM2B_DIGEST ownerPolicy;
678     TPM2B_DIGEST endorsementPolicy;
679     TPM2B_DIGEST lockoutPolicy;
680
681     // Hierarchy authValues
682     TPM2B_AUTH ownerAuth;
683     TPM2B_AUTH endorsementAuth;
684     TPM2B_AUTH lockoutAuth;
685
686     // Primary Seeds
687     TPM2B_SEED EPSeed;
688     TPM2B_SEED SPSeed;
689     TPM2B_SEED PPSeed;
690     // Note there is a nullSeed in the state_reset memory.
691
692     // Hierarchy proofs
693     TPM2B_PROOF phProof;
694     TPM2B_PROOF shProof;
695     TPM2B_PROOF ehProof;
696     // Note there is a nullProof in the state_reset memory.
697
698     /**
699      * // Reset Events
700      */
701     // A count that increments at each TPM reset and never get reset during the life
702     // time of TPM. The value of this counter is initialized to 1 during TPM
703     // manufacture process. It is used to invalidate all saved contexts after a TPM
704     // Reset.
705     UINT64 totalResetCount;
706
707     // This counter increments on each TPM Reset. The counter is reset by
708     // TPM2_Clear().
709     UINT32 resetCount;
710
711     /**
712      * // PCR
713      */
714     // This structure hold the policies for those PCR that have an update policy.
715     // This implementation only supports a single group of PCR controlled by
716     // policy. If more are required, then this structure would be changed to
717     // an array.
718     # if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
719         PCR_POLICY pcrPolicies;

```



```

720 # endif
721
722 // This structure indicates the allocation of PCR. The structure contains a
723 // list of PCR allocations for each implemented algorithm. If no PCR are
724 // allocated for an algorithm, a list entry still exists but the bit map
725 // will contain no SET bits.
726 TPML_PCR_SELECTION pcrAllocated;
727
728 //*****
729 //      Physical Presence
730
731 //*****
732 // The PP_LIST type contains a bit map of the commands that require physical
733 // to be asserted when the authorization is evaluated. Physical presence will be
734 // checked if the corresponding bit in the array is SET and if the authorization
735 // handle is TPM_RH_PLATFORM.
736 //
737 // These bits may be changed with TPM2_PP_Commands().
738 BYTE ppList[(COMMAND_COUNT + 7) / 8];
739
740 //*****
741 //      Dictionary attack values
742
743 //*****
744 // These values are used for dictionary attack tracking and control.
745
746 // the current count of unexpired
747 // authorization failures
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781

```

```

782     //          Firmware version
783     //*****
784     // The firmwareV1 and firmwareV2 values are instantiated in TimeStamp.c. This is
785     // a scheme used in development to allow determination of the linker build time
786     // of the TPM. An actual implementation would implement these values in a way that
787     // is consistent with vendor needs. The values are maintained in RAM for
simplified
788     // access with a master version in NV. These values are modified in a
789     // vendor-specific way.
790
791     // g_firmwareV1 contains the more significant 32-bits of the vendor version
number.
792     // In the reference implementation, if this value is printed as a hex
793     // value, it will have the format of YYYYMMDD
794     UINT32 firmwareV1;
795
796     // g_firmwareV1 contains the less significant 32-bits of the vendor version
number.
797     // In the reference implementation, if this value is printed as a hex
798     // value, it will have the format of 00 HH MM SS
799     UINT32 firmwareV2;
800     //*****
801     //          Timer Epoch
802     //*****
803     // timeEpoch contains a nonce that has a vendor-specific size (should not be
804     // less than 8 bytes. This nonce changes when the clock epoch changes. The clock
805     // epoch changes when there is a discontinuity in the timing of the TPM.
806     # if !CLOCK_STOPS
807         CLOCK_NONCE timeEpoch;
808     # endif
809
810 } PERSISTENT_DATA;
811
812 EXTERN PERSISTENT_DATA gp;
813
814 //*****
815 //*****
816 /*** ORDERLY DATA
817 //*****
818 //*****
819 // The data in this structure is saved to NV on each TPM2_Shutdown().
820 typedef struct orderly_data
821 {
822     //*****
823     //          TIME
824     //*****
825
826     // Clock has two parts. One is the state save part and one is the NV part. The
827     // state save version is updated on each command. When the clock rolls over, the
828     // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
829     // orderly way, then the sClock value is used to initialize the clock. If the
830     // TPM shutdown was not orderly, then the persistent value is used and the safe
831     // attribute is clear.
832
833     UINT64      clock;          // The orderly version of clock
834     TPMI_YES_NO clockSafe;     // Indicates if the clock value is
835                                // safe.
836
837     // In many implementations, the quality of the entropy available is not that
838     // high. To compensate, the current value of the drbgState can be saved and
839     // restored on each power cycle. This prevents the internal state from reverting
840     // to the initial state on each power cycle and starting with a limited amount
841     // of entropy. By keeping the old state and adding entropy, the entropy will
842     // accumulate.
843     DRBG_STATE drbgState;
844

```

```

845 // These values allow the accumulation of self-healing time across orderly shutdown
846 // of the TPM.
847 # if ACCUMULATE_SELF_HEAL_TIMER
848     UINT64 selfHealTimer; // current value of s_selfHealTimer
849     UINT64 lockoutTimer;  // current value of s_lockoutTimer
850     UINT64 time;          // current value of g_time at shutdown
851 # endif                  // ACCUMULATE_SELF_HEAL_TIMER
852
853 // These are the ACT Timeout values. They are saved with the other timers
854 # define DefineActData(N) ACT_STATE ACT_#N;
855     FOR_EACH_ACT(DefineActData)
856
857     // this is the 'signaled' attribute data for all the ACT. It is done this way so
858     // that they can be manipulated by ACT number rather than having to access a
859     // structure.
860     UINT16 signaledACT;
861     UINT16 preservedSignaled;
862
863 # if ORDERLY_DATA_PADDING != 0
864     BYTE reserved[ORDERLY_DATA_PADDING];
865 # endif
866
867 } ORDERLY_DATA;
868
869 # if ACCUMULATE_SELF_HEAL_TIMER
870 #     define s_selfHealTimer go.selfHealTimer
871 #     define s_lockoutTimer go.lockoutTimer
872 # endif // ACCUMULATE_SELF_HEAL_TIMER
873
874 # define drbgDefault go.drbgState
875
876 EXTERN ORDERLY_DATA go;
877
878 //*****
879 //*****
880 //*** STATE_CLEAR_DATA
881 //*****
882 //*****
883 // This structure contains the data that is saved on Shutdown(STATE)
884 // and restored on Startup(STATE). The values are set to their default
885 // settings on any Startup(Clear). In other words, the data is only persistent
886 // across TPM Resume.
887 //
888 // If the comments associated with a parameter indicate a default reset value, the
889 // value is applied on each Startup(CLEAR).
890
891 typedef struct state_clear_data
892 {
893     //*****
894     //          Hierarchy Control
895     //*****
896     BOOL      shEnable; // default reset is SET
897     BOOL      ehEnable; // default reset is SET
898     BOOL      phEnableNV; // default reset is SET
899     TPMI_ALG_HASH platformAlg; // default reset is TPM_ALG_NULL
900     TPM2B_DIGEST platformPolicy; // default reset is an Empty Buffer
901     TPM2B_AUTH platformAuth; // default reset is an Empty Buffer
902
903     //*****
904     //          PCR
905     //*****
906     // The set of PCR to be saved on Shutdown(STATE)
907     PCR_SAVE pcrSave; // default reset is 0...0
908
909     // This structure hold the authorization values for those PCR that have an
910     // update authorization.

```

```

911 // This implementation only supports a single group of PCR controlled by
912 // authorization. If more are required, then this structure would be changed to
913 // an array.
914 PCR_AUTHVALUE pcrAuthValues;
915
916 //*****
917 //          ACT
918 //*****
919 # define DefineActPolicySpace(N) TPMT_HA act_##N;
920     FOR_EACH_ACT(DefineActPolicySpace)
921
922 # if STATE_CLEAR_DATA_PADDING != 0
923     BYTE reserved[STATE_CLEAR_DATA_PADDING];
924 # endif
925 } STATE_CLEAR_DATA;
926
927 EXTERN STATE_CLEAR_DATA gc;
928
929 //*****
930 //*****
931 /*** State Reset Data
932 //*****
933 //*****
934 // This structure contains data is that is saved on Shutdown(STATE) and restored on
935 // the subsequent Startup(ANY). That is, the data is preserved across TPM Resume
936 // and TPM Restart.
937 //
938 // If a default value is specified in the comments this value is applied on
939 // TPM Reset.
940
941 typedef struct state_reset_data
942 {
943     //*****
944     //          Hierarchy Control
945     //*****
946     TPM2B_PROOF nullProof; // The proof value associated with
947                           // the TPM_RH_NULL hierarchy. The
948                           // default reset value is from the RNG.
949
950     TPM2B_SEED nullSeed; // The seed value for the TPM_RN_NULL
951                           // hierarchy. The default reset value
952                           // is from the RNG.
953
954     //*****
955     //          Context
956     //*****
957     // The 'clearCount' counter is incremented each time the TPM successfully executes
958     // a TPM Resume. The counter is included in each saved context that has 'stClear'
959     // SET (including descendants of keys that have 'stClear' SET). This prevents
960     these
961     // objects from being loaded after a TPM Resume.
962     // If 'clearCount' is at its maximum value when the TPM receives a
963     Shutdown(STATE),
964     // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
965     UINT32 clearCount; // The default reset value is 0.
966
967     UINT64 objectContextID; // This is the context ID for a saved
968                           // object context. The default reset
969                           // value is 0.
970     CONTEXT_SLOT contextArray[MAX_ACTIVE_SESSIONS]; // This array contains
971     // contains the values used to track
972     // the version numbers of saved
973     // contexts (see
974     // Session.c in for details). The
975     // default reset value is {0}.
976

```

```

975     CONTEXT_COUNTER contextCounter; // This is the value from which the
976                                     // 'contextID' is derived. The
977                                     // default reset value is {0}.
978
979     //*****
980     //      Command Audit
981     //*****
982     // When an audited command completes, ExecuteCommand() checks the return
983     // value. If it is TPM_RC_SUCCESS, and the command is an audited command, the
984     // TPM will extend the cpHash and rpHash for the command to this value. If this
985     // digest was the Zero Digest before the cpHash was extended, the audit counter
986     // is incremented.
987
988     TPM2B_DIGEST commandAuditDigest; // This value is set to an Empty Digest
989                                     // by TPM2_GetCommandAuditDigest() or a
990                                     // TPM Reset.
991
992     //*****
993     //      Boot counter
994     //*****
995
996     UINT32 restartCount; // This counter counts TPM Restarts.
997                       // The default reset value is 0.
998
999 //*****
1000 //      PCR
1001
1002 //*****
1003 // This counter increments whenever the PCR are updated. This counter is preserved
1004 // across TPM Resume even though the PCR are not preserved. This is because
1005 // sessions remain active across TPM Restart and the count value in the session
1006 // is compared to this counter so this counter must have values that are unique
1007 // as long as the sessions are active.
1008 // NOTE: A platform-specific specification may designate that certain PCR changes
1009 // do not increment this counter to increment.
1010
1011     UINT32 pcrCounter; // The default reset value is 0.
1012
1013 # if ALG_ECC
1014
1015     //*****
1016     //      ECDA
1017     //*****
1018
1019     UINT64 commitCounter; // This counter increments each time
1020                       // TPM2_Commit() returns
1021                       // TPM_RC_SUCCESS. The default reset
1022                       // value is 0.
1023
1024     TPM2B_NONCE commitNonce; // This random value is used to compute
1025                       // the commit values. The default reset
1026                       // value is from the RNG.
1027
1028     // This implementation relies on the number of bits in g_commitArray being a
1029     // power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
1030     BYTE commitArray[16]; // The default reset value is {0}.
1031
1032 # endif // ALG_ECC
1033 # if STATE_RESET_DATA_PADDING != 0
1034     BYTE reserved[STATE_RESET_DATA_PADDING];
1035 # endif
1036 } STATE_RESET_DATA;
1037
1038 EXTERN STATE_RESET_DATA gr;
1039
1040 /** NV Layout
1041  * The NV data organization is

```



```

1039 // 1) a PERSISTENT_DATA structure
1040 // 2) a STATE_RESET_DATA structure
1041 // 3) a STATE_CLEAR_DATA structure
1042 // 4) an ORDERLY_DATA structure
1043 // 5) the user defined NV index space
1044 # define NV_PERSISTENT_DATA (0)
1045 # define NV_STATE_RESET_DATA (NV_PERSISTENT_DATA + sizeof(PERSISTENT_DATA))
1046 # define NV_STATE_CLEAR_DATA (NV_STATE_RESET_DATA + sizeof(STATE_RESET_DATA))
1047 # define NV_ORDERLY_DATA (NV_STATE_CLEAR_DATA + sizeof(STATE_CLEAR_DATA))
1048 # define NV_INDEX_RAM_DATA (NV_ORDERLY_DATA + sizeof(ORDERLY_DATA))
1049 # define NV_USER_DYNAMIC (NV_INDEX_RAM_DATA + sizeof(s_indexOrderlyRam))
1050 # define NV_USER_DYNAMIC_END NV_MEMORY_SIZE
1051
1052 /** Global Macro Definitions
1053 // The NV_READ_PERSISTENT and NV_WRITE_PERSISTENT macros are used to access members
1054 // of the PERSISTENT_DATA structure in NV.
1055 # define NV_READ_PERSISTENT(to, from) \
1056     NvRead(&to, offsetof(PERSISTENT_DATA, from), sizeof(to))
1057
1058 # define NV_WRITE_PERSISTENT(to, from) \
1059     NvWrite(offsetof(PERSISTENT_DATA, to), sizeof(gp.to), &from)
1060
1061 # define CLEAR_PERSISTENT(item) \
1062     NvClearPersistent(offsetof(PERSISTENT_DATA, item), sizeof(gp.item))
1063
1064 # define NV_SYNC_PERSISTENT(item) NV_WRITE_PERSISTENT(item, gp.item)
1065
1066 // At the start of command processing, the index of the command is determined. This
1067 // index value is used to access the various data tables that contain per-command
1068 // information. There are multiple options for how the per-command tables can be
1069 // implemented. This is resolved in GetClosestCommandIndex().
1070 typedef UINT16 COMMAND_INDEX;
1071 # define UNIMPLEMENTED_COMMAND_INDEX ((COMMAND_INDEX) (~0))
1072
1073 typedef struct _COMMAND_FLAGS_
1074 {
1075     unsigned trialPolicy : 1; //1) If SET, one of the handles references a
1076                               // trial policy and authorization may be
1077                               // skipped. This is only allowed for a policy
1078                               // command.
1079 } COMMAND_FLAGS;
1080
1081 // This structure is used to avoid having to manage a large number of
1082 // parameters being passed through various levels of the command input processing.
1083 //
1084
1085 // The following macros are used to define the space for the CP and RP hashes. Space,
1086 // is provided for each implemented hash algorithm because it is not known what the
1087 // caller may use.
1088 # define CP_HASH(HASH, Hash) TPM2B_###HASH##_DIGEST Hash##CpHash;
1089 # define RP_HASH(HASH, Hash) TPM2B_###HASH##_DIGEST Hash##RpHash;
1090
1091 typedef struct COMMAND
1092 {
1093     TPM_ST tag; // the parsed command tag
1094     TPM_CC code; // the parsed command code
1095     COMMAND_INDEX index; // the computed command index
1096     UINT32 handleNum; // the number of entity handles in the
1097                      // handle area of the command
1098     TPM_HANDLE handles[MAX_HANDLE_NUM]; // the parsed handle values
1099     UINT32 sessionNum; // the number of sessions found
1100     INT32 parameterSize; // starts out with the parsed command size
1101                          // and is reduced and values are
1102                          // unmarshaled. Just before calling the
1103                          // command actions, this should be zero.
1104                          // After the command actions, this number

```



```

1105                                     // should grow as values are marshaled
1106                                     // in to the response buffer.
1107     INT32 authSize;                                     // this is initialized with the parsed size
1108                                                         // of authorizationSize field and should
1109                                                         // be zero when the authorizations are
1110                                                         // parsed.
1111     BYTE* parameterBuffer;                             // input to ExecuteCommand
1112     BYTE* responseBuffer;                             // input to ExecuteCommand
1113     FOR_EACH_HASH(CP_HASH)                             // space for the CP hashes
1114     FOR_EACH_HASH(RP_HASH)                             // space for the RP hashes
1115 } COMMAND;
1116
1117 // TPM2B String constants used for KDFs.
1118 // actual definition in global.c
1119 extern const TPM2B* PRIMARY_OBJECT_CREATION;
1120 extern const TPM2B* CFB_KEY;
1121 extern const TPM2B* CONTEXT_KEY;
1122 extern const TPM2B* INTEGRITY_KEY;
1123 extern const TPM2B* SECRET_KEY;
1124 extern const TPM2B* HIERARCHY_PROOF_SECRET_LABEL;
1125 extern const TPM2B* HIERARCHY_SEED_SECRET_LABEL;
1126 extern const TPM2B* HIERARCHY_FW_SECRET_LABEL;
1127 extern const TPM2B* HIERARCHY_SVN_SECRET_LABEL;
1128 extern const TPM2B* SESSION_KEY;
1129 extern const TPM2B* STORAGE_KEY;
1130 extern const TPM2B* XOR_KEY;
1131 extern const TPM2B* COMMIT_STRING;
1132 extern const TPM2B* DUPLICATE_STRING;
1133 extern const TPM2B* IDENTITY_STRING;
1134 extern const TPM2B* OBFUSCATE_STRING;
1135 # if SELF_TEST
1136 extern const TPM2B* OAEP_TEST_STRING;
1137 # endif // SELF_TEST
1138
1139 //*****
1140 /** From CryptTest.c
1141 //*****
1142 // This structure contains the self-test state values for the cryptographic modules.
1143 EXTERN CRYPTO_SELF_TEST_STATE g_cryptoSelfTestState;
1144
1145 //*****
1146 /** From Manufacture.c
1147 //*****
1148 extern BOOL g_manufactured;
1149
1150 // This value indicates if a TPM2_Startup commands has been
1151 // receive since the power on event. This flag is maintained in power
1152 // simulation module because this is the only place that may reliably set this
1153 // flag to FALSE.
1154 EXTERN BOOL g_initialized;
1155
1156 /** Private data
1157
1158 //*****
1159 /** From SessionProcess.c
1160 //*****
1161 # if defined SESSION_PROCESS_C || defined GLOBAL_C || defined MANUFACTURE_C
1162 // The following arrays are used to save command sessions information so that the
1163 // command handle/session buffer does not have to be preserved for the duration of
1164 // the command. These arrays are indexed by the session index in accordance with
1165 // the order of sessions in the session area of the command.
1166 //
1167 // Array of the authorization session handles
1168 EXTERN TPM_HANDLE s_sessionHandles[MAX_SESSION_NUM];
1169
1170 // Array of authorization session attributes

```

```

1171 EXTERN TPMA_SESSION s_attributes[MAX_SESSION_NUM];
1172
1173 // Array of handles authorized by the corresponding authorization sessions;
1174 // and if none, then TPM_RH_UNASSIGNED value is used
1175 EXTERN TPM_HANDLE s_associatedHandles[MAX_SESSION_NUM];
1176
1177 // Array of nonces provided by the caller for the corresponding sessions
1178 EXTERN TPM2B_NONCE s_nonceCaller[MAX_SESSION_NUM];
1179
1180 // Array of authorization values (HMAC's or passwords) for the corresponding
1181 // sessions
1182 EXTERN TPM2B_AUTH s_inputAuthValues[MAX_SESSION_NUM];
1183
1184 // Array of pointers to the SESSION structures for the sessions in a command
1185 EXTERN SESSION* s_usedSessions[MAX_SESSION_NUM];
1186
1187 // Special value to indicate an undefined session index
1188 #   define UNDEFINED_INDEX (0xFFFF)
1189
1190 // Index of the session used for encryption of a response parameter
1191 EXTERN UINT32 s_encryptSessionIndex;
1192
1193 // Index of the session used for decryption of a command parameter
1194 EXTERN UINT32 s_decryptSessionIndex;
1195
1196 // Index of a session used for audit
1197 EXTERN UINT32 s_auditSessionIndex;
1198
1199 // The cpHash for command audit
1200 #   if CC_GetCommandAuditDigest
1201 EXTERN TPM2B_DIGEST s_cpHashForCommandAudit;
1202 #   endif
1203
1204 // Flag indicating if NV update is pending for the lockOutAuthEnabled or
1205 // failedTries DA parameter
1206 EXTERN BOOL s_DAPendingOnNV;
1207
1208 #   endif // SESSION_PROCESS_C
1209
1210 //*****
1211 //*** From DA.c
1212 //*****
1213 #   if defined DA_C || defined GLOBAL_C || defined MANUFACTURE_C
1214 // This variable holds the accumulated time since the last time
1215 // that 'failedTries' was decremented. This value is in millisecond.
1216 #   if !ACCUMULATE_SELF_HEAL_TIMER
1217 EXTERN UINT64 s_selfHealTimer;
1218
1219 // This variable holds the accumulated time that the lockoutAuth has been
1220 // blocked.
1221 EXTERN UINT64 s_lockoutTimer;
1222 #   endif // ACCUMULATE_SELF_HEAL_TIMER
1223
1224 #   endif // DA_C
1225
1226 //*****
1227 //*** From NV.c
1228 //*****
1229 #   if defined NV_C || defined GLOBAL_C
1230 // This marks the end of the NV area. This is a run-time variable as it might
1231 // not be compile-time constant.
1232 EXTERN NV_REF s_evictNvEnd;
1233
1234 // This space is used to hold the index data for an orderly Index. It also contains
1235 // the attributes for the index.
1236 EXTERN BYTE s_indexOrderlyRam[RAM_INDEX_SPACE]; // The orderly NV Index data

```

```

1237
1238 // This value contains the current max counter value. It is written to the end of
1239 // allocatable NV space each time an index is deleted or added. This value is
1240 // initialized on Startup. The indices are searched and the maximum of all the
1241 // current counter indices and this value is the initial value for this.
1242 EXTERN UINT64 s_maxCounter;
1243
1244 // This is space used for the NV Index cache. As with a persistent object, the
1245 // contents of a referenced index are copied into the cache so that the
1246 // NV Index memory scanning and data copying can be reduced.
1247 // Only code that operates on NV Index data should use this cache directly. When
1248 // that action code runs, s_lastNvIndex will contain the index header information.
1249 // It will have been loaded when the handles were verified.
1250 // NOTE: An NV index handle can appear in many commands that do not operate on the
1251 // NV data (e.g. TPM2_StartAuthSession). However, only one NV Index at a time is
1252 // ever directly referenced by any command. If that changes, then the NV Index
1253 // caching needs to be changed to accommodate that. Currently, the code will verify
1254 // that only one NV Index is referenced by the handles of the command.
1255 EXTERN NV_INDEX s_cachedNvIndex;
1256 EXTERN NV_REF s_cachedNvRef;
1257 EXTERN BYTE* s_cachedNvRamRef;
1258
1259 // Initial NV Index/evict object iterator value
1260 # define NV_REF_INIT (NV_REF)0xFFFFFFFF
1261
1262 # endif
1263
1264 //*****
1265 //*** From Object.c
1266 //*****
1267 # if defined OBJECT_C || defined GLOBAL_C
1268 // This type is the container for an object.
1269
1270 EXTERN OBJECT s_objects[MAX_LOADED_OBJECTS];
1271
1272 # endif // OBJECT_C
1273
1274 //*****
1275 //*** From PCR.c
1276 //*****
1277 # if defined PCR_C || defined GLOBAL_C
1278 # include <platform_interface/pcrstruct.h>
1279
1280 EXTERN PCR s_pcrs[IMPLEMENTATION_PCR];
1281
1282 # endif // PCR_C
1283
1284 //*****
1285 //*** From Session.c
1286 //*****
1287 # if defined SESSION_C || defined GLOBAL_C
1288 // Container for HMAC or policy session tracking information
1289 typedef struct
1290 {
1291     BOOL occupied;
1292     SESSION session; // session structure
1293 } SESSION_SLOT;
1294
1295 EXTERN SESSION_SLOT s_sessions[MAX_LOADED_SESSIONS];
1296
1297 // The index in contextArray that has the value of the oldest saved session
1298 // context. When no context is saved, this will have a value that is greater
1299 // than or equal to MAX_ACTIVE_SESSIONS.
1300 EXTERN UINT32 s_oldestSavedSession;
1301
1302 // The number of available session slot openings. When this is 1,

```

```

1303 // a session can't be created or loaded if the GAP is maxed out.
1304 // The exception is that the oldest saved session context can always
1305 // be loaded (assuming that there is a space in memory to put it)
1306 EXTERN int s_freeSessionSlots;
1307
1308 # endif // SESSION_C
1309
1310 //*****
1311 //*** From IoBuffers.c
1312 //*****
1313 # if defined IO_BUFFER_C || defined GLOBAL_C
1314 // Each command function is allowed a structure for the inputs to the function and
1315 // a structure for the outputs. The command dispatch code unmarshals the input butter
1316 // to the command action input structure starting at the first byte of
1317 // s_actionIoBuffer. The value of s_actionIoAllocation is the number of UINT64 values
1318 // allocated. It is used to set the pointer for the response structure. The command
1319 // dispatch code will marshal the response values into the final output buffer.
1320 EXTERN UINT64 s_actionIoBuffer[768]; // action I/O buffer
1321 EXTERN UINT32 s_actionIoAllocation; // number of UIN64 allocated for the
1322 // action input structure
1323 # endif // IO_BUFFER_C
1324
1325 //*****
1326 //*** From TPMFail.c
1327 //*****
1328 // This value holds the address of the string containing the name of the function
1329 // in which the failure occurred. This address value is not useful for anything
1330 // other than helping the vendor to know in which file the failure occurred.
1331 EXTERN BOOL g_inFailureMode; // Indicates that the TPM is in failure mode
1332 # if ALLOW_FORCE_FAILURE_MODE
1333 EXTERN BOOL g_forceFailureMode; // flag to force failure mode during test
1334 # endif
1335
1336 # if FAIL_TRACE
1337 // The name of the function that triggered failure mode.
1338 EXTERN const char* s_failFunctionName;
1339 # endif // FAIL_TRACE
1340 // A numeric indicator of the function that triggered failure mode.
1341 EXTERN UINT32 s_failFunction;
1342 // The line in the file at which the error was signaled.
1343 EXTERN UINT32 s_failLine;
1344 // the reason for the failure.
1345 EXTERN UINT32 s_failCode;
1346
1347 //*****
1348 //*** From ACT_spt.c
1349 //*****
1350 // This value is used to indicate if an ACT has been updated since the last
1351 // TPM2_Startup() (one bit for each ACT). If the ACT is not updated
1352 // (TPM2_ACT_SetTimeout()) after a startup, then on each TPM2_Shutdown() the TPM will
1353 // save 1/2 of the current timer value. This prevents an attack on the ACT by saving
1354 // the counter and then running for a long period of time before doing a TPM Restart.
1355 // A quick TPM2_Shutdown() after each
1356 EXTERN UINT16 s_ActUpdated;
1357
1358 //*****
1359 //*** From CommandCodeAttributes.c
1360 //*****
1361 // This array is instantiated in CommandCodeAttributes.c when it includes
1362 // CommandCodeAttributes.h. Don't change the extern to EXTERN.
1363 extern const TPMA_CC s_ccAttr[];
1364 extern const COMMAND_ATTRIBUTES s_commandAttributes[];
1365
1366 #endif // GLOBAL_H

```

6.24 /tpm/include/private/HandleProcess.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2  // clang-format off
3
4  #if CC_Startup
5  case TPM_CC_Startup:
6      break;
7  #endif // CC_Startup
8  #if CC_Shutdown
9  case TPM_CC_Shutdown:
10     break;
11 #endif // CC_Shutdown
12 #if CC_SelfTest
13 case TPM_CC_SelfTest:
14     break;
15 #endif // CC_SelfTest
16 #if CC_IncrementalSelfTest
17 case TPM_CC_IncrementalSelfTest:
18     break;
19 #endif // CC_IncrementalSelfTest
20 #if CC_GetTestResult
21 case TPM_CC_GetTestResult:
22     break;
23 #endif // CC_GetTestResult
24 #if CC_StartAuthSession
25 case TPM_CC_StartAuthSession:
26     *handleCount = 2;
27     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
28                                     bufferRemainingSize, TRUE);
29     if (TPM_RC_SUCCESS != result)
30     {
31         return result + TPM_RC_H + TPM_RC_1;
32     }
33     result = TPMI_DH_ENTITY_Unmarshal(&handles[1], handleBufferStart,
34                                     bufferRemainingSize, TRUE);
35     if (TPM_RC_SUCCESS != result)
36     {
37         return result + TPM_RC_H + TPM_RC_2;
38     }
39     break;
40 #endif // CC_StartAuthSession
41 #if CC_PolicyRestart
42 case TPM_CC_PolicyRestart:
43     *handleCount = 1;
44     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
45                                     bufferRemainingSize);
46     if (TPM_RC_SUCCESS != result)
47     {
48         return result + TPM_RC_H + TPM_RC_1;
49     }
50     break;
51 #endif // CC_PolicyRestart
52 #if CC_Create
53 case TPM_CC_Create:
54     *handleCount = 1;
55     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
56                                     bufferRemainingSize, FALSE);
57     if (TPM_RC_SUCCESS != result)
58     {
59         return result + TPM_RC_H + TPM_RC_1;
60     }
61     break;
62 #endif // CC_Create
63 #if CC_Load
64 case TPM_CC_Load:

```

```

65     *handleCount = 1;
66     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
67                                     bufferRemainingSize, FALSE);
68     if (TPM_RC_SUCCESS != result)
69     {
70         return result + TPM_RC_H + TPM_RC_1;
71     }
72     break;
73 #endif // CC_Load
74 #if CC_LoadExternal
75 case TPM_CC_LoadExternal:
76     break;
77 #endif // CC_LoadExternal
78 #if CC_ReadPublic
79 case TPM_CC_ReadPublic:
80     *handleCount = 1;
81     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
82                                     bufferRemainingSize, FALSE);
83     if (TPM_RC_SUCCESS != result)
84     {
85         return result + TPM_RC_H + TPM_RC_1;
86     }
87     break;
88 #endif // CC_ReadPublic
89 #if CC_ActivateCredential
90 case TPM_CC_ActivateCredential:
91     *handleCount = 2;
92     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
93                                     bufferRemainingSize, FALSE);
94     if (TPM_RC_SUCCESS != result)
95     {
96         return result + TPM_RC_H + TPM_RC_1;
97     }
98     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
99                                     bufferRemainingSize, FALSE);
100    if (TPM_RC_SUCCESS != result)
101    {
102        return result + TPM_RC_H + TPM_RC_2;
103    }
104    break;
105 #endif // CC_ActivateCredential
106 #if CC_MakeCredential
107 case TPM_CC_MakeCredential:
108     *handleCount = 1;
109     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
110                                     bufferRemainingSize, FALSE);
111     if (TPM_RC_SUCCESS != result)
112     {
113         return result + TPM_RC_H + TPM_RC_1;
114     }
115     break;
116 #endif // CC_MakeCredential
117 #if CC_Unseal
118 case TPM_CC_Unseal:
119     *handleCount = 1;
120     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
121                                     bufferRemainingSize, FALSE);
122     if (TPM_RC_SUCCESS != result)
123     {
124         return result + TPM_RC_H + TPM_RC_1;
125     }
126     break;
127 #endif // CC_Unseal
128 #if CC_ObjectChangeAuth
129 case TPM_CC_ObjectChangeAuth:
130     *handleCount = 2;

```



```

131     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
132                                     bufferRemainingSize, FALSE);
133     if (TPM_RC_SUCCESS != result)
134     {
135         return result + TPM_RC_H + TPM_RC_1;
136     }
137     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
138                                     bufferRemainingSize, FALSE);
139     if (TPM_RC_SUCCESS != result)
140     {
141         return result + TPM_RC_H + TPM_RC_2;
142     }
143     break;
144 #endif // CC_ObjectChangeAuth
145 #if CC_CreateLoaded
146 case TPM_CC_CreateLoaded:
147     *handleCount = 1;
148     result = TPMI_DH_PARENT_Unmarshal(&handles[0], handleBufferStart,
149                                     bufferRemainingSize);
150     if (TPM_RC_SUCCESS != result)
151     {
152         return result + TPM_RC_H + TPM_RC_1;
153     }
154     break;
155 #endif // CC_CreateLoaded
156 #if CC_Duplicate
157 case TPM_CC_Duplicate:
158     *handleCount = 2;
159     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
160                                     bufferRemainingSize, FALSE);
161     if (TPM_RC_SUCCESS != result)
162     {
163         return result + TPM_RC_H + TPM_RC_1;
164     }
165     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
166                                     bufferRemainingSize, TRUE);
167     if (TPM_RC_SUCCESS != result)
168     {
169         return result + TPM_RC_H + TPM_RC_2;
170     }
171     break;
172 #endif // CC_Duplicate
173 #if CC_Rewrap
174 case TPM_CC_Rewrap:
175     *handleCount = 2;
176     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
177                                     bufferRemainingSize, TRUE);
178     if (TPM_RC_SUCCESS != result)
179     {
180         return result + TPM_RC_H + TPM_RC_1;
181     }
182     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
183                                     bufferRemainingSize, TRUE);
184     if (TPM_RC_SUCCESS != result)
185     {
186         return result + TPM_RC_H + TPM_RC_2;
187     }
188     break;
189 #endif // CC_Rewrap
190 #if CC_Import
191 case TPM_CC_Import:
192     *handleCount = 1;
193     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
194                                     bufferRemainingSize, FALSE);
195     if (TPM_RC_SUCCESS != result)
196     {

```

```

197         return result + TPM_RC_H + TPM_RC_1;
198     }
199     break;
200 #endif // CC_Import
201 #if CC_RSA_Encrypt
202 case TPM_CC_RSA_Encrypt:
203     *handleCount = 1;
204     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
205                                     bufferRemainingSize, FALSE);
206     if (TPM_RC_SUCCESS != result)
207     {
208         return result + TPM_RC_H + TPM_RC_1;
209     }
210     break;
211 #endif // CC_RSA_Encrypt
212 #if CC_RSA_Decrypt
213 case TPM_CC_RSA_Decrypt:
214     *handleCount = 1;
215     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
216                                     bufferRemainingSize, FALSE);
217     if (TPM_RC_SUCCESS != result)
218     {
219         return result + TPM_RC_H + TPM_RC_1;
220     }
221     break;
222 #endif // CC_RSA_Decrypt
223 #if CC_ECDH_KeyGen
224 case TPM_CC_ECDH_KeyGen:
225     *handleCount = 1;
226     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
227                                     bufferRemainingSize, FALSE);
228     if (TPM_RC_SUCCESS != result)
229     {
230         return result + TPM_RC_H + TPM_RC_1;
231     }
232     break;
233 #endif // CC_ECDH_KeyGen
234 #if CC_ECDH_ZGen
235 case TPM_CC_ECDH_ZGen:
236     *handleCount = 1;
237     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
238                                     bufferRemainingSize, FALSE);
239     if (TPM_RC_SUCCESS != result)
240     {
241         return result + TPM_RC_H + TPM_RC_1;
242     }
243     break;
244 #endif // CC_ECDH_ZGen
245 #if CC_ECC_Parameters
246 case TPM_CC_ECC_Parameters:
247     break;
248 #endif // CC_ECC_Parameters
249 #if CC_ZGen_2Phase
250 case TPM_CC_ZGen_2Phase:
251     *handleCount = 1;
252     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
253                                     bufferRemainingSize, FALSE);
254     if (TPM_RC_SUCCESS != result)
255     {
256         return result + TPM_RC_H + TPM_RC_1;
257     }
258     break;
259 #endif // CC_ZGen_2Phase
260 #if CC_ECC_Encrypt
261 case TPM_CC_ECC_Encrypt:
262     *handleCount = 1;

```

```

263     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
264                                     bufferRemainingSize, FALSE);
265     if (TPM_RC_SUCCESS != result)
266     {
267         return result + TPM_RC_H + TPM_RC_1;
268     }
269     break;
270 #endif // CC_ECC_Encrypt
271 #if CC_ECC_Decrypt
272 case TPM_CC_ECC_Decrypt:
273     *handleCount = 1;
274     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
275                                     bufferRemainingSize, FALSE);
276     if (TPM_RC_SUCCESS != result)
277     {
278         return result + TPM_RC_H + TPM_RC_1;
279     }
280     break;
281 #endif // CC_ECC_Decrypt
282 #if CC_EncryptDecrypt
283 case TPM_CC_EncryptDecrypt:
284     *handleCount = 1;
285     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
286                                     bufferRemainingSize, FALSE);
287     if (TPM_RC_SUCCESS != result)
288     {
289         return result + TPM_RC_H + TPM_RC_1;
290     }
291     break;
292 #endif // CC_EncryptDecrypt
293 #if CC_EncryptDecrypt2
294 case TPM_CC_EncryptDecrypt2:
295     *handleCount = 1;
296     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
297                                     bufferRemainingSize, FALSE);
298     if (TPM_RC_SUCCESS != result)
299     {
300         return result + TPM_RC_H + TPM_RC_1;
301     }
302     break;
303 #endif // CC_EncryptDecrypt2
304 #if CC_Hash
305 case TPM_CC_Hash:
306     break;
307 #endif // CC_Hash
308 #if CC_HMAC
309 case TPM_CC_HMAC:
310     *handleCount = 1;
311     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
312                                     bufferRemainingSize, FALSE);
313     if (TPM_RC_SUCCESS != result)
314     {
315         return result + TPM_RC_H + TPM_RC_1;
316     }
317     break;
318 #endif // CC_HMAC
319 #if CC_MAC
320 case TPM_CC_MAC:
321     *handleCount = 1;
322     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
323                                     bufferRemainingSize, FALSE);
324     if (TPM_RC_SUCCESS != result)
325     {
326         return result + TPM_RC_H + TPM_RC_1;
327     }
328     break;

```

```

329 #endif // CC_MAC
330 #if CC_GetRandom
331 case TPM_CC_GetRandom:
332     break;
333 #endif // CC_GetRandom
334 #if CC_StirRandom
335 case TPM_CC_StirRandom:
336     break;
337 #endif // CC_StirRandom
338 #if CC_HMAC_Start
339 case TPM_CC_HMAC_Start:
340     *handleCount = 1;
341     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
342                                     bufferRemainingSize, FALSE);
343     if (TPM_RC_SUCCESS != result)
344     {
345         return result + TPM_RC_H + TPM_RC_1;
346     }
347     break;
348 #endif // CC_HMAC_Start
349 #if CC_MAC_Start
350 case TPM_CC_MAC_Start:
351     *handleCount = 1;
352     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
353                                     bufferRemainingSize, FALSE);
354     if (TPM_RC_SUCCESS != result)
355     {
356         return result + TPM_RC_H + TPM_RC_1;
357     }
358     break;
359 #endif // CC_MAC_Start
360 #if CC_HashSequenceStart
361 case TPM_CC_HashSequenceStart:
362     break;
363 #endif // CC_HashSequenceStart
364 #if CC_SequenceUpdate
365 case TPM_CC_SequenceUpdate:
366     *handleCount = 1;
367     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
368                                     bufferRemainingSize, FALSE);
369     if (TPM_RC_SUCCESS != result)
370     {
371         return result + TPM_RC_H + TPM_RC_1;
372     }
373     break;
374 #endif // CC_SequenceUpdate
375 #if CC_SequenceComplete
376 case TPM_CC_SequenceComplete:
377     *handleCount = 1;
378     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
379                                     bufferRemainingSize, FALSE);
380     if (TPM_RC_SUCCESS != result)
381     {
382         return result + TPM_RC_H + TPM_RC_1;
383     }
384     break;
385 #endif // CC_SequenceComplete
386 #if CC_EventSequenceComplete
387 case TPM_CC_EventSequenceComplete:
388     *handleCount = 2;
389     result = TPMI_DH_PCR_Unmarshal(&handles[0], handleBufferStart,
390                                   bufferRemainingSize, TRUE);
391     if (TPM_RC_SUCCESS != result)
392     {
393         return result + TPM_RC_H + TPM_RC_1;
394     }

```

```

395     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
396                                     bufferRemainingSize, FALSE);
397     if (TPM_RC_SUCCESS != result)
398     {
399         return result + TPM_RC_H + TPM_RC_2;
400     }
401     break;
402 #endif // CC_EventSequenceComplete
403 #if CC_Certify
404 case TPM_CC_Certify:
405     *handleCount = 2;
406     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
407                                     bufferRemainingSize, FALSE);
408     if (TPM_RC_SUCCESS != result)
409     {
410         return result + TPM_RC_H + TPM_RC_1;
411     }
412     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
413                                     bufferRemainingSize, TRUE);
414     if (TPM_RC_SUCCESS != result)
415     {
416         return result + TPM_RC_H + TPM_RC_2;
417     }
418     break;
419 #endif // CC_Certify
420 #if CC_CertifyCreation
421 case TPM_CC_CertifyCreation:
422     *handleCount = 2;
423     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
424                                     bufferRemainingSize, TRUE);
425     if (TPM_RC_SUCCESS != result)
426     {
427         return result + TPM_RC_H + TPM_RC_1;
428     }
429     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
430                                     bufferRemainingSize, FALSE);
431     if (TPM_RC_SUCCESS != result)
432     {
433         return result + TPM_RC_H + TPM_RC_2;
434     }
435     break;
436 #endif // CC_CertifyCreation
437 #if CC_Quote
438 case TPM_CC_Quote:
439     *handleCount = 1;
440     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
441                                     bufferRemainingSize, TRUE);
442     if (TPM_RC_SUCCESS != result)
443     {
444         return result + TPM_RC_H + TPM_RC_1;
445     }
446     break;
447 #endif // CC_Quote
448 #if CC_GetSessionAuditDigest
449 case TPM_CC_GetSessionAuditDigest:
450     *handleCount = 3;
451     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
452                                     bufferRemainingSize, FALSE);
453     if (TPM_RC_SUCCESS != result)
454     {
455         return result + TPM_RC_H + TPM_RC_1;
456     }
457     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
458                                     bufferRemainingSize, TRUE);
459     if (TPM_RC_SUCCESS != result)
460     {

```

```

461         return result + TPM_RC_H + TPM_RC_2;
462     }
463     result = TPMI_SH_HMAC_Unmarshal(&handles[2], handleBufferStart,
464                                     bufferRemainingSize);
465     if (TPM_RC_SUCCESS != result)
466     {
467         return result + TPM_RC_H + TPM_RC_3;
468     }
469     break;
470 #endif // CC_GetSessionAuditDigest
471 #if CC_GetCommandAuditDigest
472 case TPM_CC_GetCommandAuditDigest:
473     *handleCount = 2;
474     result = TPMI_RH_ENDORSEMENT_Unmarshal(&handles[0], handleBufferStart,
475                                             bufferRemainingSize, FALSE);
476     if (TPM_RC_SUCCESS != result)
477     {
478         return result + TPM_RC_H + TPM_RC_1;
479     }
480     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
481                                     bufferRemainingSize, TRUE);
482     if (TPM_RC_SUCCESS != result)
483     {
484         return result + TPM_RC_H + TPM_RC_2;
485     }
486     break;
487 #endif // CC_GetCommandAuditDigest
488 #if CC_GetTime
489 case TPM_CC_GetTime:
490     *handleCount = 2;
491     result = TPMI_RH_ENDORSEMENT_Unmarshal(&handles[0], handleBufferStart,
492                                             bufferRemainingSize, FALSE);
493     if (TPM_RC_SUCCESS != result)
494     {
495         return result + TPM_RC_H + TPM_RC_1;
496     }
497     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
498                                     bufferRemainingSize, TRUE);
499     if (TPM_RC_SUCCESS != result)
500     {
501         return result + TPM_RC_H + TPM_RC_2;
502     }
503     break;
504 #endif // CC_GetTime
505 #if CC_CertifyX509
506 case TPM_CC_CertifyX509:
507     *handleCount = 2;
508     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
509                                     bufferRemainingSize, FALSE);
510     if (TPM_RC_SUCCESS != result)
511     {
512         return result + TPM_RC_H + TPM_RC_1;
513     }
514     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
515                                     bufferRemainingSize, TRUE);
516     if (TPM_RC_SUCCESS != result)
517     {
518         return result + TPM_RC_H + TPM_RC_2;
519     }
520     break;
521 #endif // CC_CertifyX509
522 #if CC_Commit
523 case TPM_CC_Commit:
524     *handleCount = 1;
525     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
526                                     bufferRemainingSize, FALSE);
527 
```



```

527     if (TPM_RC_SUCCESS != result)
528     {
529         return result + TPM_RC_H + TPM_RC_1;
530     }
531     break;
532 #endif // CC_Commit
533 #if CC_EC_Ephemeral
534 case TPM_CC_EC_Ephemeral:
535     break;
536 #endif // CC_EC_Ephemeral
537 #if CC_VerifySignature
538 case TPM_CC_VerifySignature:
539     *handleCount = 1;
540     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
541                                     bufferRemainingSize, FALSE);
542     if (TPM_RC_SUCCESS != result)
543     {
544         return result + TPM_RC_H + TPM_RC_1;
545     }
546     break;
547 #endif // CC_VerifySignature
548 #if CC_Sign
549 case TPM_CC_Sign:
550     *handleCount = 1;
551     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
552                                     bufferRemainingSize, FALSE);
553     if (TPM_RC_SUCCESS != result)
554     {
555         return result + TPM_RC_H + TPM_RC_1;
556     }
557     break;
558 #endif // CC_Sign
559 #if CC_SetCommandCodeAuditStatus
560 case TPM_CC_SetCommandCodeAuditStatus:
561     *handleCount = 1;
562     result = TPMI_DH_PROVISION_Unmarshal(&handles[0], handleBufferStart,
563                                     bufferRemainingSize);
564     if (TPM_RC_SUCCESS != result)
565     {
566         return result + TPM_RC_H + TPM_RC_1;
567     }
568     break;
569 #endif // CC_SetCommandCodeAuditStatus
570 #if CC_PCR_Extend
571 case TPM_CC_PCR_Extend:
572     *handleCount = 1;
573     result = TPMI_DH_PCR_Unmarshal(&handles[0], handleBufferStart,
574                                     bufferRemainingSize, TRUE);
575     if (TPM_RC_SUCCESS != result)
576     {
577         return result + TPM_RC_H + TPM_RC_1;
578     }
579     break;
580 #endif // CC_PCR_Extend
581 #if CC_PCR_Event
582 case TPM_CC_PCR_Event:
583     *handleCount = 1;
584     result = TPMI_DH_PCR_Unmarshal(&handles[0], handleBufferStart,
585                                     bufferRemainingSize, TRUE);
586     if (TPM_RC_SUCCESS != result)
587     {
588         return result + TPM_RC_H + TPM_RC_1;
589     }
590     break;
591 #endif // CC_PCR_Event
592 #if CC_PCR_Read

```

```

593 case TPM_CC_PCR_Read:
594     break;
595 #endif // CC_PCR_Read
596 #if CC_PCR_Allocate
597 case TPM_CC_PCR_Allocate:
598     *handleCount = 1;
599     result = TPMI_RH_PLATFORM_Unmarshal(&handles[0], handleBufferStart,
600                                         bufferRemainingSize);
601     if (TPM_RC_SUCCESS != result)
602     {
603         return result + TPM_RC_H + TPM_RC_1;
604     }
605     break;
606 #endif // CC_PCR_Allocate
607 #if CC_PCR_SetAuthPolicy
608 case TPM_CC_PCR_SetAuthPolicy:
609     *handleCount = 1;
610     result = TPMI_RH_PLATFORM_Unmarshal(&handles[0], handleBufferStart,
611                                         bufferRemainingSize);
612     if (TPM_RC_SUCCESS != result)
613     {
614         return result + TPM_RC_H + TPM_RC_1;
615     }
616     break;
617 #endif // CC_PCR_SetAuthPolicy
618 #if CC_PCR_SetAuthValue
619 case TPM_CC_PCR_SetAuthValue:
620     *handleCount = 1;
621     result = TPMI_DH_PCR_Unmarshal(&handles[0], handleBufferStart,
622                                   bufferRemainingSize, FALSE);
623     if (TPM_RC_SUCCESS != result)
624     {
625         return result + TPM_RC_H + TPM_RC_1;
626     }
627     break;
628 #endif // CC_PCR_SetAuthValue
629 #if CC_PCR_Reset
630 case TPM_CC_PCR_Reset:
631     *handleCount = 1;
632     result = TPMI_DH_PCR_Unmarshal(&handles[0], handleBufferStart,
633                                   bufferRemainingSize, FALSE);
634     if (TPM_RC_SUCCESS != result)
635     {
636         return result + TPM_RC_H + TPM_RC_1;
637     }
638     break;
639 #endif // CC_PCR_Reset
640 #if CC_PolicySigned
641 case TPM_CC_PolicySigned:
642     *handleCount = 2;
643     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
644                                       bufferRemainingSize, FALSE);
645     if (TPM_RC_SUCCESS != result)
646     {
647         return result + TPM_RC_H + TPM_RC_1;
648     }
649     result = TPMI_SH_POLICY_Unmarshal(&handles[1], handleBufferStart,
650                                       bufferRemainingSize);
651     if (TPM_RC_SUCCESS != result)
652     {
653         return result + TPM_RC_H + TPM_RC_2;
654     }
655     break;
656 #endif // CC_PolicySigned
657 #if CC_PolicySecret
658 case TPM_CC_PolicySecret:

```

```

659     *handleCount = 2;
660     result = TPMI_DH_ENTITY_Unmarshal(&handles[0], handleBufferStart,
661                                     bufferRemainingSize, FALSE);
662     if (TPM_RC_SUCCESS != result)
663     {
664         return result + TPM_RC_H + TPM_RC_1;
665     }
666     result = TPMI_SH_POLICY_Unmarshal(&handles[1], handleBufferStart,
667                                     bufferRemainingSize);
668     if (TPM_RC_SUCCESS != result)
669     {
670         return result + TPM_RC_H + TPM_RC_2;
671     }
672     break;
673 #endif // CC_PolicySecret
674 #if CC_PolicyTicket
675 case TPM_CC_PolicyTicket:
676     *handleCount = 1;
677     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
678                                     bufferRemainingSize);
679     if (TPM_RC_SUCCESS != result)
680     {
681         return result + TPM_RC_H + TPM_RC_1;
682     }
683     break;
684 #endif // CC_PolicyTicket
685 #if CC_PolicyOR
686 case TPM_CC_PolicyOR:
687     *handleCount = 1;
688     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
689                                     bufferRemainingSize);
690     if (TPM_RC_SUCCESS != result)
691     {
692         return result + TPM_RC_H + TPM_RC_1;
693     }
694     break;
695 #endif // CC_PolicyOR
696 #if CC_PolicyPCR
697 case TPM_CC_PolicyPCR:
698     *handleCount = 1;
699     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
700                                     bufferRemainingSize);
701     if (TPM_RC_SUCCESS != result)
702     {
703         return result + TPM_RC_H + TPM_RC_1;
704     }
705     break;
706 #endif // CC_PolicyPCR
707 #if CC_PolicyLocality
708 case TPM_CC_PolicyLocality:
709     *handleCount = 1;
710     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
711                                     bufferRemainingSize);
712     if (TPM_RC_SUCCESS != result)
713     {
714         return result + TPM_RC_H + TPM_RC_1;
715     }
716     break;
717 #endif // CC_PolicyLocality
718 #if CC_PolicyNV
719 case TPM_CC_PolicyNV:
720     *handleCount = 3;
721     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[0], handleBufferStart,
722                                     bufferRemainingSize);
723     if (TPM_RC_SUCCESS != result)
724     {

```

```

725         return result + TPM_RC_H + TPM_RC_1;
726     }
727     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[1], handleBufferStart,
728                                         bufferRemainingSize);
729     if (TPM_RC_SUCCESS != result)
730     {
731         return result + TPM_RC_H + TPM_RC_2;
732     }
733     result = TPMI_SH_POLICY_Unmarshal(&handles[2], handleBufferStart,
734                                       bufferRemainingSize);
735     if (TPM_RC_SUCCESS != result)
736     {
737         return result + TPM_RC_H + TPM_RC_3;
738     }
739     break;
740 #endif // CC_PolicyNV
741 #if CC_PolicyCounterTimer
742 case TPM_CC_PolicyCounterTimer:
743     *handleCount = 1;
744     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
745                                       bufferRemainingSize);
746     if (TPM_RC_SUCCESS != result)
747     {
748         return result + TPM_RC_H + TPM_RC_1;
749     }
750     break;
751 #endif // CC_PolicyCounterTimer
752 #if CC_PolicyCommandCode
753 case TPM_CC_PolicyCommandCode:
754     *handleCount = 1;
755     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
756                                       bufferRemainingSize);
757     if (TPM_RC_SUCCESS != result)
758     {
759         return result + TPM_RC_H + TPM_RC_1;
760     }
761     break;
762 #endif // CC_PolicyCommandCode
763 #if CC_PolicyPhysicalPresence
764 case TPM_CC_PolicyPhysicalPresence:
765     *handleCount = 1;
766     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
767                                       bufferRemainingSize);
768     if (TPM_RC_SUCCESS != result)
769     {
770         return result + TPM_RC_H + TPM_RC_1;
771     }
772     break;
773 #endif // CC_PolicyPhysicalPresence
774 #if CC_PolicyCpHash
775 case TPM_CC_PolicyCpHash:
776     *handleCount = 1;
777     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
778                                       bufferRemainingSize);
779     if (TPM_RC_SUCCESS != result)
780     {
781         return result + TPM_RC_H + TPM_RC_1;
782     }
783     break;
784 #endif // CC_PolicyCpHash
785 #if CC_PolicyNameHash
786 case TPM_CC_PolicyNameHash:
787     *handleCount = 1;
788     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
789                                       bufferRemainingSize);
790     if (TPM_RC_SUCCESS != result)

```

```

791     {
792         return result + TPM_RC_H + TPM_RC_1;
793     }
794     break;
795 #endif // CC_PolicyNameHash
796 #if CC_PolicyDuplicationSelect
797 case TPM_CC_PolicyDuplicationSelect:
798     *handleCount = 1;
799     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
800                                     bufferRemainingSize);
801     if (TPM_RC_SUCCESS != result)
802     {
803         return result + TPM_RC_H + TPM_RC_1;
804     }
805     break;
806 #endif // CC_PolicyDuplicationSelect
807 #if CC_PolicyAuthorize
808 case TPM_CC_PolicyAuthorize:
809     *handleCount = 1;
810     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
811                                     bufferRemainingSize);
812     if (TPM_RC_SUCCESS != result)
813     {
814         return result + TPM_RC_H + TPM_RC_1;
815     }
816     break;
817 #endif // CC_PolicyAuthorize
818 #if CC_PolicyAuthValue
819 case TPM_CC_PolicyAuthValue:
820     *handleCount = 1;
821     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
822                                     bufferRemainingSize);
823     if (TPM_RC_SUCCESS != result)
824     {
825         return result + TPM_RC_H + TPM_RC_1;
826     }
827     break;
828 #endif // CC_PolicyAuthValue
829 #if CC_PolicyPassword
830 case TPM_CC_PolicyPassword:
831     *handleCount = 1;
832     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
833                                     bufferRemainingSize);
834     if (TPM_RC_SUCCESS != result)
835     {
836         return result + TPM_RC_H + TPM_RC_1;
837     }
838     break;
839 #endif // CC_PolicyPassword
840 #if CC_PolicyGetDigest
841 case TPM_CC_PolicyGetDigest:
842     *handleCount = 1;
843     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
844                                     bufferRemainingSize);
845     if (TPM_RC_SUCCESS != result)
846     {
847         return result + TPM_RC_H + TPM_RC_1;
848     }
849     break;
850 #endif // CC_PolicyGetDigest
851 #if CC_PolicyNvWritten
852 case TPM_CC_PolicyNvWritten:
853     *handleCount = 1;
854     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
855                                     bufferRemainingSize);
856     if (TPM_RC_SUCCESS != result)

```

```

857     {
858         return result + TPM_RC_H + TPM_RC_1;
859     }
860     break;
861 #endif // CC_PolicyNvWritten
862 #if CC_PolicyTemplate
863 case TPM_CC_PolicyTemplate:
864     *handleCount = 1;
865     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
866                                     bufferRemainingSize);
867     if (TPM_RC_SUCCESS != result)
868     {
869         return result + TPM_RC_H + TPM_RC_1;
870     }
871     break;
872 #endif // CC_PolicyTemplate
873 #if CC_PolicyAuthorizeNV
874 case TPM_CC_PolicyAuthorizeNV:
875     *handleCount = 3;
876     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[0], handleBufferStart,
877                                     bufferRemainingSize);
878     if (TPM_RC_SUCCESS != result)
879     {
880         return result + TPM_RC_H + TPM_RC_1;
881     }
882     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[1], handleBufferStart,
883                                     bufferRemainingSize);
884     if (TPM_RC_SUCCESS != result)
885     {
886         return result + TPM_RC_H + TPM_RC_2;
887     }
888     result = TPMI_SH_POLICY_Unmarshal(&handles[2], handleBufferStart,
889                                     bufferRemainingSize);
890     if (TPM_RC_SUCCESS != result)
891     {
892         return result + TPM_RC_H + TPM_RC_3;
893     }
894     break;
895 #endif // CC_PolicyAuthorizeNV
896 #if CC_PolicyCapability
897 case TPM_CC_PolicyCapability:
898     *handleCount = 1;
899     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
900                                     bufferRemainingSize);
901     if (TPM_RC_SUCCESS != result)
902     {
903         return result + TPM_RC_H + TPM_RC_1;
904     }
905     break;
906 #endif // CC_PolicyCapability
907 #if CC_PolicyParameters
908 case TPM_CC_PolicyParameters:
909     *handleCount = 1;
910     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
911                                     bufferRemainingSize);
912     if (TPM_RC_SUCCESS != result)
913     {
914         return result + TPM_RC_H + TPM_RC_1;
915     }
916     break;
917 #endif // CC_PolicyParameters
918 #if CC_CreatePrimary
919 case TPM_CC_CreatePrimary:
920     *handleCount = 1;
921     result = TPMI_RH_HIERARCHY_Unmarshal(&handles[0], handleBufferStart,
922                                     bufferRemainingSize);

```



```

923     if (TPM_RC_SUCCESS != result)
924     {
925         return result + TPM_RC_H + TPM_RC_1;
926     }
927     break;
928 #endif // CC_CreatePrimary
929 #if CC_HierarchyControl
930 case TPM_CC_HierarchyControl:
931     *handleCount = 1;
932     result = TPMI_RH_BASE_HIERARCHY_Unmarshal(&handles[0], handleBufferStart,
933                                              bufferRemainingSize);
934     if (TPM_RC_SUCCESS != result)
935     {
936         return result + TPM_RC_H + TPM_RC_1;
937     }
938     break;
939 #endif // CC_HierarchyControl
940 #if CC_SetPrimaryPolicy
941 case TPM_CC_SetPrimaryPolicy:
942     *handleCount = 1;
943     result = TPMI_RH_HIERARCHY_POLICY_Unmarshal(&handles[0], handleBufferStart,
944                                              bufferRemainingSize);
945     if (TPM_RC_SUCCESS != result)
946     {
947         return result + TPM_RC_H + TPM_RC_1;
948     }
949     break;
950 #endif // CC_SetPrimaryPolicy
951 #if CC_ChangePPS
952 case TPM_CC_ChangePPS:
953     *handleCount = 1;
954     result = TPMI_RH_PLATFORM_Unmarshal(&handles[0], handleBufferStart,
955                                         bufferRemainingSize);
956     if (TPM_RC_SUCCESS != result)
957     {
958         return result + TPM_RC_H + TPM_RC_1;
959     }
960     break;
961 #endif // CC_ChangePPS
962 #if CC_ChangeEPS
963 case TPM_CC_ChangeEPS:
964     *handleCount = 1;
965     result = TPMI_RH_PLATFORM_Unmarshal(&handles[0], handleBufferStart,
966                                         bufferRemainingSize);
967     if (TPM_RC_SUCCESS != result)
968     {
969         return result + TPM_RC_H + TPM_RC_1;
970     }
971     break;
972 #endif // CC_ChangeEPS
973 #if CC_Clear
974 case TPM_CC_Clear:
975     *handleCount = 1;
976     result = TPMI_RH_CLEAR_Unmarshal(&handles[0], handleBufferStart,
977                                     bufferRemainingSize);
978     if (TPM_RC_SUCCESS != result)
979     {
980         return result + TPM_RC_H + TPM_RC_1;
981     }
982     break;
983 #endif // CC_Clear
984 #if CC_ClearControl
985 case TPM_CC_ClearControl:
986     *handleCount = 1;
987     result = TPMI_RH_CLEAR_Unmarshal(&handles[0], handleBufferStart,
988                                     bufferRemainingSize);

```

```

989     if (TPM_RC_SUCCESS != result)
990     {
991         return result + TPM_RC_H + TPM_RC_1;
992     }
993     break;
994 #endif // CC_ClearControl
995 #if CC_HierarchyChangeAuth
996 case TPM_CC_HierarchyChangeAuth:
997     *handleCount = 1;
998     result = TPMI_RH_HIERARCHY_AUTH_Unmarshal(&handles[0], handleBufferStart,
999                                             bufferRemainingSize);
1000     if (TPM_RC_SUCCESS != result)
1001     {
1002         return result + TPM_RC_H + TPM_RC_1;
1003     }
1004     break;
1005 #endif // CC_HierarchyChangeAuth
1006 #if CC_DictionaryAttackLockReset
1007 case TPM_CC_DictionaryAttackLockReset:
1008     *handleCount = 1;
1009     result = TPMI_RH_LOCKOUT_Unmarshal(&handles[0], handleBufferStart,
1010                                       bufferRemainingSize);
1011     if (TPM_RC_SUCCESS != result)
1012     {
1013         return result + TPM_RC_H + TPM_RC_1;
1014     }
1015     break;
1016 #endif // CC_DictionaryAttackLockReset
1017 #if CC_DictionaryAttackParameters
1018 case TPM_CC_DictionaryAttackParameters:
1019     *handleCount = 1;
1020     result = TPMI_RH_LOCKOUT_Unmarshal(&handles[0], handleBufferStart,
1021                                       bufferRemainingSize);
1022     if (TPM_RC_SUCCESS != result)
1023     {
1024         return result + TPM_RC_H + TPM_RC_1;
1025     }
1026     break;
1027 #endif // CC_DictionaryAttackParameters
1028 #if CC_PP_Commands
1029 case TPM_CC_PP_Commands:
1030     *handleCount = 1;
1031     result = TPMI_RH_PLATFORM_Unmarshal(&handles[0], handleBufferStart,
1032                                       bufferRemainingSize);
1033     if (TPM_RC_SUCCESS != result)
1034     {
1035         return result + TPM_RC_H + TPM_RC_1;
1036     }
1037     break;
1038 #endif // CC_PP_Commands
1039 #if CC_SetAlgorithmSet
1040 case TPM_CC_SetAlgorithmSet:
1041     *handleCount = 1;
1042     result = TPMI_RH_PLATFORM_Unmarshal(&handles[0], handleBufferStart,
1043                                       bufferRemainingSize);
1044     if (TPM_RC_SUCCESS != result)
1045     {
1046         return result + TPM_RC_H + TPM_RC_1;
1047     }
1048     break;
1049 #endif // CC_SetAlgorithmSet
1050 #if CC_FieldUpgradeStart
1051 case TPM_CC_FieldUpgradeStart:
1052     *handleCount = 2;
1053     result = TPMI_RH_PLATFORM_Unmarshal(&handles[0], handleBufferStart,
1054                                       bufferRemainingSize);

```

```

1055     if (TPM_RC_SUCCESS != result)
1056     {
1057         return result + TPM_RC_H + TPM_RC_1;
1058     }
1059     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
1060                                     bufferRemainingSize, FALSE);
1061     if (TPM_RC_SUCCESS != result)
1062     {
1063         return result + TPM_RC_H + TPM_RC_2;
1064     }
1065     break;
1066 #endif // CC_FieldUpgradeStart
1067 #if CC_FieldUpgradeData
1068 case TPM_CC_FieldUpgradeData:
1069     break;
1070 #endif // CC_FieldUpgradeData
1071 #if CC_FirmwareRead
1072 case TPM_CC_FirmwareRead:
1073     break;
1074 #endif // CC_FirmwareRead
1075 #if CC_ContextSave
1076 case TPM_CC_ContextSave:
1077     *handleCount = 1;
1078     result = TPMI_DH_CONTEXT_Unmarshal(&handles[0], handleBufferStart,
1079                                     bufferRemainingSize);
1080     if (TPM_RC_SUCCESS != result)
1081     {
1082         return result + TPM_RC_H + TPM_RC_1;
1083     }
1084     break;
1085 #endif // CC_ContextSave
1086 #if CC_ContextLoad
1087 case TPM_CC_ContextLoad:
1088     break;
1089 #endif // CC_ContextLoad
1090 #if CC_FlushContext
1091 case TPM_CC_FlushContext:
1092     break;
1093 #endif // CC_FlushContext
1094 #if CC_EvictControl
1095 case TPM_CC_EvictControl:
1096     *handleCount = 2;
1097     result = TPMI_RH_PROVISION_Unmarshal(&handles[0], handleBufferStart,
1098                                     bufferRemainingSize);
1099     if (TPM_RC_SUCCESS != result)
1100     {
1101         return result + TPM_RC_H + TPM_RC_1;
1102     }
1103     result = TPMI_DH_OBJECT_Unmarshal(&handles[1], handleBufferStart,
1104                                     bufferRemainingSize, FALSE);
1105     if (TPM_RC_SUCCESS != result)
1106     {
1107         return result + TPM_RC_H + TPM_RC_2;
1108     }
1109     break;
1110 #endif // CC_EvictControl
1111 #if CC_ReadClock
1112 case TPM_CC_ReadClock:
1113     break;
1114 #endif // CC_ReadClock
1115 #if CC_ClockSet
1116 case TPM_CC_ClockSet:
1117     *handleCount = 1;
1118     result = TPMI_RH_PROVISION_Unmarshal(&handles[0], handleBufferStart,
1119                                     bufferRemainingSize);
1120     if (TPM_RC_SUCCESS != result)

```

```

1121     {
1122         return result + TPM_RC_H + TPM_RC_1;
1123     }
1124     break;
1125 #endif // CC_ClockSet
1126 #if CC_ClockRateAdjust
1127 case TPM_CC_ClockRateAdjust:
1128     *handleCount = 1;
1129     result = TPMI_RH_PROVISION_Unmarshal(&handles[0], handleBufferStart,
1130                                         bufferRemainingSize);
1131     if (TPM_RC_SUCCESS != result)
1132     {
1133         return result + TPM_RC_H + TPM_RC_1;
1134     }
1135     break;
1136 #endif // CC_ClockRateAdjust
1137 #if CC_GetCapability
1138 case TPM_CC_GetCapability:
1139     break;
1140 #endif // CC_GetCapability
1141 #if CC_TestParms
1142 case TPM_CC_TestParms:
1143     break;
1144 #endif // CC_TestParms
1145 #if CC_NV_DefineSpace
1146 case TPM_CC_NV_DefineSpace:
1147     *handleCount = 1;
1148     result = TPMI_RH_PROVISION_Unmarshal(&handles[0], handleBufferStart,
1149                                         bufferRemainingSize);
1150     if (TPM_RC_SUCCESS != result)
1151     {
1152         return result + TPM_RC_H + TPM_RC_1;
1153     }
1154     break;
1155 #endif // CC_NV_DefineSpace
1156 #if CC_NV_UndefineSpace
1157 case TPM_CC_NV_UndefineSpace:
1158     *handleCount = 2;
1159     result = TPMI_RH_PROVISION_Unmarshal(&handles[0], handleBufferStart,
1160                                         bufferRemainingSize);
1161     if (TPM_RC_SUCCESS != result)
1162     {
1163         return result + TPM_RC_H + TPM_RC_1;
1164     }
1165     result = TPMI_RH_NV_DEFINED_INDEX_Unmarshal(&handles[1], handleBufferStart,
1166                                                bufferRemainingSize);
1167     if (TPM_RC_SUCCESS != result)
1168     {
1169         return result + TPM_RC_H + TPM_RC_2;
1170     }
1171     break;
1172 #endif // CC_NV_UndefineSpace
1173 #if CC_NV_UndefineSpaceSpecial
1174 case TPM_CC_NV_UndefineSpaceSpecial:
1175     *handleCount = 2;
1176     result = TPMI_RH_NV_DEFINED_INDEX_Unmarshal(&handles[0], handleBufferStart,
1177                                                bufferRemainingSize);
1178     if (TPM_RC_SUCCESS != result)
1179     {
1180         return result + TPM_RC_H + TPM_RC_1;
1181     }
1182     result = TPMI_RH_PLATFORM_Unmarshal(&handles[1], handleBufferStart,
1183                                         bufferRemainingSize);
1184     if (TPM_RC_SUCCESS != result)
1185     {
1186         return result + TPM_RC_H + TPM_RC_2;

```

```

1187     }
1188     break;
1189 #endif // CC_NV_UndefineSpaceSpecial
1190 #if CC_NV_ReadPublic
1191 case TPM_CC_NV_ReadPublic:
1192     *handleCount = 1;
1193     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[0], handleBufferStart,
1194                                         bufferRemainingSize);
1195     if (TPM_RC_SUCCESS != result)
1196     {
1197         return result + TPM_RC_H + TPM_RC_1;
1198     }
1199     break;
1200 #endif // CC_NV_ReadPublic
1201 #if CC_NV_Write
1202 case TPM_CC_NV_Write:
1203     *handleCount = 2;
1204     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[0], handleBufferStart,
1205                                         bufferRemainingSize);
1206     if (TPM_RC_SUCCESS != result)
1207     {
1208         return result + TPM_RC_H + TPM_RC_1;
1209     }
1210     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[1], handleBufferStart,
1211                                         bufferRemainingSize);
1212     if (TPM_RC_SUCCESS != result)
1213     {
1214         return result + TPM_RC_H + TPM_RC_2;
1215     }
1216     break;
1217 #endif // CC_NV_Write
1218 #if CC_NV_Increment
1219 case TPM_CC_NV_Increment:
1220     *handleCount = 2;
1221     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[0], handleBufferStart,
1222                                         bufferRemainingSize);
1223     if (TPM_RC_SUCCESS != result)
1224     {
1225         return result + TPM_RC_H + TPM_RC_1;
1226     }
1227     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[1], handleBufferStart,
1228                                         bufferRemainingSize);
1229     if (TPM_RC_SUCCESS != result)
1230     {
1231         return result + TPM_RC_H + TPM_RC_2;
1232     }
1233     break;
1234 #endif // CC_NV_Increment
1235 #if CC_NV_Extend
1236 case TPM_CC_NV_Extend:
1237     *handleCount = 2;
1238     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[0], handleBufferStart,
1239                                         bufferRemainingSize);
1240     if (TPM_RC_SUCCESS != result)
1241     {
1242         return result + TPM_RC_H + TPM_RC_1;
1243     }
1244     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[1], handleBufferStart,
1245                                         bufferRemainingSize);
1246     if (TPM_RC_SUCCESS != result)
1247     {
1248         return result + TPM_RC_H + TPM_RC_2;
1249     }
1250     break;
1251 #endif // CC_NV_Extend
1252 #if CC_NV_SetBits

```

```

1253 case TPM_CC_NV_SetBits:
1254     *handleCount = 2;
1255     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[0], handleBufferStart,
1256                                         bufferRemainingSize);
1257     if (TPM_RC_SUCCESS != result)
1258     {
1259         return result + TPM_RC_H + TPM_RC_1;
1260     }
1261     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[1], handleBufferStart,
1262                                         bufferRemainingSize);
1263     if (TPM_RC_SUCCESS != result)
1264     {
1265         return result + TPM_RC_H + TPM_RC_2;
1266     }
1267     break;
1268 #endif // CC_NV_SetBits
1269 #if CC_NV_WriteLock
1270 case TPM_CC_NV_WriteLock:
1271     *handleCount = 2;
1272     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[0], handleBufferStart,
1273                                         bufferRemainingSize);
1274     if (TPM_RC_SUCCESS != result)
1275     {
1276         return result + TPM_RC_H + TPM_RC_1;
1277     }
1278     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[1], handleBufferStart,
1279                                         bufferRemainingSize);
1280     if (TPM_RC_SUCCESS != result)
1281     {
1282         return result + TPM_RC_H + TPM_RC_2;
1283     }
1284     break;
1285 #endif // CC_NV_WriteLock
1286 #if CC_NV_GlobalWriteLock
1287 case TPM_CC_NV_GlobalWriteLock:
1288     *handleCount = 1;
1289     result = TPMI_RH_PROVISION_Unmarshal(&handles[0], handleBufferStart,
1290                                         bufferRemainingSize);
1291     if (TPM_RC_SUCCESS != result)
1292     {
1293         return result + TPM_RC_H + TPM_RC_1;
1294     }
1295     break;
1296 #endif // CC_NV_GlobalWriteLock
1297 #if CC_NV_Read
1298 case TPM_CC_NV_Read:
1299     *handleCount = 2;
1300     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[0], handleBufferStart,
1301                                         bufferRemainingSize);
1302     if (TPM_RC_SUCCESS != result)
1303     {
1304         return result + TPM_RC_H + TPM_RC_1;
1305     }
1306     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[1], handleBufferStart,
1307                                         bufferRemainingSize);
1308     if (TPM_RC_SUCCESS != result)
1309     {
1310         return result + TPM_RC_H + TPM_RC_2;
1311     }
1312     break;
1313 #endif // CC_NV_Read
1314 #if CC_NV_ReadLock
1315 case TPM_CC_NV_ReadLock:
1316     *handleCount = 2;
1317     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[0], handleBufferStart,
1318                                         bufferRemainingSize);

```



```

1319     if (TPM_RC_SUCCESS != result)
1320     {
1321         return result + TPM_RC_H + TPM_RC_1;
1322     }
1323     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[1], handleBufferStart,
1324                                         bufferRemainingSize);
1325     if (TPM_RC_SUCCESS != result)
1326     {
1327         return result + TPM_RC_H + TPM_RC_2;
1328     }
1329     break;
1330 #endif // CC_NV_ReadLock
1331 #if CC_NV_ChangeAuth
1332 case TPM_CC_NV_ChangeAuth:
1333     *handleCount = 1;
1334     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[0], handleBufferStart,
1335                                         bufferRemainingSize);
1336     if (TPM_RC_SUCCESS != result)
1337     {
1338         return result + TPM_RC_H + TPM_RC_1;
1339     }
1340     break;
1341 #endif // CC_NV_ChangeAuth
1342 #if CC_NV_Certify
1343 case TPM_CC_NV_Certify:
1344     *handleCount = 3;
1345     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
1346                                       bufferRemainingSize, TRUE);
1347     if (TPM_RC_SUCCESS != result)
1348     {
1349         return result + TPM_RC_H + TPM_RC_1;
1350     }
1351     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[1], handleBufferStart,
1352                                       bufferRemainingSize);
1353     if (TPM_RC_SUCCESS != result)
1354     {
1355         return result + TPM_RC_H + TPM_RC_2;
1356     }
1357     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[2], handleBufferStart,
1358                                       bufferRemainingSize);
1359     if (TPM_RC_SUCCESS != result)
1360     {
1361         return result + TPM_RC_H + TPM_RC_3;
1362     }
1363     break;
1364 #endif // CC_NV_Certify
1365 #if CC_NV_DefineSpace2
1366 case TPM_CC_NV_DefineSpace2:
1367     *handleCount = 1;
1368     result = TPMI_RH_PROVISION_Unmarshal(&handles[0], handleBufferStart,
1369                                         bufferRemainingSize);
1370     if (TPM_RC_SUCCESS != result)
1371     {
1372         return result + TPM_RC_H + TPM_RC_1;
1373     }
1374     break;
1375 #endif // CC_NV_DefineSpace2
1376 #if CC_NV_ReadPublic2
1377 case TPM_CC_NV_ReadPublic2:
1378     *handleCount = 1;
1379     result = TPMI_RH_NV_INDEX_Unmarshal(&handles[0], handleBufferStart,
1380                                         bufferRemainingSize);
1381     if (TPM_RC_SUCCESS != result)
1382     {
1383         return result + TPM_RC_H + TPM_RC_1;
1384     }

```

```

1385     break;
1386 #endif // CC_NV_ReadPublic2
1387 #if CC_SetCapability
1388 case TPM_CC_SetCapability:
1389     *handleCount = 1;
1390     result = TPMI_RH_HIERARCHY_UNMARSHAL(&handles[0], handleBufferStart,
1391                                         bufferRemainingSize, TRUE);
1392     if (TPM_RC_SUCCESS != result)
1393     {
1394         return result + TPM_RC_H + TPM_RC_1;
1395     }
1396     break;
1397 #endif // CC_SetCapability
1398 #if CC_AC_GetCapability
1399 case TPM_CC_AC_GetCapability:
1400     *handleCount = 1;
1401     result = TPMI_RH_AC_Unmarshal(&handles[0], handleBufferStart,
1402                                   bufferRemainingSize);
1403     if (TPM_RC_SUCCESS != result)
1404     {
1405         return result + TPM_RC_H + TPM_RC_1;
1406     }
1407     break;
1408 #endif // CC_AC_GetCapability
1409 #if CC_AC_Send
1410 case TPM_CC_AC_Send:
1411     *handleCount = 3;
1412     result = TPMI_DH_OBJECT_Unmarshal(&handles[0], handleBufferStart,
1413                                       bufferRemainingSize, FALSE);
1414     if (TPM_RC_SUCCESS != result)
1415     {
1416         return result + TPM_RC_H + TPM_RC_1;
1417     }
1418     result = TPMI_RH_NV_AUTH_Unmarshal(&handles[1], handleBufferStart,
1419                                       bufferRemainingSize);
1420     if (TPM_RC_SUCCESS != result)
1421     {
1422         return result + TPM_RC_H + TPM_RC_2;
1423     }
1424     result = TPMI_RH_AC_Unmarshal(&handles[2], handleBufferStart,
1425                                   bufferRemainingSize);
1426     if (TPM_RC_SUCCESS != result)
1427     {
1428         return result + TPM_RC_H + TPM_RC_3;
1429     }
1430     break;
1431 #endif // CC_AC_Send
1432 #if CC_Policy_AC_SendSelect
1433 case TPM_CC_Policy_AC_SendSelect:
1434     *handleCount = 1;
1435     result = TPMI_SH_POLICY_Unmarshal(&handles[0], handleBufferStart,
1436                                       bufferRemainingSize);
1437     if (TPM_RC_SUCCESS != result)
1438     {
1439         return result + TPM_RC_H + TPM_RC_1;
1440     }
1441     break;
1442 #endif // CC_Policy_AC_SendSelect
1443 #if CC_ACT_SetTimeout
1444 case TPM_CC_ACT_SetTimeout:
1445     *handleCount = 1;
1446     result = TPMI_RH_ACT_Unmarshal(&handles[0], handleBufferStart,
1447                                    bufferRemainingSize);
1448     if (TPM_RC_SUCCESS != result)
1449     {
1450         return result + TPM_RC_H + TPM_RC_1;

```

```

1451     }
1452     break;
1453 #endif // CC_ACT_SetTimeout
1454 #if CC_Vendor_TCG_Test
1455 case TPM_CC_Vendor_TCG_Test:
1456     break;
1457 #endif // CC_Vendor_TCG_Test

```

6.25 /tpm/include/private/HashTestData.h

```

1 //
2 // Hash Test Vectors
3 //
4
5 TPM2B_TYPE(HASH_TEST_KEY, 128); // Twice the largest digest size
6 TPM2B_HASH_TEST_KEY c_hashTestKey = {
7     {128,
8         {0xa0, 0xed, 0x5c, 0x9a, 0xd2, 0x4a, 0x21, 0x40, 0x1a, 0xd0, 0x81, 0x47, 0x39,
9          0x63, 0xf9, 0x50, 0xdc, 0x59, 0x47, 0x11, 0x40, 0x13, 0x99, 0x92, 0xc0, 0x72,
10         0xa4, 0x0f, 0xe2, 0x33, 0xe4, 0x63, 0x9b, 0xb6, 0x76, 0xc3, 0x1e, 0x6f, 0x13,
11         0xee, 0xcc, 0x99, 0x71, 0xa5, 0xc0, 0xcf, 0x9a, 0x40, 0xcf, 0xdb, 0x66, 0x70,
12         0x05, 0x63, 0x54, 0x12, 0x25, 0xf4, 0xe0, 0x1b, 0x23, 0x35, 0xe3, 0x70, 0x7d,
13         0x19, 0x5f, 0x00, 0xe4, 0xf1, 0x61, 0x73, 0x05, 0xd8, 0x58, 0x7f, 0x60, 0x61,
14         0x84, 0x36, 0xec, 0xbe, 0x96, 0x1b, 0x69, 0x00, 0xf0, 0x9a, 0x6e, 0xe3, 0x26,
15         0x73, 0x0d, 0x17, 0x5b, 0x33, 0x41, 0x44, 0x9d, 0x90, 0xab, 0xd9, 0x6b, 0x7d,
16         0x48, 0x99, 0x25, 0x93, 0x29, 0x14, 0x2b, 0xce, 0x93, 0x8d, 0x8c, 0xaf, 0x31,
17         0x0e, 0x9c, 0x57, 0xd8, 0x5b, 0x57, 0x20, 0x1b, 0x9f, 0x2d, 0xa5}}};
18
19 TPM2B_TYPE(HASH_TEST_DATA, 256); // Twice the largest block size
20 TPM2B_HASH_TEST_DATA c_hashTestData = {
21     {256,
22         {0x88, 0xac, 0xc3, 0xe5, 0x5f, 0x66, 0x9d, 0x18, 0x80, 0xc9, 0x7a, 0x9c, 0xa4,
23          0x08, 0x90, 0x98, 0x0f, 0x3a, 0x53, 0x92, 0x4c, 0x67, 0x4e, 0xb7, 0x37, 0xec,
24          0x67, 0x87, 0xb6, 0xbe, 0x10, 0xca, 0x11, 0x5b, 0x4a, 0x0b, 0x45, 0xc3, 0x32,
25          0x68, 0x48, 0x69, 0xce, 0x25, 0x1b, 0xc8, 0xaf, 0x44, 0x79, 0x22, 0x83, 0xc8,
26          0xfb, 0xe2, 0x63, 0x94, 0xa2, 0x3c, 0x59, 0x3e, 0x3e, 0xc6, 0x64, 0x2c, 0x1f,
27          0x8c, 0x11, 0x93, 0x24, 0xa3, 0x17, 0xc5, 0x2f, 0x37, 0xcf, 0x95, 0x97, 0x8e,
28          0x63, 0x39, 0x68, 0xd5, 0xca, 0xba, 0x18, 0x37, 0x69, 0x6e, 0x4f, 0x19, 0xfd,
29          0x8a, 0xc0, 0x8d, 0x87, 0x3a, 0xbc, 0x31, 0x42, 0x04, 0x05, 0xef, 0xb5, 0x02,
30          0xef, 0x1e, 0x92, 0x4b, 0xb7, 0x73, 0x2c, 0x8c, 0xeb, 0x23, 0x13, 0x81, 0x34,
31          0xb9, 0xb5, 0xc1, 0x17, 0x37, 0x39, 0xf8, 0x3e, 0xe4, 0x4c, 0x06, 0xa8, 0x81,
32          0x52, 0x2f, 0xef, 0xc9, 0x9c, 0x69, 0x89, 0xbc, 0x85, 0x9c, 0x30, 0x16, 0x02,
33          0xca, 0xe3, 0x61, 0xd4, 0x0f, 0xed, 0x34, 0x1b, 0xca, 0xc1, 0x1b, 0xd1, 0xfa,
34          0xc1, 0xa2, 0xe0, 0xdf, 0x52, 0x2f, 0x0b, 0x4b, 0x9f, 0x0e, 0x45, 0x54, 0xb9,
35          0x17, 0xb6, 0xaf, 0xd6, 0xd5, 0xca, 0x90, 0x29, 0x57, 0x7b, 0x70, 0x50, 0x94,
36          0x5c, 0x8e, 0xf6, 0x4e, 0x21, 0x8b, 0xc6, 0x8b, 0xa6, 0xbc, 0xb9, 0x64, 0xd4,
37          0x4d, 0xf3, 0x68, 0xd8, 0xac, 0xde, 0xd8, 0xd8, 0xb5, 0x6d, 0xcd, 0x93, 0xeb,
38          0x28, 0xa4, 0xe2, 0x5c, 0x44, 0xef, 0xf0, 0xe1, 0x6f, 0x38, 0x1a, 0x3c, 0xe6,
39          0xef, 0xa2, 0x9d, 0xb9, 0xa8, 0x05, 0x2a, 0x95, 0xec, 0x5f, 0xdb, 0xb0, 0x25,
40          0x67, 0x9c, 0x86, 0x7a, 0x8e, 0xea, 0x51, 0xcc, 0xc3, 0xd3, 0xff, 0x6e, 0xf0,
41          0xed, 0xa3, 0xae, 0xf9, 0x5d, 0x33, 0x70, 0xf2, 0x11}}};
42
43 #if ALG_SHA1 == YES
44 TPM2B_TYPE(SHA1, 20);
45 TPM2B_SHA1 c_SHA1_digest = {
46     {20, {0xee, 0x2c, 0xef, 0x93, 0x76, 0xbd, 0xf8, 0x91, 0xbc, 0xe6,
47           0xe5, 0x57, 0x53, 0x77, 0x01, 0xb5, 0x70, 0x95, 0xe5, 0x40}}};
48 #endif
49
50 #if ALG_SHA256 == YES
51 TPM2B_TYPE(SHA256, 32);
52 TPM2B_SHA256 c_SHA256_digest = {
53     {32, {0x64, 0xe8, 0xe0, 0xc3, 0xa9, 0xa4, 0x51, 0x49, 0x10, 0x55, 0x8d,
54           0x31, 0x71, 0xe5, 0x2f, 0x69, 0x3a, 0xdc, 0xc7, 0x11, 0x32, 0x44,
55           0x61, 0xbd, 0x34, 0x39, 0x57, 0xb0, 0xa8, 0x75, 0x86, 0x1b}}};

```

```

56 #endif
57
58 #if ALG_SHA384 == YES
59 TPM2B_TYPE(SHA384, 48);
60 TPM2B_SHA384 c_SHA384_digest = {
61     {48, {0x37, 0x75, 0x29, 0xb5, 0x20, 0x15, 0x6e, 0xa3, 0x7e, 0xa3, 0x0d, 0xcd,
62           0x80, 0xa8, 0xa3, 0x3d, 0xeb, 0xe8, 0xad, 0x4e, 0x1c, 0x77, 0x94, 0x5a,
63           0xaf, 0x6c, 0xd0, 0xc1, 0xfa, 0x43, 0x3f, 0xc7, 0xb8, 0xf1, 0x01, 0xc0,
64           0x60, 0xbf, 0xf2, 0x87, 0xe8, 0x71, 0x9e, 0x51, 0x97, 0xa0, 0x09, 0x8d}}};
65 #endif
66
67 #if ALG_SHA512 == YES
68 TPM2B_TYPE(SHA512, 64);
69 TPM2B_SHA512 c_SHA512_digest = {
70     {64,
71      {0xe2, 0x7b, 0x10, 0x3d, 0x5e, 0x48, 0x58, 0x44, 0x67, 0xac, 0xa3, 0x81, 0x8c,
72       0x1d, 0xc5, 0x71, 0x66, 0x92, 0x8a, 0x89, 0xaa, 0xd4, 0x35, 0x51, 0x60, 0x37,
73       0x31, 0xd7, 0xba, 0xe7, 0x93, 0x0b, 0x16, 0x4d, 0xb3, 0xc8, 0x34, 0x98, 0x3c,
74       0xd3, 0x53, 0xde, 0x5e, 0xe8, 0x0c, 0xbc, 0xaf, 0xc9, 0x24, 0x2c, 0xcc, 0xed,
75       0xdb, 0xde, 0xba, 0x1f, 0x14, 0x14, 0x5a, 0x95, 0x80, 0xde, 0x66, 0xbd}}};
76 #endif
77
78 TPM2B_TYPE(EMPTY, 1);
79
80 #if ALG_SM3_256 == YES
81 TPM2B_EMPTY c_SM3_256_digest = {{0, {0}}};
82 #endif
83
84 #if ALG_SHA3_256 == YES
85 TPM2B_EMPTY c_SHA3_256_digest = {{0, {0}}};
86 #endif
87
88 #if ALG_SHA3_384 == YES
89 TPM2B_EMPTY c_SHA3_384_digest = {{0, {0}}};
90 #endif
91
92 #if ALG_SHA3_512 == YES
93 TPM2B_EMPTY c_SHA3_512_digest = {{0, {0}}};
94 #endif

```

6.26 /tpm/include/private/InternalRoutines.h

```

1 #ifndef INTERNAL_ROUTINES_H
2 #define INTERNAL_ROUTINES_H
3
4 #if !defined _LIB_SUPPORT_H_ && !defined _TPM_H_
5 # error "Should not be called"
6 #endif
7
8 // DRTM functions
9 // TODO_RENAME_INC_FOLDER:platform_interface refers to the TPM_CoreLib platform
  interface
10 #include <platform_interface/prototypes/_TPM_Hash_Start_fp.h>
11 #include <platform_interface/prototypes/_TPM_Hash_Data_fp.h>
12 #include <platform_interface/prototypes/_TPM_Hash_End_fp.h>
13
14 // Internal subsystem functions
15 #include "Object_fp.h"
16 #include "Context_spt_fp.h"
17 #include "Object_spt_fp.h"
18 #include "Entity_fp.h"
19 #include "Session_fp.h"
20 #include "Hierarchy_fp.h"
21 #include "NvReserved_fp.h"
22 #include "NvDynamic_fp.h"

```

```

23 #include "NV_spt_fp.h"
24 #include "ACT_spt_fp.h"
25 #include "PCR_fp.h"
26 #include "DA_fp.h"
27 // TODO_RENAME_INC_FOLDER: public refers to the TPM_CoreLib public headers
28 #include <public/prototypes/TpmFail_fp.h>
29 #include "SessionProcess_fp.h"
30
31 // Internal support functions
32 #include "CommandCodeAttributes_fp.h"
33 #include "Marshal.h"
34 #include "Time_fp.h"
35 #include "Locality_fp.h"
36 #include "PP_fp.h"
37 #include "CommandAudit_fp.h"
38 // TODO_RENAME_INC_FOLDER:platform_interface refers to the TPM_CoreLib platform
  interface
39 #include <platform_interface/prototypes/Manufacture_fp.h>
40 #include "Handle_fp.h"
41 #include "Power_fp.h"
42 #include "Response_fp.h"
43 #include "CommandDispatcher_fp.h"
44
45 #ifdef CC_AC_Send
46 # include "AC_spt_fp.h"
47 #endif // CC_AC_Send
48
49 // Miscellaneous
50 #include "Bits_fp.h"
51 #include "AlgorithmCap_fp.h"
52 #include "PropertyCap_fp.h"
53 #include "IoBuffers_fp.h"
54 #include "Memory_fp.h"
55 #include "ResponseCodeProcessing_fp.h"
56
57 // Internal cryptographic functions
58 #include "Ticket_fp.h"
59 #include "CryptUtil_fp.h"
60 #include "CryptHash_fp.h"
61 #include "CryptSym_fp.h"
62 #include "CryptPrime_fp.h"
63 #include "CryptRand_fp.h"
64 #include "CryptSelfTest_fp.h"
65 #include "MathOnByteBuffers_fp.h"
66 #include "CryptSym_fp.h"
67 #include "AlgorithmTests_fp.h"
68
69 #if ALG_RSA
70 # include "CryptRsa_fp.h"
71 # include "CryptPrimeSieve_fp.h"
72 #endif
73
74 #if ALG_ECC
75 # include "CryptEccMain_fp.h"
76 # include "CryptEccSignature_fp.h"
77 # include "CryptEccKeyExchange_fp.h"
78 # include "CryptEccCrypt_fp.h"
79 #endif
80
81 #if CC_MAC || CC_MAC_Start
82 # include "CryptSmac_fp.h"
83 # if ALG_CMAC
84 # include "CryptCmac_fp.h"
85 # endif
86 #endif
87

```

```

88 // Asymmetric Support library Interface
89 // TODO_RENAME_INC_FOLDER: needs a component prefix
90 #include <MathLibraryInterface.h>
91
92 // Linkage to platform functions
93 // TODO_RENAME_INC_FOLDER:platform_interface refers to the TPM_CoreLib platform
  interface
94 #include <platform_interface/tpm_to_platform_interface.h>
95
96 #endif

```

6.27 /tpm/include/private/KdfTestData.h

```

1 //
2 // Hash Test Vectors
3 //
4
5 #define TEST_KDF_KEY_SIZE 20
6
7 TPM2B_TYPE(KDF_TEST_KEY, TEST_KDF_KEY_SIZE);
8 TPM2B_KDF_TEST_KEY c_kdfTestKeyIn = {
9     {TEST_KDF_KEY_SIZE,
10      {0x27, 0x1F, 0xA0, 0x8B, 0xBD, 0xC5, 0x06, 0x0E, 0xC3, 0xDF,
11       0xA9, 0x28, 0xFF, 0x9B, 0x73, 0x12, 0x3A, 0x12, 0xDA, 0x0C}}};
12
13 TPM2B_TYPE(KDF_TEST_LABEL, 17);
14 TPM2B_KDF_TEST_LABEL c_kdfTestLabel = {{17,
15     {0x4B,
16      0x44,
17      0x46,
18      0x53,
19      0x45,
20      0x4C,
21      0x46,
22      0x54,
23      0x45,
24      0x53,
25      0x54,
26      0x4C,
27      0x41,
28      0x42,
29      0x45,
30      0x4C,
31      0x00}}};
32
33 TPM2B_TYPE(KDF_TEST_CONTEXT, 8);
34 TPM2B_KDF_TEST_CONTEXT c_kdfTestContextU = {
35     {8, {0xCE, 0x24, 0x4F, 0x39, 0x5D, 0xCA, 0x73, 0x91}}};
36
37 TPM2B_KDF_TEST_CONTEXT c_kdfTestContextV = {
38     {8, {0xDA, 0x50, 0x40, 0x31, 0xDD, 0xF1, 0x2E, 0x83}}};
39
40 #if ALG_SHA512 == ALG_YES
41 TPM2B_KDF_TEST_KEY c_kdfTestKeyOut = {
42     {20, {0x8b, 0xe2, 0xc1, 0xb8, 0x5b, 0x78, 0x56, 0x9b, 0x9f, 0xa7,
43          0x59, 0xf5, 0x85, 0x7c, 0x56, 0xd6, 0x84, 0x81, 0x0f, 0xd3}}};
44 # define KDF_TEST_ALG TPM_ALG_SHA512
45
46 #elif ALG_SHA384 == ALG_YES
47 TPM2B_KDF_TEST_KEY c_kdfTestKeyOut = {
48     {20, {0x1d, 0xce, 0x70, 0xc9, 0x11, 0x3e, 0xb2, 0xdb, 0xa4, 0x7b,
49          0xd9, 0xcf, 0xc7, 0x2b, 0xf4, 0x6f, 0x45, 0xb0, 0x93, 0x12}}};
50 # define KDF_TEST_ALG TPM_ALG_SHA384
51
52 #elif ALG_SHA256 == ALG_YES

```



```

53 TPM2B_KDF_TEST_KEY c_kdfTestKeyOut = {
54     {20, {0xbb, 0x02, 0x59, 0xe1, 0xc8, 0xba, 0x60, 0x7e, 0x6a, 0x2c,
55           0xd7, 0x04, 0xb6, 0x9a, 0x90, 0x2e, 0x9a, 0xde, 0x84, 0xc4}}};
56 # define KDF_TEST_ALG TPM_ALG_SHA256
57
58 #elif ALG_SHA1 == ALG_YES
59 TPM2B_KDF_TEST_KEY c_kdfTestKeyOut = {
60     {20, {0x55, 0xb5, 0xa7, 0x18, 0x4a, 0xa0, 0x74, 0x23, 0xc4, 0x7d,
61           0xae, 0x76, 0x6c, 0x26, 0xa2, 0x37, 0x7d, 0x7c, 0xf8, 0x51}}};
62 # define KDF_TEST_ALG TPM_ALG_SHA1
63 #endif

```

6.28 /tpm/include/private/LibSupport.h

```

1  // This header file is used to select the library code that gets included in the
2  // TPM build.
3
4  #ifndef LIB_SUPPORT_H
5  #define LIB_SUPPORT_H
6  // TODO_RENAME_INC_FOLDER: public refers to the TPM_CoreLib public headers
7  #include <public/tpm_radix.h>
8
9  // Include the options for hashing and symmetric. Defer the load of the math package
10 // Until the bignum parameters are defined.
11 #ifndef SYM_LIB
12 # error SYM_LIB required
13 #endif
14 #ifndef HASH_LIB
15 # error HASH_LIB required
16 #endif
17
18 #include LIB_INCLUDE(TpmTo, SYM_LIB, Sym)
19 #include LIB_INCLUDE(TpmTo, HASH_LIB, Hash)
20
21 //TODO: was #undef MIN
22 //was #undef MAX
23
24 #endif // LIB_SUPPORT_H

```

6.29 /tpm/include/private/Marshal.h

```

1  /** Introduction
2  // This file is used to provide the things needed by a module that uses the marshaling
3  // functions. It handles the variations between the marshaling choices (procedural or
4  // table-driven).
5
6  #if TABLE_DRIVEN_MARSHAL
7
8  # include "TableMarshalTypes.h"
9
10 # include "TableMarshalDefines.h"
11
12 # include "TableDrivenMarshal_fp.h"
13
14 #else
15
16 # include "Marshal_fp.h"
17
18 #endif

```

6.30 /tpm/include/private/NV.h

```

1  /** Index Type Definitions

```

```

2
3 // These definitions allow the same code to be used pre and post 1.21. The main
4 // action is to redefine the index type values from the bit values.
5 // Use TPM_NT_ORDINARY to indicate if the TPM_NT type is defined
6
7 #ifndef _NV_H_
8 #define _NV_H_
9
10 #ifndef TPM_NT_ORDINARY
11 // If TPM_NT_ORDINARY is defined, then the TPM_NT field is present in a TPMA_NV
12 # define GET_TPM_NT(attributes) GET_ATTRIBUTE(attributes, TPMA_NV, TPM_NT)
13 #else
14 // If TPM_NT_ORDINARY is not defined, then need to synthesize it from the
15 // attributes
16 # define GetNv_TPM_NV(attributes) \
17     (IS_ATTRIBUTE(attributes, TPMA_NV, COUNTER) \
18      + (IS_ATTRIBUTE(attributes, TPMA_NV, BITS) << 1) \
19      + (IS_ATTRIBUTE(attributes, TPMA_NV, EXTEND) << 2))
20 # define TPM_NT_ORDINARY (0)
21 # define TPM_NT_COUNTER (1)
22 # define TPM_NT_BITS (2)
23 # define TPM_NT_EXTEND (4)
24 #endif
25
26 /** Attribute Macros
27 // These macros are used to isolate the differences in the way that the index type
28 // changed in version 1.21 of the specification
29 #define IsNvOrdinaryIndex(attributes) (GET_TPM_NT(attributes) == TPM_NT_ORDINARY)
30
31 #define IsNvCounterIndex(attributes) (GET_TPM_NT(attributes) == TPM_NT_COUNTER)
32
33 #define IsNvBitsIndex(attributes) (GET_TPM_NT(attributes) == TPM_NT_BITS)
34
35 #define IsNvExtendIndex(attributes) (GET_TPM_NT(attributes) == TPM_NT_EXTEND)
36
37 #ifndef TPM_NT_PIN_PASS
38 # define IsNvPinPassIndex(attributes) (GET_TPM_NT(attributes) == TPM_NT_PIN_PASS)
39 #endif
40
41 #ifndef TPM_NT_PIN_FAIL
42 # define IsNvPinFailIndex(attributes) (GET_TPM_NT(attributes) == TPM_NT_PIN_FAIL)
43 #endif
44
45 typedef struct
46 {
47     UINT32 size;
48     TPM_HANDLE handle;
49 } NV_ENTRY_HEADER;
50
51 #define NV_EVICT_OBJECT_SIZE (sizeof(UINT32) + sizeof(TPM_HANDLE) + sizeof(OBJECT))
52
53 #define NV_INDEX_COUNTER_SIZE (sizeof(UINT32) + sizeof(NV_INDEX) + sizeof(UINT64))
54
55 #define NV_RAM_INDEX_COUNTER_SIZE (sizeof(NV_RAM_HEADER) + sizeof(UINT64))
56
57 typedef struct
58 {
59     UINT32 size;
60     TPM_HANDLE handle;
61     TPMA_NV attributes;
62 } NV_RAM_HEADER;
63
64 // Defines the end-of-list marker for NV. The list terminator is
65 // a UINT32 of zero, followed by the current value of s_maxCounter which is a
66 // 64-bit value. The structure is defined as an array of 3 UINT32 values so that
67 // there is no padding between the UINT32 list end marker and the UINT64 maxCounter

```

```

68 // value.
69 typedef UINT32 NV_LIST_TERMINATOR[3];
70
71 /** Orderly RAM Values
72 // The following defines are for accessing orderly RAM values.
73
74 // This is the initialize for the RAM reference iterator.
75 #define NV_RAM_REF_INIT 0
76 // This is the starting address of the RAM space used for orderly data
77 #define RAM_ORDERLY_START (&s_indexOrderlyRam[0])
78 // This is the offset within NV that is used to save the orderly data on an
79 // orderly shutdown.
80 #define NV_ORDERLY_START (NV_INDEX_RAM_DATA)
81 // This is the end of the orderly RAM space. It is actually the first byte after the
82 // last byte of orderly RAM data
83 #define RAM_ORDERLY_END (RAM_ORDERLY_START + sizeof(s_indexOrderlyRam))
84 // This is the end of the orderly space in NV memory. As with RAM_ORDERLY_END, it is
85 // actually the offset of the first byte after the end of the NV orderly data.
86 #define NV_ORDERLY_END (NV_ORDERLY_START + sizeof(s_indexOrderlyRam))
87
88 // Macro to check that an orderly RAM address is with range.
89 #define ORDERLY_RAM_ADDRESS_OK(start, offset) \
90     ((start >= RAM_ORDERLY_START) && ((start + offset - 1) < RAM_ORDERLY_END))
91
92 #define RETURN_IF_NV_IS_NOT_AVAILABLE \
93     { \
94         if(g_NvStatus != TPM_RC_SUCCESS) \
95             return g_NvStatus; \
96     }
97
98 // Routinely have to clear the orderly flag and fail if the
99 // NV is not available so that it can be cleared.
100 #define RETURN_IF_ORDERLY \
101     { \
102         if(NvClearOrderly() != TPM_RC_SUCCESS) \
103             return g_NvStatus; \
104     }
105
106 #define NV_IS_AVAILABLE (g_NvStatus == TPM_RC_SUCCESS)
107
108 #define IS_ORDERLY(value) (value < SU_DA_USED_VALUE)
109
110 #define NV_IS_ORDERLY (IS_ORDERLY(gp.orderlyState))
111
112 // Macro to set the NV UPDATE_TYPE. This deals with the fact that the update is
113 // possibly a combination of UT_NV and UT_ORDERLY.
114 #define SET_NV_UPDATE(type) g_updateNV |= (type)
115
116 #endif // _NV_H_

```

6.31 /tpm/include/private/OIDs.h

```

1 #ifndef _OIDS_H_
2 #define _OIDS_H_
3
4 // All the OIDs in this file are defined as DER-encoded values with a leading tag
5 // 0x06 (ASN1_OBJECT_IDENTIFIER), followed by a single length byte. This allows the
6 // OID size to be determined by looking at octet[1] of the OID (total size is
7 // OID[1] + 2).
8
9 // These macros allow OIDs to be defined (or not) depending on whether the associated
10 // hash algorithm is implemented.
11 // NOTE: When one of these macros is used, the NAME needs '_' on each side. The
12 // exception is when the macro is used for the hash OID when only a single '_' is
13 // used.

```

```
14 #ifndef ALG_SHA1
15 # define ALG_SHA1 NO
16 #endif
17 #if ALG_SHA1
18 # define SHA1_OID (NAME) MAKE_OID (NAME##SHA1)
19 #else
20 # define SHA1_OID (NAME)
21 #endif
22 #ifndef ALG_SHA256
23 # define ALG_SHA256 NO
24 #endif
25 #if ALG_SHA256
26 # define SHA256_OID (NAME) MAKE_OID (NAME##SHA256)
27 #else
28 # define SHA256_OID (NAME)
29 #endif
30 #ifndef ALG_SHA384
31 # define ALG_SHA384 NO
32 #endif
33 #if ALG_SHA384
34 # define SHA384_OID (NAME) MAKE_OID (NAME##SHA384)
35 #else
36 # define SHA384_OID (NAME)
37 #endif
38 #ifndef ALG_SHA512
39 # define ALG_SHA512 NO
40 #endif
41 #if ALG_SHA512
42 # define SHA512_OID (NAME) MAKE_OID (NAME##SHA512)
43 #else
44 # define SHA512_OID (NAME)
45 #endif
46 #ifndef ALG_SM3_256
47 # define ALG_SM3_256 NO
48 #endif
49 #if ALG_SM3_256
50 # define SM3_256_OID (NAME) MAKE_OID (NAME##SM3_256)
51 #else
52 # define SM3_256_OID (NAME)
53 #endif
54 #ifndef ALG_SHA3_256
55 # define ALG_SHA3_256 NO
56 #endif
57 #if ALG_SHA3_256
58 # define SHA3_256_OID (NAME) MAKE_OID (NAME##SHA3_256)
59 #else
60 # define SHA3_256_OID (NAME)
61 #endif
62 #ifndef ALG_SHA3_384
63 # define ALG_SHA3_384 NO
64 #endif
65 #if ALG_SHA3_384
66 # define SHA3_384_OID (NAME) MAKE_OID (NAME##SHA3_384)
67 #else
68 # define SHA3_384_OID (NAME)
69 #endif
70 #ifndef ALG_SHA3_512
71 # define ALG_SHA3_512 NO
72 #endif
73 #if ALG_SHA3_512
74 # define SHA3_512_OID (NAME) MAKE_OID (NAME##SHA3_512)
75 #else
76 # define SHA3_512_OID (NAME)
77 #endif
78
79 // These are encoded to take one additional byte of algorithm selector
```

```

80 #define NIST_HASH 0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 2
81 #define NIST_SIG 0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 3
82
83 // These hash OIDs used in a lot of places.
84 #define OID_SHA1_VALUE 0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A
85 SHA1_OID(_); // Expands to:
86 //     MAKE_OID(_SHA1)
87 // which expands to:
88 //     EXTERN const BYTE OID_SHA1[] INITIALIZER({OID_SHA1_VALUE})
89 // which, depending on the setting of EXTERN and
90 // INITIALIZER, expands to either:
91 //     extern const BYTE    OID_SHA1[]
92 // or
93 //     const BYTE          OID_SHA1[] = {OID_SHA1_VALUE}
94 // which is:
95 //     const BYTE          OID_SHA1[] = {0x06, 0x05, 0x2B, 0x0E,
96 //                                       0x03, 0x02, 0x1A}
97
98 #define OID_SHA256_VALUE NIST_HASH, 1
99 SHA256_OID(_);
100
101 #define OID_SHA384_VALUE NIST_HASH, 2
102 SHA384_OID(_);
103
104 #define OID_SHA512_VALUE NIST_HASH, 3
105 SHA512_OID(_);
106
107 #define OID_SM3_256_VALUE 0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, 0x83, 0x11
108 SM3_256_OID(_); // (1.2.156.10197.1.401)
109
110 #define OID_SHA3_256_VALUE NIST_HASH, 8
111 SHA3_256_OID(_);
112
113 #define OID_SHA3_384_VALUE NIST_HASH, 9
114 SHA3_384_OID(_);
115
116 #define OID_SHA3_512_VALUE NIST_HASH, 10
117 SHA3_512_OID(_);
118
119 // These are used for RSA-PSS
120 #if ALG_RSA
121
122 # define OID_MGF1_VALUE \
123     0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x01, 0x01, 0x08
124 MAKE_OID(_MGF1);
125
126 # define OID_RSAPSS_VALUE \
127     0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x01, 0x01, 0x0A
128 MAKE_OID(_RSAPSS);
129
130 // This is the OID to designate the public part of an RSA key.
131 # define OID_PKCS1_PUB_VALUE \
132     0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x01, 0x01, 0x01
133 MAKE_OID(_PKCS1_PUB);
134
135 // These are used for RSA PKCS1 signature Algorithms
136 # define OID_PKCS1_SHA1_VALUE \
137     0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x01, 0x01, 0x05
138 SHA1_OID(_PKCS1_); // (1.2.840.113549.1.1.5)
139
140 # define OID_PKCS1_SHA256_VALUE \
141     0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x01, 0x01, 0x0B
142 SHA256_OID(_PKCS1_); // (1.2.840.113549.1.1.11)
143
144 # define OID_PKCS1_SHA384_VALUE \
145     0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x01, 0x01, 0x0C

```

```

146 SHA384_OID(_PKCS1_); // (1.2.840.113549.1.1.12)
147
148 # define OID_PKCS1_SHA512_VALUE \
149     0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x01, 0x01, 0x0D
150 SHA512_OID(_PKCS1_); // (1.2.840.113549.1.1.13)
151
152 # define OID_PKCS1_SM3_256_VALUE \
153     0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, 0x83, 0x78
154 SM3_256_OID(_PKCS1_); // 1.2.156.10197.1.504
155
156 # define OID_PKCS1_SHA3_256_VALUE NIST_SIG, 14
157 SHA3_256_OID(_PKCS1_);
158 # define OID_PKCS1_SHA3_384_VALUE NIST_SIG, 15
159 SHA3_384_OID(_PKCS1_);
160 # define OID_PKCS1_SHA3_512_VALUE NIST_SIG, 16
161 SHA3_512_OID(_PKCS1_);
162
163 #endif // ALG_RSA
164
165 #if ALG_ECDSA
166
167 # define OID_ECDSA_SHA1_VALUE 0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, 0x01
168 SHA1_OID(_ECDSA_); // (1.2.840.10045.4.1) SHA1 digest signed by an ECDSA key.
169
170 # define OID_ECDSA_SHA256_VALUE \
171     0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, 0x03, 0x02
172 SHA256_OID(_ECDSA_); // (1.2.840.10045.4.3.2) SHA256 digest signed by an ECDSA key.
173
174 # define OID_ECDSA_SHA384_VALUE \
175     0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, 0x03, 0x03
176 SHA384_OID(_ECDSA_); // (1.2.840.10045.4.3.3) SHA384 digest signed by an ECDSA key.
177
178 # define OID_ECDSA_SHA512_VALUE \
179     0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, 0x03, 0x04
180 SHA512_OID(_ECDSA_); // (1.2.840.10045.4.3.4) SHA512 digest signed by an ECDSA key.
181
182 # define OID_ECDSA_SM3_256_VALUE \
183     0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, 0x83, 0x75
184 SM3_256_OID(_ECDSA_); // 1.2.156.10197.1.501
185
186 # define OID_ECDSA_SHA3_256_VALUE NIST_SIG, 10
187 SHA3_256_OID(_ECDSA_);
188 # define OID_ECDSA_SHA3_384_VALUE NIST_SIG, 11
189 SHA3_384_OID(_ECDSA_);
190 # define OID_ECDSA_SHA3_512_VALUE NIST_SIG, 12
191 SHA3_512_OID(_ECDSA_);
192
193 #endif // ALG_ECDSA
194
195 #if ALG_ECC
196
197 # define OID_ECC_PUBLIC_VALUE 0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x02, 0x01
198 MAKE_OID(_ECC_PUBLIC);
199
200 # define OID_ECC_NIST_P192_VALUE \
201     0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, 0x01, 0x01
202 # if ECC_NIST_P192
203 MAKE_OID(_ECC_NIST_P192); // (1.2.840.10045.3.1.1) 'nistP192'
204 # endif // ECC_NIST_P192
205
206 # define OID_ECC_NIST_P224_VALUE 0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x21
207 # if ECC_NIST_P224
208 MAKE_OID(_ECC_NIST_P224); // (1.3.132.0.33) 'nistP224'
209 # endif // ECC_NIST_P224
210
211 # define OID_ECC_NIST_P256_VALUE \

```



```

212     0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, 0x01, 0x07
213 # if ECC_NIST_P256
214 MAKE_OID(_ECC_NIST_P256); // (1.2.840.10045.3.1.7) 'nistP256'
215 # endif // ECC_NIST_P256
216
217 # define OID_ECC_NIST_P384_VALUE 0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x22
218 # if ECC_NIST_P384
219 MAKE_OID(_ECC_NIST_P384); // (1.3.132.0.34) 'nistP384'
220 # endif // ECC_NIST_P384
221
222 # define OID_ECC_NIST_P521_VALUE 0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x23
223 # if ECC_NIST_P521
224 MAKE_OID(_ECC_NIST_P521); // (1.3.132.0.35) 'nistP521'
225 # endif // ECC_NIST_P521
226
227 // No OIDs defined for these anonymous curves
228 # define OID_ECC_BN_P256_VALUE 0x00
229 # if ECC_BN_P256
230 MAKE_OID(_ECC_BN_P256);
231 # endif // ECC_BN_P256
232
233 # define OID_ECC_BN_P638_VALUE 0x00
234 # if ECC_BN_P638
235 MAKE_OID(_ECC_BN_P638);
236 # endif // ECC_BN_P638
237
238 # define OID_ECC_SM2_P256_VALUE \
239     0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, 0x82, 0x2D
240 # if ECC_SM2_P256
241 MAKE_OID(_ECC_SM2_P256); // Don't know where I found this OID. It needs checking
242 # endif // ECC_SM2_P256
243
244 # if ECC_BN_P256
245 # define OID_ECC_BN_P256 NULL
246 # endif // ECC_BN_P256
247
248 #endif // ALG_ECC
249
250 #define OID_SIZE(OID) (OID[1] + 2)
251
252 #endif // !_OIDS_H_

```

6.32 /tpm/include/private/PRNG_TestVectors.h

```

1  #ifndef MSBN_DRBG_TEST_VECTORS_H
2  #define MSBN_DRBG_TEST_VECTORS_H
3
4  // #if DRBG_ALGORITHM == TPM_ALG_AES && DRBG_KEY_BITS == 256
5  #if DRBG_KEY_SIZE_BITS == 256
6
7  /* (NIST test vector)
8  [AES-256 no df]
9  [PredictionResistance = False]
10 [EntropyInputLen = 384]
11 [NonceLen = 128]
12 [PersonalizationStringLen = 0]
13 [AdditionalInputLen = 0]
14
15 COUNT = 0
16 EntropyInput = 0d15aa80 b16c3a10 906cfedb 795dae0b 5b81041c 5c5bfacb
17                373d4440 d9120f7e 3d6cf909 86cf52d8 5d3e947d 8c061f91
18 Nonce = 06caef5f b538e08e 1f3b0452 03f8f4b2
19 PersonalizationString =
20 AdditionalInput =
21     INTERMEDIATE Key = be5df629 34cc1230 166a6773 345bbd6b

```

```

22         4c8869cf 8aec1c3b 1aa98bca 37cac6f1
23     INTERMEDIATE V = 3182dd1e 7638ec70 014e93bd 813e524c
24     INTERMEDIATE ReturnedBits = 28e0ebb8 21016650 8c8f65f2 207bd0a3
25 EntropyInputReseed = 6ee793a3 3955d72a d12fd80a 8a3fcf95 ed3b4dac 5795fe25
26         cf869f7c 27573bbc 56f1acae 13a65042 b340093c 464a7a22
27 AdditionalInputReseed =
28 AdditionalInput =
29 ReturnedBits = 946f5182 d54510b9 461248f5 71ca06c9
30 */
31
32 // Entropy is the size of the state. The state is the size of the key
33 // plus the IV. The IV is a block. If Key = 256 and Block = 128 then State = 384
34 # define DRBG_TEST_INITIATE_ENTROPY \
35     0x0d, 0x15, 0xaa, 0x80, 0xb1, 0x6c, 0x3a, 0x10, 0x90, 0x6c, 0xfe, 0xdb, 0x79, \
36     0x5d, 0xae, 0x0b, 0x5b, 0x81, 0x04, 0x1c, 0x5c, 0x5b, 0xfa, 0xcb, 0x37, \
37     0x3d, 0x44, 0x40, 0xd9, 0x12, 0x0f, 0x7e, 0x3d, 0x6c, 0xf9, 0x09, 0x86, \
38     0xcf, 0x52, 0xd8, 0x5d, 0x3e, 0x94, 0x7d, 0x8c, 0x06, 0x1f, 0x91
39
40 # define DRBG_TEST_RESEED_ENTROPY \
41     0x6e, 0xe7, 0x93, 0xa3, 0x39, 0x55, 0xd7, 0x2a, 0xd1, 0x2f, 0xd8, 0x0a, 0x8a, \
42     0x3f, 0xcf, 0x95, 0xed, 0x3b, 0x4d, 0xac, 0x57, 0x95, 0xfe, 0x25, 0xcf, \
43     0x86, 0x9f, 0x7c, 0x27, 0x57, 0x3b, 0xbc, 0x56, 0xf1, 0xac, 0xae, 0x13, \
44     0xa6, 0x50, 0x42, 0xb3, 0x40, 0x09, 0x3c, 0x46, 0x4a, 0x7a, 0x22
45
46 # define DRBG_TEST_GENERATED_INTERM \
47     0x28, 0xe0, 0xeb, 0xb8, 0x21, 0x01, 0x66, 0x50, 0x8c, 0x8f, 0x65, 0xf2, 0x20, \
48     0x7b, 0xd0, 0xa3
49
50 # define DRBG_TEST_GENERATED \
51     0x94, 0x6f, 0x51, 0x82, 0xd5, 0x45, 0x10, 0xb9, 0x46, 0x12, 0x48, 0xf5, 0x71, \
52     0xca, 0x06, 0xc9
53 #elif DRBG_KEY_SIZE_BITS == 128
54 /*(NIST test vector)
55 [AES-128 no df]
56 [PredictionResistance = False]
57 [EntropyInputLen = 256]
58 [NonceLen = 64]
59 [PersonalizationStringLen = 0]
60 [AdditionalInputLen = 0]
61
62 COUNT = 0
63 EntropyInput = 8fc11bdb5aabb7e093b61428e0907303cb459f3b600dad870955f22da80a44f8
64 Nonce = belf73885ddd15aa
65 PersonalizationString =
66 AdditionalInput =
67     INTERMEDIATE Key = b134ecc836df6dbd624900af118dd7e6
68     INTERMEDIATE V = 01bb09e86dabd75c9f26dbf6f9531368
69     INTERMEDIATE ReturnedBits = dc3cf6bf5bd341135f2c6811a1071c87
70 EntropyInputReseed =
71     0cd53cd5eccd5a10d7ea266111259b05574fc6ddd8bed8bd72378cf82f1dba2a
72 AdditionalInputReseed =
73 AdditionalInput =
74 ReturnedBits = b61850decfd7106d44769a8e6e8c1ad4
75 */
76
77 # define DRBG_TEST_INITIATE_ENTROPY \
78     0x8f, 0xc1, 0x1b, 0xdb, 0x5a, 0xab, 0xb7, 0xe0, 0x93, 0xb6, 0x14, 0x28, 0xe0, \
79     0x90, 0x73, 0x03, 0xcb, 0x45, 0x9f, 0x3b, 0x60, 0x0d, 0xad, 0x87, 0x09, \
80     0x55, 0xf2, 0x2d, 0xa8, 0x0a, 0x44, 0xf8
81
82 # define DRBG_TEST_RESEED_ENTROPY \
83     0x0c, 0xd5, 0x3c, 0xd5, 0xec, 0xcd, 0x5a, 0x10, 0xd7, 0xea, 0x26, 0x61, 0x11, \
84     0x25, 0x9b, 0x05, 0x57, 0x4f, 0xc6, 0xdd, 0xd8, 0xbe, 0xd8, 0xbd, 0x72, \
85     0x37, 0x8c, 0xf8, 0x2f, 0x1d, 0xba, 0x2a
86
87 # define DRBG_TEST_GENERATED_INTERM \

```

```

88     0xdc, 0x3c, 0xf6, 0xbf, 0x5b, 0xd3, 0x41, 0x13, 0x5f, 0x2c, 0x68, 0x11, 0xa1, \
89     0x07, 0x1c, 0x87
90
91 #   define DRBG_TEST_GENERATED
92     0xb6, 0x18, 0x50, 0xde, 0xcf, 0xd7, 0x10, 0x6d, 0x44, 0x76, 0x9a, 0x8e, 0x6e, \
93     0x8c, 0x1a, 0xd4
94
95 #endif
96
97 #endif //      _MSBN_DRBG_TEST_VECTORS_H

```

6.33 /tpm/include/private/RsaTestData.h

```

1  //
2  // RSA Test Vectors
3
4  #define RSA_TEST_KEY_SIZE 256
5
6  typedef struct
7  {
8      UINT16 size;
9      BYTE  buffer[RSA_TEST_KEY_SIZE];
10 } TPM2B_RSA_TEST_KEY;
11
12 typedef TPM2B_RSA_TEST_KEY TPM2B_RSA_TEST_VALUE;
13
14 typedef struct
15 {
16     UINT16 size;
17     BYTE  buffer[RSA_TEST_KEY_SIZE / 2];
18 } TPM2B_RSA_TEST_PRIME;
19
20 const TPM2B_RSA_TEST_KEY c_rsaPublicModulus =
21 {256,
22  {0x91, 0x12, 0xf5, 0x07, 0x9d, 0x5f, 0x6b, 0x1c, 0x90, 0xf6, 0xcc, 0x87, 0xde,
23   0x3a, 0x7a, 0x15, 0xdc, 0x54, 0x07, 0x6c, 0x26, 0x8f, 0x25, 0xef, 0x7e, 0x66,
24   0xc0, 0xe3, 0x82, 0x12, 0x2f, 0xab, 0x52, 0x82, 0x1e, 0x85, 0xbc, 0x53, 0xba,
25   0x2b, 0x01, 0xad, 0x01, 0xc7, 0x8d, 0x46, 0x4f, 0x7d, 0xdd, 0x7e, 0xdc, 0xb0,
26   0xad, 0xf6, 0x0c, 0xa1, 0x62, 0x92, 0x97, 0x8a, 0x3e, 0x6f, 0x7e, 0x3e, 0xf6,
27   0x9a, 0xcc, 0xf9, 0xa9, 0x86, 0x77, 0xb6, 0x85, 0x43, 0x42, 0x04, 0x13, 0x65,
28   0xe2, 0xad, 0x36, 0xc9, 0xbf, 0xc1, 0x97, 0x84, 0x6f, 0xee, 0x7c, 0xda, 0x58,
29   0xd2, 0xae, 0x07, 0x00, 0xaf, 0xc5, 0x5f, 0x4d, 0x3a, 0x98, 0xb0, 0xed, 0x27,
30   0x7c, 0xc2, 0xce, 0x26, 0x5d, 0x87, 0xe1, 0xe3, 0xa9, 0x69, 0x88, 0x4f, 0x8c,
31   0x08, 0x31, 0x18, 0xae, 0x93, 0x16, 0xe3, 0x74, 0xde, 0xd3, 0xf6, 0x16, 0xaf,
32   0xa3, 0xac, 0x37, 0x91, 0x8d, 0x10, 0xc6, 0x6b, 0x64, 0x14, 0x3a, 0xd9, 0xfc,
33   0xe4, 0xa0, 0xf2, 0xd1, 0x01, 0x37, 0x4f, 0x4a, 0xeb, 0xe5, 0xec, 0x98, 0xc5,
34   0xd9, 0x4b, 0x30, 0xd2, 0x80, 0x2a, 0x5a, 0x18, 0x5a, 0x7d, 0xd4, 0x3d, 0xb7,
35   0x62, 0x98, 0xce, 0x6d, 0xa2, 0x02, 0x6e, 0x45, 0xaa, 0x95, 0x73, 0xe0, 0xaa,
36   0x75, 0x57, 0xb1, 0x3d, 0x1b, 0x05, 0x75, 0x23, 0x6b, 0x20, 0x69, 0x9e, 0x14,
37   0xb0, 0x7f, 0xac, 0xae, 0xd2, 0xc7, 0x48, 0x3b, 0xe4, 0x56, 0x11, 0x34, 0x1e,
38   0x05, 0x1a, 0x30, 0x20, 0xef, 0x68, 0x93, 0x6b, 0x9d, 0x7e, 0xdd, 0xba, 0x96,
39   0x50, 0xcc, 0x1c, 0x81, 0xb4, 0x59, 0xb9, 0x74, 0x36, 0xd9, 0x97, 0xdc, 0x8f,
40   0x17, 0x82, 0x72, 0xb3, 0x59, 0xf6, 0x23, 0xfa, 0x84, 0xf7, 0x6d, 0xf2, 0x05,
41   0xff, 0xf1, 0xb9, 0xcc, 0xe9, 0xa2, 0x82, 0x01, 0xfb}};
42
43 const TPM2B_RSA_TEST_PRIME c_rsaPrivatePrime =
44 {RSA_TEST_KEY_SIZE / 2,
45  {0xb7, 0xa0, 0x90, 0xc7, 0x92, 0x09, 0xde, 0x71, 0x03, 0x37, 0x4a, 0xb5, 0x2f,
46   0xda, 0x61, 0xb8, 0x09, 0x1b, 0xba, 0x99, 0x70, 0x45, 0xc1, 0x0b, 0x15, 0x12,
47   0x71, 0x8a, 0xb3, 0x2a, 0x4d, 0x5a, 0x41, 0x9b, 0x73, 0x89, 0x80, 0x0a, 0x8f,
48   0x18, 0x4c, 0x8b, 0xa2, 0x5b, 0xda, 0xbd, 0x43, 0xbe, 0xdc, 0x76, 0x4d, 0x71,
49   0x0f, 0xb9, 0xfc, 0x7a, 0x09, 0xfe, 0x4f, 0xac, 0x63, 0xd9, 0x2e, 0x50, 0x3a,
50   0xa1, 0x37, 0xc6, 0xf2, 0xa1, 0x89, 0x12, 0xe7, 0x72, 0x64, 0x2b, 0xba, 0xc1,
51   0x1f, 0xca, 0x9d, 0xb7, 0xaa, 0x3a, 0xa9, 0xd3, 0xa6, 0x6f, 0x73, 0x02, 0xbb,
52   0x85, 0x5d, 0x9a, 0xb9, 0x5c, 0x08, 0x83, 0x22, 0x20, 0x49, 0x91, 0x5f, 0x4b,

```

```

53         0x86, 0xbc, 0x3f, 0x76, 0x43, 0x08, 0x97, 0xbf, 0x82, 0x55, 0x36, 0x2d, 0x8b,
54         0x6e, 0x9e, 0xfb, 0xc1, 0x67, 0x6a, 0x43, 0xa2, 0x46, 0x81, 0x71}};
55
56 const BYTE c_RsaTestValue[RSA_TEST_KEY_SIZE] =
57 {0x2a, 0x24, 0x3a, 0xbb, 0x50, 0x1d, 0xd4, 0x2a, 0xf9, 0x18, 0x32, 0x34, 0xa2,
58  0x0f, 0xea, 0x5c, 0x91, 0x77, 0xe9, 0xe1, 0x09, 0x83, 0xdc, 0x5f, 0x71, 0x64,
59  0x5b, 0xeb, 0x57, 0x79, 0xa0, 0x41, 0xc9, 0xe4, 0x5a, 0x0b, 0xf4, 0x9f, 0xdb,
60  0x84, 0x04, 0xa6, 0x48, 0x24, 0xf6, 0x3f, 0x66, 0x1f, 0xa8, 0x04, 0x5c, 0xf0,
61  0x7a, 0x6b, 0x4a, 0x9c, 0x7e, 0x21, 0xb6, 0xda, 0x6b, 0x65, 0x9c, 0x3a, 0x68,
62  0x50, 0x13, 0x1e, 0xa4, 0xb7, 0xca, 0xec, 0xd3, 0xcc, 0xb2, 0x9b, 0x8c, 0x87,
63  0xa4, 0x6a, 0xba, 0xc2, 0x06, 0x3f, 0x40, 0x48, 0x7b, 0xa8, 0xb8, 0x2c, 0x03,
64  0x14, 0x33, 0xf3, 0x1d, 0xe9, 0xbd, 0x6f, 0x54, 0x66, 0xb4, 0x69, 0x5e, 0xbc,
65  0x80, 0x7c, 0xe9, 0x6a, 0x43, 0x7f, 0xb8, 0x6a, 0xa0, 0x5f, 0x5d, 0x7a, 0x20,
66  0xfd, 0x7a, 0x39, 0xe1, 0xea, 0x0e, 0x94, 0x91, 0x28, 0x63, 0x7a, 0xac, 0xc9,
67  0xa5, 0x3a, 0x6d, 0x31, 0x7b, 0x7c, 0x54, 0x56, 0x99, 0x56, 0xbb, 0xb7, 0xa1,
68  0x2d, 0xd2, 0x5c, 0x91, 0x5f, 0x1c, 0xd3, 0x06, 0x7f, 0x34, 0x53, 0x2f, 0x4c,
69  0xd1, 0x8b, 0xd2, 0x9e, 0xdc, 0xc3, 0x94, 0x0a, 0xe1, 0x0f, 0xa5, 0x15, 0x46,
70  0x2a, 0x8e, 0x10, 0xc2, 0xfe, 0xb7, 0x5e, 0x2d, 0x0d, 0xd1, 0x25, 0xfc, 0xe4,
71  0xf7, 0x02, 0x19, 0xfe, 0xb6, 0xe4, 0x95, 0x9c, 0x17, 0x4a, 0x9b, 0xdb, 0xab,
72  0xc7, 0x79, 0xe3, 0x5e, 0x40, 0xd0, 0x56, 0x6d, 0x25, 0x0a, 0x72, 0x65, 0x80,
73  0x92, 0x9a, 0xa8, 0x07, 0x70, 0x32, 0x14, 0xfb, 0xfe, 0x08, 0xeb, 0x13, 0xb4,
74  0x07, 0x68, 0xb4, 0x58, 0x39, 0xbe, 0x8e, 0x78, 0x3a, 0x59, 0x3f, 0x9c, 0x4c,
75  0xe9, 0xa8, 0x64, 0x68, 0xf7, 0xb9, 0x6e, 0x20, 0xf5, 0xcb, 0xca, 0x47, 0xf2,
76  0x17, 0xaa, 0x8b, 0xbc, 0x13, 0x14, 0x84, 0xf6, 0xab};
77
78 const TPM2B_RSA_TEST_VALUE c_RsaepKvt =
79 {RSA_TEST_KEY_SIZE,
80  {0x73, 0xbd, 0x65, 0x49, 0xda, 0x7b, 0xb8, 0x50, 0x9e, 0x87, 0xf0, 0x0a, 0x8a,
81   0x9a, 0x07, 0xb6, 0x00, 0x82, 0x10, 0x14, 0x60, 0xd8, 0x01, 0xfc, 0xc5, 0x18,
82   0xea, 0x49, 0x5f, 0x13, 0xcf, 0x65, 0x66, 0x30, 0x6c, 0x60, 0x3f, 0x24, 0x3c,
83   0xfb, 0xe2, 0x31, 0x16, 0x99, 0x7e, 0x31, 0x98, 0xab, 0x93, 0xb8, 0x07, 0x53,
84   0xcc, 0xdb, 0x7f, 0x44, 0xd9, 0xee, 0x5d, 0xe8, 0x5f, 0x97, 0x5f, 0xe8, 0x1f,
85   0x88, 0x52, 0x24, 0x7b, 0xac, 0x62, 0x95, 0xb7, 0x7d, 0xf5, 0xf8, 0x9f, 0x5a,
86   0xa8, 0x24, 0x9a, 0x76, 0x71, 0x2a, 0x35, 0x2a, 0xa1, 0x08, 0xbb, 0x95, 0xe3,
87   0x64, 0xdc, 0xdb, 0xc2, 0x33, 0xa9, 0x5f, 0xbe, 0x4c, 0xc4, 0xcc, 0x28, 0xc9,
88   0x25, 0xff, 0xee, 0x17, 0x15, 0x9a, 0x50, 0x90, 0x0e, 0x15, 0xb4, 0xea, 0x6a,
89   0x09, 0xe6, 0xff, 0xa4, 0xee, 0xc7, 0x7e, 0xce, 0xa9, 0x73, 0xe4, 0xa0, 0x56,
90   0xbd, 0x53, 0x2a, 0xe4, 0xc0, 0x2b, 0xa8, 0x9b, 0x09, 0x30, 0x72, 0x62, 0x0f,
91   0xf9, 0xf6, 0xa1, 0x52, 0xd2, 0x8a, 0x37, 0xee, 0xa5, 0xc8, 0x47, 0xe1, 0x99,
92   0x21, 0x47, 0xeb, 0xdd, 0x37, 0xaa, 0xe4, 0xbd, 0x55, 0x46, 0x5a, 0x5a, 0x5d,
93   0xfb, 0x7b, 0xfc, 0xff, 0xbf, 0x26, 0x71, 0xf6, 0x1e, 0xad, 0xbc, 0xbf, 0x33,
94   0xca, 0xe1, 0x92, 0x8f, 0x2a, 0x89, 0x6c, 0x45, 0x24, 0xd1, 0xa6, 0x52, 0x56,
95   0x24, 0x5e, 0x90, 0x47, 0xe5, 0xcb, 0x12, 0xb0, 0x32, 0xf9, 0xa6, 0xbb, 0xea,
96   0x37, 0xa9, 0xbd, 0xef, 0x23, 0xef, 0x63, 0x07, 0x6c, 0xc4, 0x4e, 0x64, 0x3c,
97   0xc6, 0x11, 0x84, 0x7d, 0x65, 0xd6, 0x5d, 0x7a, 0x17, 0x58, 0xa5, 0xf7, 0x74,
98   0x3b, 0x42, 0xe3, 0xd2, 0xda, 0x5f, 0x6f, 0xe0, 0x1e, 0x4b, 0xcf, 0x46, 0xe2,
99   0xdf, 0x3e, 0x41, 0x8e, 0x0e, 0xb0, 0x3f, 0x8b, 0x65}};
100
101 #define OAEP_TEST_LABEL "OAEP Test Value"
102
103 #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
104
105 const TPM2B_RSA_TEST_VALUE c_OaepKvt =
106 {RSA_TEST_KEY_SIZE,
107  {0x32, 0x68, 0x84, 0x0b, 0x9c, 0xc9, 0x25, 0x26, 0xd9, 0xc0, 0xd0, 0xb1, 0xde,
108   0x60, 0x55, 0xae, 0x33, 0xe5, 0xcf, 0x6c, 0x85, 0xbe, 0x0d, 0x71, 0x11, 0xe1,
109   0x45, 0x60, 0xbb, 0x42, 0x3d, 0xf3, 0xb1, 0x18, 0x84, 0x7b, 0xc6, 0x5d, 0xce,
110   0x1d, 0x5f, 0x9a, 0x97, 0xcf, 0xb1, 0x97, 0x9a, 0x85, 0x7c, 0xa7, 0xa1, 0x63,
111   0x23, 0xb6, 0x74, 0x0f, 0x1a, 0xee, 0x29, 0x51, 0xeb, 0x50, 0x8f, 0x3c, 0x8e,
112   0x4e, 0x31, 0x38, 0xdc, 0x11, 0xfc, 0x9a, 0x4e, 0xaf, 0x93, 0xc9, 0x7f, 0x6e,
113   0x35, 0xf3, 0xc9, 0xe4, 0x89, 0x14, 0x53, 0xe2, 0xc2, 0x1a, 0xf7, 0x6b, 0x9b,
114   0xf0, 0x7a, 0xa4, 0x69, 0x52, 0xe0, 0x24, 0x8f, 0xea, 0x31, 0xa7, 0x5c, 0x43,
115   0xb0, 0x65, 0xc9, 0xfe, 0xba, 0xfe, 0x80, 0x9e, 0xa5, 0xc0, 0xf5, 0x8d, 0xce,
116   0x41, 0xf9, 0x83, 0x0d, 0x8e, 0x0f, 0xef, 0x3d, 0x1f, 0x6a, 0xcc, 0x8a, 0x3d,
117   0x3b, 0xdf, 0x22, 0x38, 0xd7, 0x34, 0x58, 0x7b, 0x55, 0xc9, 0xf6, 0xbc, 0x7c,
118   0x4c, 0x3f, 0xd7, 0xde, 0x4e, 0x30, 0xa9, 0x69, 0xf3, 0x5f, 0x56, 0x8f, 0xc2,

```

```

119     0xe7, 0x75, 0x79, 0xb8, 0xa5, 0xc8, 0x0d, 0xc0, 0xcd, 0xb6, 0xc9, 0x63, 0xad,
120     0x7c, 0xe4, 0x8f, 0x39, 0x60, 0x4d, 0x7d, 0xdb, 0x34, 0x49, 0x2a, 0x47, 0xde,
121     0xc0, 0x42, 0x4a, 0x19, 0x94, 0x2e, 0x50, 0x21, 0x03, 0x47, 0xff, 0x73, 0xb3,
122     0xb7, 0x89, 0xcc, 0x7b, 0x2c, 0xeb, 0x03, 0xa7, 0x9a, 0x06, 0xfd, 0xed, 0x19,
123     0xbb, 0x82, 0xa0, 0x13, 0xe9, 0xfa, 0xac, 0x06, 0x5f, 0xc5, 0xa9, 0x2b, 0xda,
124     0x88, 0x23, 0xa2, 0x5d, 0xc2, 0x7f, 0xda, 0xc8, 0x5a, 0x94, 0x31, 0xc1, 0x21,
125     0xd7, 0x1e, 0x6b, 0xd7, 0x89, 0xb1, 0x93, 0x80, 0xab, 0xd1, 0x37, 0xf2, 0x6f,
126     0x50, 0xcd, 0x2a, 0xea, 0xb1, 0xc4, 0xcd, 0xcb, 0xb5}};
127
128 const TPM2B_RSA_TEST_VALUE c_RsaesKvt =
129 {RSA_TEST_KEY_SIZE,
130  {0x29, 0xa4, 0x2f, 0xbb, 0x8a, 0x14, 0x05, 0x1e, 0x3c, 0x72, 0x76, 0x77, 0x38,
131   0xe7, 0x73, 0xe3, 0x6e, 0x24, 0x4b, 0x38, 0xd2, 0x1a, 0xcf, 0x23, 0x58, 0x78,
132   0x36, 0x82, 0x23, 0x6e, 0x6b, 0xef, 0x2c, 0x3d, 0xf2, 0xe8, 0xd6, 0xc6, 0x87,
133   0x8e, 0x78, 0x9b, 0x27, 0x39, 0xc0, 0xd6, 0xef, 0x4d, 0x0b, 0xfc, 0x51, 0x27,
134   0x18, 0xf3, 0x51, 0x5e, 0x4d, 0x96, 0x3a, 0xe2, 0x15, 0xe2, 0x7e, 0x42, 0xf4,
135   0x16, 0xd5, 0xc6, 0x52, 0x5d, 0x17, 0x44, 0x76, 0x09, 0x7a, 0xcf, 0xe3, 0x30,
136   0xe3, 0x84, 0xf6, 0x6f, 0x3a, 0x33, 0xfb, 0x32, 0x0d, 0x1d, 0xe7, 0x7c, 0x80,
137   0x82, 0x4f, 0xed, 0xda, 0x87, 0x11, 0x9c, 0xc3, 0x7e, 0x85, 0xbd, 0x18, 0x58,
138   0x08, 0x2b, 0x23, 0x37, 0xe7, 0x9d, 0xd0, 0xd1, 0x79, 0xe2, 0x05, 0xbd, 0xf5,
139   0x4f, 0x0e, 0x0f, 0xdb, 0x4a, 0x74, 0xeb, 0x09, 0x01, 0xb3, 0xca, 0xbd, 0xa6,
140   0x7b, 0x09, 0xb1, 0x13, 0x77, 0x30, 0x4d, 0x87, 0x41, 0x06, 0x57, 0x2e, 0x5f,
141   0x36, 0x6e, 0xfc, 0x35, 0x69, 0xfe, 0x0a, 0x24, 0x6c, 0x98, 0x8c, 0xda, 0x97,
142   0xf4, 0xfb, 0xc7, 0x83, 0x2d, 0x3e, 0x7d, 0xc0, 0x5c, 0x34, 0xfd, 0x11, 0x2a,
143   0x12, 0xa7, 0xae, 0x4a, 0xde, 0xc8, 0x4e, 0xcf, 0xf4, 0x85, 0x63, 0x77, 0xc6,
144   0x33, 0x34, 0xe0, 0x27, 0xe4, 0x9e, 0x91, 0x0b, 0x4b, 0x85, 0xf0, 0xb0, 0x79,
145   0xaa, 0x7c, 0xc6, 0xff, 0x3b, 0xbc, 0x04, 0x73, 0xb8, 0x95, 0xd7, 0x31, 0x54,
146   0x3b, 0x56, 0xec, 0x52, 0x15, 0xd7, 0x3e, 0x62, 0xf5, 0x82, 0x99, 0x3e, 0x2a,
147   0xc0, 0x4b, 0x2e, 0x06, 0x57, 0x6d, 0x3f, 0x3e, 0x77, 0x1f, 0x2b, 0x2d, 0xc5,
148   0xb9, 0x3b, 0x68, 0x56, 0x73, 0x70, 0x32, 0x6b, 0x6b, 0x65, 0x25, 0x76, 0x45,
149   0x6c, 0x45, 0xf1, 0x6c, 0x59, 0xfc, 0x94, 0xa7, 0x15}};
150
151 const TPM2B_RSA_TEST_VALUE c_RsapssKvt =
152 {RSA_TEST_KEY_SIZE,
153  {0x01, 0xfe, 0xd5, 0x83, 0x0b, 0x15, 0xba, 0x90, 0x2c, 0xdf, 0xf7, 0x26, 0xb7,
154   0x8f, 0xb1, 0xd7, 0x0b, 0xfd, 0x83, 0xf9, 0x95, 0xd5, 0xd7, 0xb5, 0xc5, 0xc5,
155   0x4a, 0xde, 0xd5, 0xe6, 0x20, 0x78, 0xca, 0x73, 0x77, 0x3d, 0x61, 0x36, 0x48,
156   0xae, 0x3e, 0x8f, 0xee, 0x43, 0x29, 0x96, 0xdf, 0x3f, 0x1c, 0x97, 0x5a, 0xbe,
157   0xe5, 0xa2, 0x7e, 0x5b, 0xd0, 0xc0, 0x29, 0x39, 0x83, 0x81, 0x77, 0x24, 0x43,
158   0xdb, 0x3c, 0x64, 0x4d, 0xf0, 0x23, 0xe4, 0xae, 0x0f, 0x78, 0x31, 0x8c, 0xda,
159   0x0c, 0xec, 0xf1, 0xdf, 0x09, 0xf2, 0x14, 0x6a, 0x4d, 0xaf, 0x36, 0x81, 0x6e,
160   0xbd, 0xbe, 0x36, 0x79, 0x88, 0x98, 0xb6, 0x6f, 0x5a, 0xad, 0xcf, 0x7c, 0xee,
161   0xe0, 0xdd, 0x00, 0xbe, 0x59, 0x97, 0x88, 0x00, 0x34, 0xc0, 0x8b, 0x48, 0x42,
162   0x05, 0x04, 0x5a, 0xb7, 0x85, 0x38, 0xa0, 0x35, 0xd7, 0x3b, 0x51, 0xb8, 0x7b,
163   0x81, 0x83, 0xee, 0xff, 0x76, 0x6f, 0x50, 0x39, 0x4d, 0xab, 0x89, 0x63, 0x07,
164   0x6d, 0xf5, 0xe5, 0x01, 0x10, 0x56, 0xfe, 0x93, 0x06, 0x8f, 0xd3, 0xc9, 0x41,
165   0xab, 0xc9, 0xdf, 0x6e, 0x59, 0xa8, 0xc3, 0x1d, 0xbf, 0x96, 0x4a, 0x59, 0x80,
166   0x3c, 0x90, 0x3a, 0x59, 0x56, 0x4c, 0x6d, 0x44, 0x6d, 0xeb, 0xdc, 0x73, 0xcd,
167   0xc1, 0xec, 0xb8, 0x41, 0xbf, 0x89, 0x8c, 0x03, 0x69, 0x4c, 0xaf, 0x3f, 0xc1,
168   0xc5, 0xc7, 0xe7, 0x7d, 0xa7, 0x83, 0x39, 0x70, 0xa2, 0x6b, 0x83, 0xbc, 0xbe,
169   0xf5, 0xbf, 0x1c, 0xee, 0x6e, 0xa3, 0x22, 0x1e, 0x25, 0x2f, 0x16, 0x68, 0x69,
170   0x5a, 0x1d, 0xfa, 0x2c, 0x3a, 0x0f, 0x67, 0xe1, 0x77, 0x12, 0xe8, 0x3d, 0xba,
171   0xaa, 0xef, 0x96, 0x9c, 0x1f, 0x64, 0x32, 0xf4, 0xa7, 0xb3, 0x3f, 0x7d, 0x61,
172   0xbb, 0x9a, 0x27, 0xad, 0xfb, 0x2f, 0x33, 0xc4, 0x70}};
173
174 const TPM2B_RSA_TEST_VALUE c_RsassaKvt =
175 {RSA_TEST_KEY_SIZE,
176  {0x67, 0x4e, 0xdd, 0xc2, 0xd2, 0x6d, 0xe0, 0x03, 0xc4, 0xc2, 0x41, 0xd3, 0xd4,
177   0x61, 0x30, 0xd0, 0xe1, 0x68, 0x31, 0x4a, 0xda, 0xd9, 0xc2, 0x5d, 0xaa, 0xa2,
178   0x7b, 0xfb, 0x44, 0x02, 0xf5, 0xd6, 0xd8, 0x2e, 0xcd, 0x13, 0x36, 0xc9, 0x4b,
179   0xdb, 0x1a, 0x4b, 0x66, 0x1b, 0x4f, 0x9c, 0xb7, 0x17, 0xac, 0x53, 0x37, 0x4f,
180   0x21, 0xbd, 0x0c, 0x66, 0xac, 0x06, 0x65, 0x52, 0x9f, 0x04, 0xf6, 0xa5, 0x22,
181   0x5b, 0xf7, 0xe6, 0x0d, 0x3c, 0x9f, 0x41, 0x19, 0x09, 0x88, 0x7c, 0x41, 0x4c,
182   0x2f, 0x9c, 0x8b, 0x3c, 0xdd, 0x7c, 0x28, 0x78, 0x24, 0xd2, 0x09, 0xa6, 0x5b,
183   0xf7, 0x3c, 0x88, 0x7e, 0x73, 0x5a, 0x2d, 0x36, 0x02, 0x4f, 0x65, 0xb0, 0xcb,
184   0xc8, 0xdc, 0xac, 0xa2, 0xda, 0x8b, 0x84, 0x91, 0x71, 0xe4, 0x30, 0x8b, 0xb6,

```



```

185     0x12, 0xf2, 0xf0, 0xd0, 0xa0, 0x38, 0xcf, 0x75, 0xb7, 0x20, 0xcb, 0x35, 0x51,
186     0x52, 0x6b, 0xc4, 0xf4, 0x21, 0x95, 0xc2, 0xf7, 0x9a, 0x13, 0xc1, 0x1a, 0x7b,
187     0x8f, 0x77, 0xda, 0x19, 0x48, 0xbb, 0x6d, 0x14, 0x5d, 0xba, 0x65, 0xb4, 0x9e,
188     0x43, 0x42, 0x58, 0x98, 0x0b, 0x91, 0x46, 0xd8, 0x4c, 0xf3, 0x4c, 0xaf, 0x2e,
189     0x02, 0xa6, 0xb2, 0x49, 0x12, 0x62, 0x43, 0x4e, 0xa8, 0xac, 0xbf, 0xfd, 0xfa,
190     0x37, 0x24, 0xea, 0x69, 0x1c, 0xf5, 0xae, 0xfa, 0x08, 0x82, 0x30, 0xc3, 0xc0,
191     0xf8, 0x9a, 0x89, 0x33, 0xe1, 0x40, 0x6d, 0x18, 0x5c, 0x7b, 0x90, 0x48, 0xbf,
192     0x37, 0xdb, 0xea, 0xfb, 0x0e, 0xd4, 0x2e, 0x11, 0xfa, 0xa9, 0x86, 0xff, 0x00,
193     0x0b, 0x7b, 0xca, 0x09, 0x64, 0x6a, 0x8f, 0x0c, 0x0e, 0x09, 0x14, 0x36, 0x4a,
194     0x74, 0x31, 0x18, 0x5b, 0x18, 0xeb, 0xea, 0x83, 0xc3, 0x66, 0x68, 0xa6, 0x7d,
195     0x43, 0x06, 0x0f, 0x99, 0x60, 0xce, 0x65, 0x08, 0xf6}};
196
197 #endif // SHA1
198
199 #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
200
201 const TPM2B_RSA_TEST_VALUE c_OaepKvt =
202     {RSA_TEST_KEY_SIZE,
203     {0x33, 0x20, 0x6e, 0x21, 0xc3, 0xf6, 0xcd, 0xf8, 0xd7, 0x5d, 0x9f, 0xe9, 0x05,
204     0x14, 0x8c, 0x7c, 0xbb, 0x69, 0x24, 0x9e, 0x52, 0x8f, 0xaf, 0x84, 0x73, 0x21,
205     0x2c, 0x85, 0xa5, 0x30, 0x4d, 0xb6, 0xb8, 0xfa, 0x15, 0x9b, 0xc7, 0x8f, 0xc9,
206     0x7a, 0x72, 0x4b, 0x85, 0xa4, 0x1c, 0xc5, 0xd8, 0xe4, 0x92, 0xb3, 0xec, 0xd9,
207     0xa8, 0xca, 0x5e, 0x74, 0x73, 0x89, 0x7f, 0xb4, 0xac, 0x7e, 0x68, 0x12, 0xb2,
208     0x53, 0x27, 0x4b, 0xbf, 0xd0, 0x71, 0x69, 0x46, 0x9f, 0xef, 0xf4, 0x70, 0x60,
209     0xf8, 0xd7, 0xae, 0xc7, 0x5a, 0x27, 0x38, 0x25, 0x2d, 0x25, 0xab, 0x96, 0x56,
210     0x66, 0x3a, 0x23, 0x40, 0xa8, 0xdb, 0xbc, 0x86, 0xe8, 0xf3, 0xd2, 0x58, 0x0b,
211     0x44, 0xfc, 0x94, 0x1e, 0xb7, 0x5d, 0xb4, 0x57, 0xb5, 0xf3, 0x56, 0xee, 0x9b,
212     0xcf, 0x97, 0x91, 0x29, 0x36, 0xe3, 0x06, 0x13, 0xa2, 0xea, 0xd6, 0xd6, 0x0b,
213     0x86, 0x0b, 0x1a, 0x27, 0xe6, 0x22, 0xc4, 0x7b, 0xff, 0xde, 0x0f, 0xbf, 0x79,
214     0xc8, 0x1b, 0xed, 0xf1, 0x27, 0x62, 0xb5, 0x8b, 0xf9, 0xd9, 0x76, 0x90, 0xf6,
215     0xcc, 0x83, 0x0f, 0xce, 0xce, 0x2e, 0x63, 0x7a, 0x9b, 0xf4, 0x48, 0x5b, 0xd7,
216     0x81, 0x2c, 0x3a, 0xdb, 0x59, 0x0d, 0x4d, 0x9e, 0x46, 0xe9, 0x9e, 0x92, 0x22,
217     0x27, 0x1c, 0xb0, 0x67, 0x8a, 0xe6, 0x8a, 0x16, 0x8a, 0xdf, 0x95, 0x76, 0x24,
218     0x82, 0xad, 0xf1, 0xbc, 0x97, 0xbf, 0xd3, 0x5e, 0x6e, 0x14, 0x0c, 0x5b, 0x25,
219     0xfe, 0x58, 0xfa, 0x64, 0xe5, 0x14, 0x46, 0xb7, 0x58, 0xc6, 0x3f, 0x7f, 0x42,
220     0xd2, 0x8e, 0x45, 0x13, 0x41, 0x85, 0x12, 0x2e, 0x96, 0x19, 0xd0, 0x5e, 0x7d,
221     0x34, 0x06, 0x32, 0x2b, 0xc8, 0xd9, 0x0d, 0x6c, 0x06, 0x36, 0xa0, 0xff, 0x47,
222     0x57, 0x2c, 0x25, 0xbc, 0x8a, 0xa5, 0xe2, 0xc7, 0xe3}};
223
224 const TPM2B_RSA_TEST_VALUE c_RsaesKvt =
225     {RSA_TEST_KEY_SIZE,
226     {0x39, 0xfc, 0x10, 0x5d, 0xf4, 0x45, 0x3d, 0x94, 0x53, 0x06, 0x89, 0x24, 0xe7,
227     0xe8, 0xfd, 0x03, 0xac, 0xfd, 0xbd, 0xb2, 0x28, 0xd3, 0x4a, 0x52, 0xc5, 0xd4,
228     0xdb, 0x17, 0xd4, 0x24, 0x05, 0xc4, 0xeb, 0x6a, 0xce, 0x1d, 0xbb, 0x37, 0xcb,
229     0x09, 0xd8, 0x6c, 0x83, 0x19, 0x93, 0xd4, 0xe2, 0x88, 0x88, 0x9b, 0xaf, 0x92,
230     0x16, 0xc4, 0x15, 0xbd, 0x49, 0x13, 0x22, 0xb7, 0x84, 0xcf, 0x23, 0xf2, 0x6f,
231     0x0c, 0x3e, 0x8f, 0xde, 0x04, 0x09, 0x31, 0x2d, 0x99, 0xdf, 0xe6, 0x74, 0x70,
232     0x30, 0xde, 0x8c, 0xad, 0x32, 0x86, 0xe2, 0x7c, 0x12, 0x90, 0x21, 0xf3, 0x86,
233     0xb7, 0xe2, 0x64, 0xca, 0x98, 0xcc, 0x64, 0x4b, 0xef, 0x57, 0x4f, 0x5a, 0x16,
234     0x6e, 0xd7, 0x2f, 0x5b, 0xf6, 0x07, 0xad, 0x33, 0xb4, 0x8f, 0x3b, 0x3a, 0x8b,
235     0xd9, 0x06, 0x2b, 0xed, 0x3c, 0x3c, 0x76, 0xf6, 0x21, 0x31, 0xe3, 0xfb, 0x2c,
236     0x45, 0x61, 0x42, 0xba, 0xe0, 0xc3, 0x72, 0x63, 0xd0, 0x6b, 0x8f, 0x36, 0x26,
237     0xfb, 0x9e, 0x89, 0x0e, 0x44, 0x9a, 0xc1, 0x84, 0x5e, 0x84, 0x8d, 0xb6, 0xea,
238     0xf1, 0x0d, 0x66, 0xc7, 0xdb, 0x44, 0xbd, 0x19, 0x7c, 0x05, 0xbe, 0xc4, 0xab,
239     0x88, 0x32, 0xbe, 0xc7, 0x63, 0x31, 0xe6, 0x38, 0xd4, 0xe5, 0xb8, 0x4b, 0xf5,
240     0x0e, 0x55, 0x9a, 0x3a, 0xe6, 0x0a, 0xec, 0xee, 0xe2, 0xa8, 0x88, 0x04, 0xf2,
241     0xb8, 0xaa, 0x5a, 0xd8, 0x97, 0x5d, 0xa0, 0xa8, 0x42, 0xfb, 0xd9, 0xde, 0x80,
242     0xae, 0x4c, 0xb3, 0xa1, 0x90, 0x47, 0x57, 0x03, 0x10, 0x78, 0xa6, 0x8f, 0x11,
243     0xba, 0x4b, 0xce, 0x2d, 0x56, 0xa4, 0xe1, 0xbd, 0xf8, 0xa0, 0xa4, 0xd5, 0x48,
244     0x3c, 0x63, 0x20, 0x00, 0x38, 0xa0, 0xd1, 0xe6, 0x12, 0xe9, 0x1d, 0xd8, 0x49,
245     0xe3, 0xd5, 0x24, 0xb5, 0xc5, 0x3a, 0x1f, 0xb0, 0xd4}};
246
247 const TPM2B_RSA_TEST_VALUE c_RsapssKvt =
248     {RSA_TEST_KEY_SIZE,
249     {0x74, 0x89, 0x29, 0x3e, 0x1b, 0xac, 0xc6, 0x85, 0xca, 0xf0, 0x63, 0x43, 0x30,
250     0x7d, 0x1c, 0x9b, 0x2f, 0xbd, 0x4d, 0x69, 0x39, 0x5e, 0x85, 0xe2, 0xef, 0x86,

```



```

251     0x0a, 0xc6, 0x6b, 0xa6, 0x08, 0x19, 0x6c, 0x56, 0x38, 0x24, 0x55, 0x92, 0x84,
252     0x9b, 0x1b, 0x8b, 0x04, 0xcf, 0x24, 0x14, 0x24, 0x13, 0x0e, 0x8b, 0x82, 0x6f,
253     0x96, 0xc8, 0x9a, 0x68, 0xfc, 0x4c, 0x02, 0xf0, 0xdc, 0xcd, 0x36, 0x25, 0x31,
254     0xd5, 0x82, 0xcf, 0xc9, 0x69, 0x72, 0xf6, 0x1d, 0xab, 0x68, 0x20, 0x2e, 0x2d,
255     0x19, 0x49, 0xf0, 0x2e, 0xad, 0xd2, 0xda, 0xaf, 0xff, 0xb6, 0x92, 0x83, 0x5b,
256     0x8a, 0x06, 0x2d, 0x0c, 0x32, 0x11, 0x32, 0x3b, 0x77, 0x17, 0xf6, 0x50, 0xfb,
257     0xf8, 0x57, 0xc9, 0xc7, 0x9b, 0x9e, 0xc6, 0xd1, 0xa9, 0x55, 0xf0, 0x22, 0x35,
258     0xda, 0xca, 0x3c, 0x8e, 0xc6, 0x9a, 0xd8, 0x25, 0xc8, 0x5e, 0x93, 0x0d, 0xaa,
259     0xa7, 0x06, 0xaf, 0x11, 0x29, 0x99, 0xe7, 0x7c, 0xee, 0x49, 0x82, 0x30, 0xba,
260     0x2c, 0xe2, 0x40, 0x8f, 0x0a, 0xa6, 0x7b, 0x24, 0x75, 0xc5, 0xcd, 0x03, 0x12,
261     0xf4, 0xb2, 0x4b, 0x3a, 0xd1, 0x91, 0x3c, 0x20, 0x0e, 0x58, 0x2b, 0x31, 0xf8,
262     0x8b, 0xee, 0xbc, 0x1f, 0x95, 0x35, 0x58, 0x6a, 0x73, 0xee, 0x99, 0xb0, 0x01,
263     0x42, 0x4f, 0x66, 0xc0, 0x66, 0xbb, 0x35, 0x86, 0xeb, 0xd9, 0x7b, 0x55, 0x77,
264     0x2d, 0x54, 0x78, 0x19, 0x49, 0xe8, 0xcc, 0xfd, 0xb1, 0xcb, 0x49, 0xc9, 0xea,
265     0x20, 0xab, 0xed, 0xb5, 0xed, 0xfe, 0xb2, 0xb5, 0xa8, 0xcf, 0x05, 0x06, 0xd5,
266     0x7d, 0x2b, 0xbb, 0x0b, 0x65, 0x6b, 0x2b, 0x6d, 0x55, 0x95, 0x85, 0x44, 0x8b,
267     0x12, 0x05, 0xf3, 0x4b, 0xd4, 0x8e, 0x3d, 0x68, 0x2d, 0x29, 0x9c, 0x05, 0x79,
268     0xd6, 0xfc, 0x72, 0x90, 0x6a, 0xab, 0x46, 0x38, 0x81}};
269
270 const TPM2B_RSA_TEST_VALUE c_RsassaKvt =
271 {RSA_TEST_KEY_SIZE,
272  {0x8a, 0xb1, 0x0a, 0xb5, 0xe4, 0x02, 0xf7, 0xdd, 0x45, 0x2a, 0xcc, 0x2b, 0x6b,
273   0x8c, 0x0e, 0x9a, 0x92, 0x4f, 0x9b, 0xc5, 0xe4, 0x8b, 0x82, 0xb9, 0xb0, 0xd9,
274   0x87, 0x8c, 0xcb, 0xf0, 0xb0, 0x59, 0xa5, 0x92, 0x21, 0xa0, 0xa7, 0x61, 0x5c,
275   0xed, 0xa8, 0x6e, 0x22, 0x29, 0x46, 0xc7, 0x86, 0x37, 0x4b, 0x1b, 0x1e, 0x94,
276   0x93, 0xc8, 0x4c, 0x17, 0x7a, 0xae, 0x59, 0x91, 0xf8, 0x83, 0x84, 0xc4, 0x8c,
277   0x38, 0xc2, 0x35, 0x0e, 0x7e, 0x50, 0x67, 0x76, 0xe7, 0xd3, 0xec, 0x6f, 0x0d,
278   0xa0, 0x5c, 0x2f, 0x0a, 0x80, 0x28, 0xd3, 0xc5, 0x7d, 0x2d, 0x1a, 0x0b, 0x96,
279   0xd6, 0xe5, 0x98, 0x05, 0x8c, 0x4d, 0xa0, 0x1f, 0x8c, 0xb6, 0xfb, 0xb1, 0xcf,
280   0xe9, 0xcb, 0x38, 0x27, 0x60, 0x64, 0x17, 0xca, 0xf4, 0x8b, 0x61, 0xb7, 0x1d,
281   0xb6, 0x20, 0x9d, 0x40, 0x2a, 0x1c, 0xfd, 0x55, 0x40, 0x4b, 0x95, 0x39, 0x52,
282   0x18, 0x3b, 0xab, 0x44, 0xe8, 0x83, 0x4b, 0x7c, 0x47, 0xfb, 0xed, 0x06, 0x9c,
283   0xcd, 0x4f, 0xba, 0x81, 0xd6, 0xb7, 0x31, 0xcf, 0x5c, 0x23, 0xf8, 0x25, 0xab,
284   0x95, 0x77, 0x0a, 0x8f, 0x46, 0xef, 0xfb, 0x59, 0xb8, 0x04, 0xd7, 0x1e, 0xf5,
285   0xaf, 0x6a, 0x1a, 0x26, 0x9b, 0xae, 0xf4, 0xf5, 0x7f, 0x84, 0x6f, 0x3c, 0xed,
286   0xf8, 0x24, 0x0b, 0x43, 0xd1, 0xba, 0x74, 0x89, 0x4e, 0x39, 0xfe, 0xab, 0xa5,
287   0x16, 0xa5, 0x28, 0xee, 0x96, 0x84, 0x3e, 0x16, 0x6d, 0x5f, 0x4e, 0x0b, 0x7d,
288   0x94, 0x16, 0x1b, 0x8c, 0xf9, 0xaa, 0x9b, 0xc0, 0x49, 0x02, 0x4c, 0x3e, 0x62,
289   0xff, 0xfe, 0xa2, 0x20, 0x33, 0x5e, 0xa6, 0xdd, 0xda, 0x15, 0x2d, 0xb7, 0xcd,
290   0xda, 0xff, 0xb1, 0x0b, 0x45, 0x7b, 0xd3, 0xa0, 0x42, 0x29, 0xab, 0xa9, 0x73,
291   0xe9, 0xa4, 0xd9, 0x8d, 0xac, 0xa1, 0x88, 0x2c, 0x2d}};
292
293 #endif // SHA256
294
295 #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
296
297 const TPM2B_RSA_TEST_VALUE c_OaepKvt =
298 {RSA_TEST_KEY_SIZE,
299  {0x0f, 0x3c, 0x42, 0x4d, 0x8c, 0x91, 0x96, 0x05, 0x3c, 0xfd, 0x59, 0x3b, 0x7f,
300   0x29, 0xbc, 0x03, 0x67, 0xc1, 0xff, 0x74, 0xe7, 0x09, 0xf4, 0x13, 0x45, 0xbe,
301   0x13, 0x1d, 0xc9, 0x86, 0x94, 0xfe, 0xed, 0xa6, 0xe8, 0x3a, 0xcb, 0x89, 0x4d,
302   0xec, 0x86, 0x63, 0x4c, 0xdb, 0xf1, 0x95, 0xee, 0xc1, 0x46, 0xc5, 0x3b, 0xd8,
303   0xf8, 0xa2, 0x41, 0x6a, 0x60, 0x8b, 0x9e, 0x5e, 0x7f, 0x20, 0x16, 0xe3, 0x69,
304   0xb6, 0x2d, 0x92, 0xfc, 0x60, 0xa2, 0x74, 0x88, 0xd5, 0xc7, 0xa6, 0xd1, 0xff,
305   0xe3, 0x45, 0x02, 0x51, 0x39, 0xd9, 0xf3, 0x56, 0x0b, 0x91, 0x80, 0xe0, 0x6c,
306   0xa8, 0xc3, 0x78, 0xef, 0x34, 0x22, 0x8c, 0xf5, 0xfb, 0x47, 0x98, 0x5d, 0x57,
307   0x8e, 0x3a, 0xb9, 0xff, 0x92, 0x04, 0xc7, 0xc2, 0x6e, 0xfa, 0x14, 0xc1, 0xb9,
308   0x68, 0x15, 0x5c, 0x12, 0xe8, 0xa8, 0xbe, 0xea, 0xe8, 0x8d, 0x9b, 0x48, 0x28,
309   0x35, 0xdb, 0x4b, 0x52, 0xc1, 0x2d, 0x85, 0x47, 0x83, 0xd0, 0xe9, 0xae, 0x90,
310   0x6e, 0x65, 0xd4, 0x34, 0x7f, 0x81, 0xce, 0x69, 0xf0, 0x96, 0x62, 0xf7, 0xec,
311   0x41, 0xd5, 0xc2, 0xe3, 0x4b, 0xba, 0x9c, 0x8a, 0x02, 0xce, 0xf0, 0x5d, 0x14,
312   0xf7, 0x09, 0x42, 0x8e, 0x4a, 0x27, 0xfe, 0x3e, 0x66, 0x42, 0x99, 0x03, 0xe1,
313   0x69, 0xbd, 0xdb, 0x7f, 0x9b, 0x70, 0xeb, 0x4e, 0x9c, 0xac, 0x45, 0x67, 0x91,
314   0x9f, 0x75, 0x10, 0xc6, 0xfc, 0x14, 0xe1, 0x28, 0xc1, 0x0e, 0xe0, 0x7e, 0xc0,
315   0x5c, 0x1d, 0xee, 0xe8, 0xff, 0x45, 0x79, 0x51, 0x86, 0x08, 0xe6, 0x39, 0xac,
316   0xb5, 0xfd, 0xb8, 0xf1, 0xdd, 0x2e, 0xf4, 0xb2, 0x1a, 0x69, 0x0d, 0xd9, 0x98,

```

```

317         0x8e, 0xdb, 0x85, 0x61, 0x70, 0x20, 0x82, 0x91, 0x26, 0x87, 0x80, 0xc4, 0x6a,
318         0xd8, 0x3b, 0x91, 0x4d, 0xd3, 0x33, 0x84, 0xad, 0xb7}}};
319
320 const TPM2B_RSA_TEST_VALUE c_RsaesKvt =
321     {RSA_TEST_KEY_SIZE,
322     {0x44, 0xd5, 0x9f, 0xbc, 0x48, 0x03, 0x3d, 0x9f, 0x22, 0x91, 0x2a, 0xab, 0x3c,
323     0x31, 0x71, 0xab, 0x86, 0x3f, 0x0f, 0x6f, 0x59, 0x5b, 0x93, 0x27, 0xbc, 0xbc,
324     0xcd, 0x29, 0x38, 0x43, 0x2a, 0x3b, 0x3b, 0xd2, 0xb3, 0x45, 0x40, 0xba, 0x15,
325     0xb4, 0x45, 0xe3, 0x56, 0xab, 0xff, 0xb3, 0x20, 0x26, 0x39, 0xcc, 0x48, 0xc5,
326     0x5d, 0x41, 0x0d, 0x2f, 0x57, 0x7f, 0x9d, 0x16, 0x2e, 0x26, 0x57, 0xc7, 0x6b,
327     0xf3, 0x36, 0x54, 0xbd, 0xb6, 0x1d, 0x46, 0x4e, 0x13, 0x50, 0xd7, 0x61, 0x9d,
328     0x8d, 0x7b, 0xeb, 0x21, 0x9f, 0x79, 0xf3, 0xfd, 0xe0, 0x1b, 0xa8, 0xed, 0x6d,
329     0x29, 0x33, 0x0d, 0x65, 0x94, 0x24, 0x1e, 0x62, 0x88, 0x6b, 0x2b, 0x4e, 0x39,
330     0xf5, 0x80, 0x39, 0xca, 0x76, 0x95, 0xbc, 0x7c, 0x27, 0x1d, 0xdd, 0x3a, 0x11,
331     0xf1, 0x3e, 0x54, 0x03, 0xb7, 0x43, 0x91, 0x99, 0x33, 0xfe, 0x9d, 0x14, 0x2c,
332     0x87, 0x9a, 0x95, 0x18, 0x1f, 0x02, 0x04, 0x6a, 0xe2, 0xb7, 0x81, 0x14, 0x13,
333     0x45, 0x16, 0xfb, 0xe4, 0xb7, 0x8f, 0xab, 0x2b, 0xd7, 0x60, 0x34, 0x8a, 0x55,
334     0xbc, 0x01, 0x8c, 0x49, 0x02, 0x29, 0xf1, 0x9c, 0x94, 0x98, 0x44, 0xd0, 0x94,
335     0xcb, 0xd4, 0x85, 0x4c, 0x3b, 0x77, 0x72, 0x99, 0xd5, 0x4b, 0xc6, 0x3b, 0xe4,
336     0xd2, 0xc8, 0xe9, 0x6a, 0x23, 0x18, 0x3b, 0x3b, 0x5e, 0x32, 0xec, 0x70, 0x84,
337     0x5d, 0xbb, 0x6a, 0x8f, 0x0c, 0x5f, 0x55, 0xa5, 0x30, 0x34, 0x48, 0xbb, 0xc2,
338     0xdf, 0x12, 0xb9, 0x81, 0xad, 0x36, 0x3f, 0xf0, 0x24, 0x16, 0x48, 0x04, 0x4a,
339     0x7f, 0xfd, 0x9f, 0x4c, 0xea, 0xfe, 0x1d, 0x83, 0xd0, 0x81, 0xad, 0x25, 0x6c,
340     0x5f, 0x45, 0x36, 0x91, 0xf0, 0xd5, 0x8b, 0x53, 0x0a, 0xdf, 0xec, 0x9f, 0x04,
341     0x58, 0xc4, 0x35, 0xa0, 0x78, 0x1f, 0x68, 0xe0, 0x22}}};
342
343 const TPM2B_RSA_TEST_VALUE c_RsapssKvt =
344     {RSA_TEST_KEY_SIZE,
345     {0x3f, 0x3a, 0x82, 0x6d, 0x42, 0xe3, 0x8b, 0x4f, 0x45, 0x9c, 0xda, 0x6c, 0xbe,
346     0xbe, 0xcd, 0x00, 0x98, 0xfb, 0xbe, 0x59, 0x30, 0xc6, 0x3c, 0xaa, 0xb3, 0x06,
347     0x27, 0xb5, 0xda, 0xfa, 0xb2, 0xc3, 0x43, 0xb7, 0xbd, 0xe9, 0xd3, 0x23, 0xed,
348     0x80, 0xce, 0x74, 0xb3, 0xb8, 0x77, 0x8d, 0xe6, 0x8d, 0x3c, 0xe5, 0xf5, 0xd7,
349     0x80, 0xcf, 0x38, 0x55, 0x76, 0xd7, 0x87, 0xa8, 0xd6, 0x3a, 0xcf, 0xfd, 0xd8,
350     0x91, 0x65, 0xab, 0x43, 0x66, 0x50, 0xb7, 0x9a, 0x13, 0x6b, 0x45, 0x80, 0x76,
351     0x86, 0x22, 0x27, 0x72, 0xf7, 0xbb, 0x65, 0x22, 0x5c, 0x55, 0x60, 0xd8, 0x84,
352     0x9f, 0xf2, 0x61, 0x52, 0xac, 0xf2, 0x4f, 0x5b, 0x7b, 0x21, 0xe1, 0xf5, 0x4b,
353     0x8f, 0x01, 0xf2, 0x4b, 0xcf, 0xd3, 0xfb, 0x74, 0x5e, 0x6e, 0x96, 0xb4, 0xa8,
354     0x0f, 0x01, 0x9b, 0x26, 0x54, 0x0a, 0x70, 0x55, 0x26, 0xb7, 0x0b, 0xe8, 0x01,
355     0x68, 0x66, 0x0d, 0x6f, 0xb5, 0xfc, 0x66, 0xbd, 0x9e, 0x44, 0xed, 0x6a, 0x1e,
356     0x3c, 0x3b, 0x61, 0x5d, 0xe8, 0xdb, 0x99, 0x5b, 0x67, 0xbf, 0x94, 0xfb, 0xe6,
357     0x8c, 0x4b, 0x07, 0xcb, 0x43, 0x3a, 0x0d, 0xb1, 0x1b, 0x10, 0x66, 0x81, 0xe2,
358     0x0d, 0xe7, 0xd1, 0xca, 0x85, 0xa7, 0x50, 0x82, 0x2d, 0xbf, 0xed, 0xcf, 0x43,
359     0x6d, 0xdb, 0x2c, 0x7b, 0x73, 0x20, 0xfe, 0x73, 0x3f, 0x19, 0xc6, 0xdb, 0x69,
360     0xb8, 0xc3, 0xd3, 0xf4, 0xe5, 0x64, 0xf8, 0x36, 0x8e, 0xd5, 0xd8, 0x09, 0x2a,
361     0x5f, 0x26, 0x70, 0xa1, 0xd9, 0x5b, 0x14, 0xf8, 0x22, 0xe9, 0x9d, 0x22, 0x51,
362     0xf4, 0x52, 0xc1, 0x6f, 0x53, 0xf5, 0xca, 0x0d, 0xda, 0x39, 0x8c, 0x29, 0x42,
363     0xe8, 0x58, 0x89, 0xbb, 0xd1, 0x2e, 0xc5, 0xdb, 0x86, 0x8d, 0xaf, 0xec, 0x58,
364     0x36, 0x8d, 0x8d, 0x57, 0x23, 0xd5, 0xdd, 0xb9, 0x24}}};
365
366 const TPM2B_RSA_TEST_VALUE c_RsassaKvt =
367     {RSA_TEST_KEY_SIZE,
368     {0x39, 0x10, 0x58, 0x7d, 0x6d, 0xa8, 0xd5, 0x90, 0x07, 0xd6, 0x2b, 0x13, 0xe9,
369     0xd8, 0x93, 0x7e, 0xf3, 0x5d, 0x71, 0xe0, 0xf0, 0x33, 0x3a, 0x4a, 0x22, 0xf3,
370     0xe6, 0x95, 0xd3, 0x8e, 0x8c, 0x41, 0xe7, 0xb3, 0x13, 0xde, 0x4a, 0x45, 0xd3,
371     0xd1, 0xfb, 0xb1, 0x3f, 0x9b, 0x39, 0xa5, 0x50, 0x58, 0xef, 0xb6, 0x3a, 0x43,
372     0xdd, 0x54, 0xab, 0xda, 0x9d, 0x32, 0x49, 0xe4, 0x57, 0x96, 0xe5, 0x1b, 0x1d,
373     0x8f, 0x33, 0x8e, 0x07, 0x67, 0x56, 0x14, 0xc1, 0x18, 0x78, 0xa2, 0x52, 0xe6,
374     0x2e, 0x07, 0x81, 0xbe, 0xd8, 0xca, 0x76, 0x63, 0x68, 0xc5, 0x47, 0xa2, 0x92,
375     0x5e, 0x4c, 0xfd, 0x14, 0xc7, 0x46, 0x14, 0xbe, 0xc7, 0x85, 0xef, 0xe6, 0xb8,
376     0x46, 0xcb, 0x3a, 0x67, 0x66, 0x89, 0xc6, 0xee, 0x9d, 0x64, 0xf5, 0x0d, 0x09,
377     0x80, 0x9a, 0x6f, 0x0e, 0xeb, 0xe4, 0xb9, 0xe9, 0xab, 0x90, 0x4f, 0xe7, 0x5a,
378     0xc8, 0xca, 0xf6, 0x16, 0x0a, 0x82, 0xbd, 0xb7, 0x76, 0x59, 0x08, 0x2d, 0xd9,
379     0x40, 0x5d, 0xaa, 0xa5, 0xef, 0xfb, 0xe3, 0x81, 0x2c, 0x2c, 0x5c, 0xa8, 0x16,
380     0xbd, 0x63, 0x20, 0xc2, 0x4d, 0x3b, 0x51, 0xaa, 0x62, 0x1f, 0x06, 0xe5, 0xbb,
381     0x78, 0x44, 0x04, 0x0c, 0x5c, 0xe1, 0x1b, 0x6b, 0x9d, 0x21, 0x10, 0xaf, 0x48,
382     0x48, 0x98, 0x97, 0x77, 0xc2, 0x73, 0xb4, 0x98, 0x64, 0xcc, 0x94, 0x2c, 0x29,

```

```

383     0x28, 0x45, 0x36, 0xd1, 0xc5, 0xd0, 0x2f, 0x97, 0x27, 0x92, 0x65, 0x22, 0xbb,
384     0x63, 0x79, 0xea, 0xf5, 0xff, 0x77, 0x0f, 0x4b, 0x56, 0x8a, 0x9f, 0xad, 0x1a,
385     0x97, 0x67, 0x39, 0x69, 0xb8, 0x4c, 0x6c, 0xc2, 0x56, 0xc5, 0x7a, 0xa8, 0x14,
386     0x5a, 0x24, 0x7a, 0xa4, 0x6e, 0x55, 0xb2, 0x86, 0x1d, 0xf4, 0x62, 0x5a, 0x2d,
387     0x87, 0x6d, 0xde, 0x99, 0x78, 0x2d, 0xef, 0xd7, 0xdc}};
388
389 #endif // SHA384
390
391 #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
392
393 const TPM2B_RSA_TEST_VALUE c_OaepKvt =
394     {RSA_TEST_KEY_SIZE,
395      {0x48, 0x45, 0xa7, 0x70, 0xb2, 0x41, 0xb7, 0x48, 0x5e, 0x79, 0x8c, 0xdf, 0x1c,
396       0xc6, 0x7e, 0xbb, 0x11, 0x80, 0x82, 0x52, 0xbf, 0x40, 0x3d, 0x90, 0x03, 0x6e,
397       0x20, 0x3a, 0xb9, 0x65, 0xc8, 0x51, 0x4c, 0xbd, 0x9c, 0xa9, 0x43, 0x89, 0xd0,
398       0x57, 0x0c, 0xa3, 0x69, 0x22, 0x7e, 0x82, 0x2a, 0x1c, 0x1d, 0x5a, 0x80, 0x84,
399       0x81, 0xbb, 0x5e, 0x5e, 0xd0, 0xc1, 0x66, 0x9a, 0xac, 0x00, 0xba, 0x14, 0xa2,
400       0xe9, 0xd0, 0x3a, 0x89, 0x5a, 0x63, 0xe2, 0xec, 0x92, 0x05, 0xf4, 0x47, 0x66,
401       0x12, 0x7f, 0xdb, 0xa7, 0x3c, 0x5b, 0x67, 0xe1, 0x55, 0xca, 0x0a, 0x27, 0xbf,
402       0x39, 0x89, 0x11, 0x05, 0xba, 0x9b, 0x5a, 0x9b, 0x65, 0x44, 0xad, 0x78, 0xcf,
403       0x8f, 0x94, 0xf6, 0x9a, 0xb4, 0x52, 0x39, 0x0e, 0x00, 0xba, 0xbc, 0xe0, 0xbd,
404       0x6f, 0x81, 0x2d, 0x76, 0x42, 0x66, 0x70, 0x07, 0x77, 0xbf, 0x09, 0x88, 0x2a,
405       0x0c, 0xb1, 0x56, 0x3e, 0xee, 0xfd, 0xdc, 0xb6, 0x3c, 0x0d, 0xc5, 0xa4, 0x0d,
406       0x10, 0x32, 0x80, 0x3e, 0x1e, 0xfe, 0x36, 0x8f, 0xb5, 0x42, 0xc1, 0x21, 0x7b,
407       0xdf, 0xdf, 0x4a, 0xd2, 0x68, 0x0c, 0x01, 0x9f, 0x4a, 0xfd, 0xd4, 0xec, 0xf7,
408       0x49, 0x06, 0xab, 0xed, 0xc6, 0xd5, 0x1b, 0x63, 0x76, 0x38, 0xc8, 0x6c, 0xc7,
409       0x4f, 0xcb, 0x29, 0x8a, 0x0e, 0x6f, 0x33, 0xaf, 0x69, 0x31, 0x8e, 0xa7, 0xdd,
410       0x9a, 0x36, 0xde, 0x9b, 0xf1, 0x0b, 0xfb, 0x20, 0xa0, 0x6d, 0x33, 0x31, 0xc9,
411       0x9e, 0xb4, 0x2e, 0xc5, 0x40, 0x0e, 0x60, 0x71, 0x36, 0x75, 0x05, 0xf9, 0x37,
412       0xe0, 0xca, 0x8e, 0x8f, 0x56, 0xe0, 0xea, 0x9b, 0xeb, 0x17, 0xf3, 0xca, 0x40,
413       0xc3, 0x48, 0x01, 0xba, 0xdc, 0xc6, 0x4b, 0x2b, 0x5b, 0x7b, 0x5c, 0x81, 0xa6,
414       0xbb, 0xc7, 0x43, 0xc0, 0xbe, 0xc0, 0x30, 0x7b, 0x55}};
415
416 const TPM2B_RSA_TEST_VALUE c_RsaesKvt =
417     {RSA_TEST_KEY_SIZE,
418      {0x74, 0x83, 0xfa, 0x52, 0x65, 0x50, 0x68, 0xd0, 0x82, 0x05, 0x72, 0x70, 0x78,
419       0x1c, 0xac, 0x10, 0x23, 0xc5, 0x07, 0xf8, 0x93, 0xd2, 0xeb, 0x65, 0x87, 0xbb,
420       0x47, 0xc2, 0xfb, 0x30, 0x9e, 0x61, 0x4c, 0xac, 0x04, 0x57, 0x5a, 0x7c, 0xeb,
421       0x29, 0x08, 0x84, 0x86, 0x89, 0x1e, 0x8f, 0x07, 0x32, 0xa3, 0x8b, 0x70, 0xe7,
422       0xa2, 0x9f, 0x9c, 0x42, 0x71, 0x3d, 0x23, 0x59, 0x82, 0x5e, 0x8a, 0xde, 0xd6,
423       0xfb, 0xd8, 0xc5, 0x8b, 0xc0, 0xdb, 0x10, 0x38, 0x87, 0xd3, 0xbf, 0x04, 0xb0,
424       0x66, 0xb9, 0x85, 0x81, 0x54, 0x4c, 0x69, 0xdc, 0xba, 0x78, 0xf3, 0x4a, 0xdb,
425       0x25, 0xa2, 0xf2, 0x34, 0x55, 0xdd, 0xaa, 0xa5, 0xc4, 0xed, 0x55, 0x06, 0x0e,
426       0x2a, 0x30, 0x77, 0xab, 0x82, 0x79, 0xf0, 0xcd, 0x9d, 0x6f, 0x09, 0xa0, 0xc8,
427       0x82, 0xc9, 0xe0, 0x61, 0xda, 0x40, 0xcd, 0x17, 0x59, 0xc0, 0xef, 0x95, 0x6d,
428       0xa3, 0x6d, 0x1c, 0x2b, 0xee, 0x24, 0xef, 0xd8, 0x4a, 0x55, 0x6c, 0xd6, 0x26,
429       0x42, 0x32, 0x17, 0xfd, 0x6a, 0xb3, 0x4f, 0xde, 0x07, 0x2f, 0x10, 0xd4, 0xac,
430       0x14, 0xea, 0x89, 0x68, 0xcc, 0xd3, 0x07, 0xb7, 0xcf, 0xba, 0x39, 0x20, 0x63,
431       0x20, 0x7b, 0x44, 0x8b, 0x48, 0x60, 0x5d, 0x3a, 0x2a, 0x0a, 0xe9, 0x68, 0xab,
432       0x15, 0x46, 0x27, 0x64, 0xb5, 0x82, 0x06, 0x29, 0xe7, 0x25, 0xca, 0x46, 0x48,
433       0x6e, 0x2a, 0x34, 0x57, 0x4b, 0x81, 0x75, 0xae, 0xb6, 0xfd, 0x6f, 0x51, 0x5f,
434       0x04, 0x59, 0xc7, 0x15, 0x1f, 0xe0, 0x68, 0xf7, 0x36, 0x2d, 0xdf, 0xc8, 0x9d,
435       0x05, 0x27, 0x2d, 0x3f, 0x2b, 0x59, 0x5d, 0xcb, 0xf3, 0xc4, 0x92, 0x6e, 0x00,
436       0xa8, 0x8d, 0xd0, 0x69, 0xe5, 0x59, 0xda, 0xba, 0x4f, 0x38, 0xf5, 0xa0, 0x8b,
437       0xf1, 0x73, 0xe9, 0x0d, 0xee, 0x64, 0xe5, 0xa2, 0xd8}};
438
439 const TPM2B_RSA_TEST_VALUE c_RsapssKvt =
440     {RSA_TEST_KEY_SIZE,
441      {0x1b, 0xca, 0x8b, 0x18, 0x15, 0x3b, 0x95, 0x5b, 0x0a, 0x89, 0x10, 0x03, 0x7f,
442       0x7c, 0xa0, 0xc9, 0x66, 0x57, 0x86, 0x6a, 0xc9, 0xeb, 0x82, 0x71, 0xf3, 0x8d,
443       0x6f, 0xa9, 0xa4, 0x2d, 0xd0, 0x22, 0xdf, 0xe9, 0xc6, 0x71, 0x5b, 0xf4, 0x27,
444       0x38, 0x5b, 0x2c, 0x8a, 0x54, 0xcc, 0x85, 0x11, 0x69, 0x6d, 0x6f, 0x42, 0xe7,
445       0x22, 0xcb, 0xd6, 0xad, 0x1a, 0xc5, 0xab, 0x6a, 0xa5, 0xfc, 0xa5, 0x70, 0x72,
446       0x4a, 0x62, 0x25, 0xd0, 0xa2, 0x16, 0x61, 0xab, 0xac, 0x31, 0xa0, 0x46, 0x24,
447       0x4f, 0xdd, 0x9a, 0x36, 0x55, 0xb6, 0x00, 0x9e, 0x23, 0x50, 0x0d, 0x53, 0x01,
448       0xb3, 0x46, 0x56, 0xb2, 0x1d, 0x33, 0x5b, 0xca, 0x41, 0x7f, 0x65, 0x7e, 0x00,

```

```

449     0x5c, 0x12, 0xff, 0x0a, 0x70, 0x5d, 0x8c, 0x69, 0x4a, 0x02, 0xee, 0x72, 0x30,
450     0xa7, 0x5c, 0xa4, 0xbb, 0xbe, 0x03, 0x0c, 0xe4, 0x5f, 0x33, 0xb6, 0x78, 0x91,
451     0x9d, 0xd8, 0xec, 0x34, 0x03, 0x2e, 0x63, 0x32, 0xc7, 0x2a, 0x36, 0x50, 0xd5,
452     0x8b, 0x0e, 0x7f, 0x54, 0x4e, 0xf4, 0x29, 0x11, 0x1b, 0xcd, 0x0f, 0x37, 0xa5,
453     0xbc, 0x61, 0x83, 0x50, 0xfa, 0x18, 0x75, 0xd9, 0xfe, 0xa7, 0xe8, 0x9b, 0xc1,
454     0x4f, 0x96, 0x37, 0x81, 0x71, 0xdf, 0x71, 0x8b, 0x89, 0x81, 0xf4, 0x95, 0xb5,
455     0x29, 0x66, 0x41, 0x0c, 0x73, 0xd7, 0x0b, 0x21, 0xb4, 0xfb, 0xf9, 0x63, 0x2f,
456     0xe9, 0x7b, 0x38, 0xaa, 0x20, 0xc3, 0x96, 0xcc, 0xb7, 0xb2, 0x24, 0xa1, 0xe0,
457     0x59, 0x9c, 0x10, 0x9e, 0x5a, 0xf7, 0xe3, 0x02, 0xe6, 0x23, 0xe2, 0x44, 0x21,
458     0x3f, 0x6e, 0x5e, 0x79, 0xb2, 0x93, 0x7d, 0xce, 0xed, 0xe2, 0xe1, 0xab, 0x98,
459     0x07, 0xa7, 0xbd, 0xbc, 0xd8, 0xf7, 0x06, 0xeb, 0xc5, 0xa6, 0x37, 0x18, 0x11,
460     0x88, 0xf7, 0x63, 0x39, 0xb9, 0x57, 0x29, 0xdc, 0x03}};
461
462 const TPM2B_RSA_TEST_VALUE c_RsassaKvt =
463 {RSA_TEST_KEY_SIZE,
464  {0x05, 0x55, 0x00, 0x62, 0x01, 0xc6, 0x04, 0x31, 0x55, 0x73, 0x3f, 0x2a, 0xf9,
465   0xd4, 0x0f, 0xc1, 0x2b, 0xeb, 0xd8, 0xc8, 0xdb, 0xb2, 0xab, 0x6c, 0x26, 0xde,
466   0x2d, 0x89, 0xc2, 0x2d, 0x36, 0x62, 0xc8, 0x22, 0x5d, 0x58, 0x03, 0xb1, 0x46,
467   0x14, 0xa5, 0xd4, 0xbc, 0x25, 0x6b, 0x7f, 0x8f, 0x14, 0x7e, 0x03, 0x2f, 0x3d,
468   0xb8, 0x39, 0xa5, 0x79, 0x13, 0x7e, 0x22, 0x2a, 0xb9, 0x3e, 0x8f, 0xaa, 0x01,
469   0x7c, 0x03, 0x12, 0x21, 0x6c, 0x2a, 0xb4, 0x39, 0x98, 0x6d, 0xff, 0x08, 0x6c,
470   0x59, 0x2d, 0xdc, 0xc6, 0xf1, 0x77, 0x62, 0x10, 0xa6, 0xcc, 0xe2, 0x71, 0x8e,
471   0x97, 0x00, 0x87, 0x5b, 0x0e, 0x20, 0x00, 0x3f, 0x18, 0x63, 0x83, 0xf0, 0xe4,
472   0x0a, 0x64, 0x8c, 0xe9, 0x8c, 0x91, 0xe7, 0x89, 0x04, 0x64, 0x2c, 0x8b, 0x41,
473   0xc8, 0xac, 0xf6, 0x5a, 0x75, 0xe6, 0xa5, 0x76, 0x43, 0xcb, 0xa5, 0x33, 0x8b,
474   0x07, 0xc9, 0x73, 0x0f, 0x45, 0xa4, 0xc3, 0xac, 0xc1, 0xc3, 0xe6, 0xe7, 0x21,
475   0x66, 0x1c, 0xba, 0xbf, 0xea, 0x3e, 0x39, 0xfa, 0xb2, 0xe2, 0x8f, 0xfe, 0x9c,
476   0xb4, 0x85, 0x89, 0x33, 0x2a, 0x0c, 0xc8, 0x5d, 0x58, 0xe1, 0x89, 0x12, 0xe9,
477   0x4d, 0x42, 0xb3, 0x1f, 0x99, 0x0c, 0x3e, 0xd8, 0xb2, 0xeb, 0xf5, 0x88, 0xfb,
478   0xe1, 0x4b, 0x8e, 0xdc, 0xd3, 0xa8, 0xda, 0xbe, 0x04, 0x45, 0xbf, 0x56, 0xc6,
479   0x54, 0x70, 0x00, 0xb8, 0x66, 0x46, 0x3a, 0xa3, 0x1e, 0xb6, 0xeb, 0x1a, 0xa0,
480   0x0b, 0xd3, 0x9a, 0x9a, 0x52, 0xda, 0x60, 0x69, 0xb7, 0xef, 0x93, 0x47, 0x38,
481   0xab, 0x1a, 0xa0, 0x22, 0x6e, 0x76, 0x06, 0xb6, 0x74, 0xaf, 0x74, 0x8f, 0x51,
482   0xc0, 0x89, 0x5a, 0x4b, 0xbe, 0x6a, 0x91, 0x18, 0x25, 0x7d, 0xa6, 0x77, 0xe6,
483   0xfd, 0xc2, 0x62, 0x36, 0x07, 0xc6, 0xef, 0x79, 0xc9}};
484
485 #endif // SHA512

```

6.34 /tpm/include/private/SelfTest.h

```

1  /** Introduction
2  // This file contains the structure definitions for the self-test. It also contains
3  // macros for use when the self-test is implemented.
4  #ifndef SELF_TEST_H
5  #define SELF_TEST_H
6
7  /** Defines
8
9  // Was typing this a lot
10 #define SELF_TEST_FAILURE FAIL(FATAL_ERROR_SELF_TEST)
11
12 // Use the definition of key sizes to set algorithm values for key size.
13 #define AES_ENTRIES (AES_128 + AES_192 + AES_256)
14 #define SM4_ENTRIES (SM4_128)
15 #define CAMELLIA_ENTRIES (CAMELLIA_128 + CAMELLIA_192 + CAMELLIA_256)
16
17 #define NUM_SYMS (AES_ENTRIES + SM4_ENTRIES + CAMELLIA_ENTRIES)
18
19 typedef UINT32 SYM_INDEX;
20
21 // These two defines deal with the fact that the TPM_ALG_ID table does not delimit
22 // the symmetric mode values with a SYM_MODE_FIRST and SYM_MODE_LAST
23 #define SYM_MODE_FIRST ALG_CTR_VALUE
24 #define SYM_MODE_LAST ALG_ECB_VALUE
25

```

```

26 #define NUM_SYM_MODES (SYM_MODE_LAST - SYM_MODE_FIRST + 1)
27
28 // Define a type to hold a bit vector for the modes.
29 #if NUM_SYM_MODES <= 0
30 # error "No symmetric modes implemented"
31 #elif NUM_SYM_MODES <= 8
32 typedef BYTE SYM_MODES;
33 #elif NUM_SYM_MODES <= 16
34 typedef UINT16 SYM_MODES;
35 #elif NUM_SYM_MODES <= 32
36 typedef UINT32 SYM_MODES;
37 #else
38 # error "Too many symmetric modes"
39 #endif
40
41 typedef struct SYMMETRIC_TEST_VECTOR
42 {
43     const TPM_ALG_ID alg;           // the algorithm
44     const UINT16 keyBits;          // bits in the key
45     const BYTE* key;               // The test key
46     const UINT32 ivSize;           // block size of the algorithm
47     const UINT32 dataInOutSize;    // size to encrypt/decrypt
48     const BYTE* dataIn;            // data to encrypt
49     const BYTE* dataOut[NUM_SYM_MODES]; // data to decrypt
50 } SYMMETRIC_TEST_VECTOR;
51
52 #if ALG_SHA512
53 # define DEFAULT_TEST_HASH          ALG_SHA512_VALUE
54 # define DEFAULT_TEST_DIGEST_SIZE   SHA512_DIGEST_SIZE
55 # define DEFAULT_TEST_HASH_BLOCK_SIZE SHA512_BLOCK_SIZE
56 #elif ALG_SHA384
57 # define DEFAULT_TEST_HASH          ALG_SHA384_VALUE
58 # define DEFAULT_TEST_DIGEST_SIZE   SHA384_DIGEST_SIZE
59 # define DEFAULT_TEST_HASH_BLOCK_SIZE SHA384_BLOCK_SIZE
60 #elif ALG_SHA256
61 # define DEFAULT_TEST_HASH          ALG_SHA256_VALUE
62 # define DEFAULT_TEST_DIGEST_SIZE   SHA256_DIGEST_SIZE
63 # define DEFAULT_TEST_HASH_BLOCK_SIZE SHA256_BLOCK_SIZE
64 #elif ALG_SHA1
65 # define DEFAULT_TEST_HASH          ALG_SHA1_VALUE
66 # define DEFAULT_TEST_DIGEST_SIZE   SHA1_DIGEST_SIZE
67 # define DEFAULT_TEST_HASH_BLOCK_SIZE SHA1_BLOCK_SIZE
68 #endif
69
70 #endif // _SELF_TEST_H_

```

6.35 /tpm/include/private/SymmetricTest.h

```

1 /** Introduction
2
3 // This file contains the structures and data definitions for the symmetric tests.
4 // This file references the header file that contains the actual test vectors. This
5 // organization was chosen so that the program that is used to generate the test
6 // vector values does not have to also re-generate this data.
7 #ifndef SELF_TEST_DATA
8 # error "This file may only be included in AlgorithmTests.c"
9 #endif
10
11 #ifndef _SYMMETRIC_TEST_H
12 # define _SYMMETRIC_TEST_H
13 # include "SymmetricTestData.h"
14
15 /** Symmetric Test Structures
16
17 const SYMMETRIC_TEST_VECTOR c_symTestValues[NUM_SYMS + 1] = {

```



```

18 # if ALG_AES && AES_128
19     {TPM_ALG_AES,
20      128,
21      key_AES128,
22      16,
23      sizeof(dataIn_AES128),
24      dataIn_AES128,
25      {dataOut_AES128_CTR,
26       dataOut_AES128_OFB,
27       dataOut_AES128_CBC,
28       dataOut_AES128_CFB,
29       dataOut_AES128_ECB}},
30 # endif
31 # if ALG_AES && AES_192
32     {TPM_ALG_AES,
33      192,
34      key_AES192,
35      16,
36      sizeof(dataIn_AES192),
37      dataIn_AES192,
38      {dataOut_AES192_CTR,
39       dataOut_AES192_OFB,
40       dataOut_AES192_CBC,
41       dataOut_AES192_CFB,
42       dataOut_AES192_ECB}},
43 # endif
44 # if ALG_AES && AES_256
45     {TPM_ALG_AES,
46      256,
47      key_AES256,
48      16,
49      sizeof(dataIn_AES256),
50      dataIn_AES256,
51      {dataOut_AES256_CTR,
52       dataOut_AES256_OFB,
53       dataOut_AES256_CBC,
54       dataOut_AES256_CFB,
55       dataOut_AES256_ECB}},
56 # endif
57 // There are no SM4 test values yet so...
58 # if ALG_SM4 && SM4_128 && 0
59     {TPM_ALG_SM4,
60      128,
61      key_SM4128,
62      16,
63      sizeof(dataIn_SM4128),
64      dataIn_SM4128,
65      {dataOut_SM4128_CTR,
66       dataOut_SM4128_OFB,
67       dataOut_SM4128_CBC,
68       dataOut_SM4128_CFB,
69       dataOut_AES128_ECB}},
70 # endif
71     {0}};
72
73 #endif // _SYMMETRIC_TEST_H

```

6.36 /tpm/include/private/SymmetricTestData.h

```

1 // This is a vector for testing either encrypt or decrypt. The premise for decrypt
2 // is that the IV for decryption is the same as the IV for encryption. However,
3 // the ivOut value may be different for encryption and decryption. We will encrypt
4 // at least two blocks. This means that the chaining value will be used for each
5 // of the schemes (if any) and that implicitly checks that the chaining value
6 // is handled properly.

```



```

7
8  #if AES_128
9
10 const BYTE key_AES128[] = {0x2b,
11                             0x7e,
12                             0x15,
13                             0x16,
14                             0x28,
15                             0xae,
16                             0xd2,
17                             0xa6,
18                             0xab,
19                             0xf7,
20                             0x15,
21                             0x88,
22                             0x09,
23                             0xcf,
24                             0x4f,
25                             0x3c};
26
27 const BYTE dataIn_AES128[] = {0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
28                               0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
29                               0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
30                               0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
31
32 const BYTE dataOut_AES128_ECB[] = {0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60,
33                                     0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97,
34                                     0xf5, 0xd3, 0xd5, 0x85, 0x03, 0xb9, 0x69, 0x9d,
35                                     0xe7, 0x85, 0x89, 0x5a, 0x96, 0xfd, 0xba, 0xaf};
36
37 const BYTE dataOut_AES128_CBC[] = {0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46,
38                                     0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
39                                     0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee,
40                                     0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2};
41
42 const BYTE dataOut_AES128_CFB[] = {0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
43                                     0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
44                                     0xc8, 0xa6, 0x45, 0x37, 0xa0, 0xb3, 0xa9, 0x3f,
45                                     0xcd, 0xe3, 0xcd, 0xad, 0x9f, 0x1c, 0xe5, 0x8b};
46
47 const BYTE dataOut_AES128_OFB[] = {0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
48                                     0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
49                                     0x77, 0x89, 0x50, 0x8d, 0x16, 0x91, 0x8f, 0x03,
50                                     0xf5, 0x3c, 0x52, 0xda, 0xc5, 0x4e, 0xd8, 0x25};
51
52 const BYTE dataOut_AES128_CTR[] = {0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
53                                     0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
54                                     0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff,
55                                     0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd, 0xff};
56 #endif
57
58 #if AES_192
59
60 const BYTE key_AES192[] = {0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
61                             0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
62                             0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b};
63
64 const BYTE dataIn_AES192[] = {0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
65                               0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
66                               0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
67                               0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
68
69 const BYTE dataOut_AES192_ECB[] = {0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f,
70                                     0xf7, 0x12, 0xa2, 0x14, 0x57, 0x1f, 0xa5, 0xcc,
71                                     0x97, 0x41, 0x04, 0x84, 0x6d, 0x0a, 0xd3, 0xad,
72                                     0x77, 0x34, 0xec, 0xb3, 0xec, 0xee, 0x4e, 0xef};

```

```

73
74  const BYTE dataOut_AES192_CBC[] = {0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d,
75                                     0x71, 0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
76                                     0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4,
77                                     0xe5, 0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a};
78
79  const BYTE dataOut_AES192_CFB[] = {0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
80                                     0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
81                                     0x67, 0xce, 0x7f, 0x7f, 0x81, 0x17, 0x36, 0x21,
82                                     0x96, 0x1a, 0x2b, 0x70, 0x17, 0x1d, 0x3d, 0x7a};
83
84  const BYTE dataOut_AES192_OFB[] = {0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
85                                     0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
86                                     0xfc, 0xc2, 0x8b, 0x8d, 0x4c, 0x63, 0x83, 0x7c,
87                                     0x09, 0xe8, 0x17, 0x00, 0xc1, 0x10, 0x04, 0x01};
88
89  const BYTE dataOut_AES192_CTR[] = {0x1a, 0xbc, 0x93, 0x24, 0x17, 0x52, 0x1c, 0xa2,
90                                     0x4f, 0x2b, 0x04, 0x59, 0xfe, 0x7e, 0x6e, 0x0b,
91                                     0x09, 0x03, 0x39, 0xec, 0x0a, 0xa6, 0xfa, 0xef,
92                                     0xd5, 0xcc, 0xc2, 0xc6, 0xf4, 0xce, 0x8e, 0x94};
93  #endif
94
95  #if AES_256
96
97  const BYTE key_AES256[] = {0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
98                             0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
99                             0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
100                             0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4};
101
102  const BYTE dataIn_AES256[] = {0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
103                                0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
104                                0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
105                                0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
106
107  const BYTE dataOut_AES256_ECB[] = {0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c,
108                                     0x06, 0x4b, 0x5a, 0x7e, 0x3d, 0xb1, 0x81, 0xf8,
109                                     0x59, 0x1c, 0xcb, 0x10, 0xd4, 0x10, 0xed, 0x26,
110                                     0xdc, 0x5b, 0xa7, 0x4a, 0x31, 0x36, 0x28, 0x70};
111
112  const BYTE dataOut_AES256_CBC[] = {0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba,
113                                     0x77, 0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
114                                     0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d,
115                                     0x67, 0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d};
116
117  const BYTE dataOut_AES256_CFB[] = {0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
118                                     0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
119                                     0x39, 0xff, 0xed, 0x14, 0x3b, 0x28, 0xb1, 0xc8,
120                                     0x32, 0x11, 0x3c, 0x63, 0x31, 0xe5, 0x40, 0x7b};
121
122  const BYTE dataOut_AES256_OFB[] = {0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
123                                     0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
124                                     0x4f, 0xeb, 0xdc, 0x67, 0x40, 0xd2, 0x0b, 0x3a,
125                                     0xc8, 0x8f, 0x6a, 0xd8, 0x2a, 0x4f, 0xb0, 0x8d};
126
127  const BYTE dataOut_AES256_CTR[] = {0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
128                                     0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
129                                     0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
130                                     0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5};
131  #endif

```

6.37 /tpm/include/private/TableMarshal.h

```

1  #ifndef TABLE_MARSHAL_H
2  #define TABLE_MARSHAL_H
3

```

```

4  // These are the basic unmarshaling types. This is in the first byte of
5  // each structure descriptor that is passed to Marshal()/Unmarshal() for processing.
6  #define UINT_MTYPE      0
7  #define VALUES_MTYPE   (UINT_MTYPE + 1)
8  #define TABLE_MTYPE    (VALUES_MTYPE + 1)
9  #define MIN_MAX_MTYPE    (TABLE_MTYPE + 1)
10 #define ATTRIBUTES_MTYPE (MIN_MAX_MTYPE + 1)
11 #define STRUCTURE_MTYPE  (ATTRIBUTES_MTYPE + 1)
12 #define TPM2B_MTYPE      (STRUCTURE_MTYPE + 1)
13 #define TPM2BS_MTYPE     (TPM2B_MTYPE + 1)
14 #define LIST_MTYPE       (TPM2BS_MTYPE + 1) // TPML
15 #define ERROR_MTYPE      (LIST_MTYPE + 1)
16 #define NULL_MTYPE       (ERROR_MTYPE + 1)
17 #define COMPOSITE_MTYPE  (NULL_MTYPE + 1)
18
19 /*** The Marshal Index
20 // A structure is used to hold the values that guide the marshaling/unmarshaling of
21 // each of the types. Each structure has a name and an address. For a structure to
22 // define a TPMS_name, the structure is a TPMS_name_MARSHAL_STRUCT and its
23 // index is TPMS_name_MARSHAL_INDEX. So, to get the proper structure, use the
24 // associated marshal index. The marshal index is passed to Marshal() or Unmarshal()
25 // and those functions look up the proper structure.
26 //
27 // To handle structures that allow a null value, the upper bit of each marshal
28 // index indicates if the null value is allowed. This is the NULL_FLAG. It is defined
29 // in TableMarshalIndex.h because it is needed by code outside of the marshaling
30 // code.
31
32 // A structure will have a list of marshal indexes to indicate what to unmarshal. When
33 // that index appears in a structure/union, the value will contain a flag to indicate
34 // that the NULL_FLAG should be SET on the call to Unmarshal() to unmarshal the type.
35 // The caller simply takes the entry and passes it to Unmarshal() to indicate that the
36 // NULL_FLAG is SET. There is also the opportunity to SET the NULL_FLAG in the called
37 // structure if the NULL_FLAG was set in the call to the calling structure. This is
38 // indicated by:
39 #define NULL_MASK ~(NULL_FLAG)
40
41 // When looking up the value to marshal, the upper bit of the marshal index is
42 // masked to yield the actual index. The MSb is the flag bit that indicates if a
43 // null flag is set. Code does not verify that the bit is clear when the called object
44 // does not take a flag as this is a benign error.
45
46 // the modifier byte as used by each MTYPE shown as a structure. They are expressed
47 // as a bit maps below. However, the code uses masking and not bit fields. The types
48 // show below are just to help in understanding.
49 // NOTE: LSB0 bit numbering is assumed in these typedefs.
50 //
51 // When used in an UINT_MTYPE
52 typedef struct integerModifier
53 {
54     unsigned size    : 2;
55     unsigned sign     : 1;
56     unsigned unused   : 7;
57 } integerModifier;
58
59 // When used in a VALUES_MTYPE
60 typedef struct valuesModifier
61 {
62     unsigned size      : 2;
63     unsigned sign       : 1;
64     unsigned unused     : 5;
65     unsigned takesNull : 1;
66 } valuesModifier;
67
68 // When used in a TABLE_MTYPE
69 typedef struct tableModifier

```

```

70 {
71     unsigned size      : 2;
72     unsigned sign      : 1;
73     unsigned unused     : 3;
74     unsigned hasBits    : 1;
75     unsigned takesNull  : 1;
76 } tableModifier;
77
78 // the modifier byte for MIN_MAX_MTYPE
79 typedef struct minMaxModifier
80 {
81     unsigned size      : 2;
82     unsigned sign      : 1;
83     unsigned unused     : 3;
84     unsigned hasBits    : 1;
85     unsigned takesNull  : 1;
86 } minMaxModifier;
87
88 // the modifier byte for ATTRIBUTES_MTYPE
89 typedef struct attributesModifier
90 {
91     unsigned size      : 2;
92     unsigned sign      : 1;
93     unsigned unused     : 5;
94 } attributesModifier;
95
96 // the modifier byte is not present in a STRUCTURE_MTYPE or an TPM2B_MTYPE
97
98 // the modifier byte for a TPM2BS_MTYPE
99 typedef struct tpm2bsModifier
100 {
101     unsigned offset     : 4;
102     unsigned unused     : 2;
103     unsigned sizeEqual  : 1;
104     unsigned propagateNull : 1;
105 } tpm2bsModifier;
106
107 // the modifier byte for a LIST_MTYPE
108 typedef struct listModifier
109 {
110     unsigned offset     : 4;
111     unsigned unused     : 2;
112     unsigned sizeEqual  : 1;
113     unsigned propagateNull : 1;
114 } listModifier;
115
116 /*** Modifier Octet Values
117 // These are used in anything that is an integer value. These would not be in
118 // structure modifier bytes (they would be used in values in structures but not the
119 // STRUCTURE_MTYPE header.
120 #define ONE_BYTES      (0)
121 #define TWO_BYTES      (1)
122 #define FOUR_BYTES     (2)
123 #define EIGHT_BYTES    (3)
124 #define SIZE_MASK      (0x3)
125 #define IS_SIGNED      (1 << 2) // when the unmarshaled type is a signed value
126 #define SIGNED_MASK    (SIZE_MASK | IS_SIGNED)
127
128 // This may be used for any type except a UINT_MTYPE
129 #define TAKES_NULL      (1 << 7) // when the type takes a null
130
131 // When referencing a structure, this flag indicates if a null is to be propagated
132 // to the referenced structure or type.
133 #define PROPAGATE_NULL  (TAKES_NULL)
134
135 // Can be used in min-max or table structures.

```

```

136 #define HAS_BITS (1 << 6) // when bit mask is present
137
138 // In a union, we need to know if this is a union of constant arrays.
139 #define IS_ARRAY_UNION (1 << 6)
140
141 // In a TPM2BS_MTYPE
142 #define SIZE_EQUAL (1 << 6)
143 #define OFFSET_MASK (0xF)
144
145 // Right now, there are three spare bits in the modifiers field.
146
147 // Within the descriptor word of each entry in a StructMarsh_mst, there is a selector
148 // field to determine which of the sub-types the entry represents and a field that is
149 // used to reference another structure entry. This is a 6-bit field allowing a
150 // structure to have 64 entries. This should be more than enough as the structures are
151 // not that long. As of now, only 10-bits of the descriptor word leaving room for
152 // expansion.
153
154 // These are the values used in a STRUCTURE_MTYPE to identify the sub-type of the
155 // thing being processed
156 #define SIMPLE_STYPE 0
157 #define UNION_STYPE 1
158 #define ARRAY_STYPE 2
159
160 // The code used GET_ to get the element type and the compiler uses SET_ to initialize
161 // the value. The element type is the three bits (2:0).
162 #define GET_ELEMENT_TYPE(val) (val & 7)
163 #define SET_ELEMENT_TYPE(val) (val & 7)
164
165 // When an entry is an array or union, this references the structure entry that
166 // contains the dimension or selector value. The code then uses this number to look up
167 // the structure entry for that element to find out what it and where is it in memory.
168 // When this is not a reference, it is a simple type and it could be used as an array
169 // value or a union selector. When a simple value, this field contains the size
170 // of the associated value (ONE_BYTES, TWO_BYTES ...)
171 //
172 // The entry size/number is 6 bits (13:8).
173 #define GET_ELEMENT_NUMBER(val) (((val) >> 8) & 0x3F)
174 #define SET_ELEMENT_NUMBER(val) (((val) & 0x3F) << 8)
175 #define GET_ELEMENT_SIZE(val) GET_ELEMENT_NUMBER(val)
176 #define SET_ELEMENT_SIZE(val) SET_ELEMENT_NUMBER(val)
177 // This determines if the null flag is propagated to this type. If generate, the
178 // NULL_FLAG is SET in the index value. This flag is one bit (7)
179 #define ELEMENT_PROPAGATE (PROPAGATE_NULL)
180
181 #define INDEX_MASK ((UINT16) NULL_MASK)
182
183 // This is used in all bit-field checks. These are used when a value that is checked
184 // is conditional (dependent on the compilation). For example, if AES_128 is (NO),
185 // then the bit associated with AES_128 will be 0. In some cases, the bit value is
186 // found by checking that the input is within the range of the table, and then using
187 // the (val - min) value to index the bit. This would be used when verifying that
188 // a particular algorithm is implemented. In other cases, there is a bit for each
189 // value in a table. For example, if checking the key sizes, there is a list of
190 // possible key sizes allowed by the algorithm registry and a bit field to indicate
191 // if that key size is allowed in the implementation. The smallest bit field has
192 // 32-bits because it is implemented as part of the 'values' array in structures
193 // that allow bit fields.
194 #define IS_BIT_SET32(bit, bits) \
195     (((UINT32*)bits)[bit >> 5] & (1 << (bit & 0x1F))) != 0
196
197 // For a COMPOSITE_MTYPE, the qualifiers byte has an element size and count.
198 #define SET_ELEMENT_COUNT(count) ((count & 0x1F) << 3)
199 #define GET_ELEMENT_COUNT(val) ((val >> 3) & 0x1F)
200
201 #endif // _TABLE_MARSHAL_H_

```

6.38 /tpm/include/private/TableMarshalDefines.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
```

6.39 /tpm/include/private/TableMarshalMainTable.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
```

6.40 /tpm/include/private/TableMarshalPrototypes.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
```

6.41 /tpm/include/private/TableMarshalRedefines.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
```

6.42 /tpm/include/private/TableMarshalTypes.h

```
1 // clang-format off
2 /*(Auto-generated)
3 * Created by NewMarshal; Version 1.4 Apr 7, 2019
4 * Date: Mar 6, 2020 Time: 01:50:10PM
5 */
6
7 #ifndef TABLE_MARSHAL_TYPES_H_
8 #define TABLE_MARSHAL_TYPES_H_
9
10 typedef UINT16 marshalIndex_t;
11
12 /*** Structure Entries
13 // A structure contains a list of elements to unmarshal. Each of the entries is a
14 // UINT16. The structure descriptor is:
15
16 // The 'values' array contains indicators for the things to marshal. The 'elements'
17 // parameter indicates how many different entities are unmarshaled. This number
18 // nominally corresponds to the number of rows in the Part 2 table that describes
19 // the structure (the number of rows minus the title row and any error code rows).
20
21 // A schematic of a simple structure entry is shown here but the values are not
22 // actually in a structure. As shown, the third value is the offset in the structure
23 // where the value is placed when unmarshaled, or fetched from when marshaling. This
24 // is sufficient when the element type indicated by 'index' is always a simple type
25 // and never a union or array. This is just shown for illustrative purposes.
26 typedef struct simpleStructureEntry_t {
27     UINT16 qualifiers; // indicates the type of entry (array, union
28                       // etc.)
29     marshalIndex_t index; // the index into the appropriate array of
30                           // the descriptor of this type
31     UINT16 offset; // where this comes from or is placed
32 } simpleStructureEntry_t;
33
34 typedef const struct UintMarshal_mst
35 {
36     UINT8 marshalType; // UINT_MTYPE
37     UINT8 modifiers; // size and signed indicator.
38 } UintMarshal_mst;
39
40 typedef struct UnionMarshal_mst
41 {
42     UINT8 countOfselectors;
43     UINT8 modifiers; // NULL_SELECTOR
```



```

44     UINT16      offsetOfUnmarshalTypes;
45     UINT32      selectors[1];
46 //     UINT16      marshalingTypes[1]; // This is not part of the prototypical
47 //                                     // entry. It is here to show where the
48 //                                     // marshaling types will be in a union
49 } UnionMarshal_mst;
50
51 typedef struct NullUnionMarshal_mst
52 {
53     UINT8      count;
54 } NullUnionMarshal_mst;
55
56 typedef struct MarshalHeader_mst
57 {
58     UINT8      marshalType; // VALUES_MTYPE
59     UINT8      modifiers;
60     UINT8      errorCode;
61 } MarshalHeader_mst;
62
63 typedef const struct ArrayMarshal_mst // used in a structure
64 {
65     marshalIndex_t type;
66     UINT16      stride;
67 } ArrayMarshal_mst;
68
69 typedef const struct StructMarshal_mst
70 {
71     UINT8      marshalType; // STRUCTURE_MTYPE
72     UINT8      elements;
73     UINT16      values[1]; // three times elements
74 } StructMarshal_mst;
75
76 typedef const struct ValuesMarshal_mst
77 {
78     UINT8      marshalType; // VALUES_MTYPE
79     UINT8      modifiers;
80     UINT8      errorCode;
81     UINT8      ranges;
82     UINT8      singles;
83     UINT32      values[1];
84 } ValuesMarshal_mst;
85
86 typedef const struct TableMarshal_mst
87 {
88     UINT8      marshalType; // TABLE_MTYPE
89     UINT8      modifiers;
90     UINT8      errorCode;
91     UINT8      singles;
92     UINT32      values[1];
93 } TableMarshal_mst;
94
95 typedef const struct MinMaxMarshal_mst
96 {
97     UINT8      marshalType; // MIN_MAX_MTYPE
98     UINT8      modifiers;
99     UINT8      errorCode;
100    UINT32      values[2];
101 } MinMaxMarshal_mst;
102
103 typedef const struct Tpm2bMarshal_mst
104 {
105     UINT8      unmarshalType; // TPM2B_MTYPE
106     UINT16      sizeIndex; // reference to type for this size value
107 } Tpm2bMarshal_mst;
108
109 typedef const struct Tpm2bsMarshal_mst

```

```

110 {
111     UINT8        unmarshalType;    // TPM2BS_MTYPE
112     UINT8        modifiers;        // size= and offset (2 - 7)
113     UINT16       sizeIndex;        // index of the size value;
114     UINT16       dataIndex;        // the structure
115 } Tpm2bsMarshal_mst;
116
117 typedef const struct ListMarshal_mst
118 {
119     UINT8        unmarshalType;    // LIST_MTYPE (for TPML)
120     UINT8        modifiers;        // size offset 2-7
121     UINT16       sizeIndex;        // reference to the minmax structure that
122                                     // unmarshals the size parameter
123     UINT16       arrayRef;        // reference to an array definition (type
124                                     // and stride)
125 } ListMarshal_mst;
126
127 typedef const struct AttributesMarshal_mst
128 {
129     UINT8        unmarshashType;    // ATTRIBUTE_MTYPE
130     UINT8        modifiers;        // size (ONE_BYTES, TWO_BYTES, or FOUR_BYTES
131     UINT32       attributeMask;    // the values that must be zero.
132 } AttributesMarshal_mst;
133
134 typedef const struct CompositeMarshal_mst
135 {
136     UINT8        unmarshashType;    // COMPOSITE_MTYPE
137     UINT8        modifiers;        // number of entries and size
138     marshalIndex_t types[1];        // array of unmarshaling types
139 } CompositeMarshal_mst;
140
141 typedef const struct TPM_ECC_CURVE_mst {
142     UINT8        marshalType;
143     UINT8        modifiers;
144     UINT8        errorCode;
145     UINT32       values[4];
146 } TPM_ECC_CURVE_mst;
147
148 typedef const struct TPM_CLOCK_ADJUST_mst {
149     UINT8        marshalType;
150     UINT8        modifiers;
151     UINT8        errorCode;
152     UINT32       values[2];
153 } TPM_CLOCK_ADJUST_mst;
154
155 typedef const struct TPM_EO_mst {
156     UINT8        marshalType;
157     UINT8        modifiers;
158     UINT8        errorCode;
159     UINT32       values[2];
160 } TPM_EO_mst;
161
162 typedef const struct TPM_SU_mst {
163     UINT8        marshalType;
164     UINT8        modifiers;
165     UINT8        errorCode;
166     UINT8        entries;
167     UINT32       values[2];
168 } TPM_SU_mst;
169
170 typedef const struct TPM_SE_mst {
171     UINT8        marshalType;
172     UINT8        modifiers;
173     UINT8        errorCode;
174     UINT8        entries;
175     UINT32       values[3];

```

```
176 } TPM_SE_mst;
177
178 typedef const struct TPM_CAP_mst {
179     UINT8      marshalType;
180     UINT8      modifiers;
181     UINT8      errorCode;
182     UINT8      ranges;
183     UINT8      singles;
184     UINT32     values[3];
185 } TPM_CAP_mst;
186
187 typedef const struct TPMI_YES_NO_mst {
188     UINT8      marshalType;
189     UINT8      modifiers;
190     UINT8      errorCode;
191     UINT8      entries;
192     UINT32     values[2];
193 } TPMI_YES_NO_mst;
194
195 typedef const struct TPMI_DH_OBJECT_mst {
196     UINT8      marshalType;
197     UINT8      modifiers;
198     UINT8      errorCode;
199     UINT8      ranges;
200     UINT8      singles;
201     UINT32     values[5];
202 } TPMI_DH_OBJECT_mst;
203
204 typedef const struct TPMI_DH_PARENT_mst {
205     UINT8      marshalType;
206     UINT8      modifiers;
207     UINT8      errorCode;
208     UINT8      ranges;
209     UINT8      singles;
210     UINT32     values[20];
211 } TPMI_DH_PARENT_mst;
212
213 typedef const struct TPMI_DH_PERSISTENT_mst {
214     UINT8      marshalType;
215     UINT8      modifiers;
216     UINT8      errorCode;
217     UINT32     values[2];
218 } TPMI_DH_PERSISTENT_mst;
219
220 typedef const struct TPMI_DH_ENTITY_mst {
221     UINT8      marshalType;
222     UINT8      modifiers;
223     UINT8      errorCode;
224     UINT8      ranges;
225     UINT8      singles;
226     UINT32     values[15];
227 } TPMI_DH_ENTITY_mst;
228
229 typedef const struct TPMI_DH_PCR_mst {
230     UINT8      marshalType;
231     UINT8      modifiers;
232     UINT8      errorCode;
233     UINT32     values[3];
234 } TPMI_DH_PCR_mst;
235
236 typedef const struct TPMI_SH_AUTH_SESSION_mst {
237     UINT8      marshalType;
238     UINT8      modifiers;
239     UINT8      errorCode;
240     UINT8      ranges;
241     UINT8      singles;
```

```
242     UINT32      values[5];
243 } TPMI_SH_AUTH_SESSION_mst;
244
245 typedef const struct TPMI_SH_HMAC_mst {
246     UINT8      marshalType;
247     UINT8      modifiers;
248     UINT8      errorCode;
249     UINT32      values[2];
250 } TPMI_SH_HMAC_mst;
251
252 typedef const struct TPMI_SH_POLICY_mst {
253     UINT8      marshalType;
254     UINT8      modifiers;
255     UINT8      errorCode;
256     UINT32      values[2];
257 } TPMI_SH_POLICY_mst;
258
259 typedef const struct TPMI_DH_CONTEXT_mst {
260     UINT8      marshalType;
261     UINT8      modifiers;
262     UINT8      errorCode;
263     UINT8      ranges;
264     UINT8      singles;
265     UINT32      values[6];
266 } TPMI_DH_CONTEXT_mst;
267
268 typedef const struct TPMI_DH_SAVED_mst {
269     UINT8      marshalType;
270     UINT8      modifiers;
271     UINT8      errorCode;
272     UINT8      ranges;
273     UINT8      singles;
274     UINT32      values[7];
275 } TPMI_DH_SAVED_mst;
276
277 typedef const struct TPMI_RH_HIERARCHY_mst {
278     UINT8      marshalType;
279     UINT8      modifiers;
280     UINT8      errorCode;
281     UINT8      ranges;
282     UINT8      singles;
283     UINT32      values[16];
284 } TPMI_RH_HIERARCHY_mst;
285
286 typedef const struct TPMI_RH_ENABLES_mst {
287     UINT8      marshalType;
288     UINT8      modifiers;
289     UINT8      errorCode;
290     UINT8      entries;
291     UINT32      values[5];
292 } TPMI_RH_ENABLES_mst;
293
294 typedef const struct TPMI_RH_HIERARCHY_AUTH_mst {
295     UINT8      marshalType;
296     UINT8      modifiers;
297     UINT8      errorCode;
298     UINT8      entries;
299     UINT32      values[4];
300 } TPMI_RH_HIERARCHY_AUTH_mst;
301
302 typedef const struct TPMI_RH_HIERARCHY_POLICY_mst {
303     UINT8      marshalType;
304     UINT8      modifiers;
305     UINT8      errorCode;
306     UINT8      ranges;
307     UINT8      singles;
```

```
308     UINT32         values[6];
309 } TPMI_RH_HIERARCHY_POLICY_mst;
310
311 typedef const struct TPMI_RH_BASE_HIERARCHY_mst {
312     UINT8         marshalType;
313     UINT8         modifiers;
314     UINT8         errorCode;
315     UINT8         entries;
316     UINT32         values[3];
317 } TPMI_RH_BASE_HIERARCHY_mst;
318
319 typedef const struct TPMI_RH_PLATFORM_mst {
320     UINT8         marshalType;
321     UINT8         modifiers;
322     UINT8         errorCode;
323     UINT8         entries;
324     UINT32         values[1];
325 } TPMI_RH_PLATFORM_mst;
326
327 typedef const struct TPMI_RH_OWNER_mst {
328     UINT8         marshalType;
329     UINT8         modifiers;
330     UINT8         errorCode;
331     UINT8         entries;
332     UINT32         values[2];
333 } TPMI_RH_OWNER_mst;
334
335 typedef const struct TPMI_RH_ENDORSEMENT_mst {
336     UINT8         marshalType;
337     UINT8         modifiers;
338     UINT8         errorCode;
339     UINT8         entries;
340     UINT32         values[2];
341 } TPMI_RH_ENDORSEMENT_mst;
342
343 typedef const struct TPMI_RH_PROVISION_mst {
344     UINT8         marshalType;
345     UINT8         modifiers;
346     UINT8         errorCode;
347     UINT8         entries;
348     UINT32         values[2];
349 } TPMI_RH_PROVISION_mst;
350
351 typedef const struct TPMI_RH_CLEAR_mst {
352     UINT8         marshalType;
353     UINT8         modifiers;
354     UINT8         errorCode;
355     UINT8         entries;
356     UINT32         values[2];
357 } TPMI_RH_CLEAR_mst;
358
359 typedef const struct TPMI_RH_NV_AUTH_mst {
360     UINT8         marshalType;
361     UINT8         modifiers;
362     UINT8         errorCode;
363     UINT8         ranges;
364     UINT8         singles;
365     UINT32         values[4];
366 } TPMI_RH_NV_AUTH_mst;
367
368 typedef const struct TPMI_RH_LOCKOUT_mst {
369     UINT8         marshalType;
370     UINT8         modifiers;
371     UINT8         errorCode;
372     UINT8         entries;
373     UINT32         values[1];
```

```
374 } TPMI_RH_LOCKOUT_mst;
375
376 typedef const struct TPMI_RH_NV_INDEX_mst {
377     UINT8      marshalType;
378     UINT8      modifiers;
379     UINT8      errorCode;
380     UINT32     values[2];
381 } TPMI_RH_NV_INDEX_mst;
382
383 typedef const struct TPMI_RH_AC_mst {
384     UINT8      marshalType;
385     UINT8      modifiers;
386     UINT8      errorCode;
387     UINT32     values[2];
388 } TPMI_RH_AC_mst;
389
390 typedef const struct TPMI_RH_ACT_mst {
391     UINT8      marshalType;
392     UINT8      modifiers;
393     UINT8      errorCode;
394     UINT32     values[2];
395 } TPMI_RH_ACT_mst;
396
397 typedef const struct TPMI_ALG_HASH_mst {
398     UINT8      marshalType;
399     UINT8      modifiers;
400     UINT8      errorCode;
401     UINT32     values[5];
402 } TPMI_ALG_HASH_mst;
403
404 typedef const struct TPMI_ALG_ASYM_mst {
405     UINT8      marshalType;
406     UINT8      modifiers;
407     UINT8      errorCode;
408     UINT32     values[5];
409 } TPMI_ALG_ASYM_mst;
410
411 typedef const struct TPMI_ALG_SYM_mst {
412     UINT8      marshalType;
413     UINT8      modifiers;
414     UINT8      errorCode;
415     UINT32     values[5];
416 } TPMI_ALG_SYM_mst;
417
418 typedef const struct TPMI_ALG_SYM_OBJECT_mst {
419     UINT8      marshalType;
420     UINT8      modifiers;
421     UINT8      errorCode;
422     UINT32     values[5];
423 } TPMI_ALG_SYM_OBJECT_mst;
424
425 typedef const struct TPMI_ALG_SYM_MODE_mst {
426     UINT8      marshalType;
427     UINT8      modifiers;
428     UINT8      errorCode;
429     UINT32     values[4];
430 } TPMI_ALG_SYM_MODE_mst;
431
432 typedef const struct TPMI_ALG_KDF_mst {
433     UINT8      marshalType;
434     UINT8      modifiers;
435     UINT8      errorCode;
436     UINT32     values[4];
437 } TPMI_ALG_KDF_mst;
438
439 typedef const struct TPMI_ALG_SIG_SCHEME_mst {
```



```

440     UINT8      marshalType;
441     UINT8      modifiers;
442     UINT8      errorCode;
443     UINT32     values[4];
444 } TPMI_ALG_SIG_SCHEME_mst;
445
446 typedef const struct TPMI_ECC_KEY_EXCHANGE_mst {
447     UINT8      marshalType;
448     UINT8      modifiers;
449     UINT8      errorCode;
450     UINT32     values[4];
451 } TPMI_ECC_KEY_EXCHANGE_mst;
452
453 typedef const struct TPMI_ST_COMMAND_TAG_mst {
454     UINT8      marshalType;
455     UINT8      modifiers;
456     UINT8      errorCode;
457     UINT8      entries;
458     UINT32     values[2];
459 } TPMI_ST_COMMAND_TAG_mst;
460
461 typedef const struct TPMI_ALG_MAC_SCHEME_mst {
462     UINT8      marshalType;
463     UINT8      modifiers;
464     UINT8      errorCode;
465     UINT32     values[5];
466 } TPMI_ALG_MAC_SCHEME_mst;
467
468 typedef const struct TPMI_ALG_CIPHER_MODE_mst {
469     UINT8      marshalType;
470     UINT8      modifiers;
471     UINT8      errorCode;
472     UINT32     values[4];
473 } TPMI_ALG_CIPHER_MODE_mst;
474
475 typedef const struct TPMS_EMPTY_mst
476 {
477     UINT8      marshalType;
478     UINT8      elements;
479     UINT16     values[3];
480 } TPMS_EMPTY_mst;
481
482 typedef const struct TPMS_ALGORITHM_DESCRIPTION_mst
483 {
484     UINT8      marshalType;
485     UINT8      elements;
486     UINT16     values[6];
487 } TPMS_ALGORITHM_DESCRIPTION_mst;
488
489 typedef struct TPMU_HA_mst
490 {
491     BYTE      countOfselectors;
492     BYTE      modifiers;
493     UINT16     offsetOfUnmarshalTypes;
494     UINT32     selectors[9];
495     UINT16     marshalingTypes[9];
496 } TPMU_HA_mst;
497
498 typedef const struct TPMT_HA_mst
499 {
500     UINT8      marshalType;
501     UINT8      elements;
502     UINT16     values[6];
503 } TPMT_HA_mst;
504
505 typedef const struct TPMS_PCR_SELECT_mst

```

```
506 {
507     UINT8    marshalType;
508     UINT8    elements;
509     UINT16    values[6];
510 } TPMS_PCR_SELECT_mst;
511
512 typedef const struct TPMS_PCR_SELECTION_mst
513 {
514     UINT8    marshalType;
515     UINT8    elements;
516     UINT16    values[9];
517 } TPMS_PCR_SELECTION_mst;
518
519 typedef const struct TPMT_TK_CREATION_mst
520 {
521     UINT8    marshalType;
522     UINT8    elements;
523     UINT16    values[9];
524 } TPMT_TK_CREATION_mst;
525
526 typedef const struct TPMT_TK_VERIFIED_mst
527 {
528     UINT8    marshalType;
529     UINT8    elements;
530     UINT16    values[9];
531 } TPMT_TK_VERIFIED_mst;
532
533 typedef const struct TPMT_TK_AUTH_mst
534 {
535     UINT8    marshalType;
536     UINT8    elements;
537     UINT16    values[9];
538 } TPMT_TK_AUTH_mst;
539
540 typedef const struct TPMT_TK_HASHCHECK_mst
541 {
542     UINT8    marshalType;
543     UINT8    elements;
544     UINT16    values[9];
545 } TPMT_TK_HASHCHECK_mst;
546
547 typedef const struct TPMS_ALG_PROPERTY_mst
548 {
549     UINT8    marshalType;
550     UINT8    elements;
551     UINT16    values[6];
552 } TPMS_ALG_PROPERTY_mst;
553
554 typedef const struct TPMS_TAGGED_PROPERTY_mst
555 {
556     UINT8    marshalType;
557     UINT8    elements;
558     UINT16    values[6];
559 } TPMS_TAGGED_PROPERTY_mst;
560
561 typedef const struct TPMS_TAGGED_PCR_SELECT_mst
562 {
563     UINT8    marshalType;
564     UINT8    elements;
565     UINT16    values[9];
566 } TPMS_TAGGED_PCR_SELECT_mst;
567
568 typedef const struct TPMS_TAGGED_POLICY_mst
569 {
570     UINT8    marshalType;
571     UINT8    elements;
```

```

572     UINT16    values[6];
573 } TPMS_TAGGED_POLICY_mst;
574
575 typedef const struct TPMS_ACT_DATA_mst
576 {
577     UINT8     marshalType;
578     UINT8     elements;
579     UINT16    values[9];
580 } TPMS_ACT_DATA_mst;
581
582 typedef struct TPMU_CAPABILITIES_mst
583 {
584     BYTE      countOfselectors;
585     BYTE      modifiers;
586     UINT16    offsetOfUnmarshalTypes;
587     UINT32    selectors[11];
588     UINT16    marshalingTypes[11];
589 } TPMU_CAPABILITIES_mst;
590
591 typedef const struct TPMS_CAPABILITY_DATA_mst
592 {
593     UINT8     marshalType;
594     UINT8     elements;
595     UINT16    values[6];
596 } TPMS_CAPABILITY_DATA_mst;
597
598 typedef const struct TPMS_CLOCK_INFO_mst
599 {
600     UINT8     marshalType;
601     UINT8     elements;
602     UINT16    values[12];
603 } TPMS_CLOCK_INFO_mst;
604
605 typedef const struct TPMS_TIME_INFO_mst
606 {
607     UINT8     marshalType;
608     UINT8     elements;
609     UINT16    values[6];
610 } TPMS_TIME_INFO_mst;
611
612 typedef const struct TPMS_TIME_ATTEST_INFO_mst
613 {
614     UINT8     marshalType;
615     UINT8     elements;
616     UINT16    values[6];
617 } TPMS_TIME_ATTEST_INFO_mst;
618
619 typedef const struct TPMS_CERTIFY_INFO_mst
620 {
621     UINT8     marshalType;
622     UINT8     elements;
623     UINT16    values[6];
624 } TPMS_CERTIFY_INFO_mst;
625
626 typedef const struct TPMS_QUOTE_INFO_mst
627 {
628     UINT8     marshalType;
629     UINT8     elements;
630     UINT16    values[6];
631 } TPMS_QUOTE_INFO_mst;
632
633 typedef const struct TPMS_COMMAND_AUDIT_INFO_mst
634 {
635     UINT8     marshalType;
636     UINT8     elements;
637     UINT16    values[12];

```

```

638 } TPMS_COMMAND_AUDIT_INFO_mst;
639
640 typedef const struct TPMS_SESSION_AUDIT_INFO_mst
641 {
642     UINT8    marshalType;
643     UINT8    elements;
644     UINT16   values[6];
645 } TPMS_SESSION_AUDIT_INFO_mst;
646
647 typedef const struct TPMS_CREATION_INFO_mst
648 {
649     UINT8    marshalType;
650     UINT8    elements;
651     UINT16   values[6];
652 } TPMS_CREATION_INFO_mst;
653
654 typedef const struct TPMS_NV_CERTIFY_INFO_mst
655 {
656     UINT8    marshalType;
657     UINT8    elements;
658     UINT16   values[9];
659 } TPMS_NV_CERTIFY_INFO_mst;
660
661 typedef const struct TPMS_NV_DIGEST_CERTIFY_INFO_mst
662 {
663     UINT8    marshalType;
664     UINT8    elements;
665     UINT16   values[6];
666 } TPMS_NV_DIGEST_CERTIFY_INFO_mst;
667
668 typedef const struct TPMI_ST_ATTEST_mst {
669     UINT8    marshalType;
670     UINT8    modifiers;
671     UINT8    errorCode;
672     UINT8    ranges;
673     UINT8    singles;
674     UINT32   values[3];
675 } TPMI_ST_ATTEST_mst;
676
677 typedef struct TPMU_ATTEST_mst
678 {
679     BYTE      countOfselectors;
680     BYTE      modifiers;
681     UINT16    offsetOfUnmarshalTypes;
682     UINT32    selectors[8];
683     UINT16    marshalingTypes[8];
684 } TPMU_ATTEST_mst;
685
686 typedef const struct TPMS_ATTEST_mst
687 {
688     UINT8    marshalType;
689     UINT8    elements;
690     UINT16   values[21];
691 } TPMS_ATTEST_mst;
692
693 typedef const struct TPMS_AUTH_COMMAND_mst
694 {
695     UINT8    marshalType;
696     UINT8    elements;
697     UINT16   values[12];
698 } TPMS_AUTH_COMMAND_mst;
699
700 typedef const struct TPMS_AUTH_RESPONSE_mst
701 {
702     UINT8    marshalType;
703     UINT8    elements;

```

```

704     UINT16     values[9];
705 } TPMS_AUTH_RESPONSE_mst;
706
707 typedef const struct TPMI_AES_KEY_BITS_mst {
708     UINT8      marshalType;
709     UINT8      modifiers;
710     UINT8      errorCode;
711     UINT8      entries;
712     UINT32     values[3];
713 } TPMI_AES_KEY_BITS_mst;
714
715 typedef const struct TPMI_SM4_KEY_BITS_mst {
716     UINT8      marshalType;
717     UINT8      modifiers;
718     UINT8      errorCode;
719     UINT8      entries;
720     UINT32     values[1];
721 } TPMI_SM4_KEY_BITS_mst;
722
723 typedef const struct TPMI_CAMELLIA_KEY_BITS_mst {
724     UINT8      marshalType;
725     UINT8      modifiers;
726     UINT8      errorCode;
727     UINT8      entries;
728     UINT32     values[3];
729 } TPMI_CAMELLIA_KEY_BITS_mst;
730
731 typedef struct TPMU_SYM_KEY_BITS_mst
732 {
733     BYTE        countOfselectors;
734     BYTE        modifiers;
735     UINT16      offsetOfUnmarshalTypes;
736     UINT32      selectors[6];
737     UINT16      marshalingTypes[6];
738 } TPMU_SYM_KEY_BITS_mst;
739
740 typedef struct TPMU_SYM_MODE_mst
741 {
742     BYTE        countOfselectors;
743     BYTE        modifiers;
744     UINT16      offsetOfUnmarshalTypes;
745     UINT32      selectors[6];
746     UINT16      marshalingTypes[6];
747 } TPMU_SYM_MODE_mst;
748
749 typedef const struct TPMT_SYM_DEF_mst
750 {
751     UINT8      marshalType;
752     UINT8      elements;
753     UINT16     values[9];
754 } TPMT_SYM_DEF_mst;
755
756 typedef const struct TPMT_SYM_DEF_OBJECT_mst
757 {
758     UINT8      marshalType;
759     UINT8      elements;
760     UINT16     values[9];
761 } TPMT_SYM_DEF_OBJECT_mst;
762
763 typedef const struct TPMS_SYMCIPHER_PARMS_mst
764 {
765     UINT8      marshalType;
766     UINT8      elements;
767     UINT16     values[3];
768 } TPMS_SYMCIPHER_PARMS_mst;
769

```

```

770 typedef const struct TPMS_DERIVE_mst
771 {
772     UINT8    marshalType;
773     UINT8    elements;
774     UINT16    values[6];
775 } TPMS_DERIVE_mst;
776
777 typedef const struct TPMS_SENSITIVE_CREATE_mst
778 {
779     UINT8    marshalType;
780     UINT8    elements;
781     UINT16    values[6];
782 } TPMS_SENSITIVE_CREATE_mst;
783
784 typedef const struct TPMS_SCHEME_HASH_mst
785 {
786     UINT8    marshalType;
787     UINT8    elements;
788     UINT16    values[3];
789 } TPMS_SCHEME_HASH_mst;
790
791 typedef const struct TPMS_SCHEME_ECDA_mst
792 {
793     UINT8    marshalType;
794     UINT8    elements;
795     UINT16    values[6];
796 } TPMS_SCHEME_ECDA_mst;
797
798 typedef const struct TPMS_ALG_KEYEDHASH_SCHEME_mst {
799     UINT8    marshalType;
800     UINT8    modifiers;
801     UINT8    errorCode;
802     UINT32    values[4];
803 } TPMS_ALG_KEYEDHASH_SCHEME_mst;
804
805 typedef const struct TPMS_SCHEME_XOR_mst
806 {
807     UINT8    marshalType;
808     UINT8    elements;
809     UINT16    values[6];
810 } TPMS_SCHEME_XOR_mst;
811
812 typedef struct TPMU_SCHEME_KEYEDHASH_mst
813 {
814     BYTE    countOfselectors;
815     BYTE    modifiers;
816     UINT16    offsetOfUnmarshalTypes;
817     UINT32    selectors[3];
818     UINT16    marshalingTypes[3];
819 } TPMU_SCHEME_KEYEDHASH_mst;
820
821 typedef const struct TPMT_KEYEDHASH_SCHEME_mst
822 {
823     UINT8    marshalType;
824     UINT8    elements;
825     UINT16    values[6];
826 } TPMT_KEYEDHASH_SCHEME_mst;
827
828 typedef struct TPMU_SIG_SCHEME_mst
829 {
830     BYTE    countOfselectors;
831     BYTE    modifiers;
832     UINT16    offsetOfUnmarshalTypes;
833     UINT32    selectors[8];
834     UINT16    marshalingTypes[8];
835 } TPMU_SIG_SCHEME_mst;

```



```
836
837 typedef const struct TPMT_SIG_SCHEME_mst
838 {
839     UINT8    marshalType;
840     UINT8    elements;
841     UINT16    values[6];
842 } TPMT_SIG_SCHEME_mst;
843
844 typedef struct TPMU_KDF_SCHEME_mst
845 {
846     BYTE        countOfselectors;
847     BYTE        modifiers;
848     UINT16        offsetOfUnmarshalTypes;
849     UINT32        selectors[5];
850     UINT16        marshalingTypes[5];
851 } TPMU_KDF_SCHEME_mst;
852
853 typedef const struct TPMT_KDF_SCHEME_mst
854 {
855     UINT8    marshalType;
856     UINT8    elements;
857     UINT16    values[6];
858 } TPMT_KDF_SCHEME_mst;
859
860 typedef const struct TPMI_ALG_ASYNC_SCHEME_mst {
861     UINT8    marshalType;
862     UINT8    modifiers;
863     UINT8    errorCode;
864     UINT32    values[4];
865 } TPMI_ALG_ASYNC_SCHEME_mst;
866
867 typedef struct TPMU_ASYNC_SCHEME_mst
868 {
869     BYTE        countOfselectors;
870     BYTE        modifiers;
871     UINT16        offsetOfUnmarshalTypes;
872     UINT32        selectors[11];
873     UINT16        marshalingTypes[11];
874 } TPMU_ASYNC_SCHEME_mst;
875
876 typedef const struct TPMI_ALG_RSA_SCHEME_mst {
877     UINT8    marshalType;
878     UINT8    modifiers;
879     UINT8    errorCode;
880     UINT32    values[4];
881 } TPMI_ALG_RSA_SCHEME_mst;
882
883 typedef const struct TPMT_RSA_SCHEME_mst
884 {
885     UINT8    marshalType;
886     UINT8    elements;
887     UINT16    values[6];
888 } TPMT_RSA_SCHEME_mst;
889
890 typedef const struct TPMI_ALG_RSA_DECRYPT_mst {
891     UINT8    marshalType;
892     UINT8    modifiers;
893     UINT8    errorCode;
894     UINT32    values[4];
895 } TPMI_ALG_RSA_DECRYPT_mst;
896
897 typedef const struct TPMT_RSA_DECRYPT_mst
898 {
899     UINT8    marshalType;
900     UINT8    elements;
901     UINT16    values[6];
```

```

902 } TPMT_RSA_DECRYPT_mst;
903
904 typedef const struct TPMT_RSA_KEY_BITS_mst {
905     UINT8      marshalType;
906     UINT8      modifiers;
907     UINT8      errorCode;
908     UINT8      entries;
909     UINT32     values[3];
910 } TPMT_RSA_KEY_BITS_mst;
911
912 typedef const struct TPMS_ECC_POINT_mst
913 {
914     UINT8      marshalType;
915     UINT8      elements;
916     UINT16     values[6];
917 } TPMS_ECC_POINT_mst;
918
919 typedef const struct TPMT_ALG_ECC_SCHEME_mst {
920     UINT8      marshalType;
921     UINT8      modifiers;
922     UINT8      errorCode;
923     UINT32     values[4];
924 } TPMT_ALG_ECC_SCHEME_mst;
925
926 typedef const struct TPMT_ECC_CURVE_mst {
927     UINT8      marshalType;
928     UINT8      modifiers;
929     UINT8      errorCode;
930     UINT32     values[3];
931 } TPMT_ECC_CURVE_mst;
932
933 typedef const struct TPMT_ECC_SCHEME_mst
934 {
935     UINT8      marshalType;
936     UINT8      elements;
937     UINT16     values[6];
938 } TPMT_ECC_SCHEME_mst;
939
940 typedef const struct TPMS_ALGORITHM_DETAIL_ECC_mst
941 {
942     UINT8      marshalType;
943     UINT8      elements;
944     UINT16     values[33];
945 } TPMS_ALGORITHM_DETAIL_ECC_mst;
946
947 typedef const struct TPMS_SIGNATURE_RSA_mst
948 {
949     UINT8      marshalType;
950     UINT8      elements;
951     UINT16     values[6];
952 } TPMS_SIGNATURE_RSA_mst;
953
954 typedef const struct TPMS_SIGNATURE_ECC_mst
955 {
956     UINT8      marshalType;
957     UINT8      elements;
958     UINT16     values[9];
959 } TPMS_SIGNATURE_ECC_mst;
960
961 typedef struct TPMU_SIGNATURE_mst
962 {
963     BYTE      countOfselectors;
964     BYTE      modifiers;
965     UINT16     offsetOfUnmarshalTypes;
966     UINT32     selectors[8];
967     UINT16     marshalingTypes[8];

```

```

968 } TPMU_SIGNATURE_mst;
969
970 typedef const struct TPMT_SIGNATURE_mst
971 {
972     UINT8    marshalType;
973     UINT8    elements;
974     UINT16   values[6];
975 } TPMT_SIGNATURE_mst;
976
977 typedef struct TPMU_ENCRYPTED_SECRET_mst
978 {
979     BYTE      countOfselectors;
980     BYTE      modifiers;
981     UINT16    offsetOfUnmarshalTypes;
982     UINT32    selectors[4];
983     UINT16    marshalingTypes[4];
984 } TPMU_ENCRYPTED_SECRET_mst;
985
986 typedef const struct TPMI_ALG_PUBLIC_mst {
987     UINT8    marshalType;
988     UINT8    modifiers;
989     UINT8    errorCode;
990     UINT32    values[4];
991 } TPMI_ALG_PUBLIC_mst;
992
993 typedef struct TPMU_PUBLIC_ID_mst
994 {
995     BYTE      countOfselectors;
996     BYTE      modifiers;
997     UINT16    offsetOfUnmarshalTypes;
998     UINT32    selectors[4];
999     UINT16    marshalingTypes[4];
1000 } TPMU_PUBLIC_ID_mst;
1001
1002 typedef const struct TPMS_KEYEDHASH_PARMS_mst
1003 {
1004     UINT8    marshalType;
1005     UINT8    elements;
1006     UINT16   values[3];
1007 } TPMS_KEYEDHASH_PARMS_mst;
1008
1009 typedef const struct TPMS_RSA_PARMS_mst
1010 {
1011     UINT8    marshalType;
1012     UINT8    elements;
1013     UINT16   values[12];
1014 } TPMS_RSA_PARMS_mst;
1015
1016 typedef const struct TPMS_ECC_PARMS_mst
1017 {
1018     UINT8    marshalType;
1019     UINT8    elements;
1020     UINT16   values[12];
1021 } TPMS_ECC_PARMS_mst;
1022
1023 typedef struct TPMU_PUBLIC_PARMS_mst
1024 {
1025     BYTE      countOfselectors;
1026     BYTE      modifiers;
1027     UINT16    offsetOfUnmarshalTypes;
1028     UINT32    selectors[4];
1029     UINT16    marshalingTypes[4];
1030 } TPMU_PUBLIC_PARMS_mst;
1031
1032 typedef const struct TPMT_PUBLIC_PARMS_mst
1033 {

```

```

1034     UINT8     marshalType;
1035     UINT8     elements;
1036     UINT16    values[6];
1037 } TPMT_PUBLIC_PARMS_mst;
1038
1039 typedef const struct TPMT_PUBLIC_mst
1040 {
1041     UINT8     marshalType;
1042     UINT8     elements;
1043     UINT16    values[18];
1044 } TPMT_PUBLIC_mst;
1045
1046 typedef struct TPMU_SENSITIVE_COMPOSITE_mst
1047 {
1048     BYTE      countOfselectors;
1049     BYTE      modifiers;
1050     UINT16    offsetOfUnmarshalTypes;
1051     UINT32    selectors[4];
1052     UINT16    marshalingTypes[4];
1053 } TPMU_SENSITIVE_COMPOSITE_mst;
1054
1055 typedef const struct TPMT_SENSITIVE_mst
1056 {
1057     UINT8     marshalType;
1058     UINT8     elements;
1059     UINT16    values[12];
1060 } TPMT_SENSITIVE_mst;
1061
1062 typedef const struct TPMS_NV_PIN_COUNTER_PARAMETERS_mst
1063 {
1064     UINT8     marshalType;
1065     UINT8     elements;
1066     UINT16    values[6];
1067 } TPMS_NV_PIN_COUNTER_PARAMETERS_mst;
1068
1069 typedef const struct TPMS_NV_PUBLIC_mst
1070 {
1071     UINT8     marshalType;
1072     UINT8     elements;
1073     UINT16    values[15];
1074 } TPMS_NV_PUBLIC_mst;
1075
1076 typedef const struct TPMS_CONTEXT_DATA_mst
1077 {
1078     UINT8     marshalType;
1079     UINT8     elements;
1080     UINT16    values[6];
1081 } TPMS_CONTEXT_DATA_mst;
1082
1083 typedef const struct TPMS_CONTEXT_mst
1084 {
1085     UINT8     marshalType;
1086     UINT8     elements;
1087     UINT16    values[12];
1088 } TPMS_CONTEXT_mst;
1089
1090 typedef const struct TPMS_CREATION_DATA_mst
1091 {
1092     UINT8     marshalType;
1093     UINT8     elements;
1094     UINT16    values[21];
1095 } TPMS_CREATION_DATA_mst;
1096
1097 typedef const struct TPM_AT_mst {
1098     UINT8     marshalType;
1099     UINT8     modifiers;

```

```

1100     UINT8      errorCode;
1101     UINT8      entries;
1102     UINT32     values[4];
1103 } TPM_AT_mst;
1104
1105 typedef const struct TPMS_AC_OUTPUT_mst
1106 {
1107     UINT8      marshalType;
1108     UINT8      elements;
1109     UINT16     values[6];
1110 } TPMS_AC_OUTPUT_mst;
1111
1112 typedef const struct Type02_mst {
1113     UINT8      marshalType;
1114     UINT8      modifiers;
1115     UINT8      errorCode;
1116     UINT32     values[2];
1117 } Type02_mst;
1118
1119 typedef const struct Type03_mst {
1120     UINT8      marshalType;
1121     UINT8      modifiers;
1122     UINT8      errorCode;
1123     UINT32     values[2];
1124 } Type03_mst;
1125
1126 typedef const struct Type04_mst {
1127     UINT8      marshalType;
1128     UINT8      modifiers;
1129     UINT8      errorCode;
1130     UINT32     values[2];
1131 } Type04_mst;
1132
1133 typedef const struct Type06_mst {
1134     UINT8      marshalType;
1135     UINT8      modifiers;
1136     UINT8      errorCode;
1137     UINT32     values[2];
1138 } Type06_mst;
1139
1140 typedef const struct Type08_mst {
1141     UINT8      marshalType;
1142     UINT8      modifiers;
1143     UINT8      errorCode;
1144     UINT32     values[2];
1145 } Type08_mst;
1146
1147 typedef const struct Type10_mst {
1148     UINT8      marshalType;
1149     UINT8      modifiers;
1150     UINT8      errorCode;
1151     UINT8      entries;
1152     UINT32     values[1];
1153 } Type10_mst;
1154
1155 typedef const struct Type11_mst {
1156     UINT8      marshalType;
1157     UINT8      modifiers;
1158     UINT8      errorCode;
1159     UINT8      entries;
1160     UINT32     values[1];
1161 } Type11_mst;
1162
1163 typedef const struct Type12_mst {
1164     UINT8      marshalType;
1165     UINT8      modifiers;

```

```
1166     UINT8      errorCode;
1167     UINT8      entries;
1168     UINT32     values[2];
1169 } Type12_mst;
1170
1171 typedef const struct Type13_mst {
1172     UINT8      marshalType;
1173     UINT8      modifiers;
1174     UINT8      errorCode;
1175     UINT8      entries;
1176     UINT32     values[1];
1177 } Type13_mst;
1178
1179 typedef const struct Type15_mst {
1180     UINT8      marshalType;
1181     UINT8      modifiers;
1182     UINT8      errorCode;
1183     UINT32     values[2];
1184 } Type15_mst;
1185
1186 typedef const struct Type17_mst {
1187     UINT8      marshalType;
1188     UINT8      modifiers;
1189     UINT8      errorCode;
1190     UINT32     values[2];
1191 } Type17_mst;
1192
1193 typedef const struct Type18_mst {
1194     UINT8      marshalType;
1195     UINT8      modifiers;
1196     UINT8      errorCode;
1197     UINT32     values[2];
1198 } Type18_mst;
1199
1200 typedef const struct Type19_mst {
1201     UINT8      marshalType;
1202     UINT8      modifiers;
1203     UINT8      errorCode;
1204     UINT32     values[2];
1205 } Type19_mst;
1206
1207 typedef const struct Type20_mst {
1208     UINT8      marshalType;
1209     UINT8      modifiers;
1210     UINT8      errorCode;
1211     UINT32     values[2];
1212 } Type20_mst;
1213
1214 typedef const struct Type22_mst {
1215     UINT8      marshalType;
1216     UINT8      modifiers;
1217     UINT8      errorCode;
1218     UINT32     values[2];
1219 } Type22_mst;
1220
1221 typedef const struct Type23_mst {
1222     UINT8      marshalType;
1223     UINT8      modifiers;
1224     UINT8      errorCode;
1225     UINT32     values[2];
1226 } Type23_mst;
1227
1228 typedef const struct Type24_mst {
1229     UINT8      marshalType;
1230     UINT8      modifiers;
1231     UINT8      errorCode;
```



```
1232     UINT32     values[2];
1233 } Type24_mst;
1234
1235 typedef const struct Type25_mst {
1236     UINT8     marshalType;
1237     UINT8     modifiers;
1238     UINT8     errorCode;
1239     UINT32     values[2];
1240 } Type25_mst;
1241
1242 typedef const struct Type26_mst {
1243     UINT8     marshalType;
1244     UINT8     modifiers;
1245     UINT8     errorCode;
1246     UINT32     values[2];
1247 } Type26_mst;
1248
1249 typedef const struct Type27_mst {
1250     UINT8     marshalType;
1251     UINT8     modifiers;
1252     UINT8     errorCode;
1253     UINT32     values[2];
1254 } Type27_mst;
1255
1256 typedef const struct Type29_mst {
1257     UINT8     marshalType;
1258     UINT8     modifiers;
1259     UINT8     errorCode;
1260     UINT32     values[2];
1261 } Type29_mst;
1262
1263 typedef const struct Type30_mst {
1264     UINT8     marshalType;
1265     UINT8     modifiers;
1266     UINT8     errorCode;
1267     UINT32     values[2];
1268 } Type30_mst;
1269
1270 typedef const struct Type33_mst {
1271     UINT8     marshalType;
1272     UINT8     modifiers;
1273     UINT8     errorCode;
1274     UINT32     values[2];
1275 } Type33_mst;
1276
1277 typedef const struct Type34_mst {
1278     UINT8     marshalType;
1279     UINT8     modifiers;
1280     UINT8     errorCode;
1281     UINT32     values[2];
1282 } Type34_mst;
1283
1284 typedef const struct Type35_mst {
1285     UINT8     marshalType;
1286     UINT8     modifiers;
1287     UINT8     errorCode;
1288     UINT32     values[2];
1289 } Type35_mst;
1290
1291 typedef const struct Type38_mst {
1292     UINT8     marshalType;
1293     UINT8     modifiers;
1294     UINT8     errorCode;
1295     UINT32     values[2];
1296 } Type38_mst;
1297
```

```

1298 typedef const struct Type41_mst {
1299     UINT8      marshalType;
1300     UINT8      modifiers;
1301     UINT8      errorCode;
1302     UINT32     values[2];
1303 } Type41_mst;
1304
1305 typedef const struct Type42_mst {
1306     UINT8      marshalType;
1307     UINT8      modifiers;
1308     UINT8      errorCode;
1309     UINT32     values[2];
1310 } Type42_mst;
1311
1312 typedef const struct Type44_mst {
1313     UINT8      marshalType;
1314     UINT8      modifiers;
1315     UINT8      errorCode;
1316     UINT32     values[2];
1317 } Type44_mst;
1318
1319 // This structure combines all the individual marshaling structures to build
1320 // something that can be referenced by offset rather than full address
1321 typedef const struct MarshalData_st {
1322     UIntMarshal_mst      UINT8_DATA;
1323     UIntMarshal_mst      UINT16_DATA;
1324     UIntMarshal_mst      UINT32_DATA;
1325     UIntMarshal_mst      UINT64_DATA;
1326     UIntMarshal_mst      INT8_DATA;
1327     UIntMarshal_mst      INT16_DATA;
1328     UIntMarshal_mst      INT32_DATA;
1329     UIntMarshal_mst      INT64_DATA;
1330     UIntMarshal_mst      UINT0_DATA;
1331     TPM_ECC_CURVE_mst     TPM_ECC_CURVE_DATA;
1332     TPM_CLOCK_ADJUST_mst TPM_CLOCK_ADJUST_DATA;
1333     TPM_EO_mst            TPM_EO_DATA;
1334     TPM_SU_mst            TPM_SU_DATA;
1335     TPM_SE_mst            TPM_SE_DATA;
1336     TPM_CAP_mst           TPM_CAP_DATA;
1337     AttributesMarshal_mst TPMA_ALGORITHM_DATA;
1338     AttributesMarshal_mst TPMA_OBJECT_DATA;
1339     AttributesMarshal_mst TPMA_SESSION_DATA;
1340     AttributesMarshal_mst TPMA_ACT_DATA;
1341     TPMI_YES_NO_mst       TPMI_YES_NO_DATA;
1342     TPMI_DH_OBJECT_mst     TPMI_DH_OBJECT_DATA;
1343     TPMI_DH_PARENT_mst     TPMI_DH_PARENT_DATA;
1344     TPMI_DH_PERSISTENT_mst TPMI_DH_PERSISTENT_DATA;
1345     TPMI_DH_ENTITY_mst     TPMI_DH_ENTITY_DATA;
1346     TPMI_DH_PCR_mst        TPMI_DH_PCR_DATA;
1347     TPMI_SH_AUTH_SESSION_mst TPMI_SH_AUTH_SESSION_DATA;
1348     TPMI_SH_HMAC_mst       TPMI_SH_HMAC_DATA;
1349     TPMI_SH_POLICY_mst     TPMI_SH_POLICY_DATA;
1350     TPMI_DH_CONTEXT_mst    TPMI_DH_CONTEXT_DATA;
1351     TPMI_DH_SAVED_mst      TPMI_DH_SAVED_DATA;
1352     TPMI_RH_HIERARCHY_mst   TPMI_RH_HIERARCHY_DATA;
1353     TPMI_RH_ENABLES_mst     TPMI_RH_ENABLES_DATA;
1354     TPMI_RH_HIERARCHY_AUTH_mst TPMI_RH_HIERARCHY_AUTH_DATA;
1355     TPMI_RH_HIERARCHY_POLICY_mst TPMI_RH_HIERARCHY_POLICY_DATA;
1356     TPMI_RH_BASE_HIERARCHY_mst TPMI_RH_BASE_HIERARCHY_DATA;
1357     TPMI_RH_PLATFORM_mst    TPMI_RH_PLATFORM_DATA;
1358     TPMI_RH_OWNER_mst       TPMI_RH_OWNER_DATA;
1359     TPMI_RH_ENDORSEMENT_mst TPMI_RH_ENDORSEMENT_DATA;
1360     TPMI_RH_PROVISION_mst    TPMI_RH_PROVISION_DATA;
1361     TPMI_RH_CLEAR_mst        TPMI_RH_CLEAR_DATA;
1362     TPMI_RH_NV_AUTH_mst      TPMI_RH_NV_AUTH_DATA;
1363     TPMI_RH_LOCKOUT_mst      TPMI_RH_LOCKOUT_DATA;

```

1364	TPMI_RH_NV_INDEX_mst	TPMI_RH_NV_INDEX_DATA;
1365	TPMI_RH_AC_mst	TPMI_RH_AC_DATA;
1366	TPMI_RH_ACT_mst	TPMI_RH_ACT_DATA;
1367	TPMI_ALG_HASH_mst	TPMI_ALG_HASH_DATA;
1368	TPMI_ALG_ASYM_mst	TPMI_ALG_ASYM_DATA;
1369	TPMI_ALG_SYM_mst	TPMI_ALG_SYM_DATA;
1370	TPMI_ALG_SYM_OBJECT_mst	TPMI_ALG_SYM_OBJECT_DATA;
1371	TPMI_ALG_SYM_MODE_mst	TPMI_ALG_SYM_MODE_DATA;
1372	TPMI_ALG_KDF_mst	TPMI_ALG_KDF_DATA;
1373	TPMI_ALG_SIG_SCHEME_mst	TPMI_ALG_SIG_SCHEME_DATA;
1374	TPMI_ECC_KEY_EXCHANGE_mst	TPMI_ECC_KEY_EXCHANGE_DATA;
1375	TPMI_ST_COMMAND_TAG_mst	TPMI_ST_COMMAND_TAG_DATA;
1376	TPMI_ALG_MAC_SCHEME_mst	TPMI_ALG_MAC_SCHEME_DATA;
1377	TPMI_ALG_CIPHER_MODE_mst	TPMI_ALG_CIPHER_MODE_DATA;
1378	TPMS_EMPTY_mst	TPMS_EMPTY_DATA;
1379	TPMS_ALGORITHM_DESCRIPTION_mst	TPMS_ALGORITHM_DESCRIPTION_DATA;
1380	TPMU_HA_mst	TPMU_HA_DATA;
1381	TPMT_HA_mst	TPMT_HA_DATA;
1382	Tpm2bMarshal_mst	TPM2B_DIGEST_DATA;
1383	Tpm2bMarshal_mst	TPM2B_DATA_DATA;
1384	Tpm2bMarshal_mst	TPM2B_EVENT_DATA;
1385	Tpm2bMarshal_mst	TPM2B_MAX_BUFFER_DATA;
1386	Tpm2bMarshal_mst	TPM2B_MAX_NV_BUFFER_DATA;
1387	Tpm2bMarshal_mst	TPM2B_TIMEOUT_DATA;
1388	Tpm2bMarshal_mst	TPM2B_IV_DATA;
1389	NullUnionMarshal_mst	NULL_UNION_DATA;
1390	Tpm2bMarshal_mst	TPM2B_NAME_DATA;
1391	TPMS_PCR_SELECT_mst	TPMS_PCR_SELECT_DATA;
1392	TPMS_PCR_SELECTION_mst	TPMS_PCR_SELECTION_DATA;
1393	TPMT_TK_CREATION_mst	TPMT_TK_CREATION_DATA;
1394	TPMT_TK_VERIFIED_mst	TPMT_TK_VERIFIED_DATA;
1395	TPMT_TK_AUTH_mst	TPMT_TK_AUTH_DATA;
1396	TPMT_TK_HASHCHECK_mst	TPMT_TK_HASHCHECK_DATA;
1397	TPMS_ALG_PROPERTY_mst	TPMS_ALG_PROPERTY_DATA;
1398	TPMS_TAGGED_PROPERTY_mst	TPMS_TAGGED_PROPERTY_DATA;
1399	TPMS_TAGGED_PCR_SELECT_mst	TPMS_TAGGED_PCR_SELECT_DATA;
1400	TPMS_TAGGED_POLICY_mst	TPMS_TAGGED_POLICY_DATA;
1401	TPMS_ACT_DATA_mst	TPMS_ACT_DATA_DATA;
1402	ListMarshal_mst	TPML_CC_DATA;
1403	ListMarshal_mst	TPML_CCA_DATA;
1404	ListMarshal_mst	TPML_ALG_DATA;
1405	ListMarshal_mst	TPML_HANDLE_DATA;
1406	ListMarshal_mst	TPML_DIGEST_DATA;
1407	ListMarshal_mst	TPML_DIGEST_VALUES_DATA;
1408	ListMarshal_mst	TPML_PCR_SELECTION_DATA;
1409	ListMarshal_mst	TPML_ALG_PROPERTY_DATA;
1410	ListMarshal_mst	TPML_TAGGED_TPM_PROPERTY_DATA;
1411	ListMarshal_mst	TPML_TAGGED_PCR_PROPERTY_DATA;
1412	ListMarshal_mst	TPML_ECC_CURVE_DATA;
1413	ListMarshal_mst	TPML_TAGGED_POLICY_DATA;
1414	ListMarshal_mst	TPML_ACT_DATA_DATA;
1415	TPMU_CAPABILITIES_mst	TPMU_CAPABILITIES_DATA;
1416	TPMS_CAPABILITY_DATA_mst	TPMS_CAPABILITY_DATA_DATA;
1417	TPMS_CLOCK_INFO_mst	TPMS_CLOCK_INFO_DATA;
1418	TPMS_TIME_INFO_mst	TPMS_TIME_INFO_DATA;
1419	TPMS_TIME_ATTEST_INFO_mst	TPMS_TIME_ATTEST_INFO_DATA;
1420	TPMS_CERTIFY_INFO_mst	TPMS_CERTIFY_INFO_DATA;
1421	TPMS_QUOTE_INFO_mst	TPMS_QUOTE_INFO_DATA;
1422	TPMS_COMMAND_AUDIT_INFO_mst	TPMS_COMMAND_AUDIT_INFO_DATA;
1423	TPMS_SESSION_AUDIT_INFO_mst	TPMS_SESSION_AUDIT_INFO_DATA;
1424	TPMS_CREATION_INFO_mst	TPMS_CREATION_INFO_DATA;
1425	TPMS_NV_CERTIFY_INFO_mst	TPMS_NV_CERTIFY_INFO_DATA;
1426	TPMS_NV_DIGEST_CERTIFY_INFO_mst	TPMS_NV_DIGEST_CERTIFY_INFO_DATA;
1427	TPMI_ST_ATTEST_mst	TPMI_ST_ATTEST_DATA;
1428	TPMU_ATTEST_mst	TPMU_ATTEST_DATA;
1429	TPMS_ATTEST_mst	TPMS_ATTEST_DATA;

1430	Tpm2bMarshal_mst	TPM2B_ATTEST_DATA;
1431	TPMS_AUTH_COMMAND_mst	TPMS_AUTH_COMMAND_DATA;
1432	TPMS_AUTH_RESPONSE_mst	TPMS_AUTH_RESPONSE_DATA;
1433	TPMI_AES_KEY_BITS_mst	TPMI_AES_KEY_BITS_DATA;
1434	TPMI_SM4_KEY_BITS_mst	TPMI_SM4_KEY_BITS_DATA;
1435	TPMI_CAMELLIA_KEY_BITS_mst	TPMI_CAMELLIA_KEY_BITS_DATA;
1436	TPMU_SYM_KEY_BITS_mst	TPMU_SYM_KEY_BITS_DATA;
1437	TPMU_SYM_MODE_mst	TPMU_SYM_MODE_DATA;
1438	TPMT_SYM_DEF_mst	TPMT_SYM_DEF_DATA;
1439	TPMT_SYM_DEF_OBJECT_mst	TPMT_SYM_DEF_OBJECT_DATA;
1440	Tpm2bMarshal_mst	TPM2B_SYM_KEY_DATA;
1441	TPMS_SYMCIPHER_PARMS_mst	TPMS_SYMCIPHER_PARMS_DATA;
1442	Tpm2bMarshal_mst	TPM2B_LABEL_DATA;
1443	TPMS_DERIVE_mst	TPMS_DERIVE_DATA;
1444	Tpm2bMarshal_mst	TPM2B_DERIVE_DATA;
1445	Tpm2bMarshal_mst	TPM2B_SENSITIVE_DATA_DATA;
1446	TPMS_SENSITIVE_CREATE_mst	TPMS_SENSITIVE_CREATE_DATA;
1447	Tpm2bsMarshal_mst	TPM2B_SENSITIVE_CREATE_DATA;
1448	TPMS_SCHEME_HASH_mst	TPMS_SCHEME_HASH_DATA;
1449	TPMS_SCHEME_ECDSA_mst	TPMS_SCHEME_ECDSA_DATA;
1450	TPMI_ALG_KEYEDHASH_SCHEME_mst	TPMI_ALG_KEYEDHASH_SCHEME_DATA;
1451	TPMS_SCHEME_XOR_mst	TPMS_SCHEME_XOR_DATA;
1452	TPMU_SCHEME_KEYEDHASH_mst	TPMU_SCHEME_KEYEDHASH_DATA;
1453	TPMT_KEYEDHASH_SCHEME_mst	TPMT_KEYEDHASH_SCHEME_DATA;
1454	TPMU_SIG_SCHEME_mst	TPMU_SIG_SCHEME_DATA;
1455	TPMT_SIG_SCHEME_mst	TPMT_SIG_SCHEME_DATA;
1456	TPMU_KDF_SCHEME_mst	TPMU_KDF_SCHEME_DATA;
1457	TPMT_KDF_SCHEME_mst	TPMT_KDF_SCHEME_DATA;
1458	TPMI_ALG_ASYM_SCHEME_mst	TPMI_ALG_ASYM_SCHEME_DATA;
1459	TPMU_ASYM_SCHEME_mst	TPMU_ASYM_SCHEME_DATA;
1460	TPMI_ALG_RSA_SCHEME_mst	TPMI_ALG_RSA_SCHEME_DATA;
1461	TPMT_RSA_SCHEME_mst	TPMT_RSA_SCHEME_DATA;
1462	TPMI_ALG_RSA_DECRYPT_mst	TPMI_ALG_RSA_DECRYPT_DATA;
1463	TPMT_RSA_DECRYPT_mst	TPMT_RSA_DECRYPT_DATA;
1464	Tpm2bMarshal_mst	TPM2B_PUBLIC_KEY_RSA_DATA;
1465	TPMI_RSA_KEY_BITS_mst	TPMI_RSA_KEY_BITS_DATA;
1466	Tpm2bMarshal_mst	TPM2B_PRIVATE_KEY_RSA_DATA;
1467	Tpm2bMarshal_mst	TPM2B_ECC_PARAMETER_DATA;
1468	TPMS_ECC_POINT_mst	TPMS_ECC_POINT_DATA;
1469	Tpm2bsMarshal_mst	TPM2B_ECC_POINT_DATA;
1470	TPMI_ALG_ECC_SCHEME_mst	TPMI_ALG_ECC_SCHEME_DATA;
1471	TPMI_ECC_CURVE_mst	TPMI_ECC_CURVE_DATA;
1472	TPMT_ECC_SCHEME_mst	TPMT_ECC_SCHEME_DATA;
1473	TPMS_ALGORITHM_DETAIL_ECC_mst	TPMS_ALGORITHM_DETAIL_ECC_DATA;
1474	TPMS_SIGNATURE_RSA_mst	TPMS_SIGNATURE_RSA_DATA;
1475	TPMS_SIGNATURE_ECC_mst	TPMS_SIGNATURE_ECC_DATA;
1476	TPMU_SIGNATURE_mst	TPMU_SIGNATURE_DATA;
1477	TPMT_SIGNATURE_mst	TPMT_SIGNATURE_DATA;
1478	TPMU_ENCRYPTED_SECRET_mst	TPMU_ENCRYPTED_SECRET_DATA;
1479	Tpm2bMarshal_mst	TPM2B_ENCRYPTED_SECRET_DATA;
1480	TPMI_ALG_PUBLIC_mst	TPMI_ALG_PUBLIC_DATA;
1481	TPMU_PUBLIC_ID_mst	TPMU_PUBLIC_ID_DATA;
1482	TPMS_KEYEDHASH_PARMS_mst	TPMS_KEYEDHASH_PARMS_DATA;
1483	TPMS_RSA_PARMS_mst	TPMS_RSA_PARMS_DATA;
1484	TPMS_ECC_PARMS_mst	TPMS_ECC_PARMS_DATA;
1485	TPMU_PUBLIC_PARMS_mst	TPMU_PUBLIC_PARMS_DATA;
1486	TPMT_PUBLIC_PARMS_mst	TPMT_PUBLIC_PARMS_DATA;
1487	TPMT_PUBLIC_mst	TPMT_PUBLIC_DATA;
1488	Tpm2bsMarshal_mst	TPM2B_PUBLIC_DATA;
1489	Tpm2bMarshal_mst	TPM2B_TEMPLATE_DATA;
1490	Tpm2bMarshal_mst	TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA;
1491	TPMU_SENSITIVE_COMPOSITE_mst	TPMU_SENSITIVE_COMPOSITE_DATA;
1492	TPMT_SENSITIVE_mst	TPMT_SENSITIVE_DATA;
1493	Tpm2bsMarshal_mst	TPM2B_SENSITIVE_DATA;
1494	Tpm2bMarshal_mst	TPM2B_PRIVATE_DATA;
1495	Tpm2bMarshal_mst	TPM2B_ID_OBJECT_DATA;

```

1496     TPMS_NV_PIN_COUNTER_PARAMETERS_mst TPMS_NV_PIN_COUNTER_PARAMETERS_DATA;
1497     AttributesMarshal_mst               TPMA_NV_DATA;
1498     TPMS_NV_PUBLIC_mst                  TPMS_NV_PUBLIC_DATA;
1499     Tpm2bsMarshal_mst                   TPM2B_NV_PUBLIC_DATA;
1500     Tpm2bMarshal_mst                    TPM2B_CONTEXT_SENSITIVE_DATA;
1501     TPMS_CONTEXT_DATA_mst              TPMS_CONTEXT_DATA_DATA;
1502     Tpm2bMarshal_mst                   TPM2B_CONTEXT_DATA_DATA;
1503     TPMS_CONTEXT_mst                    TPMS_CONTEXT_DATA;
1504     TPMS_CREATION_DATA_mst              TPMS_CREATION_DATA_DATA;
1505     Tpm2bsMarshal_mst                   TPM2B_CREATION_DATA_DATA;
1506     TPM_AT_mst                          TPM_AT_DATA;
1507     TPMS_AC_OUTPUT_mst                  TPMS_AC_OUTPUT_DATA;
1508     ListMarshal_mst                     TPML_AC_CAPABILITIES_DATA;
1509     MinMaxMarshal_mst                   Type00_DATA;
1510     MinMaxMarshal_mst                   Type01_DATA;
1511     Type02_mst                          Type02_DATA;
1512     Type03_mst                          Type03_DATA;
1513     Type04_mst                          Type04_DATA;
1514     MinMaxMarshal_mst                   Type05_DATA;
1515     Type06_mst                          Type06_DATA;
1516     MinMaxMarshal_mst                   Type07_DATA;
1517     Type08_mst                          Type08_DATA;
1518     Type10_mst                          Type10_DATA;
1519     Type11_mst                          Type11_DATA;
1520     Type12_mst                          Type12_DATA;
1521     Type13_mst                          Type13_DATA;
1522     Type15_mst                          Type15_DATA;
1523     Type17_mst                          Type17_DATA;
1524     Type18_mst                          Type18_DATA;
1525     Type19_mst                          Type19_DATA;
1526     Type20_mst                          Type20_DATA;
1527     Type22_mst                          Type22_DATA;
1528     Type23_mst                          Type23_DATA;
1529     Type24_mst                          Type24_DATA;
1530     Type25_mst                          Type25_DATA;
1531     Type26_mst                          Type26_DATA;
1532     Type27_mst                          Type27_DATA;
1533     MinMaxMarshal_mst                   Type28_DATA;
1534     Type29_mst                          Type29_DATA;
1535     Type30_mst                          Type30_DATA;
1536     MinMaxMarshal_mst                   Type31_DATA;
1537     MinMaxMarshal_mst                   Type32_DATA;
1538     Type33_mst                          Type33_DATA;
1539     Type34_mst                          Type34_DATA;
1540     Type35_mst                          Type35_DATA;
1541     MinMaxMarshal_mst                   Type36_DATA;
1542     MinMaxMarshal_mst                   Type37_DATA;
1543     Type38_mst                          Type38_DATA;
1544     MinMaxMarshal_mst                   Type39_DATA;
1545     MinMaxMarshal_mst                   Type40_DATA;
1546     Type41_mst                          Type41_DATA;
1547     Type42_mst                          Type42_DATA;
1548     MinMaxMarshal_mst                   Type43_DATA;
1549     Type44_mst                          Type44_DATA;
1550 } MarshalData_st;
1551
1552 #endif // _TABLE_MARSHAL_TYPES_H_
1553 // clang-format on

```

6.43 /tpm/include/private/Tpm.h

```

1 // Root header file for building any TPM.lib code
2
3 #ifndef _TPM_H_
4 #define _TPM_H_

```

```

5 // TODO_RENAME_INC_FOLDER: public refers to the TPM_CoreLib public headers
6 #include <public/tpm_public.h>
7
8 #include "TpmAlgorithmDefines.h"
9 #include "LibSupport.h" // Types from the library. These need to come before
10 // Global.h because some of the structures in
11 // that file depend on the structures used by the
12 // cryptographic libraries.
13 #include "GpMacros.h" // Define additional macros
14 #include "Global.h" // Define other TPM types
15 #include "InternalRoutines.h" // Function prototypes
16
17 #endif // _TPM_H_

```

6.44 /tpm/include/private/TpmASN1.h

```

1 /** Introduction
2 // This file contains the macro and structure definitions for the X509 commands and
3 // functions.
4
5 #ifndef TPMASN1_H
6 #define TPMASN1_H
7
8 /** Includes
9
10 #include "Tpm.h"
11 #include "OIDs.h"
12
13 /** Defined Constants
14 /**** ASN.1 Universal Types (Class 00b)
15 #define ASN1_EOC 0x00
16 #define ASN1_BOOLEAN 0x01
17 #define ASN1_INTEGER 0x02
18 #define ASN1_BITSTRING 0x03
19 #define ASN1_OCTET_STRING 0x04
20 #define ASN1_NULL 0x05
21 #define ASN1_OBJECT_IDENTIFIER 0x06
22 #define ASN1_OBJECT_DESCRIPTOR 0x07
23 #define ASN1_EXTERNAL 0x08
24 #define ASN1_REAL 0x09
25 #define ASN1_ENUMERATED 0x0A
26 #define ASN1_EMBEDDED 0x0B
27 #define ASN1_UTF8String 0x0C
28 #define ASN1_RELATIVE_OID 0x0D
29 #define ASN1_SEQUENCE 0x10 // Primitive + Constructed + 0x10
30 #define ASN1_SET 0x11 // Primitive + Constructed + 0x11
31 #define ASN1_NumericString 0x12
32 #define ASN1_PrintableString 0x13
33 #define ASN1_T61String 0x14
34 #define ASN1_VideoString 0x15
35 #define ASN1_IA5String 0x16
36 #define ASN1_UTCTime 0x17
37 #define ASN1_GeneralizeTime 0x18
38 #define ASN1_VisibleString 0x1A
39 #define ASN1_GeneralString 0x1B
40 #define ASN1_UniversalString 0x1C
41 #define ASN1_CHARACTER STRING 0x1D
42 #define ASN1_BMPString 0x1E
43 #define ASN1_CONSTRUCTED 0x20
44
45 #define ASN1_APPLICATION_SPECIFIC 0xA0
46
47 #define ASN1_CONSTRUCTED_SEQUENCE (ASN1_SEQUENCE + ASN1_CONSTRUCTED)
48
49 #define MAX_DEPTH 10 // maximum push depth for marshaling context.

```



```

50
51 /** Macros
52
53 /**** Unmarshaling Macros
54 #ifndef GOTO_ERROR_UNLESS
55 # error missing GOTO_ERROR_UNLESS definition
56 #endif
57
58 // Checks the validity of the size making sure that there is no wrap around
59 #define CHECK_SIZE(context, length) \
60     GOTO_ERROR_UNLESS(((length) + (context)->offset) >= (context)->offset) \
61     && (((length) + (context)->offset) <= (context)->size))
62 #define NEXT_OCTET(context) ((context)->buffer[(context)->offset++])
63 #define PEEK_NEXT(context) ((context)->buffer[(context)->offset])
64
65 /**** Marshaling Macros
66
67 // Marshaling works in reverse order. The offset is set to the top of the buffer and,
68 // as the buffer is filled, 'offset' counts down to zero. When the full thing is
69 // encoded it can be moved to the top of the buffer. This happens when the last
70 // context is closed.
71
72 #define CHECK_SPACE(context, length) GOTO_ERROR_UNLESS(context->offset > length)
73
74 /** Structures
75
76 typedef struct ASN1UnmarshalContext
77 {
78     BYTE* buffer; // pointer to the buffer
79     INT16 size; // size of the buffer (a negative number indicates
80                // a parsing failure).
81     INT16 offset; // current offset into the buffer (a negative number
82                  // indicates a parsing failure). Not used
83     BYTE tag; // The last unmarshaled tag
84 } ASN1UnmarshalContext;
85
86 typedef struct ASN1MarshalContext
87 {
88     BYTE* buffer; // pointer to the start of the buffer
89     INT16 offset; // place on the top where the last entry was added
90                  // items are added from the bottom up.
91     INT16 end; // the end offset of the current value
92     INT16 depth; // how many pushed end values.
93     INT16 ends[MAX_DEPTH];
94 } ASN1MarshalContext;
95
96 #endif // _TPMASN1_H_

```

6.45 /tpm/include/private/X509.h

```

1 /**** Introduction
2 // This file contains the macro and structure definitions for the X509 commands and
3 // functions.
4
5 #ifndef _X509_H_
6 #define _X509_H_
7
8 /**** Includes
9
10 #include "Tpm.h"
11 #include "TpmASN1.h"
12
13 /**** Defined Constants
14
15 /**** X509 Application-specific types

```

```

16 #define X509_SELECTION          0xA0
17 #define X509_ISSUER_UNIQUE_ID   0xA1
18 #define X509_SUBJECT_UNIQUE_ID  0xA2
19 #define X509_EXTENSIONS         0xA3
20
21 // These defines give the order in which values appear in the TBSCertificate
22 // of an x.509 certificate. These values are used to index into an array of
23 //
24 #define ENCODED_SIZE_REF        0
25 #define VERSION_REF             (ENCODED_SIZE_REF + 1)
26 #define SERIAL_NUMBER_REF      (VERSION_REF + 1)
27 #define SIGNATURE_REF          (SERIAL_NUMBER_REF + 1)
28 #define ISSUER_REF              (SIGNATURE_REF + 1)
29 #define VALIDITY_REF           (ISSUER_REF + 1)
30 #define SUBJECT_KEY_REF        (VALIDITY_REF + 1)
31 #define SUBJECT_PUBLIC_KEY_REF (SUBJECT_KEY_REF + 1)
32 #define EXTENSIONS_REF         (SUBJECT_PUBLIC_KEY_REF + 1)
33 #define REF_COUNT              (EXTENSIONS_REF + 1)
34
35 /** Structures
36
37 // Used to access the fields of a TBSSignature some of which are in the in_CertifyX509
38 // structure and some of which are in the out_CertifyX509 structure.
39 typedef struct stringRef
40 {
41     BYTE* buf;
42     INT16 len;
43 } stringRef;
44
45 // This is defined to avoid bit by bit comparisons within a UINT32
46 typedef union x509KeyUsageUnion
47 {
48     TPMA_X509_KEY_USAGE x509;
49     UINT32 integer;
50 } x509KeyUsageUnion;
51
52 /** Global X509 Constants
53 // These values are instanced by X509_spt.c and referenced by other X509-related
54 // files.
55
56 // This is the DER-encoded value for the Key Usage OID (2.5.29.15). This is the
57 // full OID, not just the numeric value
58 #define OID_KEY_USAGE_EXTENSION_VALUE 0x06, 0x03, 0x55, 0x1D, 0x0F
59 MAKE_OID(_KEY_USAGE_EXTENSION);
60
61 // This is the DER-encoded value for the TCG-defined TPMA_OBJECT OID
62 // (2.23.133.10.1.1)
63 #define OID_TCG_TPMA_OBJECT_VALUE 0x06, 0x07, 0x67, 0x81, 0x05, 0x0a, 0x01, 0x01, 0x01
64 MAKE_OID(_TCG_TPMA_OBJECT);
65
66 #ifdef _X509_SPT_
67 // If a bit is SET in KEY_USAGE_SIGN is also SET in keyUsage then
68 // the associated key has to have 'sign' SET.
69 const x509KeyUsageUnion KEY_USAGE_SIGN = {TPMA_X509_KEY_USAGE_INITIALIZER(
70     /* bits_at_0 */ 0,
71     /* decipheronly */ 0,
72     /* encipheronly */ 0,
73     /* crlsign */ 1,
74     /* keycertsign */ 1,
75     /* keyagreement */ 0,
76     /* dataencipherment */ 0,
77     /* keyencipherment */ 0,
78     /* nonrepudiation */ 0,
79     /* digitalsignature */ 1});
80 // If a bit is SET in KEY_USAGE_DECRYPT is also SET in keyUsage then
81 // the associated key has to have 'decrypt' SET.

```

```

82  const x509KeyUsageUnion KEY_USAGE_DECRYPT = {TPMA_X509_KEY_USAGE_INITIALIZER(
83      /* bits_at_0      */ 0,
84      /* decipheronly    */ 1,
85      /* encipheronly    */ 1,
86      /* crlsign         */ 0,
87      /* keycertsign     */ 0,
88      /* keyagreement    */ 1,
89      /* dataencipherment */ 1,
90      /* keyencipherment */ 1,
91      /* nonrepudiation  */ 0,
92      /* digitalsignature */ 0});
93  #else
94  extern x509KeyUsageUnion KEY_USAGE_SIGN;
95  extern x509KeyUsageUnion KEY_USAGE_DECRYPT;
96  #endif
97
98  #endif // _X509_H_

```

6.46 /tpm/include/private/prototypes/ActivateCredential_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_ActivateCredential // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ACTIVATECREDENTIAL_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ACTIVATECREDENTIAL_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT activateHandle;
12     TPMI_DH_OBJECT keyHandle;
13     TPM2B_ID_OBJECT credentialBlob;
14     TPM2B_ENCRYPTED_SECRET secret;
15 } ActivateCredential_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_DIGEST certInfo;
21 } ActivateCredential_Out;
22
23 // Response code modifiers
24 #   define RC_ActivateCredential_activateHandle (TPM_RC_H + TPM_RC_1)
25 #   define RC_ActivateCredential_keyHandle      (TPM_RC_H + TPM_RC_2)
26 #   define RC_ActivateCredential_credentialBlob (TPM_RC_P + TPM_RC_1)
27 #   define RC_ActivateCredential_secret        (TPM_RC_P + TPM_RC_2)
28
29 // Function prototype
30 TPM_RC
31 TPM2_ActivateCredential(ActivateCredential_In* in, ActivateCredential_Out* out);
32
33 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ACTIVATECREDENTIAL_FP_H_
34 #endif // CC_ActivateCredential

```

6.47 /tpm/include/private/prototypes/ACT_SetTimeout_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_ACT_SetTimeout // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ACT_SETTIMEOUT_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ACT_SETTIMEOUT_FP_H_
7

```

```

8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_ACT actHandle;
12     UINT32      startTimeout;
13 } ACT_SetTimeout_In;
14
15 // Response code modifiers
16 # define RC_ACT_SetTimeout_actHandle (TPM_RC_H + TPM_RC_1)
17 # define RC_ACT_SetTimeout_startTimeout (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_ACT_SetTimeout(ACT_SetTimeout_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ACT_SETTIMEOUT_FP_H
24 #endif // CC_ACT_SetTimeout

```

6.48 /tpm/include/private/prototypes/ACT_spt_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes 1.00
3 * Date: Oct 24, 2019 Time: 10:38:43AM
4 */
5
6 #ifndef _ACT_SPT_FP_H
7 #define _ACT_SPT_FP_H
8
9 /*** ActStartup()
10 // This function is called by TPM2_Startup() to initialize the ACT counter values.
11 BOOL ActStartup(STARTUP_TYPE type);
12
13 /*** ActGetSignaled()
14 // This function returns the state of the signaled flag associated with an ACT.
15 BOOL ActGetSignaled(TPM_RH actHandle);
16
17 /*** ActShutdown()
18 // This function saves the current state of the counters
19 BOOL ActShutdown(TPM_SU state //IN: the type of the shutdown.
20 );
21
22 /*** ActIsImplemented()
23 // This function determines if an ACT is implemented in both the TPM and the platform
24 // code.
25 BOOL ActIsImplemented(UINT32 act);
26
27 /*** ActCounterUpdate()
28 // This function updates the ACT counter. If the counter already has a pending update,
29 // it returns TPM_RC_RETRY so that the update can be tried again later.
30 TPM_RC
31 ActCounterUpdate(TPM_RH handle, //IN: the handle of the act
32                 UINT32 newValue //IN: the value to set in the ACT
33 );
34
35 /*** ActGetCapabilityData()
36 // This function returns the list of ACT data
37 // Return Type: TPMI_YES_NO
38 // YES if more ACT data is available
39 // NO if no more ACT data to
40 TPMI_YES_NO
41 ActGetCapabilityData(TPM_HANDLE actHandle, // IN: the handle for the starting ACT
42                    UINT32 maxCount, // IN: maximum allowed return values
43                    TPML_ACT_DATA* actList // OUT: ACT data list
44 );
45

```

```

46  /*** ActGetOneCapability()
47  // This function returns an ACT's capability, if present.
48  BOOL ActGetOneCapability(TPM_HANDLE    actHandle, // IN: the handle for the ACT
49                          TPMS_ACT_DATA* actData  // OUT: ACT data
50  );
51
52  #endif // _ACT_SPT_FP_H_

```

6.49 /tpm/include/private/prototypes/AC_GetCapability_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_AC_GetCapability // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_AC_GETCAPABILITY_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_AC_GETCAPABILITY_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_AC ac;
12     TPM_AT      capability;
13     UINT32      count;
14 } AC_GetCapability_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPMI_YES_NO      moreData;
20     TPML_AC_CAPABILITIES capabilitiesData;
21 } AC_GetCapability_Out;
22
23 // Response code modifiers
24 #   define RC_AC_GetCapability_ac      (TPM_RC_H + TPM_RC_1)
25 #   define RC_AC_GetCapability_capability (TPM_RC_P + TPM_RC_1)
26 #   define RC_AC_GetCapability_count   (TPM_RC_P + TPM_RC_2)
27
28 // Function prototype
29 TPM_RC
30 TPM2_AC_GetCapability(AC_GetCapability_In* in, AC_GetCapability_Out* out);
31
32 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_AC_GETCAPABILITY_FP_H_
33 #endif // CC_AC_GetCapability

```

6.50 /tpm/include/private/prototypes/AC_Send_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_AC_Send // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_AC_SEND_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_AC_SEND_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT  sendObject;
12     TPMI_RH_NV_AUTH  authHandle;
13     TPMI_RH_AC       ac;
14     TPM2B_MAX_BUFFER acDataIn;
15 } AC_Send_In;
16
17 // Output structure definition
18 typedef struct

```

```

19 {
20     TPMS_AC_OUTPUT acDataOut;
21 } AC_Send_Out;
22
23 // Response code modifiers
24 # define RC_AC_Send_sendObject (TPM_RC_H + TPM_RC_1)
25 # define RC_AC_Send_authHandle (TPM_RC_H + TPM_RC_2)
26 # define RC_AC_Send_ac          (TPM_RC_H + TPM_RC_3)
27 # define RC_AC_Send_acDataIn    (TPM_RC_P + TPM_RC_1)
28
29 // Function prototype
30 TPM_RC
31 TPM2_AC_Send(AC_Send_In* in, AC_Send_Out* out);
32
33 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_AC_SEND_FP_H_
34 #endif // CC_AC_Send

```

6.51 /tpm/include/private/prototypes/AC_spt_fp.h

```

1  /*(Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Mar 4, 2020 Time: 02:36:44PM
4   */
5
6 #ifndef _AC_SPT_FP_H_
7 #define _AC_SPT_FP_H_
8
9 /*** AcToCapabilities()
10 // This function returns a pointer to a list of AC capabilities.
11 TPML_AC_CAPABILITIES* AcToCapabilities(TPMI_RH_AC component // IN: component
12 );
13
14 /*** AcIsAccessible()
15 // Function to determine if an AC handle references an actual AC
16 // Return Type: BOOL
17 BOOL AcIsAccessible(TPM_HANDLE acHandle);
18
19 /*** AcCapabilitiesGet()
20 // This function returns a list of capabilities associated with an AC
21 // Return Type: TPMI_YES_NO
22 // YES if there are more handles available
23 // NO all the available handles has been returned
24 TPMI_YES_NO
25 AcCapabilitiesGet(TPMI_RH_AC component, // IN: the component
26                 TPM_AT type, // IN: start capability type
27                 UINT32 count, // IN: requested number
28                 TPML_AC_CAPABILITIES* capabilityList // OUT: list of handle
29 );
30
31 /*** AcSendObject()
32 // Stub to handle sending of an AC object
33 // Return Type: TPM_RC
34 TPM_RC
35 AcSendObject(TPM_HANDLE acHandle, // IN: Handle of AC receiving object
36             OBJECT* object, // IN: object structure to send
37             TPMS_AC_OUTPUT* acDataOut // OUT: results of operation
38 );
39
40 #endif // _AC_SPT_FP_H_

```

6.52 /tpm/include/private/prototypes/AlgorithmCap_fp.h

```

1  /*(Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017

```



```

3  *   Date: Mar 28, 2019   Time: 08:25:19PM
4  */
5
6  #ifndef _ALGORITHM_CAP_FP_H_
7  #define _ALGORITHM_CAP_FP_H_
8
9  /** AlgorithmCapGetImplemented()
10 // This function is used by TPM2_GetCapability() to return a list of the
11 // implemented algorithms.
12 //
13 // Return Type: TPMT_YES_NO
14 // YES         more algorithms to report
15 // NO         no more algorithms to report
16 TPMT_YES_NO
17 AlgorithmCapGetImplemented(TPM_ALG_ID algID, // IN: the starting algorithm ID
18                          UINT32 count, // IN: count of returned algorithms
19                          TPML_ALG_PROPERTY* algList // OUT: algorithm list
20 );
21
22 /** AlgorithmCapGetOneImplemented()
23 // This function returns whether a single algorithm was implemented, along
24 // with its properties (if implemented).
25 BOOL AlgorithmCapGetOneImplemented(
26     TPM_ALG_ID algID, // IN: the algorithm ID
27     TPMS_ALG_PROPERTY* algProperty // OUT: algorithm properties
28 );
29
30 /** AlgorithmGetImplementedVector()
31 // This function returns the bit vector of the implemented algorithms.
32 LIB_EXPORT
33 void AlgorithmGetImplementedVector(
34     ALGORITHM_VECTOR* implemented // OUT: the implemented bits are SET
35 );
36
37 #endif // _ALGORITHM_CAP_FP_H_

```

6.53 /tpm/include/private/prototypes/AlgorithmTests_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 4, 2020   Time: 02:36:44PM
4  */
5
6  #ifndef _ALGORITHM_TESTS_FP_H_
7  #define _ALGORITHM_TESTS_FP_H_
8
9  #if SELF_TEST
10
11 /** TestAlgorithm()
12 // Dispatches to the correct test function for the algorithm or gets a list of
13 // testable algorithms.
14 //
15 // If 'toTest' is not NULL, then the test decisions are based on the algorithm
16 // selections in 'toTest'. Otherwise, 'g_toTest' is used. When bits are clear in
17 // 'g_toTest' they will also be cleared 'toTest'.
18 //
19 // If there doesn't happen to be a test for the algorithm, its associated bit is
20 // quietly cleared.
21 //
22 // If 'alg' is zero (TPM_ALG_ERROR), then the toTest vector is cleared of any bits
23 // for which there is no test (i.e. no tests are actually run but the vector is
24 // cleared).
25 //
26 // Note: 'toTest' will only ever have bits set for implemented algorithms but 'alg'
27 // can be anything.

```

```

28 // Return Type: TPM_RC
29 //     TPM_RC_CANCELED    test was canceled
30 LIB_EXPORT
31 TPM_RC
32 TestAlgorithm(TPM_ALG_ID alg, ALGORITHM_VECTOR* toTest);
33 #endif // SELF_TESTS
34
35 #endif // _ALGORITHM_TESTS_FP_H

```

6.54 /tpm/include/private/prototypes/Attest_spt_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 28, 2019 Time: 08:25:18PM
4  */
5
6 #ifndef _ATTEST_SPT_FP_H
7 #define _ATTEST_SPT_FP_H
8
9 /***FillInAttestInfo()
10 // Fill in common fields of TPMS_ATTEST structure.
11 void FillInAttestInfo(
12     TPMI_DH_OBJECT    signHandle, // IN: handle of signing object
13     TPMT_SIG_SCHEME*  scheme,     // IN/OUT: scheme to be used for signing
14     TPM2B_DATA*       data,       // IN: qualifying data
15     TPMS_ATTEST*      attest      // OUT: attest structure
16 );
17
18 /***SignAttestInfo()
19 // Sign a TPMS_ATTEST structure. If signHandle is TPM_RH_NULL, a null signature
20 // is returned.
21 //
22 // Return Type: TPM_RC
23 //     TPM_RC_ATTRIBUTES 'signHandle' references not a signing key
24 //     TPM_RC_SCHEME     'scheme' is not compatible with 'signHandle' type
25 //     TPM_RC_VALUE      digest generated for the given 'scheme' is greater than
26 //                       the modulus of 'signHandle' (for an RSA key);
27 //                       invalid commit status or failed to generate "r" value
28 //                       (for an ECC key)
29 TPM_RC
30 SignAttestInfo(OBJECT*      signKey, // IN: sign object
31                TPMT_SIG_SCHEME* scheme, // IN: sign scheme
32                TPMS_ATTEST*  certifyInfo, // IN: the data to be signed
33                TPM2B_DATA*    qualifyingData, // IN: extra data for the signing
34                                // process
35                TPM2B_ATTEST*  attest, // OUT: marshaled attest blob to be
36                                // signed
37                TPMT_SIGNATURE* signature // OUT: signature
38 );
39
40 /*** IsSigningObject()
41 // Checks to see if the object is OK for signing. This is here rather than in
42 // Object_spt.c because all the attestation commands use this file but not
43 // Object_spt.c.
44 // Return Type: BOOL
45 //     TRUE(1)    object may sign
46 //     FALSE(0)   object may not sign
47 BOOL IsSigningObject(OBJECT* object // IN:
48 );
49
50 #endif // _ATTEST_SPT_FP_H

```

6.55 /tpm/include/private/prototypes/Bits_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 28, 2019   Time: 08:25:19PM
4  */
5
6  #ifndef _BITS_FP_H_
7  #define _BITS_FP_H_
8
9  /*** TestBit()
10 // This function is used to check the setting of a bit in an array of bits.
11 // Return Type: BOOL
12 //     TRUE(1)         bit is set
13 //     FALSE(0)        bit is not set
14 BOOL TestBit(unsigned int bitNum,      // IN: number of the bit in 'bArray'
15             BYTE*      bArray,        // IN: array containing the bits
16             unsigned int bytesInArray // IN: size in bytes of 'bArray'
17 );
18
19 /*** SetBit()
20 // This function will set the indicated bit in 'bArray'.
21 void SetBit(unsigned int bitNum,      // IN: number of the bit in 'bArray'
22            BYTE*      bArray,        // IN: array containing the bits
23            unsigned int bytesInArray // IN: size in bytes of 'bArray'
24 );
25
26 /*** ClearBit()
27 // This function will clear the indicated bit in 'bArray'.
28 void ClearBit(unsigned int bitNum,    // IN: number of the bit in 'bArray'.
29              BYTE*      bArray,      // IN: array containing the bits
30              unsigned int bytesInArray // IN: size in bytes of 'bArray'
31 );
32
33 #endif // _BITS_FP_H_

```

6.56 /tpm/include/private/prototypes/CertifyCreation_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_CertifyCreation // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CERTIFYCREATION_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CERTIFYCREATION_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT    signHandle;
12     TPMI_DH_OBJECT    objectHandle;
13     TPM2B_DATA         qualifyingData;
14     TPM2B_DIGEST       creationHash;
15     TPMT_SIG_SCHEME    inScheme;
16     TPMT_TK_CREATION   creationTicket;
17 } CertifyCreation_In;
18
19 // Output structure definition
20 typedef struct
21 {
22     TPM2B_ATTEST    certifyInfo;
23     TPMT_SIGNATURE  signature;
24 } CertifyCreation_Out;
25
26 // Response code modifiers
27 #   define RC_CertifyCreation_signHandle    (TPM_RC_H + TPM_RC_1)

```

```

28 #   define RC_CertifyCreation_objectHandle    (TPM_RC_H + TPM_RC_2)
29 #   define RC_CertifyCreation_qualifyingData (TPM_RC_P + TPM_RC_1)
30 #   define RC_CertifyCreation_creationHash    (TPM_RC_P + TPM_RC_2)
31 #   define RC_CertifyCreation_inScheme        (TPM_RC_P + TPM_RC_3)
32 #   define RC_CertifyCreation_creationTicket  (TPM_RC_P + TPM_RC_4)
33
34 // Function prototype
35 TPM_RC
36 TPM2_CertifyCreation(CertifyCreation_In* in, CertifyCreation_Out* out);
37
38 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CERTIFYCREATION_FP_H_
39 #endif // CC_CertifyCreation

```

6.57 /tpm/include/private/prototypes/CertifyX509_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_CertifyX509 // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CERTIFYX509_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CERTIFYX509_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT    objectHandle;
12     TPMI_DH_OBJECT    signHandle;
13     TPM2B_DATA         reserved;
14     TPMT_SIG_SCHEME    inScheme;
15     TPM2B_MAX_BUFFER   partialCertificate;
16 } CertifyX509_In;
17
18 // Output structure definition
19 typedef struct
20 {
21     TPM2B_MAX_BUFFER   addedToCertificate;
22     TPM2B_DIGEST        tbsDigest;
23     TPMT_SIGNATURE      signature;
24 } CertifyX509_Out;
25
26 // Response code modifiers
27 #   define RC_CertifyX509_objectHandle    (TPM_RC_H + TPM_RC_1)
28 #   define RC_CertifyX509_signHandle      (TPM_RC_H + TPM_RC_2)
29 #   define RC_CertifyX509_reserved        (TPM_RC_P + TPM_RC_1)
30 #   define RC_CertifyX509_inScheme        (TPM_RC_P + TPM_RC_2)
31 #   define RC_CertifyX509_partialCertificate (TPM_RC_P + TPM_RC_3)
32
33 // Function prototype
34 TPM_RC
35 TPM2_CertifyX509(CertifyX509_In* in, CertifyX509_Out* out);
36
37 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CERTIFYX509_FP_H_
38 #endif // CC_CertifyX509

```

6.58 /tpm/include/private/prototypes/Certify_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Certify // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CERTIFY_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CERTIFY_FP_H_
7
8 // Input structure definition

```

```

9  typedef struct
10 {
11     TPMI_DH_OBJECT  objectHandle;
12     TPMI_DH_OBJECT  signHandle;
13     TPM2B_DATA      qualifyingData;
14     TPMT_SIG_SCHEME inScheme;
15 } Certify_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_ATTEST  certifyInfo;
21     TPMT_SIGNATURE signature;
22 } Certify_Out;
23
24 // Response code modifiers
25 # define RC_Certify_objectHandle (TPM_RC_H + TPM_RC_1)
26 # define RC_Certify_signHandle   (TPM_RC_H + TPM_RC_2)
27 # define RC_Certify_qualifyingData (TPM_RC_P + TPM_RC_1)
28 # define RC_Certify_inScheme     (TPM_RC_P + TPM_RC_2)
29
30 // Function prototype
31 TPM_RC
32 TPM2_Certify(Certify_In* in, Certify_Out* out);
33
34 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CERTIFY_FP_H_
35 #endif // CC_Certify

```

6.59 /tpm/include/private/prototypes/ChangeEPS_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_ChangeEPS // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CHANGEEPS_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CHANGEEPS_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_PLATFORM authHandle;
12 } ChangeEPS_In;
13
14 // Response code modifiers
15 #   define RC_ChangeEPS_authHandle (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_ChangeEPS(ChangeEPS_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CHANGEEPS_FP_H_
22 #endif // CC_ChangeEPS

```

6.60 /tpm/include/private/prototypes/ChangePPS_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_ChangePPS // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CHANGEPPS_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CHANGEPPS_FP_H_
7
8  // Input structure definition
9  typedef struct

```

```

10 {
11     TPMI_RH_PLATFORM authHandle;
12 } ChangePPS_In;
13
14 // Response code modifiers
15 # define RC_ChangePPS_authHandle (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_ChangePPS(ChangePPS_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CHANGEPPS_FP_H_
22 #endif // CC_ChangePPS

```

6.61 /tpm/include/private/prototypes/ClearControl_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ClearControl // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLEARCONTROL_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLEARCONTROL_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_CLEAR auth;
12     TPMI_YES_NO disable;
13 } ClearControl_In;
14
15 // Response code modifiers
16 # define RC_ClearControl_auth (TPM_RC_H + TPM_RC_1)
17 # define RC_ClearControl_disable (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_ClearControl(ClearControl_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLEARCONTROL_FP_H_
24 #endif // CC_ClearControl

```

6.62 /tpm/include/private/prototypes/Clear_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Clear // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLEAR_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLEAR_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_CLEAR authHandle;
12 } Clear_In;
13
14 // Response code modifiers
15 # define RC_Clear_authHandle (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_Clear(Clear_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLEAR_FP_H_

```



```
22 #endif // CC_Clear
```

6.63 /tpm/include/private/prototypes/ClockRateAdjust_fp.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ClockRateAdjust // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLOCKRATEADJUST_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLOCKRATEADJUST_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PROVISION auth;
12     TPM_CLOCK_ADJUST rateAdjust;
13 } ClockRateAdjust_In;
14
15 // Response code modifiers
16 #   define RC_ClockRateAdjust_auth (TPM_RC_H + TPM_RC_1)
17 #   define RC_ClockRateAdjust_rateAdjust (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_ClockRateAdjust(ClockRateAdjust_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLOCKRATEADJUST_FP_H_
24 #endif // CC_ClockRateAdjust
```

6.64 /tpm/include/private/prototypes/ClockSet_fp.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ClockSet // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLOCKSET_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLOCKSET_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PROVISION auth;
12     UINT64 newTime;
13 } ClockSet_In;
14
15 // Response code modifiers
16 #   define RC_ClockSet_auth (TPM_RC_H + TPM_RC_1)
17 #   define RC_ClockSet_newTime (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_ClockSet(ClockSet_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CLOCKSET_FP_H_
24 #endif // CC_ClockSet
```

6.65 /tpm/include/private/prototypes/CommandAudit_fp.h

```
1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Apr 2, 2019 Time: 04:23:27PM
4 */
5
```

```

6  #ifndef _COMMAND_AUDIT_FP_H
7  #define _COMMAND_AUDIT_FP_H
8
9  /*** CommandAuditPreInstall_Init()
10 // This function initializes the command audit list. This function simulates
11 // the behavior of manufacturing. A function is used instead of a structure
12 // definition because this is easier than figuring out the initialization value
13 // for a bit array.
14 //
15 // This function would not be implemented outside of a manufacturing or
16 // simulation environment.
17 void CommandAuditPreInstall_Init(void);
18
19 /*** CommandAuditStartup()
20 // This function clears the command audit digest on a TPM Reset.
21 BOOL CommandAuditStartup(STARTUP_TYPE type // IN: start up type
22 );
23
24 /*** CommandAuditSet()
25 // This function will SET the audit flag for a command. This function
26 // will not SET the audit flag for a command that is not implemented. This
27 // ensures that the audit status is not SET when TPM2_GetCapability() is
28 // used to read the list of audited commands.
29 //
30 // This function is only used by TPM2_SetCommandCodeAuditStatus().
31 //
32 // The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the
33 // changes to be saved to NV after it is setting and clearing bits.
34 // Return Type: BOOL
35 //     TRUE(1)      command code audit status was changed
36 //     FALSE(0)     command code audit status was not changed
37 BOOL CommandAuditSet(TPM_CC commandCode // IN: command code
38 );
39
40 /*** CommandAuditClear()
41 // This function will CLEAR the audit flag for a command. It will not CLEAR the
42 // audit flag for TPM_CC_SetCommandCodeAuditStatus().
43 //
44 // This function is only used by TPM2_SetCommandCodeAuditStatus().
45 //
46 // The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the
47 // changes to be saved to NV after it is setting and clearing bits.
48 // Return Type: BOOL
49 //     TRUE(1)      command code audit status was changed
50 //     FALSE(0)     command code audit status was not changed
51 BOOL CommandAuditClear(TPM_CC commandCode // IN: command code
52 );
53
54 /*** CommandAuditIsRequired()
55 // This function indicates if the audit flag is SET for a command.
56 // Return Type: BOOL
57 //     TRUE(1)      command is audited
58 //     FALSE(0)     command is not audited
59 BOOL CommandAuditIsRequired(COMMAND_INDEX commandIndex // IN: command index
60 );
61
62 /*** CommandAuditCapGetCCList()
63 // This function returns a list of commands that have their audit bit SET.
64 //
65 // The list starts at the input commandCode.
66 // Return Type: TPMT_YES_NO
67 //     YES          if there are more command code available
68 //     NO           all the available command code has been returned
69 TPMT_YES_NO
70 CommandAuditCapGetCCList(TPM_CC commandCode, // IN: start command code
71                          UINT32 count,      // IN: count of returned TPM_CC

```

```

72         TPML_CC* commandList    // OUT: list of TPM_CC
73     );
74
75     /*** CommandAuditCapGetOneCC()
76     // This function returns true if a command has its audit bit set.
77     BOOL CommandAuditCapGetOneCC(TPM_CC commandCode // IN: command code
78     );
79
80     /*** CommandAuditGetDigest
81     // This command is used to create a digest of the commands being audited. The
82     // commands are processed in ascending numeric order with a list of TPM_CC being
83     // added to a hash. This operates as if all the audited command codes were
84     // concatenated and then hashed.
85     void CommandAuditGetDigest(TPM2B_DIGEST* digest // OUT: command digest
86     );
87
88     #endif // _COMMAND_AUDIT_FP_H_

```

6.66 /tpm/include/private/prototypes/CommandCodeAttributes_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 28, 2019 Time: 08:25:19PM
4  */
5
6  #ifndef _COMMAND_CODE_ATTRIBUTES_FP_H_
7  #define _COMMAND_CODE_ATTRIBUTES_FP_H_
8
9  /*** GetClosestCommandIndex()
10 // This function returns the command index for the command with a value that is
11 // equal to or greater than the input value
12 // Return Type: COMMAND_INDEX
13 // UNIMPLEMENTED_COMMAND_INDEX    command is not implemented
14 // other                          index of a command
15 COMMAND_INDEX
16 GetClosestCommandIndex(TPM_CC commandCode // IN: the command code to start at
17 );
18
19 /*** CommandCodeToCommandIndex()
20 // This function returns the index in the various attributes arrays of the
21 // command.
22 // Return Type: COMMAND_INDEX
23 // UNIMPLEMENTED_COMMAND_INDEX    command is not implemented
24 // other                          index of the command
25 COMMAND_INDEX
26 CommandCodeToCommandIndex(TPM_CC commandCode // IN: the command code to look up
27 );
28
29 /*** GetNextCommandIndex()
30 // This function returns the index of the next implemented command.
31 // Return Type: COMMAND_INDEX
32 // UNIMPLEMENTED_COMMAND_INDEX    no more implemented commands
33 // other                          the index of the next implemented command
34 COMMAND_INDEX
35 GetNextCommandIndex(COMMAND_INDEX commandIndex // IN: the starting index
36 );
37
38 /*** GetCommandCode()
39 // This function returns the commandCode associated with the command index
40 TPM_CC
41 GetCommandCode(COMMAND_INDEX commandIndex // IN: the command index
42 );
43
44 /*** CommandAuthRole()
45 //

```

```

46 // This function returns the authorization role required of a handle.
47 //
48 // Return Type: AUTH_ROLE
49 // AUTH_NONE      no authorization is required
50 // AUTH_USER      user role authorization is required
51 // AUTH_ADMIN     admin role authorization is required
52 // AUTH_DUP       duplication role authorization is required
53 AUTH_ROLE
54 CommandAuthRole(COMMAND_INDEX commandIndex, // IN: command index
55                 UINT32      handleIndex    // IN: handle index (zero based)
56 );
57
58 /*** EncryptSize()
59 // This function returns the size of the decrypt size field. This function returns
60 // 0 if encryption is not allowed
61 // Return Type: int
62 // 0      encryption not allowed
63 // 2      size field is two bytes
64 // 4      size field is four bytes
65 int EncryptSize(COMMAND_INDEX commandIndex // IN: command index
66 );
67
68 /*** DecryptSize()
69 // This function returns the size of the decrypt size field. This function returns
70 // 0 if decryption is not allowed
71 // Return Type: int
72 // 0      encryption not allowed
73 // 2      size field is two bytes
74 // 4      size field is four bytes
75 int DecryptSize(COMMAND_INDEX commandIndex // IN: command index
76 );
77
78 /*** IsSessionAllowed()
79 //
80 // This function indicates if the command is allowed to have sessions.
81 //
82 // This function must not be called if the command is not known to be implemented.
83 //
84 // Return Type: BOOL
85 // TRUE(1)      session is allowed with this command
86 // FALSE(0)     session is not allowed with this command
87 BOOL IsSessionAllowed(COMMAND_INDEX commandIndex // IN: the command to be checked
88 );
89
90 /*** IsHandleInResponse()
91 // This function determines if a command has a handle in the response
92 BOOL IsHandleInResponse(COMMAND_INDEX commandIndex);
93
94 /*** IsWriteOperation()
95 // Checks to see if an operation will write to an NV Index and is subject to being
96 // blocked by read-lock
97 BOOL IsWriteOperation(COMMAND_INDEX commandIndex // IN: Command to check
98 );
99
100 /*** IsReadOperation()
101 // Checks to see if an operation will write to an NV Index and is
102 // subject to being blocked by write-lock.
103 BOOL IsReadOperation(COMMAND_INDEX commandIndex // IN: Command to check
104 );
105
106 /*** CommandCapGetCCList()
107 // This function returns a list of implemented commands and command attributes
108 // starting from the command in 'commandCode'.
109 // Return Type: TPMI_YES_NO
110 // YES      more command attributes are available
111 // NO      no more command attributes are available

```

```

112 TPMI_YES_NO
113 CommandCapGetCCList(TPM_CC commandCode, // IN: start command code
114                     UINT32 count,        // IN: maximum count for number of entries in
115                                         // 'commandList'
116                     TPML_CCA* commandList // OUT: list of TPMA_CC
117 );
118
119 /*** CommandCapGetOneCC()
120 // This function checks whether a command is implemented, and returns its
121 // attributes if so.
122 BOOL CommandCapGetOneCC(TPM_CC commandCode, // IN: command code
123                         TPMA_CC* commandAttributes // OUT: Command attributes
124 );
125
126 /*** IsVendorCommand()
127 // Function indicates if a command index references a vendor command.
128 // Return Type: BOOL
129 // TRUE(1)      command is a vendor command
130 // FALSE(0)     command is not a vendor command
131 BOOL IsVendorCommand(COMMAND_INDEX commandIndex // IN: command index to check
132 );
133
134 #endif // _COMMAND_CODE_ATTRIBUTES_FP_H_

```

6.67 /tpm/include/private/prototypes/CommandDispatcher_fp.h

```

1 /* (Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 7, 2020 Time: 07:06:44PM
4 */
5
6 #ifndef _COMMAND_DISPATCHER_FP_H_
7 #define _COMMAND_DISPATCHER_FP_H_
8
9 /*** ParseHandleBuffer()
10 // This is the table-driven version of the handle buffer unmarshaling code
11 TPM_RC
12 ParseHandleBuffer(COMMAND* command);
13
14 /*** CommandDispatcher()
15 // Function to unmarshal the command parameters, call the selected action code, and
16 // marshal the response parameters.
17 TPM_RC
18 CommandDispatcher(COMMAND* command);
19
20 #endif // _COMMAND_DISPATCHER_FP_H_

```

6.68 /tpm/include/private/prototypes/Commit_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Commit // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_COMMIT_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_COMMIT_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT      signHandle;
12     TPM2B_ECC_POINT      P1;
13     TPM2B_SENSITIVE_DATA s2;
14     TPM2B_ECC_PARAMETER y2;
15 } Commit_In;

```

```

16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_ECC_POINT K;
21     TPM2B_ECC_POINT L;
22     TPM2B_ECC_POINT E;
23     UINT16          counter;
24 } Commit_Out;
25
26 // Response code modifiers
27 #   define RC_Commit_signHandle (TPM_RC_H + TPM_RC_1)
28 #   define RC_Commit_P1        (TPM_RC_P + TPM_RC_1)
29 #   define RC_Commit_s2        (TPM_RC_P + TPM_RC_2)
30 #   define RC_Commit_y2        (TPM_RC_P + TPM_RC_3)
31
32 // Function prototype
33 TPM_RC
34 TPM2_Commit(Commit_In* in, Commit_Out* out);
35
36 #   endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_COMMIT_FP_H_
37 #endif // CC_Commit

```

6.69 /tpm/include/private/prototypes/ContextLoad_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ContextLoad // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CONTEXTLOAD_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CONTEXTLOAD_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMS_CONTEXT context;
12 } ContextLoad_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPMI_DH_CONTEXT loadedHandle;
18 } ContextLoad_Out;
19
20 // Response code modifiers
21 #   define RC_ContextLoad_context (TPM_RC_P + TPM_RC_1)
22
23 // Function prototype
24 TPM_RC
25 TPM2_ContextLoad(ContextLoad_In* in, ContextLoad_Out* out);
26
27 #   endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CONTEXTLOAD_FP_H_
28 #endif // CC_ContextLoad

```

6.70 /tpm/include/private/prototypes/ContextSave_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ContextSave // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CONTEXTSAVE_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CONTEXTSAVE_FP_H_
7
8 // Input structure definition

```



```

9  typedef struct
10 {
11     TPMI_DH_CONTEXT saveHandle;
12 } ContextSave_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPMS_CONTEXT context;
18 } ContextSave_Out;
19
20 // Response code modifiers
21 #   define RC_ContextSave_saveHandle (TPM_RC_H + TPM_RC_1)
22
23 // Function prototype
24 TPM_RC
25 TPM2_ContextSave(ContextSave_In* in, ContextSave_Out* out);
26
27 #   endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CONTEXTSAVE_FP_H_
28 #endif // CC_ContextSave

```

6.71 /tpm/include/private/prototypes/Context_spt_fp.h

```

1  /*(Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Mar 28, 2019 Time: 08:25:18PM
4   */
5
6  #ifndef _CONTEXT_SPT_FP_H_
7  #define _CONTEXT_SPT_FP_H_
8
9  /*** ComputeContextProtectionKey()
10 // This function retrieves the symmetric protection key for context encryption
11 // It is used by TPM2_ConextSave and TPM2_ContextLoad to create the symmetric
12 // encryption key and iv
13 // Return Type: TPM_RC
14 //     TPM_RC_FW_LIMITED      The requested hierarchy is FW-limited, but the TPM
15 //                             does not support FW-limited objects or the TPM failed
16 //                             to derive the Firmware Secret.
17 //     TPM_RC_SVN_LIMITED     The requested hierarchy is SVN-limited, but the TPM
18 //                             does not support SVN-limited objects or the TPM
19 //                             failed to derive the Firmware SVN Secret for the
20 //                             requested SVN.
21 TPM_RC ComputeContextProtectionKey(TPMS_CONTEXT* contextBlob, // IN: context blob
22     TPM2B_SYM_KEY* symKey, // OUT: the symmetric key
23     TPM2B_IV* iv // OUT: the IV.
24 );
25
26 /*** ComputeContextIntegrity()
27 // Generate the integrity hash for a context
28 //     It is used by TPM2_ContextSave to create an integrity hash
29 //     and by TPM2_ContextLoad to compare an integrity hash
30 // Return Type: TPM_RC
31 //     TPM_RC_FW_LIMITED      The requested hierarchy is FW-limited, but the TPM
32 //                             does not support FW-limited objects or the TPM failed
33 //                             to derive the Firmware Secret.
34 //     TPM_RC_SVN_LIMITED     The requested hierarchy is SVN-limited, but the TPM
35 //                             does not support SVN-limited objects or the TPM
36 //                             failed to derive the Firmware SVN Secret for the
37 //                             requested SVN.
38 TPM_RC ComputeContextIntegrity(TPMS_CONTEXT* contextBlob, // IN: context blob
39     TPM2B_DIGEST* integrity // OUT: integrity
40 );
41
42 /*** SequenceDataExport()

```

```

43 // This function is used scan through the sequence object and
44 // either modify the hash state data for export (contextSave) or to
45 // import it into the internal format (contextLoad).
46 // This function should only be called after the sequence object has been copied
47 // to the context buffer (contextSave) or from the context buffer into the sequence
48 // object. The presumption is that the context buffer version of the data is the
49 // same size as the internal representation so nothing outside of the hash context
50 // area gets modified.
51 void SequenceDataExport(
52     HASH_OBJECT*      object,          // IN: an internal hash object
53     HASH_OBJECT_BUFFER* exportObject // OUT: a sequence context in a buffer
54 );
55
56 /*** SequenceDataImport()
57 // This function is used scan through the sequence object and
58 // either modify the hash state data for export (contextSave) or to
59 // import it into the internal format (contextLoad).
60 // This function should only be called after the sequence object has been copied
61 // to the context buffer (contextSave) or from the context buffer into the sequence
62 // object. The presumption is that the context buffer version of the data is the
63 // same size as the internal representation so nothing outside of the hash context
64 // area gets modified.
65 void SequenceDataImport(
66     HASH_OBJECT*      object,          // IN/OUT: an internal hash object
67     HASH_OBJECT_BUFFER* exportObject // IN/OUT: a sequence context in a buffer
68 );
69
70 #endif // _CONTEXT_SPT_FP_H_

```

6.72 /tpm/include/private/prototypes/CreateLoaded_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_CreateLoaded // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATELOADED_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATELOADED_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_PARENT      parentHandle;
12     TPM2B_SENSITIVE_CREATE inSensitive;
13     TPM2B_TEMPLATE      inPublic;
14 } CreateLoaded_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM_HANDLE      objectHandle;
20     TPM2B_PRIVATE outPrivate;
21     TPM2B_PUBLIC outPublic;
22     TPM2B_NAME      name;
23 } CreateLoaded_Out;
24
25 // Response code modifiers
26 #   define RC_CreateLoaded_parentHandle (TPM_RC_H + TPM_RC_1)
27 #   define RC_CreateLoaded_inSensitive (TPM_RC_P + TPM_RC_1)
28 #   define RC_CreateLoaded_inPublic (TPM_RC_P + TPM_RC_2)
29
30 // Function prototype
31 TPM_RC
32 TPM2_CreateLoaded(CreateLoaded_In* in, CreateLoaded_Out* out);
33
34 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATELOADED_FP_H_

```

```
35 #endif    // CC_CreateLoaded
```

6.73 /tpm/include/private/prototypes/CreatePrimary_fp.h

```
1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_CreatePrimary    // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATEPRIMARY_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATEPRIMARY_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_HIERARCHY    primaryHandle;
12     TPM2B_SENSITIVE_CREATE inSensitive;
13     TPM2B_PUBLIC          inPublic;
14     TPM2B_DATA            outsideInfo;
15     TPML_PCR_SELECTION    creationPCR;
16 } CreatePrimary_In;
17
18 // Output structure definition
19 typedef struct
20 {
21     TPM_HANDLE          objectHandle;
22     TPM2B_PUBLIC         outPublic;
23     TPM2B_CREATION_DATA  creationData;
24     TPM2B_DIGEST         creationHash;
25     TPMT_TK_CREATION      creationTicket;
26     TPM2B_NAME           name;
27 } CreatePrimary_Out;
28
29 // Response code modifiers
30 #   define RC_CreatePrimary_primaryHandle (TPM_RC_H + TPM_RC_1)
31 #   define RC_CreatePrimary_inSensitive   (TPM_RC_P + TPM_RC_1)
32 #   define RC_CreatePrimary_inPublic      (TPM_RC_P + TPM_RC_2)
33 #   define RC_CreatePrimary_outsideInfo   (TPM_RC_P + TPM_RC_3)
34 #   define RC_CreatePrimary_creationPCR   (TPM_RC_P + TPM_RC_4)
35
36 // Function prototype
37 TPM_RC
38 TPM2_CreatePrimary(CreatePrimary_In* in, CreatePrimary_Out* out);
39
40 # endif    // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATEPRIMARY_FP_H_
41 #endif    // CC_CreatePrimary
```

6.74 /tpm/include/private/prototypes/Create_fp.h

```
1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_Create    // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATE_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATE_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT        parentHandle;
12     TPM2B_SENSITIVE_CREATE inSensitive;
13     TPM2B_PUBLIC          inPublic;
14     TPM2B_DATA            outsideInfo;
15     TPML_PCR_SELECTION    creationPCR;
16 } Create_In;
```

```

17
18 // Output structure definition
19 typedef struct
20 {
21     TPM2B_PRIVATE      outPrivate;
22     TPM2B_PUBLIC       outPublic;
23     TPM2B_CREATION_DATA creationData;
24     TPM2B_DIGEST       creationHash;
25     TPMT_TK_CREATION    creationTicket;
26 } Create_Out;
27
28 // Response code modifiers
29 #   define RC_Create_parentHandle (TPM_RC_H + TPM_RC_1)
30 #   define RC_Create_inSensitive (TPM_RC_P + TPM_RC_1)
31 #   define RC_Create_inPublic    (TPM_RC_P + TPM_RC_2)
32 #   define RC_Create_outsideInfo (TPM_RC_P + TPM_RC_3)
33 #   define RC_Create_creationPCR (TPM_RC_P + TPM_RC_4)
34
35 // Function prototype
36 TPM_RC
37 TPM2_Create(Create_In* in, Create_Out* out);
38
39 #   endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_CREATE_FP_H_
40 #endif // CC_Create

```

6.75 /tpm/include/private/prototypes/CryptCmac_fp.h

```

1 /* (Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 28, 2019 Time: 08:25:18PM
4  */
5
6 #ifndef _CRYPT_CMAC_FP_H_
7 #define _CRYPT_CMAC_FP_H_
8
9 #if ALG_CMAC
10
11 /*** CryptCmacStart()
12 // This is the function to start the CMAC sequence operation. It initializes the
13 // dispatch functions for the data and end operations for CMAC and initializes the
14 // parameters that are used for the processing of data, including the key, key size
15 // and block cipher algorithm.
16 UINT16
17 CryptCmacStart(
18     SMAC_STATE* state, TPMU_PUBLIC_PARMS* keyParms, TPM_ALG_ID macAlg, TPM2B* key);
19
20 /*** CryptCmacData()
21 // This function is used to add data to the CMAC sequence computation. The function
22 // will XOR new data into the IV. If the buffer is full, and there is additional
23 // input data, the data is encrypted into the IV buffer, the new data is then
24 // XOR into the IV. When the data runs out, the function returns without encrypting
25 // even if the buffer is full. The last data block of a sequence will not be
26 // encrypted until the call to CryptCmacEnd(). This is to allow the proper subkey
27 // to be computed and applied before the last block is encrypted.
28 void CryptCmacData(SMAC_STATES* state, UINT32 size, const BYTE* buffer);
29
30 /*** CryptCmacEnd()
31 // This is the completion function for the CMAC. It does padding, if needed, and
32 // selects the subkey to be applied before the last block is encrypted.
33 UINT16
34 CryptCmacEnd(SMAC_STATES* state, UINT32 outSize, BYTE* outBuffer);
35 #endif
36
37 #endif // _CRYPT_CMAC_FP_H_

```

6.76 /tpm/include/private/prototypes/CryptEccCrypt_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Feb 28, 2020   Time: 03:04:48PM
4  */
5
6  #ifndef _CRYPT_ECC_CRYPT_FP_H_
7  #define _CRYPT_ECC_CRYPT_FP_H_
8
9  #if CC_ECC_Encrypt || CC_ECC_Encrypt
10
11  /*** CryptEccSelectScheme()
12  // This function is used by TPM2_ECC_Decrypt and TPM2_ECC_Encrypt. It sets scheme
13  // either the input scheme or the key scheme. If they key scheme is not TPM_ALG_NULL
14  // then the input scheme must be TPM_ALG_NULL or the same as the key scheme. If
15  // not, then the function returns FALSE.
16  // Return Type: BOOL
17  //     TRUE      'scheme' is set
18  //     FALSE     'scheme' is not valid (it may have been changed).
19  BOOL CryptEccSelectScheme(OBJECT*      key,      //IN: key containing default scheme
20                          TPMT_KDF_SCHEME* scheme // IN: a decrypt scheme
21  );
22
23  /*** CryptEccEncrypt()
24  //This function performs ECC-based data obfuscation. The only scheme that is currently
25  // supported is MGF1 based. See Part 1, Annex D for details.
26  // Return Type: TPM_RC
27  //     TPM_RC_CURVE      unsupported curve
28  //     TPM_RC_HASH       hash not allowed
29  //     TPM_RC_SCHEME     'scheme' is not supported
30  //     TPM_RC_NO_RESULT  internal error in big number processing
31  LIB_EXPORT TPM_RC CryptEccEncrypt(
32      OBJECT*      key,      // IN: public key of recipient
33      TPMT_KDF_SCHEME* scheme, // IN: scheme to use.
34      TPM2B_MAX_BUFFER* plainText, // IN: the text to obfuscate
35      TPMS_ECC_POINT* c1,      // OUT: public ephemeral key
36      TPM2B_MAX_BUFFER* c2,    // OUT: obfuscated text
37      TPM2B_DIGEST* c3        // OUT: digest of ephemeral key
38                          // and plainText
39  );
40
41  /*** CryptEccDecrypt()
42  // This function performs ECC decryption and integrity check of the input data.
43  // Return Type: TPM_RC
44  //     TPM_RC_CURVE      unsupported curve
45  //     TPM_RC_HASH       hash not allowed
46  //     TPM_RC_SCHEME     'scheme' is not supported
47  //     TPM_RC_NO_RESULT  internal error in big number processing
48  //     TPM_RC_VALUE      C3 did not match hash of recovered data
49  LIB_EXPORT TPM_RC CryptEccDecrypt(
50      OBJECT*      key,      // IN: key used for data recovery
51      TPMT_KDF_SCHEME* scheme, // IN: scheme to use.
52      TPM2B_MAX_BUFFER* plainText, // OUT: the recovered text
53      TPMS_ECC_POINT* c1,      // IN: public ephemeral key
54      TPM2B_MAX_BUFFER* c2,    // IN: obfuscated text
55      TPM2B_DIGEST* c3        // IN: digest of ephemeral key
56                          // and plainText
57  );
58  #endif // CC_ECC_Encrypt || CC_ECC_Encrypt
59
60  #endif // _CRYPT_ECC_CRYPT_FP_H_

```

6.77 /tpm/include/private/prototypes/CryptEccKeyExchange_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 28, 2019   Time: 08:25:18PM
4  */
5
6  #ifndef _CRYPT_ECC_KEY_EXCHANGE_FP_H_
7  #define _CRYPT_ECC_KEY_EXCHANGE_FP_H_
8
9  #if CC_ZGen_2Phase == YES
10
11  /*** CryptEcc2PhaseKeyExchange()
12  // This function is the dispatch routine for the EC key exchange functions that use
13  // two ephemeral and two static keys.
14  // Return Type: TPM_RC
15  // TPM_RC_SCHEME scheme is not defined
16  LIB_EXPORT TPM_RC CryptEcc2PhaseKeyExchange(
17      TPMS_ECC_POINT* outZ1, // OUT: a computed point
18      TPMS_ECC_POINT* outZ2, // OUT: and optional second point
19      TPM_ECC_CURVE curveId, // IN: the curve for the computations
20      TPM_ALG_ID scheme, // IN: the key exchange scheme
21      TPM2B_ECC_PARAMETER* dsA, // IN: static private TPM key
22      TPM2B_ECC_PARAMETER* deA, // IN: ephemeral private TPM key
23      TPMS_ECC_POINT* QsB, // IN: static public party B key
24      TPMS_ECC_POINT* QeB // IN: ephemeral public party B key
25  );
26  # if ALG_SM2
27
28  /*** SM2KeyExchange()
29  // This function performs the key exchange defined in SM2.
30  // The first step is to compute
31  // 'tA' = ('dsA' + 'deA' avf(Xe,A)) mod 'n'
32  // Then, compute the 'Z' value from
33  // 'outZ' = ('h' 'tA' mod 'n') ('QsA' + [avf('QeB.x')]('QeB')).
34  // The function will compute the ephemeral public key from the ephemeral
35  // private key.
36  // All points are required to be on the curve of 'inQsA'. The function will fail
37  // catastrophically if this is not the case
38  // Return Type: TPM_RC
39  // TPM_RC_NO_RESULT the value for dsA does not give a valid point on the
40  // curve
41  LIB_EXPORT TPM_RC SM2KeyExchange(
42      TPMS_ECC_POINT* outZ, // OUT: the computed point
43      TPM_ECC_CURVE curveId, // IN: the curve for the computations
44      TPM2B_ECC_PARAMETER* dsAIn, // IN: static private TPM key
45      TPM2B_ECC_PARAMETER* deAIn, // IN: ephemeral private TPM key
46      TPMS_ECC_POINT* QsBIn, // IN: static public party B key
47      TPMS_ECC_POINT* QeBIn // IN: ephemeral public party B key
48  );
49  # endif
50  #endif // CC_ZGen_2Phase
51
52  #endif // _CRYPT_ECC_KEY_EXCHANGE_FP_H_

```

6.78 /tpm/include/private/prototypes/CryptEccMain_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Apr 2, 2019   Time: 03:18:00PM
4  */
5
6  #ifndef _CRYPT_ECC_MAIN_FP_H_
7  #define _CRYPT_ECC_MAIN_FP_H_
8

```



```

9  #if ALG_ECC
10
11  /** Functions
12  # if SIMULATION
13  void EccSimulationEnd(void);
14  # endif // SIMULATION
15
16  /** CryptEccInit()
17  // This function is called at _TPM_Init
18  BOOL CryptEccInit(void);
19
20  /** CryptEccStartup()
21  // This function is called at TPM2_Startup().
22  BOOL CryptEccStartup(void);
23
24  /** ClearPoint2B(generic)
25  // Initialize the size values of a TPMS_ECC_POINT structure.
26  void ClearPoint2B(TPMS_ECC_POINT* p // IN: the point
27  );
28
29  /** CryptEccGetParametersByCurveId()
30  // This function returns a pointer to the curve data that is associated with
31  // the indicated curveId.
32  // If there is no curve with the indicated ID, the function returns NULL. This
33  // function is in this module so that it can be called by GetCurve data.
34  // Return Type: const TPM_ECC_CURVE_METADATA
35  //      NULL      curve with the indicated TPM_ECC_CURVE is not implemented
36  //      != NULL    pointer to the curve data
37  LIB_EXPORT const TPM_ECC_CURVE_METADATA* CryptEccGetParametersByCurveId(
38  TPM_ECC_CURVE curveId // IN: the curveId
39  );
40
41  /** CryptEccGetKeySizeForCurve()
42  // This function returns the key size in bits of the indicated curve.
43  LIB_EXPORT UINT16 CryptEccGetKeySizeForCurve(TPM_ECC_CURVE curveId // IN: the curve
44  );
45
46  /** CryptEccGetOID()
47  const BYTE* CryptEccGetOID(TPM_ECC_CURVE curveId);
48
49  /** CryptEccGetCurveByIndex()
50  // This function returns the number of the 'i'-th implemented curve. The normal
51  // use would be to call this function with 'i' starting at 0. When the 'i' is greater
52  // than or equal to the number of implemented curves, TPM_ECC_NONE is returned.
53  LIB_EXPORT TPM_ECC_CURVE CryptEccGetCurveByIndex(UINT16 i);
54
55  /** CryptCapGetECCCurve()
56  // This function returns the list of implemented ECC curves.
57  // Return Type: TPMI_YES_NO
58  //      YES      if no more ECC curve is available
59  //      NO      if there are more ECC curves not reported
60  TPMI_YES_NO
61  CryptCapGetECCCurve(TPM_ECC_CURVE curveID, // IN: the starting ECC curve
62  UINT32 maxCount, // IN: count of returned curves
63  TPML_ECC_CURVE* curveList // OUT: ECC curve list
64  );
65
66  /** CryptCapGetOneECCCurve()
67  // This function returns whether the ECC curve is implemented.
68  BOOL CryptCapGetOneECCCurve(TPM_ECC_CURVE curveID // IN: the ECC curve
69  );
70
71  /** CryptGetCurveSignScheme()
72  // This function will return a pointer to the scheme of the curve.
73  const TPMT_ECC_SCHEME* CryptGetCurveSignScheme(
74  TPM_ECC_CURVE curveId // IN: The curve selector

```

```

75 );
76
77 /*** CryptGenerateR()
78 // This function computes the commit random value for a split signing scheme.
79 //
80 // If 'c' is NULL, it indicates that 'r' is being generated
81 // for TPM2_Commit.
82 // If 'c' is not NULL, the TPM will validate that the 'gr.commitArray'
83 // bit associated with the input value of 'c' is SET. If not, the TPM
84 // returns FALSE and no 'r' value is generated.
85 // Return Type: BOOL
86 //     TRUE(1)           r value computed
87 //     FALSE(0)          no r value computed
88 BOOL CryptGenerateR(TPM2B_ECC_PARAMETER* r,           // OUT: the generated random value
89                    UINT16* c,                       // IN/OUT: count value.
90                    TPMI_ECC_CURVE curveID,          // IN: the curve for the value
91                    TPM2B_NAME* name,               // IN: optional name of a key to
92                                                    // associate with 'r'
93 );
94
95 /*** CryptCommit()
96 // This function is called when the count value is committed. The 'gr.commitArray'
97 // value associated with the current count value is SET and g_commitCounter is
98 // incremented. The low-order 16 bits of old value of the counter is returned.
99 UINT16
100 CryptCommit(void);
101
102 /*** CryptEndCommit()
103 // This function is called when the signing operation using the committed value
104 // is completed. It clears the gr.commitArray bit associated with the count
105 // value so that it can't be used again.
106 void CryptEndCommit(UINT16 c // IN: the counter value of the commitment
107 );
108
109 /*** CryptEccGetParameters()
110 // This function returns the ECC parameter details of the given curve.
111 // Return Type: BOOL
112 //     TRUE(1)           success
113 //     FALSE(0)          unsupported ECC curve ID
114 BOOL CryptEccGetParameters(
115     TPM_ECC_CURVE curveId, // IN: ECC curve ID
116     TPMS_ALGORITHM_DETAIL_ECC* parameters // OUT: ECC parameters
117 );
118
119 /*** TpmEcc_IsValidPrivateEcc()
120 // Checks that 0 < 'x' < 'q'
121 BOOL TpmEcc_IsValidPrivateEcc(const Crypt_Int* x, // IN: private key to check
122                               const Crypt_EccCurve* E // IN: the curve to check
123 );
124
125 LIB_EXPORT BOOL CryptEccIsValidPrivateKey(TPM2B_ECC_PARAMETER* d,
126                                           TPM_ECC_CURVE curveId);
127
128 /*** TpmEcc_PointMult()
129 // This function does a point multiply of the form 'R' = ['d']'S' + ['u']'Q' where the
130 // parameters are Crypt_Int* values. If 'S' is NULL and d is not NULL, then it
131 // computes
132 // 'R' = ['d']'G' + ['u']'Q' or just 'R' = ['d']'G' if 'u' and 'Q' are NULL.
133 // If 'skipChecks' is TRUE, then the function will not verify that the inputs are
134 // correct for the domain. This would be the case when the values were created by the
135 // CryptoEngine code.
136 // It will return TPM_RC_NO_RESULT if the resulting point is the point at infinity.
137 // Return Type: TPM_RC
138 //     TPM_RC_NO_RESULT          result of multiplication is a point at infinity
139 //     TPM_RC_ECC_POINT          'S' or 'Q' is not on the curve
140 //     TPM_RC_VALUE              'd' or 'u' is not < n

```

```

140 TPM_RC
141 TpmEcc_PointMult(Crypt_Point*      R, // OUT: computed point
142                 const Crypt_Point* S, // IN: optional point to multiply by 'd'
143                 const Crypt_Int*   d, // IN: scalar for [d]S or [d]G
144                 const Crypt_Point* Q, // IN: optional second point
145                 const Crypt_Int*   u, // IN: optional second scalar
146                 const Crypt_EccCurve* E // IN: curve parameters
147 );
148
149 /***TpmEcc_GenPrivateScalar()
150 // This function gets random values that are the size of the key plus 64 bits. The
151 // value is reduced (mod ('q' - 1)) and incremented by 1 ('q' is the order of the
152 // curve. This produces a value ('d') such that 1 <= 'd' < 'q'. This is the method
153 // of FIPS 186-4 Section B.4.1 "Key Pair Generation Using Extra Random Bits".
154 // Return Type: BOOL
155 //      TRUE(1)      success
156 //      FALSE(0)     failure generating private key
157 BOOL TpmEcc_GenPrivateScalar(
158     Crypt_Int*      dOut, // OUT: the qualified random value
159     const Crypt_EccCurve* E, // IN: curve for which the private key
160                               // needs to be appropriate
161     RAND_STATE* rand // IN: state for DRBG
162 );
163
164 /*** TpmEcc_GenerateKeyPair()
165 // This function gets a private scalar from the source of random bits and does
166 // the point multiply to get the public key.
167 BOOL TpmEcc_GenerateKeyPair(Crypt_Int*      bnD, // OUT: private scalar
168                             Crypt_Point*    ecQ, // OUT: public point
169                             const Crypt_EccCurve* E, // IN: curve for the point
170                             RAND_STATE* rand // IN: DRBG state to use
171 );
172
173 /***CryptEccNewKeyPair(***)
174 // This function creates an ephemeral ECC. It is ephemeral in that
175 // is expected that the private part of the key will be discarded
176 LIB_EXPORT TPM_RC CryptEccNewKeyPair(
177     TPMS_ECC_POINT*    Qout, // OUT: the public point
178     TPM2B_ECC_PARAMETER* dOut, // OUT: the private scalar
179     TPM_ECC_CURVE      curveId // IN: the curve for the key
180 );
181
182 /*** CryptEccPointMultiply()
183 // This function computes 'R' := ['dIn']'G' + ['uIn']'QIn'. Where 'dIn' and
184 // 'uIn' are scalars, 'G' and 'QIn' are points on the specified curve and 'G' is the
185 // default generator of the curve.
186 //
187 // The 'xOut' and 'yOut' parameters are optional and may be set to NULL if not
188 // used.
189 //
190 // It is not necessary to provide 'uIn' if 'QIn' is specified but one of 'uIn' and
191 // 'dIn' must be provided. If 'dIn' and 'QIn' are specified but 'uIn' is not
192 // provided, then 'R' = ['dIn']'QIn'.
193 //
194 // If the multiply produces the point at infinity, the TPM_RC_NO_RESULT is returned.
195 //
196 // The sizes of 'xOut' and 'yOut' will be set to be the size of the degree of
197 // the curve
198 //
199 // It is a fatal error if 'dIn' and 'uIn' are both unspecified (NULL) or if 'Qin'
200 // or 'Rout' is unspecified.
201 //
202 // Return Type: TPM_RC
203 //      TPM_RC_ECC_POINT      the point 'Pin' or 'Qin' is not on the curve
204 //      TPM_RC_NO_RESULT      the product point is at infinity
205 //      TPM_RC_CURVE          bad curve

```

```

206 //      TPM_RC_VALUE          'dIn' or 'uIn' out of range
207 //
208 LIB_EXPORT TPM_RC CryptEccPointMultiply(
209     TPMS_ECC_POINT*      Rout,      // OUT: the product point R
210     TPM_ECC_CURVE        curveId,   // IN: the curve to use
211     TPMS_ECC_POINT*      Pin,       // IN: first point (can be null)
212     TPM2B_ECC_PARAMETER* dIn,       // IN: scalar value for [dIn]Qin
213                                     // the Pin
214     TPMS_ECC_POINT*      Qin,       // IN: point Q
215     TPM2B_ECC_PARAMETER* uIn        // IN: scalar value for the multiplier
216                                     // of Q
217 );
218
219 /*** CryptEccIsPointOnCurve()
220 // This function is used to test if a point is on a defined curve. It does this
221 // by checking that 'y'^2 mod 'p' = 'x'^3 + 'a'*'x' + 'b' mod 'p'.
222 //
223 // It is a fatal error if 'Q' is not specified (is NULL).
224 // Return Type: BOOL
225 //      TRUE(1)          point is on curve
226 //      FALSE(0)         point is not on curve or curve is not supported
227 LIB_EXPORT BOOL CryptEccIsPointOnCurve(
228     TPM_ECC_CURVE        curveId,   // IN: the curve selector
229     TPMS_ECC_POINT*      Qin        // IN: the point.
230 );
231
232 /*** CryptEccGenerateKey()
233 // This function generates an ECC key pair based on the input parameters.
234 // This routine uses KDFa to produce candidate numbers. The method is according
235 // to FIPS 186-3, section B.1.2 "Key Pair Generation by Testing Candidates."
236 // According to the method in FIPS 186-3, the resulting private value 'd' should be
237 // 1 <= 'd' < 'n' where 'n' is the order of the base point.
238 //
239 // It is a fatal error if 'Qout', 'dOut', is not provided (is NULL).
240 //
241 // If the curve is not supported
242 // If 'seed' is not provided, then a random number will be used for the key
243 // Return Type: TPM_RC
244 //      TPM_RC_CURVE          curve is not supported
245 //      TPM_RC_NO_RESULT      could not verify key with signature (FIPS only)
246 LIB_EXPORT TPM_RC CryptEccGenerateKey(
247     TPMT_PUBLIC* publicArea,        // IN/OUT: The public area template for
248                                     // the new key. The public key
249                                     // area will be replaced computed
250                                     // ECC public key
251     TPMT_SENSITIVE* sensitive,      // OUT: the sensitive area will be
252                                     // updated to contain the private
253                                     // ECC key and the symmetric
254                                     // encryption key
255     RAND_STATE* rand                // IN: if not NULL, the deterministic
256                                     // RNG state
257 );
258 #endif // ALG_ECC
259
260 #endif // _CRYPT_ECC_MAIN_FP_H_

```

6.79 /tpm/include/private/prototypes/CryptEccSignature_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 28, 2019 Time: 08:25:18PM
4 */
5
6 #ifndef _CRYPT_ECC_SIGNATURE_FP_H_
7 #define _CRYPT_ECC_SIGNATURE_FP_H_

```

```

8
9  #if ALG_ECC
10
11  /*** CryptEccSign()
12  // This function is the dispatch function for the various ECC-based
13  // signing schemes.
14  // There is a bit of ugliness to the parameter passing. In order to test this,
15  // we sometime would like to use a deterministic RNG so that we can get the same
16  // signatures during testing. The easiest way to do this for most schemes is to
17  // pass in a deterministic RNG and let it return canned values during testing.
18  // There is a competing need for a canned parameter to use in ECDA. To accommodate
19  // both needs with minimal fuss, a special type of RAND_STATE is defined to carry
20  // the address of the commit value. The setup and handling of this is not very
21  // different for the caller than what was in previous versions of the code.
22  // Return Type: TPM_RC
23  //      TPM_RC_SCHEME          'scheme' is not supported
24  LIB_EXPORT TPM_RC CryptEccSign(TPMT_SIGNATURE* signature, // OUT: signature
25                                OBJECT* signKey, // IN: ECC key to sign the hash
26                                const TPM2B_DIGEST* digest, // IN: digest to sign
27                                TPMT_ECC_SCHEME* scheme, // IN: signing scheme
28                                RAND_STATE* rand);
29
30  /*** CryptEccValidateSignature()
31  // This function validates an EcDsa or EcSchnorr signature.
32  // The point 'Qin' needs to have been validated to be on the curve of 'curveId'.
33  // Return Type: TPM_RC
34  //      TPM_RC_SIGNATURE      not a valid signature
35  LIB_EXPORT TPM_RC CryptEccValidateSignature(
36      TPMT_SIGNATURE* signature, // IN: signature to be verified
37      OBJECT* signKey, // IN: ECC key signed the hash
38      const TPM2B_DIGEST* digest // IN: digest that was signed
39  );
40
41  /***CryptEccCommitCompute()
42  // This function performs the point multiply operations required by TPM2_Commit.
43  //
44  // If 'B' or 'M' is provided, they must be on the curve defined by 'curveId'. This
45  // routine does not check that they are on the curve and results are unpredictable
46  // if they are not.
47  //
48  // It is a fatal error if 'r' is NULL. If 'B' is not NULL, then it is a
49  // fatal error if 'd' is NULL or if 'K' and 'L' are both NULL.
50  // If 'M' is not NULL, then it is a fatal error if 'E' is NULL.
51  //
52  // Return Type: TPM_RC
53  //      TPM_RC_NO_RESULT      if 'K', 'L' or 'E' was computed to be the point
54  //                          at infinity
55  //      TPM_RC_CANCELED      a cancel indication was asserted during this
56  //                          function
57  LIB_EXPORT TPM_RC CryptEccCommitCompute(
58      TPMS_ECC_POINT* K, // OUT: [d]B or [r]Q
59      TPMS_ECC_POINT* L, // OUT: [r]B
60      TPMS_ECC_POINT* E, // OUT: [r]M
61      TPM_ECC_CURVE curveId, // IN: the curve for the computations
62      TPMS_ECC_POINT* M, // IN: M (optional)
63      TPMS_ECC_POINT* B, // IN: B (optional)
64      TPM2B_ECC_PARAMETER* d, // IN: d (optional)
65      TPM2B_ECC_PARAMETER* r // IN: the computed r value (required)
66  );
67  #endif // ALG_ECC
68
69  #endif // _CRYPT_ECC_SIGNATURE_FP_H_

```


6.80 /tpm/include/private/prototypes/CryptHash_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Feb 28, 2020   Time: 03:04:48PM
4  */
5
6  #ifndef _CRYPT_HASH_FP_H_
7  #define _CRYPT_HASH_FP_H_
8
9  /*** CryptHashInit()
10 // This function is called by _TPM_Init to perform the initialization operations for
11 // the library.
12 BOOL CryptHashInit(void);
13
14 /*** CryptHashStartup()
15 // This function is called by TPM2_Startup(). It checks that the size of the
16 // HashDefArray is consistent with the HASH_COUNT.
17 BOOL CryptHashStartup(void);
18
19 /*** CryptGetHashDef()
20 // This function accesses the hash descriptor associated with a hash a
21 // algorithm. The function returns a pointer to a 'null' descriptor if hashAlg is
22 // TPM_ALG_NULL or not a defined algorithm.
23 PHASH_DEF
24 CryptGetHashDef(TPM_ALG_ID hashAlg);
25
26 /*** CryptHashIsValidAlg()
27 // This function tests to see if an algorithm ID is a valid hash algorithm. If
28 // flag is true, then TPM_ALG_NULL is a valid hash.
29 // Return Type: BOOL
30 //     TRUE(1)      hashAlg is a valid, implemented hash on this TPM
31 //     FALSE(0)     hashAlg is not valid for this TPM
32 BOOL CryptHashIsValidAlg(TPM_ALG_ID hashAlg, // IN: the algorithm to check
33                          BOOL flag // IN: TRUE if TPM_ALG_NULL is to be treated
34                                // as a valid hash
35 );
36
37 /*** CryptHashGetAlgByIndex()
38 // This function is used to iterate through the hashes. TPM_ALG_NULL
39 // is returned for all indexes that are not valid hashes.
40 // If the TPM implements 3 hashes, then an 'index' value of 0 will
41 // return the first implemented hash and an 'index' of 2 will return the
42 // last. All other index values will return TPM_ALG_NULL.
43 //
44 // Return Type: TPM_ALG_ID
45 // TPM_ALG_***      a hash algorithm
46 // TPM_ALG_NULL     this can be used as a stop value
47 LIB_EXPORT TPM_ALG_ID CryptHashGetAlgByIndex(UINT32 index // IN: the index
48 );
49
50 /*** CryptHashGetDigestSize()
51 // Returns the size of the digest produced by the hash. If 'hashAlg' is not a hash
52 // algorithm, the TPM will FAIL.
53 // Return Type: UINT16
54 //     0          TPM_ALG_NULL
55 //     > 0        the digest size
56 //
57 LIB_EXPORT UINT16 CryptHashGetDigestSize(
58     TPM_ALG_ID hashAlg // IN: hash algorithm to look up
59 );
60
61 /*** CryptHashGetBlockSize()
62 // Returns the size of the block used by the hash. If 'hashAlg' is not a hash
63 // algorithm, the TPM will FAIL.
64 // Return Type: UINT16

```



```

65 // 0      TPM_ALG_NULL
66 // > 0    the digest size
67 //
68 LIB_EXPORT UINT16 CryptHashGetBlockSize(
69     TPM_ALG_ID hashAlg // IN: hash algorithm to look up
70 );
71
72 /*** CryptHashGetOid()
73 // This function returns a pointer to DER-encoded OID for a hash algorithm. All OIDs
74 // are full OID values including the Tag (0x06) and length byte.
75 LIB_EXPORT const BYTE* CryptHashGetOid(TPM_ALG_ID hashAlg);
76
77 /*** CryptHashGetContextAlg()
78 // This function returns the hash algorithm associated with a hash context.
79 TPM_ALG_ID
80 CryptHashGetContextAlg(PHASH_STATE state // IN: the context to check
81 );
82
83 /*** CryptHashCopyState
84 // This function is used to clone a HASH_STATE.
85 LIB_EXPORT void CryptHashCopyState(HASH_STATE* out, // OUT: destination of the state
86     const HASH_STATE* in // IN: source of the state
87 );
88
89 /*** CryptHashExportState()
90 // This function is used to export a hash or HMAC hash state. This function
91 // would be called when preparing to context save a sequence object.
92 void CryptHashExportState(
93     PCHASH_STATE internalFmt, // IN: the hash state formatted for use by
94     // library
95     PEXPORT_HASH_STATE externalFmt // OUT: the exported hash state
96 );
97
98 /*** CryptHashImportState()
99 // This function is used to import the hash state. This function
100 // would be called to import a hash state when the context of a sequence object
101 // was being loaded.
102 void CryptHashImportState(
103     PHASH_STATE internalFmt, // OUT: the hash state formatted for use by
104     // the library
105     PCEXPORT_HASH_STATE externalFmt // IN: the exported hash state
106 );
107
108 /*** CryptHashStart()
109 // Functions starts a hash stack
110 // Start a hash stack and returns the digest size. As a side effect, the
111 // value of 'stateSize' in hashState is updated to indicate the number of bytes
112 // of state that were saved. This function calls GetHashServer() and that function
113 // will put the TPM into failure mode if the hash algorithm is not supported.
114 //
115 // This function does not use the sequence parameter. If it is necessary to import
116 // or export context, this will start the sequence in a local state
117 // and export the state to the input buffer. Will need to add a flag to the state
118 // structure to indicate that it needs to be imported before it can be used.
119 // (BLEH).
120 // Return Type: UINT16
121 // 0      hash is TPM_ALG_NULL
122 // >0     digest size
123 LIB_EXPORT UINT16 CryptHashStart(
124     PHASH_STATE hashState, // OUT: the running hash state
125     TPM_ALG_ID hashAlg // IN: hash algorithm
126 );
127
128 /*** CryptDigestUpdate()
129 // Add data to a hash or HMAC, SMAC stack.
130 //

```

```

131 void CryptDigestUpdate(PHASH_STATE hashState, // IN: the hash context information
132                        UINT32      dataSize,   // IN: the size of data to be added
133                        const BYTE* data        // IN: data to be hashed
134 );
135
136 /*** CryptHashEnd()
137 // Complete a hash or HMAC computation. This function will place the smaller of
138 // 'digestSize' or the size of the digest in 'dOut'. The number of bytes in the
139 // placed in the buffer is returned. If there is a failure, the returned value
140 // is <= 0.
141 // Return Type: UINT16
142 //      0      no data returned
143 //      > 0    the number of bytes in the digest or dOutSize, whichever is smaller
144 LIB_EXPORT UINT16 CryptHashEnd(PHASH_STATE hashState, // IN: the state of hash stack
145                               UINT32      dOutSize,   // IN: size of digest buffer
146                               BYTE*      dOut        // OUT: hash digest
147 );
148
149 /*** CryptHashBlock()
150 // Start a hash, hash a single block, update 'digest' and return the size of
151 // the results.
152 //
153 // The 'digestSize' parameter can be smaller than the digest. If so, only the more
154 // significant bytes are returned.
155 // Return Type: UINT16
156 //      >= 0    number of bytes placed in 'dOut'
157 LIB_EXPORT UINT16 CryptHashBlock(TPM_ALG_ID hashAlg, // IN: The hash algorithm
158                                 UINT32      dataSize, // IN: size of buffer to hash
159                                 const BYTE* data,      // IN: the buffer to hash
160                                 UINT32 dOutSize, // IN: size of the digest buffer
161                                 BYTE* dOut      // OUT: digest buffer
162 );
163
164 /*** CryptDigestUpdate2B()
165 // This function updates a digest (hash or HMAC) with a TPM2B.
166 //
167 // This function can be used for both HMAC and hash functions so the
168 // 'digestState' is void so that either state type can be passed.
169 LIB_EXPORT void CryptDigestUpdate2B(PHASH_STATE state, // IN: the digest state
170                                    const TPM2B* bIn    // IN: 2B containing the data
171 );
172
173 /*** CryptHashEnd2B()
174 // This function is the same as CryptCompleteHash() but the digest is
175 // placed in a TPM2B. This is the most common use and this is provided
176 // for specification clarity. 'digest.size' should be set to indicate the number of
177 // bytes to place in the buffer
178 // Return Type: UINT16
179 //      >=0    the number of bytes placed in 'digest.buffer'
180 LIB_EXPORT UINT16 CryptHashEnd2B(
181     PHASH_STATE state, // IN: the hash state
182     P2B         digest // IN: the size of the buffer Out: requested
183     //      number of bytes
184 );
185
186 /*** CryptDigestUpdateInt()
187 // This function is used to include an integer value to a hash stack. The function
188 // marshals the integer into its canonical form before calling CryptDigestUpdate().
189 LIB_EXPORT void CryptDigestUpdateInt(
190     void* state, // IN: the state of hash stack
191     UINT32 intSize, // IN: the size of 'intValue' in bytes
192     UINT64 intValue // IN: integer value to be hashed
193 );
194
195 /*** CryptHmacStart()
196 // This function is used to start an HMAC using a temp

```

```

197 // hash context. The function does the initialization
198 // of the hash with the HMAC key XOR iPad and updates the
199 // HMAC key XOR oPad.
200 //
201 // The function returns the number of bytes in a digest produced by 'hashAlg'.
202 // Return Type: UINT16
203 // >= 0      number of bytes in digest produced by 'hashAlg' (may be zero)
204 //
205 LIB_EXPORT UINT16 CryptHmacStart(PHMAC_STATE state, // IN/OUT: the state buffer
206                                TPM_ALG_ID hashAlg, // IN: the algorithm to use
207                                UINT16 keySize, // IN: the size of the HMAC key
208                                const BYTE* key // IN: the HMAC key
209 );
210
211 /*** CryptHmacEnd()
212 // This function is called to complete an HMAC. It will finish the current
213 // digest, and start a new digest. It will then add the oPadKey and the
214 // completed digest and return the results in dOut. It will not return more
215 // than dOutSize bytes.
216 // Return Type: UINT16
217 // >= 0      number of bytes in 'dOut' (may be zero)
218 LIB_EXPORT UINT16 CryptHmacEnd(PHMAC_STATE state, // IN: the hash state buffer
219                               UINT32 dOutSize, // IN: size of digest buffer
220                               BYTE* dOut // OUT: hash digest
221 );
222
223 /*** CryptHmacStart2B()
224 // This function starts an HMAC and returns the size of the digest
225 // that will be produced.
226 //
227 // This function is provided to support the most common use of starting an HMAC
228 // with a TPM2B key.
229 //
230 // The caller must provide a block of memory in which the hash sequence state
231 // is kept. The caller should not alter the contents of this buffer until the
232 // hash sequence is completed or abandoned.
233 //
234 // Return Type: UINT16
235 // > 0      the digest size of the algorithm
236 // = 0      the hashAlg was TPM_ALG_NULL
237 LIB_EXPORT UINT16 CryptHmacStart2B(
238     PHMAC_STATE hmacState, // OUT: the state of HMAC stack. It will be used
239                             // in HMAC update and completion
240     TPMI_ALG_HASH hashAlg, // IN: hash algorithm
241     P2B key // IN: HMAC key
242 );
243
244 /*** CryptHmacEnd2B()
245 // This function is the same as CryptHmacEnd() but the HMAC result
246 // is returned in a TPM2B which is the most common use.
247 // Return Type: UINT16
248 // >=0      the number of bytes placed in 'digest'
249 LIB_EXPORT UINT16 CryptHmacEnd2B(
250     PHMAC_STATE hmacState, // IN: the state of HMAC stack
251     P2B digest // OUT: HMAC
252 );
253
254 /*** Mask and Key Generation Functions
255 /*** CryptMGF_KDF()
256 // This function performs MGF1/KDF1 or KDF2 using the selected hash. KDF1 and KDF2 are
257 // T('n') = T('n'-1) || H('seed' || 'counter') with the difference being that, with
258 // KDF1, 'counter' starts at 0 but with KDF2, 'counter' starts at 1. The caller
259 // determines which version by setting the initial value of counter to either 0 or 1.
260 // Note: Any value that is not 0 is considered to be 1.
261 //
262 // This function returns the length of the mask produced which

```

```

263 // could be zero if the digest algorithm is not supported
264 // Return Type: UINT16
265 //      0      hash algorithm was TPM_ALG_NULL
266 //      > 0    should be the same as 'mSize'
267 LIB_EXPORT UINT16 CryptMGF_KDF(UINT32 mSize, // IN: length of the mask to be produced
268                                BYTE* mask, // OUT: buffer to receive the mask
269                                TPM_ALG_ID hashAlg, // IN: hash to use
270                                UINT32 seedSize, // IN: size of the seed
271                                BYTE* seed, // IN: seed size
272                                UINT32 counter // IN: counter initial value
273 );
274
275 /*** CryptKDFa()
276 // This function performs the key generation according to Part 1 of the
277 // TPM specification.
278 //
279 // This function returns the number of bytes generated which may be zero.
280 //
281 // The 'key' and 'keyStream' pointers are not allowed to be NULL. The other
282 // pointer values may be NULL. The value of 'sizeInBits' must be no larger
283 // than (2^18)-1 = 256K bits (32385 bytes).
284 //
285 // The 'once' parameter is set to allow incremental generation of a large
286 // value. If this flag is TRUE, 'sizeInBits' will be used in the HMAC computation
287 // but only one iteration of the KDF is performed. This would be used for
288 // XOR obfuscation so that the mask value can be generated in digest-sized
289 // chunks rather than having to be generated all at once in an arbitrarily
290 // large buffer and then XORed into the result. If 'once' is TRUE, then
291 // 'sizeInBits' must be a multiple of 8.
292 //
293 // Any error in the processing of this command is considered fatal.
294 // Return Type: UINT16
295 //      0      hash algorithm is not supported or is TPM_ALG_NULL
296 //      > 0    the number of bytes in the 'keyStream' buffer
297 LIB_EXPORT UINT16 CryptKDFa(
298     TPM_ALG_ID hashAlg, // IN: hash algorithm used in HMAC
299     const TPM2B* key, // IN: HMAC key
300     const TPM2B* label, // IN: a label for the KDF
301     const TPM2B* contextU, // IN: context U
302     const TPM2B* contextV, // IN: context V
303     UINT32 sizeInBits, // IN: size of generated key in bits
304     BYTE* keyStream, // OUT: key buffer
305     UINT32* counterInOut, // IN/OUT: caller may provide the iteration
306                          // counter for incremental operations to
307                          // avoid large intermediate buffers.
308     UINT16 blocks, // IN: If non-zero, this is the maximum number
309                  // of blocks to be returned, regardless
310                  // of sizeInBits
311 );
312
313 /*** CryptKDFe()
314 // This function implements KDFe() as defined in TPM specification part 1.
315 //
316 // This function returns the number of bytes generated which may be zero.
317 //
318 // The 'Z' and 'keyStream' pointers are not allowed to be NULL. The other
319 // pointer values may be NULL. The value of 'sizeInBits' must be no larger
320 // than (2^18)-1 = 256K bits (32385 bytes).
321 // Any error in the processing of this command is considered fatal.
322 // Return Type: UINT16
323 //      0      hash algorithm is not supported or is TPM_ALG_NULL
324 //      > 0    the number of bytes in the 'keyStream' buffer
325 //
326 LIB_EXPORT UINT16 CryptKDFe(TPM_ALG_ID hashAlg, // IN: hash algorithm used in HMAC
327                             TPM2B* Z, // IN: Z
328                             const TPM2B* label, // IN: a label value for the KDF

```

```

329         TPM2B*      partyUInfo, // IN: PartyUInfo
330         TPM2B*      partyVInfo, // IN: PartyVInfo
331         UINT32 sizeInBits, // IN: size of generated key in bits
332         BYTE*  keyStream // OUT: key buffer
333     );
334
335 #endif // _CRYPT_HASH_FP_H_

```

6.81 /tpm/include/private/prototypes/CryptPrimeSieve_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Aug 30, 2019   Time: 02:11:54PM
4  */
5
6  #ifndef _CRYPT_PRIME_SIEVE_FP_H_
7  #define _CRYPT_PRIME_SIEVE_FP_H_
8
9  #if RSA_KEY_SIEVE
10
11  /*** RsaAdjustPrimeLimit()
12  // This used during the sieve process. The iterator for getting the
13  // next prime (RsaNextPrime()) will return primes until it hits the
14  // limit (primeLimit) set up by this function. This causes the sieve
15  // process to stop when an appropriate number of primes have been
16  // sieved.
17  LIB_EXPORT void RsaAdjustPrimeLimit(uint32_t requestedPrimes);
18
19  /*** RsaNextPrime()
20  // This the iterator used during the sieve process. The input is the
21  // last prime returned (or any starting point) and the output is the
22  // next higher prime. The function returns 0 when the primeLimit is
23  // reached.
24  LIB_EXPORT uint32_t RsaNextPrime(uint32_t lastPrime);
25
26  /*** FindNthSetBit()
27  // This function finds the nth SET bit in a bit array. The 'n' parameter is
28  // between 1 and the number of bits in the array (always a multiple of 8).
29  // If called when the array does not have n bits set, it will return -1
30  // Return Type: unsigned int
31  //   <0      no bit is set or no bit with the requested number is set
32  //   >=0      the number of the bit in the array that is the nth set
33  LIB_EXPORT int FindNthSetBit(
34      const UINT16 aSize, // IN: the size of the array to check
35      const BYTE* a,      // IN: the array to check
36      const UINT32 n      // IN, the number of the SET bit
37  );
38
39  /*** PrimeSieve()
40  // This function does a prime sieve over the input 'field' which has as its
41  // starting address the value in bnN. Since this initializes the Sieve
42  // using a precomputed field with the bits associated with 3, 5 and 7 already
43  // turned off, the value of pnN may need to be adjusted by a few counts to allow
44  // the precomputed field to be used without modification.
45  //
46  // To get better performance, one could address the issue of developing the
47  // composite numbers. When the size of the prime gets large, the time for doing
48  // the divisions goes up, noticeably. It could be better to develop larger composite
49  // numbers even if they need to be Crypt_Int*'s themselves. The object would be to
50  // reduce the number of times that the large prime is divided into a few large
51  // divides and then use smaller divides to get to the final 16 bit (or smaller)
52  // remainders.
53  LIB_EXPORT UINT32 PrimeSieve(Crypt_Int* bnN, // IN/OUT: number to sieve
54      UINT32 fieldSize, // IN: size of the field area in bytes
55      BYTE* field // IN: field

```



```

56 );
57 # ifdef SIEVE_DEBUG
58
59 /***SetFieldSize()
60 // Function to set the field size used for prime generation. Used for tuning.
61 LIB_EXPORT uint32_t SetFieldSize(uint32_t newFieldSize);
62 # endif // SIEVE_DEBUG
63
64 /*** PrimeSelectWithSieve()
65 // This function will sieve the field around the input prime candidate. If the
66 // sieve field is not empty, one of the one bits in the field is chosen for testing
67 // with Miller-Rabin. If the value is prime, 'pnP' is updated with this value
68 // and the function returns success. If this value is not prime, another
69 // pseudo-random candidate is chosen and tested. This process repeats until
70 // all values in the field have been checked. If all bits in the field have
71 // been checked and none is prime, the function returns FALSE and a new random
72 // value needs to be chosen.
73 // Return Type: TPM_RC
74 //     TPM_RC_FAILURE      TPM in failure mode, probably due to entropy source
75 //     TPM_RC_SUCCESS      candidate is probably prime
76 //     TPM_RC_NO_RESULT    candidate is not prime and couldn't find an alternative
77 //                          in the field
78 LIB_EXPORT TPM_RC PrimeSelectWithSieve(
79     Crypt_Int* candidate, // IN/OUT: The candidate to filter
80     UINT32     e,         // IN: the exponent
81     RAND_STATE* rand      // IN: the random number generator state
82 );
83 # if RSA_INSTRUMENT
84
85 /*** PrintTuple()
86 char* PrintTuple(UINT32* i);
87
88 /*** RsaSimulationEnd()
89 void RsaSimulationEnd(void);
90
91 /*** GetSieveStats()
92 LIB_EXPORT void GetSieveStats(
93     uint32_t* trials, uint32_t* emptyFields, uint32_t* averageBits);
94 # endif
95 #endif // RSA_KEY_SIEVE
96 #if !RSA_INSTRUMENT
97
98 /*** RsaSimulationEnd()
99 // Stub for call when not doing instrumentation.
100 void RsaSimulationEnd(void);
101 #endif
102
103 #endif // _CRYPT_PRIME_SIEVE_FP_H_

```

6.82 /tpm/include/private/prototypes/CryptPrime_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Aug 30, 2019   Time: 02:11:54PM
4  */
5
6  #ifndef _CRYPT_PRIME_FP_H_
7  #define _CRYPT_PRIME_FP_H_
8
9  /*** IsPrimeInt()
10 // This will do a test of a word of up to 32-bits in size.
11 BOOL IsPrimeInt(uint32_t n);
12
13 /*** TpmMath_IsProbablyPrime()
14 // This function is used when the key sieve is not implemented. This function

```



```

15 // Will try to eliminate some of the obvious things before going on
16 // to perform MillerRabin as a final verification of primeness.
17 BOOL TpmMath_IsProbablyPrime(Crypt_Int* prime, // IN:
18                             RAND_STATE* rand // IN: the random state just
19                             // in case Miller-Rabin is required
20 );
21
22 /*** MillerRabinRounds()
23 // Function returns the number of Miller-Rabin rounds necessary to give an
24 // error probability equal to the security strength of the prime. These values
25 // are from FIPS 186-3.
26 UINT32
27 MillerRabinRounds(UINT32 bits // IN: Number of bits in the RSA prime
28 );
29
30 /*** MillerRabin()
31 // This function performs a Miller-Rabin test from FIPS 186-3. It does
32 // 'iterations' trials on the number. In all likelihood, if the number
33 // is not prime, the first test fails.
34 // Return Type: BOOL
35 // TRUE(1)          probably prime
36 // FALSE(0)         composite
37 BOOL MillerRabin(Crypt_Int* bnW, RAND_STATE* rand);
38 #if ALG_RSA
39
40 /*** RsaCheckPrime()
41 // This will check to see if a number is prime and appropriate for an
42 // RSA prime.
43 //
44 // This has different functionality based on whether we are using key
45 // sieving or not. If not, the number checked to see if it is divisible by
46 // the public exponent, then the number is adjusted either up or down
47 // in order to make it a better candidate. It is then checked for being
48 // probably prime.
49 //
50 // If sieving is used, the number is used to root a sieving process.
51 //
52 TPM_RC
53 RsaCheckPrime(Crypt_Int* prime, UINT32 exponent, RAND_STATE* rand);
54
55 /*** TpmRsa_GeneratePrimeForRSA()
56 // Function to generate a prime of the desired size with the proper attributes
57 // for an RSA prime.
58 TPM_RC
59 TpmRsa_GeneratePrimeForRSA(
60     Crypt_Int* prime, // IN/OUT: points to the BN that will get the
61                       // random value
62     UINT32 bits, // IN: number of bits to get
63     UINT32 exponent, // IN: the exponent
64     RAND_STATE* rand // IN: the random state
65 );
66 #endif // ALG_RSA
67
68 #endif // _CRYPT_PRIME_FP_H_

```

6.83 /tpm/include/private/prototypes/CryptRand_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 4, 2020 Time: 02:36:44PM
4 */
5
6 #ifndef _CRYPT_RAND_FP_H_
7 #define _CRYPT_RAND_FP_H_
8

```

```

9  /*** DRBG_GetEntropy()
10 // Even though this implementation never fails, it may get blocked
11 // indefinitely long in the call to get entropy from the platform
12 // (DRBG_GetEntropy32()).
13 // This function is only used during instantiation of the DRBG for
14 // manufacturing and on each start-up after an non-orderly shutdown.
15 //
16 // Return Type: BOOL
17 //     TRUE(1)      requested entropy returned
18 //     FALSE(0)     entropy Failure
19 BOOL DRBG_GetEntropy(UINT32 requiredEntropy, // IN: requested number of bytes of full
20 //                                     // entropy
21 //                                     BYTE* entropy // OUT: buffer to return collected entropy
22 );
23
24 /*** IncrementIv()
25 // This function increments the IV value by 1. It is used by EncryptDRBG().
26 void IncrementIv(DRBG_IV* iv);
27
28 /*** DRBG_Reseed()
29 // This function is used when reseeding of the DRBG is required. If
30 // entropy is provided, it is used in lieu of using hardware entropy.
31 // Note: the provided entropy must be the required size.
32 //
33 // Return Type: BOOL
34 //     TRUE(1)      reseed succeeded
35 //     FALSE(0)     reseed failed, probably due to the entropy generation
36 BOOL DRBG_Reseed(DRBG_STATE* drbgState, // IN: the state to update
37                 DRBG_SEED* providedEntropy, // IN: entropy
38                 DRBG_SEED* additionalData // IN:
39 );
40
41 /*** DRBG_SelfTest()
42 // This is run when the DRBG is instantiated and at startup.
43 //
44 // Return Type: BOOL
45 //     TRUE(1)      test OK
46 //     FALSE(0)     test failed
47 BOOL DRBG_SelfTest(void);
48
49 /*** CryptRandomStir()
50 // This function is used to cause a reseed. A DRBG_SEED amount of entropy is
51 // collected from the hardware and then additional data is added.
52 //
53 // Return Type: TPM_RC
54 //     TPM_RC_NO_RESULT failure of the entropy generator
55 LIB_EXPORT TPM_RC CryptRandomStir(UINT16 additionalDataSize, BYTE* additionalData);
56
57 /*** CryptRandomGenerate()
58 // Generate a 'randomSize' number of random bytes.
59 LIB_EXPORT UINT16 CryptRandomGenerate(UINT16 randomSize, BYTE* buffer);
60
61 /*** DRBG_InstantiateSeededKdf()
62 // This function is used to instantiate a KDF-based RNG. This is used for derivations.
63 // This function always returns TRUE.
64 LIB_EXPORT BOOL DRBG_InstantiateSeededKdf(
65     KDF_STATE* state, // OUT: buffer to hold the state
66     TPM_ALG_ID hashAlg, // IN: hash algorithm
67     TPM_ALG_ID kdf, // IN: the KDF to use
68     TPM2B* seed, // IN: the seed to use
69     const TPM2B* label, // IN: a label for the generation process.
70     TPM2B* context, // IN: the context value
71     UINT32 limit // IN: Maximum number of bits from the KDF
72 );
73
74 /*** DRBG_AdditionalData()

```

```

75 // Function to reseed the DRBG with additional entropy. This is normally called
76 // before computing the protection value of a primary key in the Endorsement
77 // hierarchy.
78 LIB_EXPORT void DRBG_AdditionalData(DRBG_STATE* drbgState, // IN:OUT state to update
79                                     TPM2B* additionalData // IN: value to incorporate
80 );
81
82 /*** DRBG_InstantiateSeeded()
83 // This function is used to instantiate a random number generator from seed values.
84 // The nominal use of this generator is to create sequences of pseudo-random
85 // numbers from a seed value.
86 //
87 // Return Type: TPM_RC
88 // TPM_RC_FAILURE DRBG self-test failure
89 LIB_EXPORT TPM_RC DRBG_InstantiateSeeded(
90     DRBG_STATE* drbgState, // IN/OUT: buffer to hold the state
91     const TPM2B* seed,      // IN: the seed to use
92     const TPM2B* purpose,   // IN: a label for the generation process.
93     const TPM2B* name,      // IN: name of the object
94     const TPM2B* additional // IN: additional data
95 );
96
97 /*** CryptRandStartup()
98 // This function is called when TPM_Startup is executed. This function always returns
99 // TRUE.
100 LIB_EXPORT BOOL CryptRandStartup(void);
101
102 /*** CryptRandInit()
103 // This function is called when _TPM_Init is being processed.
104 //
105 // Return Type: BOOL
106 // TRUE(1) success
107 // FALSE(0) failure
108 LIB_EXPORT BOOL CryptRandInit(void);
109
110 /*** DRBG_Generate()
111 // This function generates a random sequence according SP800-90A.
112 // If 'random' is not NULL, then 'randomSize' bytes of random values are generated.
113 // If 'random' is NULL or 'randomSize' is zero, then the function returns
114 // zero without generating any bits or updating the reseed counter.
115 // This function returns the number of bytes produced which could be less than the
116 // number requested if the request is too large ("too large" is implementation
117 // dependent.)
118 LIB_EXPORT UINT16 DRBG_Generate(
119     RAND_STATE* state,
120     BYTE* random, // OUT: buffer to receive the random values
121     UINT16 randomSize // IN: the number of bytes to generate
122 );
123
124 /*** DRBG_Instantiate()
125 // This is CTR_DRBG_Instantiate_algorithm() from [SP 800-90A 10.2.1.3.1].
126 // This is called when a the TPM DRBG is to be instantiated. This is
127 // called to instantiate a DRBG used by the TPM for normal
128 // operations.
129 //
130 // Return Type: BOOL
131 // TRUE(1) instantiation succeeded
132 // FALSE(0) instantiation failed
133 LIB_EXPORT BOOL DRBG_Instantiate(
134     DRBG_STATE* drbgState, // OUT: the instantiated value
135     UINT16 pSize,          // IN: Size of personalization string
136     BYTE* personalization // IN: The personalization string
137 );
138
139 /*** DRBG_Uninstantiate()
140 // This is Uninstantiate_function() from [SP 800-90A 9.4].

```

```

141 //
142 // Return Type: TPM_RC
143 // TPM_RC_VALUE not a valid state
144 LIB_EXPORT TPM_RC DRBG_Uninstantiate(
145     DRBG_STATE* drbgState // IN/OUT: working state to erase
146 );
147
148 #endif // _CRYPT_RAND_FP_H_

```

6.84 /tpm/include/private/prototypes/CryptRsa_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Apr 2, 2019 Time: 03:18:00PM
4  */
5
6  #ifndef _CRYPT_RSA_FP_H_
7  #define _CRYPT_RSA_FP_H_
8
9  #if ALG_RSA
10
11  /*** CryptRsaInit()
12  // Function called at _TPM_Init().
13  BOOL CryptRsaInit(void);
14
15  /*** CryptRsaStartup()
16  // Function called at TPM2_Startup()
17  BOOL CryptRsaStartup(void);
18
19  /*** CryptRsaPssSaltSize()
20  // This function computes the salt size used in PSS. It is broken out so that
21  // the X509 code can get the same value that is used by the encoding function in this
22  // module.
23  INT16
24  CryptRsaPssSaltSize(INT16 hashSize, INT16 outSize);
25
26  /*** MakeDerTag()
27  // Construct the DER value that is used in RSASSA
28  // Return Type: INT16
29  // > 0 size of value
30  // <= 0 no hash exists
31  INT16
32  MakeDerTag(TPM_ALG_ID hashAlg, INT16 sizeOfBuffer, BYTE* buffer);
33
34  /*** CryptRsaSelectScheme()
35  // This function is used by TPM2_RSA_Decrypt and TPM2_RSA_Encrypt. It sets up
36  // the rules to select a scheme between input and object default.
37  // This function assume the RSA object is loaded.
38  // If a default scheme is defined in object, the default scheme should be chosen,
39  // otherwise, the input scheme should be chosen.
40  // In the case that both the object and 'scheme' are not TPM_ALG_NULL, then
41  // if the schemes are the same, the input scheme will be chosen.
42  // if the scheme are not compatible, a NULL pointer will be returned.
43  //
44  // The return pointer may point to a TPM_ALG_NULL scheme.
45  TPMT_RSA_DECRYPT* CryptRsaSelectScheme(
46     TPMI_DH_OBJECT rsaHandle, // IN: handle of an RSA key
47     TPMT_RSA_DECRYPT* scheme // IN: a sign or decrypt scheme
48 );
49
50  /*** CryptRsaLoadPrivateExponent()
51  // This function is called to generate the private exponent of an RSA key.
52  // Return Type: TPM_RC
53  // TPM_RC_BINDING public and private parts of 'rsaKey' are not matched
54  TPM_RC

```

```

55 CryptRsaLoadPrivateExponent(TPMT_PUBLIC* publicArea, TPMT_SENSITIVE* sensitive);
56
57 /*** CryptRsaEncrypt()
58 // This is the entry point for encryption using RSA. Encryption is
59 // use of the public exponent. The padding parameter determines what
60 // padding will be used.
61 //
62 // The 'cOutSize' parameter must be at least as large as the size of the key.
63 //
64 // If the padding is RSA_PAD_NONE, 'dIn' is treated as a number. It must be
65 // lower in value than the key modulus.
66 // NOTE: If dIn has fewer bytes than cOut, then we don't add low-order zeros to
67 // dIn to make it the size of the RSA key for the call to RSAEP. This is
68 // because the high order bytes of dIn might have a numeric value that is
69 // greater than the value of the key modulus. If this had low-order zeros
70 // added, it would have a numeric value larger than the modulus even though
71 // it started out with a lower numeric value.
72 //
73 // Return Type: TPM_RC
74 //     TPM_RC_VALUE      'cOutSize' is too small (must be the size
75 //                        of the modulus)
76 //     TPM_RC_SCHEME     'padType' is not a supported scheme
77 //
78 LIB_EXPORT TPM_RC CryptRsaEncrypt(
79     TPM2B_PUBLIC_KEY_RSA* cOut,    // OUT: the encrypted data
80     TPM2B* dIn,                  // IN: the data to encrypt
81     OBJECT* key,                 // IN: the key used for encryption
82     TPMT_RSA_DECRYPT* scheme,     // IN: the type of padding and hash
83                                     // if needed
84     const TPM2B* label,          // IN: in case it is needed
85     RAND_STATE* rand             // IN: random number generator
86                                     // state (mostly for testing)
87 );
88
89 /*** CryptRsaDecrypt()
90 // This is the entry point for decryption using RSA. Decryption is
91 // use of the private exponent. The 'padType' parameter determines what
92 // padding was used.
93 //
94 // Return Type: TPM_RC
95 //     TPM_RC_SIZE      'cInSize' is not the same as the size of the public
96 //                        modulus of 'key'; or numeric value of the encrypted
97 //                        data is greater than the modulus
98 //     TPM_RC_VALUE     'dOutSize' is not large enough for the result
99 //     TPM_RC_SCHEME    'padType' is not supported
100 //
101 LIB_EXPORT TPM_RC CryptRsaDecrypt(
102     TPM2B* dOut,    // OUT: the decrypted data
103     TPM2B* cIn,    // IN: the data to decrypt
104     OBJECT* key,   // IN: the key to use for decryption
105     TPMT_RSA_DECRYPT* scheme, // IN: the padding scheme
106     const TPM2B* label // IN: in case it is needed for the scheme
107 );
108
109 /*** CryptRsaSign()
110 // This function is used to generate an RSA signature of the type indicated in
111 // 'scheme'.
112 //
113 // Return Type: TPM_RC
114 //     TPM_RC_SCHEME    'scheme' or 'hashAlg' are not supported
115 //     TPM_RC_VALUE     'hInSize' does not match 'hashAlg' (for RSASSA)
116 //
117 LIB_EXPORT TPM_RC CryptRsaSign(TPMT_SIGNATURE* sigOut,
118     OBJECT* key, // IN: key to use
119     TPM2B_DIGEST* hIn, // IN: the digest to sign
120     RAND_STATE* rand // IN: the random number generator

```



```

121                                     //      to use (mostly for testing)
122 );
123
124 /*** CryptRsaValidateSignature()
125 // This function is used to validate an RSA signature. If the signature is valid
126 // TPM_RC_SUCCESS is returned. If the signature is not valid, TPM_RC_SIGNATURE is
127 // returned. Other return codes indicate either parameter problems or fatal errors.
128 //
129 // Return Type: TPM_RC
130 //      TPM_RC_SIGNATURE      the signature does not check
131 //      TPM_RC_SCHEME         unsupported scheme or hash algorithm
132 //
133 LIB_EXPORT TPM_RC CryptRsaValidateSignature(
134     TPMT_SIGNATURE* sig,    // IN: signature
135     OBJECT*         key,    // IN: public modulus
136     TPM2B_DIGEST*   digest // IN: The digest being validated
137 );
138
139 /*** CryptRsaGenerateKey()
140 // Generate an RSA key from a provided seed
141 // Return Type: TPM_RC
142 //      TPM_RC_CANCELED      operation was canceled
143 //      TPM_RC_RANGE         public exponent is not supported
144 //      TPM_RC_VALUE         could not find a prime using the provided parameters
145 LIB_EXPORT TPM_RC CryptRsaGenerateKey(
146     TPMT_PUBLIC*   publicArea,
147     TPMT_SENSITIVE* sensitive,
148     RAND_STATE*    rand // IN: if not NULL, the deterministic
149                        //      RNG state
150 );
151 #endif // ALG_RSA
152
153 #endif // _CRYPT_RSA_FP_H_

```

6.85 /tpm/include/private/prototypes/CryptSelfTest_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar  4, 2020 Time: 02:36:44PM
4  */
5
6  #ifndef _CRYPT_SELF_TEST_FP_H_
7  #define _CRYPT_SELF_TEST_FP_H_
8
9  /*** CryptSelfTest()
10 // This function is called to start/complete a full self-test.
11 // If 'fullTest' is NO, then only the untested algorithms will be run. If
12 // 'fullTest' is YES, then 'g_untestedDecryptionAlgorithms' is reinitialized and then
13 // all tests are run.
14 // This implementation of the reference design does not support processing outside
15 // the framework of a TPM command. As a consequence, this command does not
16 // complete until all tests are done. Since this can take a long time, the TPM
17 // will check after each test to see if the command is canceled. If so, then the
18 // TPM will returned TPM_RC_CANCELED. To continue with the self-tests, call
19 // TPM2_SelfTest(fullTest == No) and the TPM will complete the testing.
20 // Return Type: TPM_RC
21 //      TPM_RC_CANCELED      if the command is canceled
22 LIB_EXPORT
23 TPM_RC
24 CryptSelfTest(TPMI_YES_NO fullTest // IN: if full test is required
25 );
26
27 /*** CryptIncrementalSelfTest()
28 // This function is used to perform an incremental self-test. This implementation
29 // will perform the toTest values before returning. That is, it assumes that the

```



```

30 // TPM cannot perform background tasks between commands.
31 //
32 // This command may be canceled. If it is, then there is no return result.
33 // However, this command can be run again and the incremental progress will not
34 // be lost.
35 // Return Type: TPM_RC
36 //     TPM_RC_CANCELED      processing of this command was canceled
37 //     TPM_RC_TESTING       if toTest list is not empty
38 //     TPM_RC_VALUE         an algorithm in the toTest list is not implemented
39 TPM_RC
40 CryptIncrementalSelfTest(TPML_ALG* toTest,    // IN: list of algorithms to be tested
41                        TPML_ALG* toDoList    // OUT: list of algorithms needing test
42 );
43
44 /*** CryptInitializeToTest()
45 // This function will initialize the data structures for testing all the
46 // algorithms. This should not be called unless CryptAlgsSetImplemented() has
47 // been called
48 void CryptInitializeToTest(void);
49
50 /*** CryptTestAlgorithm()
51 // Only point of contact with the actual self tests. If a self-test fails, there
52 // is no return and the TPM goes into failure mode.
53 // The call to TestAlgorithm uses an algorithm selector and a bit vector. When the
54 // test is run, the corresponding bit in 'toTest' and in 'g_toTest' is CLEAR. If
55 // 'toTest' is NULL, then only the bit in 'g_toTest' is CLEAR.
56 // There is a special case for the call to TestAlgorithm(). When 'alg' is
57 // ALG_ERROR, TestAlgorithm() will CLEAR any bit in 'toTest' for which it has
58 // no test. This allows the knowledge about which algorithms have test to be
59 // accessed through the interface that provides the test.
60 // Return Type: TPM_RC
61 //     TPM_RC_CANCELED      test was canceled
62 LIB_EXPORT
63 TPM_RC
64 CryptTestAlgorithm(TPM_ALG_ID alg, ALGORITHM_VECTOR* toTest);
65
66 #endif // _CRYPT_SELF_TEST_FP_H

```

6.86 /tpm/include/private/prototypes/CryptSmac_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 28, 2019 Time: 08:25:19PM
4  */
5
6  #ifndef _CRYPT_SMAC_FP_H_
7  #define _CRYPT_SMAC_FP_H_
8
9  #if SMAC_IMPLEMENTED
10
11  /*** CryptSmacStart()
12  // Function to start an SMAC.
13  UINT16
14  CryptSmacStart(HASH_STATE* state,
15                TPMU_PUBLIC_PARMS* keyParameters,
16                TPM_ALG_ID macAlg, // IN: the type of MAC
17                TPM2B* key);
18
19  /*** CryptMacStart()
20  // Function to start either an HMAC or an SMAC. Cannot reuse the CryptHmacStart
21  // function because of the difference in number of parameters.
22  UINT16
23  CryptMacStart(HMAC_STATE* state,
24                TPMU_PUBLIC_PARMS* keyParameters,
25                TPM_ALG_ID macAlg, // IN: the type of MAC

```

```

26         TPM2B*          key);
27
28     /*** CryptMacEnd()
29     // Dispatch to the MAC end function using a size and buffer pointer.
30     UINT16
31     CryptMacEnd(HMAC_STATE* state, UINT32 size, BYTE* buffer);
32
33     /*** CryptMacEnd2B()
34     // Dispatch to the MAC end function using a 2B.
35     UINT16
36     CryptMacEnd2B(HMAC_STATE* state, TPM2B* data);
37     #endif // SMAC_IMPLEMENTED
38
39     #endif // _CRYPT_SMAC_FP_H_

```

6.87 /tpm/include/private/prototypes/CryptSym_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Apr  2, 2019   Time: 03:18:00PM
4  */
5
6  #ifndef _CRYPT_SYM_FP_H_
7  #define _CRYPT_SYM_FP_H_
8
9  /*** Initialization and Data Access Functions
10
11  /*** CryptSymInit()
12  // This function is called to do _TPM_Init processing
13  BOOL CryptSymInit(void);
14
15  /*** CryptSymStartup()
16  // This function is called to do TPM2_Startup() processing
17  BOOL CryptSymStartup(void);
18
19  /*** CryptGetSymmetricBlockSize()
20  // This function returns the block size of the algorithm. The table of bit sizes has
21  // an entry for each allowed key size. The entry for a key size is 0 if the TPM does
22  // not implement that key size. The key size table is delimited with a negative number
23  // (-1). After the delimiter is a list of block sizes with each entry corresponding
24  // to the key bit size. For most symmetric algorithms, the block size is the same
25  // regardless of the key size but this arrangement allows them to be different.
26  // Return Type: INT16
27  //   <= 0    cipher not supported
28  //   > 0      the cipher block size in bytes
29  LIB_EXPORT INT16 CryptGetSymmetricBlockSize(
30      TPM_ALG_ID symmetricAlg, // IN: the symmetric algorithm
31      UINT16     keySizeInBits // IN: the key size
32  );
33
34  /*** Symmetric Encryption
35  // This function performs symmetric encryption based on the mode.
36  // Return Type: TPM_RC
37  //   TPM_RC_SIZE      'dSize' is not a multiple of the block size for an
38  //                     algorithm that requires it
39  //   TPM_RC_FAILURE    Fatal error
40  LIB_EXPORT TPM_RC CryptSymmetricEncrypt(
41      BYTE*      dOut, // OUT:
42      TPM_ALG_ID algorithm, // IN: the symmetric algorithm
43      UINT16     keySizeInBits, // IN: key size in bits
44      const BYTE* key, // IN: key buffer. The size of this buffer
45                      // in bytes is (keySizeInBits + 7) / 8
46      TPM2B_IV* ivInOut, // IN/OUT: IV for decryption.
47      TPM_ALG_ID mode, // IN: Mode to use
48      INT32      dSize, // IN: data size (may need to be a

```

```

49                                     // multiple of the blockSize)
50     const BYTE* dIn                  // IN: data buffer
51 );
52
53 /*** CryptSymmetricDecrypt()
54 // This function performs symmetric decryption based on the mode.
55 // Return Type: TPM_RC
56 //     TPM_RC_FAILURE      A fatal error
57 //     TPM_RCS_SIZE        'dSize' is not a multiple of the block size for an
58 //                          algorithm that requires it
59 LIB_EXPORT TPM_RC CryptSymmetricDecrypt(
60     BYTE*      dOut,                // OUT: decrypted data
61     TPM_ALG_ID algorithm,           // IN: the symmetric algorithm
62     UINT16     keySizeInBits,       // IN: key size in bits
63     const BYTE* key,                // IN: key buffer. The size of this buffer
64                                     // in bytes is (keySizeInBits + 7) / 8
65     TPM2B_IV*  ivInOut,             // IN/OUT: IV for decryption.
66     TPM_ALG_ID mode,                // IN: Mode to use
67     INT32      dSize,               // IN: data size (may need to be a
68                                     // multiple of the blockSize)
69     const BYTE* dIn                  // IN: data buffer
70 );
71
72 /*** CryptSymKeyValidate()
73 // Validate that a provided symmetric key meets the requirements of the TPM
74 // Return Type: TPM_RC
75 //     TPM_RC_KEY_SIZE     Key size specifiers do not match
76 //     TPM_RC_KEY          Key is not allowed
77 TPM_RC
78 CryptSymKeyValidate(TPMT_SYM_DEF_OBJECT* symDef, TPM2B_SYM_KEY* key);
79
80 #endif // _CRYPT_SYM_FP_H

```

6.88 /tpm/include/private/prototypes/CryptUtil_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Aug 30, 2019   Time: 02:11:54PM
4  */
5
6  #ifndef CRYPT_UTIL_FP_H
7  #define CRYPT_UTIL_FP_H
8
9  /*** CryptIsSchemeAnonymous()
10 // This function is used to test a scheme to see if it is an anonymous scheme
11 // The only anonymous scheme is ECDSA. ECDSA can be used to do things
12 // like U-Prove.
13 BOOL CryptIsSchemeAnonymous(TPM_ALG_ID scheme // IN: the scheme algorithm to test
14 );
15
16 /*** ParmDecryptSym()
17 // This function performs parameter decryption using symmetric block cipher.
18 void ParmDecryptSym(TPM_ALG_ID symAlg,          // IN: the symmetric algorithm
19     TPM_ALG_ID hash,                          // IN: hash algorithm for KDFa
20     UINT16     keySizeInBits,                  // IN: the key size in bits
21     TPM2B*     key,                           // IN: KDF HMAC key
22     TPM2B*     nonceCaller,                   // IN: nonce caller
23     TPM2B*     nonceTpm,                     // IN: nonce TPM
24     UINT32     dataSize,                      // IN: size of parameter buffer
25     BYTE*      data                          // OUT: buffer to be decrypted
26 );
27
28 /*** ParmEncryptSym()
29 // This function performs parameter encryption using symmetric block cipher.
30 void ParmEncryptSym(TPM_ALG_ID symAlg,          // IN: symmetric algorithm

```

```

31         TPM_ALG_ID hash,           // IN: hash algorithm for KDFa
32         UINT16      keySizeInBits, // IN: symmetric key size in bits
33         TPM2B*      key,           // IN: KDF HMAC key
34         TPM2B*      nonceCaller,   // IN: nonce caller
35         TPM2B*      nonceTpm,      // IN: nonce TPM
36         UINT32      dataSize,      // IN: size of parameter buffer
37         BYTE*       data           // OUT: buffer to be encrypted
38     );
39
40     /*** CryptXORObfuscation()
41     // This function implements XOR obfuscation. It should not be called if the
42     // hash algorithm is not implemented. The only return value from this function
43     // is TPM_RC_SUCCESS.
44     void CryptXORObfuscation(TPM_ALG_ID hash, // IN: hash algorithm for KDF
45                             TPM2B*      key, // IN: KDF key
46                             TPM2B*      contextU, // IN: contextU
47                             TPM2B*      contextV, // IN: contextV
48                             UINT32      dataSize, // IN: size of data buffer
49                             BYTE*      data // IN/OUT: data to be XORed in place
50     );
51
52     /*** CryptInit()
53     // This function is called when the TPM receives a _TPM_Init indication.
54     //
55     // NOTE: The hash algorithms do not have to be tested, they just need to be
56     // available. They have to be tested before the TPM can accept HMAC authorization
57     // or return any result that relies on a hash algorithm.
58     // Return Type: BOOL
59     //     TRUE(1)      initializations succeeded
60     //     FALSE(0)     initialization failed and caller should place the TPM into
61     //                  Failure Mode
62     BOOL CryptInit(void);
63
64     /*** CryptStartup()
65     // This function is called by TPM2_Startup() to initialize the functions in
66     // this cryptographic library and in the provided CryptoLibrary. This function
67     // and CryptUtilInit() are both provided so that the implementation may move the
68     // initialization around to get the best interaction.
69     // Return Type: BOOL
70     //     TRUE(1)      startup succeeded
71     //     FALSE(0)     startup failed and caller should place the TPM into
72     //                  Failure Mode
73     BOOL CryptStartup(STARTUP_TYPE type // IN: the startup type
74     );
75
76     /*** Algorithm-Independent Functions
77     /*** Introduction
78     // These functions are used generically when a function of a general type
79     // (e.g., symmetric encryption) is required. The functions will modify the
80     // parameters as required to interface to the indicated algorithms.
81     //
82     /*** CryptIsAsymAlgorithm()
83     // This function indicates if an algorithm is an asymmetric algorithm.
84     // Return Type: BOOL
85     //     TRUE(1)      if it is an asymmetric algorithm
86     //     FALSE(0)     if it is not an asymmetric algorithm
87     BOOL CryptIsAsymAlgorithm(TPM_ALG_ID algID // IN: algorithm ID
88     );
89
90     /*** CryptSecretEncrypt()
91     // This function creates a secret value and its associated secret structure using
92     // an asymmetric algorithm.
93     //
94     // This function is used by TPM2_Rewrap() TPM2_MakeCredential(),

```

```

97 // and TPM2_Duplicate().
98 // Return Type: TPM_RC
99 //     TPM_RC_ATTRIBUTES 'keyHandle' does not reference a valid decryption key
100 //     TPM_RC_KEY         invalid ECC key (public point is not on the curve)
101 //     TPM_RC_SCHEME      RSA key with an unsupported padding scheme
102 //     TPM_RC_VALUE       numeric value of the data to be decrypted is greater
103 //                        than the RSA key modulus
104 TPM_RC
105 CryptSecretEncrypt(OBJECT*      encryptKey, // IN: encryption key object
106                   const TPM2B* label,      // IN: a null-terminated string as L
107                   TPM2B_DATA* data,        // OUT: secret value
108                   TPM2B_ENCRYPTED_SECRET* secret // OUT: secret structure
109 );
110
111 /*** CryptSecretDecrypt()
112 // Decrypt a secret value by asymmetric (or symmetric) algorithm
113 // This function is used for ActivateCredential and Import for asymmetric
114 // decryption, and StartAuthSession for both asymmetric and symmetric
115 // decryption process
116 //
117 // Return Type: TPM_RC
118 //     TPM_RC_ATTRIBUTES RSA key is not a decryption key
119 //     TPM_RC_BINDING     Invalid RSA key (public and private parts are not
120 //                        cryptographically bound.
121 //     TPM_RC_ECC_POINT   ECC point in the secret is not on the curve
122 //     TPM_RC_INSUFFICIENT failed to retrieve ECC point from the secret
123 //     TPM_RC_NO_RESULT   multiplication resulted in ECC point at infinity
124 //     TPM_RC_SIZE        data to decrypt is not of the same size as RSA key
125 //     TPM_RC_VALUE       For RSA key, numeric value of the encrypted data is
126 //                        greater than the modulus, or the recovered data is
127 //                        larger than the output buffer.
128 //                        For keyedHash or symmetric key, the secret is
129 //                        larger than the size of the digest produced by
130 //                        the name algorithm.
131 //     TPM_RC_FAILURE     internal error
132 TPM_RC
133 CryptSecretDecrypt(OBJECT*      decryptKey, // IN: decrypt key
134                   TPM2B_NONCE* nonceCaller, // IN: nonceCaller. It is needed for
135                                           // symmetric decryption. For
136                                           // asymmetric decryption, this
137                                           // parameter is NULL
138                   const TPM2B* label,      // IN: a value for L
139                   TPM2B_ENCRYPTED_SECRET* secret, // IN: input secret
140                   TPM2B_DATA* data        // OUT: decrypted secret value
141 );
142
143 /*** CryptParameterEncryption()
144 // This function does in-place encryption of a response parameter.
145 void CryptParameterEncryption(
146     TPM_HANDLE handle, // IN: encrypt session handle
147     TPM2B* nonceCaller, // IN: nonce caller
148     INT32 bufferSize, // IN: size of parameter buffer
149     UINT16 leadingSizeInByte, // IN: the size of the leading size field in
150                               // bytes
151     TPM2B_AUTH* extraKey, // IN: additional key material other than
152                          // sessionAuth
153     BYTE* buffer // IN/OUT: parameter buffer to be encrypted
154 );
155
156 /*** CryptParameterDecryption()
157 // This function does in-place decryption of a command parameter.
158 // Return Type: TPM_RC
159 //     TPM_RC_SIZE The number of bytes in the input buffer is less than
160 //                 the number of bytes to be decrypted.
161 TPM_RC
162 CryptParameterDecryption(

```



```

163     TPM_HANDLE handle,           // IN: encrypted session handle
164     TPM2B*      nonceCaller,     // IN: nonce caller
165     INT32       bufferSize,      // IN: size of parameter buffer
166     UINT16      leadingSizeInByte, // IN: the size of the leading size field in
167                                     // byte
168     TPM2B_AUTH* extraKey,        // IN: the authValue
169     BYTE*       buffer           // IN/OUT: parameter buffer to be decrypted
170 );
171
172 /*** CryptComputeSymmetricUnique()
173 // This function computes the unique field in public area for symmetric objects.
174 void CryptComputeSymmetricUnique(
175     TPMT_PUBLIC* publicArea, // IN: the object's public area
176     TPMT_SENSITIVE* sensitive, // IN: the associated sensitive area
177     TPM2B_DIGEST* unique // OUT: unique buffer
178 );
179
180 /*** CryptCreateObject()
181 // This function creates an object.
182 // For an asymmetric key, it will create a key pair and, for a parent key, a seed
183 // value for child protections.
184 //
185 // For an symmetric object, (TPM_ALG_SYMCIPHER or TPM_ALG_KEYEDHASH), it will
186 // create a secret key if the caller did not provide one. It will create a random
187 // secret seed value that is hashed with the secret value to create the public
188 // unique value.
189 //
190 // 'publicArea', 'sensitive', and 'sensitiveCreate' are the only required parameters
191 // and are the only ones that are used by TPM2_Create(). The other parameters
192 // are optional and are used when the generated Object needs to be deterministic.
193 // This is the case for both Primary Objects and Derived Objects.
194 //
195 // When a seed value is provided, a RAND_STATE will be populated and used for
196 // all operations in the object generation that require a random number. In the
197 // simplest case, TPM2_CreatePrimary() will use 'seed', 'label' and 'context' with
198 // context being the hash of the template. If the Primary Object is in
199 // the Endorsement hierarchy, it will also populate 'proof' with ehProof.
200 //
201 // For derived keys, 'seed' will be the secret value from the parent, 'label' and
202 // 'context' will be set according to the parameters of TPM2_CreateLoaded() and
203 // 'hashAlg' will be set which causes the RAND_STATE to be a KDF generator.
204 //
205 // Return Type: TPM_RC
206 //     TPM_RC_KEY          a provided key is not an allowed value
207 //     TPM_RC_KEY_SIZE     key size in the public area does not match the size
208 //                         in the sensitive creation area for a symmetric key
209 //     TPM_RC_NO_RESULT    unable to get random values (only in derivation)
210 //     TPM_RC_RANGE        for an RSA key, the exponent is not supported
211 //     TPM_RC_SIZE         sensitive data size is larger than allowed for the
212 //                         scheme for a keyed hash object
213 //     TPM_RC_VALUE        exponent is not prime or could not find a prime using
214 //                         the provided parameters for an RSA key;
215 //                         unsupported name algorithm for an ECC key
216 TPM_RC
217 CryptCreateObject(OBJECT* object, // IN: new object structure pointer
218                 TPMS_SENSITIVE_CREATE* sensitiveCreate, // IN: sensitive creation
219                 RAND_STATE* rand // IN: the random number generator
220                 // to use
221 );
222
223 /*** CryptGetSignHashAlg()
224 // Get the hash algorithm of signature from a TPMT_SIGNATURE structure.
225 // It assumes the signature is not NULL
226 // This is a function for easy access
227 TPMI_ALG_HASH
228 CryptGetSignHashAlg(TPMT_SIGNATURE* auth // IN: signature

```



```

229 );
230
231 /*** CryptIsSplitSign()
232 // This function is used to determine if the signing operation is a split
233 // signing operation that required a TPM2_Commit().
234 //
235 BOOL CryptIsSplitSign(TPM_ALG_ID scheme // IN: the algorithm selector
236 );
237
238 /*** CryptIsAsymSignScheme()
239 // This function indicates if a scheme algorithm is a sign algorithm.
240 BOOL CryptIsAsymSignScheme(TPMI_ALG_PUBLIC publicType, // IN: Type of the object
241 TPMI_ALG_ASYM_SCHEME scheme // IN: the scheme
242 );
243
244 /*** CryptIsAsymDecryptScheme()
245 // This function indicate if a scheme algorithm is a decrypt algorithm.
246 BOOL CryptIsAsymDecryptScheme(TPMI_ALG_PUBLIC publicType, // IN: Type of the object
247 TPMI_ALG_ASYM_SCHEME scheme // IN: the scheme
248 );
249
250 /*** CryptSelectSignScheme()
251 // This function is used by the attestation and signing commands. It implements
252 // the rules for selecting the signature scheme to use in signing. This function
253 // requires that the signing key either be TPM_RH_NULL or be loaded.
254 //
255 // If a default scheme is defined in object, the default scheme should be chosen,
256 // otherwise, the input scheme should be chosen.
257 // In the case that both object and input scheme has a non-NULL scheme
258 // algorithm, if the schemes are compatible, the input scheme will be chosen.
259 //
260 // This function should not be called if 'signObject->publicArea.type' ==
261 // ALG_SYMCIPHER.
262 //
263 // Return Type: BOOL
264 // TRUE(1) scheme selected
265 // FALSE(0) both 'scheme' and key's default scheme are empty; or
266 // 'scheme' is empty while key's default scheme requires
267 // explicit input scheme (split signing); or
268 // non-empty default key scheme differs from 'scheme'
269 BOOL CryptSelectSignScheme(OBJECT* signObject, // IN: signing key
270 TPMT_SIG_SCHEME* scheme // IN/OUT: signing scheme
271 );
272
273 /*** CryptSign()
274 // Sign a digest with asymmetric key or HMAC.
275 // This function is called by attestation commands and the generic TPM2_Sign
276 // command.
277 // This function checks the key scheme and digest size. It does not
278 // check if the sign operation is allowed for restricted key. It should be
279 // checked before the function is called.
280 // The function will assert if the key is not a signing key.
281 //
282 // Return Type: TPM_RC
283 // TPM_RC_SCHEME 'signScheme' is not compatible with the signing key type
284 // TPM_RC_VALUE 'digest' value is greater than the modulus of
285 // 'signHandle' or size of 'hashData' does not match hash
286 // algorithm in 'signScheme' (for an RSA key);
287 // invalid commit status or failed to generate "r" value
288 // (for an ECC key)
289 TPM_RC
290 CryptSign(OBJECT* signKey, // IN: signing key
291 TPMT_SIG_SCHEME* signScheme, // IN: sign scheme.
292 TPM2B_DIGEST* digest, // IN: The digest being signed
293 TPMT_SIGNATURE* signature // OUT: signature
294 );

```

```

295
296 /*** CryptValidateSignature()
297 // This function is used to verify a signature. It is called by
298 // TPM2_VerifySignature() and TPM2_PolicySigned.
299 //
300 // Since this operation only requires use of a public key, no consistency
301 // checks are necessary for the key to signature type because a caller can load
302 // any public key that they like with any scheme that they like. This routine
303 // simply makes sure that the signature is correct, whatever the type.
304 //
305 // Return Type: TPM_RC
306 //     TPM_RC_SIGNATURE      the signature is not genuine
307 //     TPM_RC_SCHEME         the scheme is not supported
308 //     TPM_RC_HANDLE         an HMAC key was selected but the
309 //                           private part of the key is not loaded
310 TPM_RC
311 CryptValidateSignature(TPMI_DH_OBJECT keyHandle, // IN: The handle of sign key
312                      TPM2B_DIGEST* digest,      // IN: The digest being validated
313                      TPMT_SIGNATURE* signature  // IN: signature
314 );
315
316 /*** CryptGetTestResult
317 // This function returns the results of a self-test function.
318 // Note: the behavior in this function is NOT the correct behavior for a real
319 // TPM implementation. An artificial behavior is placed here due to the
320 // limitation of a software simulation environment. For the correct behavior,
321 // consult the part 3 specification for TPM2_GetTestResult().
322 TPM_RC
323 CryptGetTestResult(TPM2B_MAX_BUFFER* outData // OUT: test result data
324 );
325
326 /*** CryptValidateKeys()
327 // This function is used to verify that the key material of and object is valid.
328 // For a 'publicOnly' object, the key is verified for size and, if it is an ECC
329 // key, it is verified to be on the specified curve. For a key with a sensitive
330 // area, the binding between the public and private parts of the key are verified.
331 // If the nameAlg of the key is TPM_ALG_NULL, then the size of the sensitive area
332 // is verified but the public portion is not verified, unless the key is an RSA key.
333 // For an RSA key, the reason for loading the sensitive area is to use it. The
334 // only way to use a private RSA key is to compute the private exponent. To compute
335 // the private exponent, the public modulus is used.
336 // Return Type: TPM_RC
337 //     TPM_RC_BINDING      the public and private parts are not cryptographically
338 //                           bound
339 //     TPM_RC_HASH          cannot have a publicOnly key with nameAlg of TPM_ALG_NULL
340 //     TPM_RC_KEY           the public unique is not valid
341 //     TPM_RC_KEY_SIZE      the private area key is not valid
342 //     TPM_RC_TYPE          the types of the sensitive and private parts do not match
343 TPM_RC
344 CryptValidateKeys(TPMT_PUBLIC* publicArea,
345                  TPMT_SENSITIVE* sensitive,
346                  TPM_RC blamePublic,
347                  TPM_RC blameSensitive);
348
349 /*** CryptSelectMac()
350 // This function is used to set the MAC scheme based on the key parameters and
351 // the input scheme.
352 // Return Type: TPM_RC
353 //     TPM_RC_SCHEME      the scheme is not a valid mac scheme
354 //     TPM_RC_TYPE         the input key is not a type that supports a mac
355 //     TPM_RC_VALUE        the input scheme and the key scheme are not compatible
356 TPM_RC
357 CryptSelectMac(TPMT_PUBLIC* publicArea, TPMI_ALG_MAC_SCHEME* inMac);
358
359 /*** CryptMacIsValidForKey()
360 // Check to see if the key type is compatible with the mac type

```

```

361  BOOL CryptMacIsValidForKey(TPM_ALG_ID keyType, TPM_ALG_ID macAlg, BOOL flag);
362
363  /*** CryptSmacIsValidAlg()
364  // This function is used to test if an algorithm is a supported SMAC algorithm. It
365  // needs to be updated as new algorithms are added.
366  BOOL CryptSmacIsValidAlg(TPM_ALG_ID alg,
367                          BOOL      FLAG // IN: Indicates if TPM_ALG_NULL is valid
368  );
369
370  /*** CryptSymModeIsValid()
371  // Function checks to see if an algorithm ID is a valid, symmetric block cipher
372  // mode for the TPM. If 'flag' is SET, then TPM_ALG_NULL is a valid mode.
373  // not include the modes used for SMAC
374  BOOL CryptSymModeIsValid(TPM_ALG_ID mode, BOOL flag);
375
376  #endif // _CRYPT_UTIL_FP_H_

```

6.89 /tpm/include/private/prototypes/DA_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Apr  2, 2019   Time: 04:23:27PM
4  */
5
6  #ifndef _DA_FP_H_
7  #define _DA_FP_H_
8
9  /*** DAPreInstall_Init()
10 // This function initializes the DA parameters to their manufacturer-default
11 // values. The default values are determined by a platform-specific specification.
12 //
13 // This function should not be called outside of a manufacturing or simulation
14 // environment.
15 //
16 // The DA parameters will be restored to these initial values by TPM2_Clear().
17 void DAPreInstall_Init(void);
18
19 /*** DASTartup()
20 // This function is called by TPM2_Startup() to initialize the DA parameters.
21 // In the case of Startup(CLEAR), use of lockoutAuth will be enabled if the
22 // lockout recovery time is 0. Otherwise, lockoutAuth will not be enabled until
23 // the TPM has been continuously powered for the lockoutRecovery time.
24 //
25 // This function requires that NV be available and not rate limiting.
26 BOOL DASTartup(STARTUP_TYPE type // IN: startup type
27 );
28
29 /*** DARegisterFailure()
30 // This function is called when a authorization failure occurs on an entity
31 // that is subject to dictionary-attack protection. When a DA failure is
32 // triggered, register the failure by resetting the relevant self-healing
33 // timer to the current time.
34 void DARegisterFailure(TPM_HANDLE handle // IN: handle for failure
35 );
36
37 /*** DASelfHeal()
38 // This function is called to check if sufficient time has passed to allow
39 // decrement of failedTries or to re-enable use of lockoutAuth.
40 //
41 // This function should be called when the time interval is updated.
42 void DASelfHeal(void);
43
44 #endif // _DA_FP_H_

```

6.90 /tpm/include/private/prototypes/DictionaryAttackLockReset_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_DictionaryAttackLockReset  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_DICTIONARYATTACKLOCKRESET_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_DICTIONARYATTACKLOCKRESET_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_LOCKOUT lockHandle;
12 } DictionaryAttackLockReset_In;
13
14 // Response code modifiers
15 #  define RC_DictionaryAttackLockReset_lockHandle (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_DictionaryAttackLockReset(DictionaryAttackLockReset_In* in);
20
21 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_DICTIONARYATTACKLOCKRESET_FP_H_
22 #endif  // CC_DictionaryAttackLockReset

```

6.91 /tpm/include/private/prototypes/DictionaryAttackParameters_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_DictionaryAttackParameters  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_DICTIONARYATTACKPARAMETERS_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_DICTIONARYATTACKPARAMETERS_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_LOCKOUT lockHandle;
12     UINT32          newMaxTries;
13     UINT32          newRecoveryTime;
14     UINT32          lockoutRecovery;
15 } DictionaryAttackParameters_In;
16
17 // Response code modifiers
18 #  define RC_DictionaryAttackParameters_lockHandle (TPM_RC_H + TPM_RC_1)
19 #  define RC_DictionaryAttackParameters_newMaxTries (TPM_RC_P + TPM_RC_1)
20 #  define RC_DictionaryAttackParameters_newRecoveryTime (TPM_RC_P + TPM_RC_2)
21 #  define RC_DictionaryAttackParameters_lockoutRecovery (TPM_RC_P + TPM_RC_3)
22
23 // Function prototype
24 TPM_RC
25 TPM2_DictionaryAttackParameters(DictionaryAttackParameters_In* in);
26
27 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_DICTIONARYATTACKPARAMETERS_FP_H_
28 #endif  // CC_DictionaryAttackParameters

```

6.92 /tpm/include/private/prototypes/Duplicate_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_Duplicate  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_DUPLICATE_FP_H_

```

```

6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_DUPLICATE_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT    objectHandle;
12     TPMI_DH_OBJECT    newParentHandle;
13     TPM2B_DATA         encryptionKeyIn;
14     TPMT_SYM_DEF_OBJECT symmetricAlg;
15 } Duplicate_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_DATA         encryptionKeyOut;
21     TPM2B_PRIVATE      duplicate;
22     TPM2B_ENCRYPTED_SECRET outSymSeed;
23 } Duplicate_Out;
24
25 // Response code modifiers
26 #   define RC_Duplicate_objectHandle    (TPM_RC_H + TPM_RC_1)
27 #   define RC_Duplicate_newParentHandle (TPM_RC_H + TPM_RC_2)
28 #   define RC_Duplicate_encryptionKeyIn (TPM_RC_P + TPM_RC_1)
29 #   define RC_Duplicate_symmetricAlg    (TPM_RC_P + TPM_RC_2)
30
31 // Function prototype
32 TPM_RC
33 TPM2_Duplicate(Duplicate_In* in, Duplicate_Out* out);
34
35 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_DUPLICATE_FP_H_
36 #endif // CC_Duplicate

```

6.93 /tpm/include/private/prototypes/ECC_Decrypt_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_ECC_Decrypt // Command must be enabled
4
5  #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECC_DECRYPT_FP_H_
6  #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECC_DECRYPT_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT    keyHandle;
12     TPM2B_ECC_POINT    C1;
13     TPM2B_MAX_BUFFER    C2;
14     TPM2B_DIGEST        C3;
15     TPMT_KDF_SCHEME    inScheme;
16 } ECC_Decrypt_In;
17
18 // Output structure definition
19 typedef struct
20 {
21     TPM2B_MAX_BUFFER    plainText;
22 } ECC_Decrypt_Out;
23
24 // Response code modifiers
25 #   define RC_ECC_Decrypt_keyHandle    (TPM_RC_H + TPM_RC_1)
26 #   define RC_ECC_Decrypt_C1           (TPM_RC_P + TPM_RC_1)
27 #   define RC_ECC_Decrypt_C2           (TPM_RC_P + TPM_RC_2)
28 #   define RC_ECC_Decrypt_C3           (TPM_RC_P + TPM_RC_3)
29 #   define RC_ECC_Decrypt_inScheme     (TPM_RC_P + TPM_RC_4)
30
31 // Function prototype

```

```

32 TPM_RC
33 TPM2_ECC_Decrypt(ECC_Decrypt_In* in, ECC_Decrypt_Out* out);
34
35 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECC_DECRYPT_FP_H_
36 #endif // CC_ECC_Decrypt

```

6.94 /tpm/include/private/prototypes/ECC_Encrypt_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ECC_Encrypt // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECC_ENCRYPT_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECC_ENCRYPT_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT keyHandle;
12     TPM2B_MAX_BUFFER plainText;
13     TPMT_KDF_SCHEME inScheme;
14 } ECC_Encrypt_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM2B_ECC_POINT C1;
20     TPM2B_MAX_BUFFER C2;
21     TPM2B_DIGEST C3;
22 } ECC_Encrypt_Out;
23
24 // Response code modifiers
25 #   define RC_ECC_Encrypt_keyHandle (TPM_RC_H + TPM_RC_1)
26 #   define RC_ECC_Encrypt_plainText (TPM_RC_P + TPM_RC_1)
27 #   define RC_ECC_Encrypt_inScheme (TPM_RC_P + TPM_RC_2)
28
29 // Function prototype
30 TPM_RC
31 TPM2_ECC_Encrypt(ECC_Encrypt_In* in, ECC_Encrypt_Out* out);
32
33 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECC_ENCRYPT_FP_H_
34 #endif // CC_ECC_Encrypt

```

6.95 /tpm/include/private/prototypes/ECC_Parameters_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ECC_Parameters // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECC_PARAMETERS_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECC_PARAMETERS_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_ECC_CURVE curveID;
12 } ECC_Parameters_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPMS_ALGORITHM_DETAIL_ECC parameters;
18 } ECC_Parameters_Out;
19

```



```

20 // Response code modifiers
21 #   define RC_ECC_Parameters_curveID (TPM_RC_P + TPM_RC_1)
22
23 // Function prototype
24 TPM_RC
25 TPM2_ECC_Parameters(ECC_Parameters_In* in, ECC_Parameters_Out* out);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECC_PARAMETERS_FP_H_
28 #endif // CC_ECC_Parameters

```

6.96 /tpm/include/private/prototypes/ECDH_KeyGen_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ECDH_KeyGen // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECDH_KEYGEN_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECDH_KEYGEN_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT keyHandle;
12 } ECDH_KeyGen_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPM2B_ECC_POINT zPoint;
18     TPM2B_ECC_POINT pubPoint;
19 } ECDH_KeyGen_Out;
20
21 // Response code modifiers
22 #   define RC_ECDH_KeyGen_keyHandle (TPM_RC_H + TPM_RC_1)
23
24 // Function prototype
25 TPM_RC
26 TPM2_ECDH_KeyGen(ECDH_KeyGen_In* in, ECDH_KeyGen_Out* out);
27
28 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECDH_KEYGEN_FP_H_
29 #endif // CC_ECDH_KeyGen

```

6.97 /tpm/include/private/prototypes/ECDH_ZGen_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ECDH_ZGen // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECDH_ZGEN_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECDH_ZGEN_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT keyHandle;
12     TPM2B_ECC_POINT inPoint;
13 } ECDH_ZGen_In;
14
15 // Output structure definition
16 typedef struct
17 {
18     TPM2B_ECC_POINT outPoint;
19 } ECDH_ZGen_Out;
20

```

```

21 // Response code modifiers
22 # define RC_ECDH_ZGen_keyHandle (TPM_RC_H + TPM_RC_1)
23 # define RC_ECDH_ZGen_inPoint (TPM_RC_P + TPM_RC_1)
24
25 // Function prototype
26 TPM_RC
27 TPM2_ECDH_ZGen(ECDH_ZGen_In* in, ECDH_ZGen_Out* out);
28
29 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ECDH_ZGEN_FP_H_
30 #endif // CC_ECDH_ZGen

```

6.98 /tpm/include/private/prototypes/EC_Ephemeral_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_EC_Ephemeral // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_EC_EPHEMERAL_FP_H_
6 # define _TPM_INCLUDE_PRIVATE_PROTOTYPES_EC_EPHEMERAL_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_ECC_CURVE curveID;
12 } EC_Ephemeral_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPM2B_ECC_POINT Q;
18     UINT16 counter;
19 } EC_Ephemeral_Out;
20
21 // Response code modifiers
22 # define RC_EC_Ephemeral_curveID (TPM_RC_P + TPM_RC_1)
23
24 // Function prototype
25 TPM_RC
26 TPM2_EC_Ephemeral(EC_Ephemeral_In* in, EC_Ephemeral_Out* out);
27
28 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_EC_EPHEMERAL_FP_H_
29 #endif // CC_EC_Ephemeral

```

6.99 /tpm/include/private/prototypes/EncryptDecrypt2_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_EncryptDecrypt2 // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ENCRYPTDECRYPT2_FP_H_
6 # define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ENCRYPTDECRYPT2_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT keyHandle;
12     TPM2B_MAX_BUFFER inData;
13     TPMI_YES_NO decrypt;
14     TPMI_ALG_CIPHER_MODE mode;
15     TPM2B_IV ivIn;
16 } EncryptDecrypt2_In;
17
18 // Output structure definition
19 typedef struct

```

```

20 {
21     TPM2B_MAX_BUFFER outData;
22     TPM2B_IV          ivOut;
23 } EncryptDecrypt2_Out;
24
25 // Response code modifiers
26 #   define RC_EncryptDecrypt2_keyHandle (TPM_RC_H + TPM_RC_1)
27 #   define RC_EncryptDecrypt2_inData   (TPM_RC_P + TPM_RC_1)
28 #   define RC_EncryptDecrypt2_decrypt (TPM_RC_P + TPM_RC_2)
29 #   define RC_EncryptDecrypt2_mode    (TPM_RC_P + TPM_RC_3)
30 #   define RC_EncryptDecrypt2_ivIn    (TPM_RC_P + TPM_RC_4)
31
32 // Function prototype
33 TPM_RC
34 TPM2_EncryptDecrypt2(EncryptDecrypt2_In* in, EncryptDecrypt2_Out* out);
35
36 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ENCRYPTDECRYPT2_FP_H_
37 #endif // CC_EncryptDecrypt2

```

6.100 /tpm/include/private/prototypes/EncryptDecrypt_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_EncryptDecrypt // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ENCRYPTDECRYPT_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ENCRYPTDECRYPT_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT    keyHandle;
12     TPMI_YES_NO       decrypt;
13     TPMI_ALG_CIPHER_MODE mode;
14     TPM2B_IV          ivIn;
15     TPM2B_MAX_BUFFER  inData;
16 } EncryptDecrypt_In;
17
18 // Output structure definition
19 typedef struct
20 {
21     TPM2B_MAX_BUFFER outData;
22     TPM2B_IV          ivOut;
23 } EncryptDecrypt_Out;
24
25 // Response code modifiers
26 #   define RC_EncryptDecrypt_keyHandle (TPM_RC_H + TPM_RC_1)
27 #   define RC_EncryptDecrypt_decrypt   (TPM_RC_P + TPM_RC_1)
28 #   define RC_EncryptDecrypt_mode      (TPM_RC_P + TPM_RC_2)
29 #   define RC_EncryptDecrypt_ivIn      (TPM_RC_P + TPM_RC_3)
30 #   define RC_EncryptDecrypt_inData    (TPM_RC_P + TPM_RC_4)
31
32 // Function prototype
33 TPM_RC
34 TPM2_EncryptDecrypt(EncryptDecrypt_In* in, EncryptDecrypt_Out* out);
35
36 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ENCRYPTDECRYPT_FP_H_
37 #endif // CC_EncryptDecrypt

```

6.101 /tpm/include/private/prototypes/EncryptDecrypt_spt_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 28, 2019 Time: 08:25:18PM

```

```

4  */
5
6  #ifndef _ENCRYPT_DECRYPT_SPT_FP_H_
7  #define _ENCRYPT_DECRYPT_SPT_FP_H_
8
9  #if CC_EncryptDecrypt2
10
11  // Return Type: TPM_RC
12  // TPM_RC_KEY is not a symmetric decryption key with both
13  // public and private portions loaded
14  // TPM_RC_SIZE 'IvIn' size is incompatible with the block cipher mode;
15  // or 'inData' size is not an even multiple of the block
16  // size for CBC or ECB mode
17  // TPM_RC_VALUE 'keyHandle' is restricted and the argument 'mode' does
18  // not match the key's mode
19  TPM_RC
20  EncryptDecryptShared(TPMI_DH_OBJECT keyHandleIn,
21                      TPMI_YES_NO decryptIn,
22                      TPMI_ALG_SYM_MODE modeIn,
23                      TPM2B_IV* ivIn,
24                      TPM2B_MAX_BUFFER* inData,
25                      EncryptDecrypt_Out* out);
26 #endif // CC_EncryptDecrypt
27
28 #endif // _ENCRYPT_DECRYPT_SPT_FP_H_

```

6.102 /tpm/include/private/prototypes/Entity_fp.h

```

1  /* (Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 7, 2020 Time: 07:19:36PM
4  */
5
6  #ifndef _ENTITY_FP_H_
7  #define _ENTITY_FP_H_
8
9  /** Functions
10  /** EntityGetLoadStatus()
11  // This function will check that all the handles access loaded entities.
12  // Return Type: TPM_RC
13  // TPM_RC_HANDLE handle type does not match
14  // TPM_RC_REFERENCE_Hx entity is not present
15  // TPM_RC_HIERARCHY entity belongs to a disabled hierarchy
16  // TPM_RC_OBJECT_MEMORY handle is an evict object but there is no
17  // space to load it to RAM
18  TPM_RC
19  EntityGetLoadStatus(COMMAND* command // IN/OUT: command parsing structure
20  );
21
22  /** EntityGetAuthValue()
23  // This function is used to access the 'authValue' associated with a handle.
24  // This function assumes that the handle references an entity that is accessible
25  // and the handle is not for a persistent objects. That is EntityGetLoadStatus()
26  // should have been called. Also, the accessibility of the authValue should have
27  // been verified by IsAuthValueAvailable().
28  //
29  // This function copies the authorization value of the entity to 'auth'.
30  // Return Type: UINT16
31  // count number of bytes in the authValue with 0's stripped
32  UINT16
33  EntityGetAuthValue(TPMI_DH_ENTITY handle, // IN: handle of entity
34                    TPM2B_AUTH* auth // OUT: authValue of the entity
35  );
36
37  /** EntityGetAuthPolicy()

```

```

38 // This function is used to access the 'authPolicy' associated with a handle.
39 // This function assumes that the handle references an entity that is accessible
40 // and the handle is not for a persistent objects. That is EntityGetLoadStatus()
41 // should have been called. Also, the accessibility of the authPolicy should have
42 // been verified by IsAuthPolicyAvailable().
43 //
44 // This function copies the authorization policy of the entity to 'authPolicy'.
45 //
46 // The return value is the hash algorithm for the policy.
47 TPMI_ALG_HASH
48 EntityGetAuthPolicy(TPMI_DH_ENTITY handle,      // IN: handle of entity
49                   TPM2B_DIGEST* authPolicy    // OUT: authPolicy of the entity
50 );
51
52 /*** EntityGetName()
53 // This function returns the Name associated with a handle.
54 TPM2B_NAME* EntityGetName(TPMI_DH_ENTITY handle, // IN: handle of entity
55                           TPM2B_NAME* name      // OUT: name of entity
56 );
57
58 /*** EntityGetHierarchy()
59 // This function returns the hierarchy handle associated with an entity.
60 // a) A handle that is a hierarchy handle is associated with itself.
61 // b) An NV index belongs to TPM_RH_PLATFORM if TPMA_NV_PLATFORMCREATE,
62 //    is SET, otherwise it belongs to TPM_RH_OWNER
63 // c) An object handle belongs to its hierarchy.
64 TPMI_RH_HIERARCHY
65 EntityGetHierarchy(TPMI_DH_ENTITY handle // IN :handle of entity
66 );
67
68 #endif // _ENTITY_FP_H_

```

6.103 /tpm/include/private/prototypes/EventSequenceComplete_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_EventSequenceComplete // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_EVENTSEQUENCECOMPLETE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_EVENTSEQUENCECOMPLETE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_PCR      pcrHandle;
12     TPMI_DH_OBJECT   sequenceHandle;
13     TPM2B_MAX_BUFFER buffer;
14 } EventSequenceComplete_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPML_DIGEST_VALUES results;
20 } EventSequenceComplete_Out;
21
22 // Response code modifiers
23 #   define RC_EventSequenceComplete_pcrHandle      (TPM_RC_H + TPM_RC_1)
24 #   define RC_EventSequenceComplete_sequenceHandle (TPM_RC_H + TPM_RC_2)
25 #   define RC_EventSequenceComplete_buffer         (TPM_RC_P + TPM_RC_1)
26
27 // Function prototype
28 TPM_RC
29 TPM2_EventSequenceComplete(EventSequenceComplete_In* in,
30                           EventSequenceComplete_Out* out);
31

```

```

32 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_EVENTSEQUENCECOMPLETE_FP_H_
33 #endif // CC_EventSequenceComplete

```

6.104 /tpm/include/private/prototypes/EvictControl_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_EvictControl // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_EVICTCONTROL_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_EVICTCONTROL_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PROVISION auth;
12     TPMI_DH_OBJECT objectHandle;
13     TPMI_DH_PERSISTENT persistentHandle;
14 } EvictControl_In;
15
16 // Response code modifiers
17 #   define RC_EvictControl_auth (TPM_RC_H + TPM_RC_1)
18 #   define RC_EvictControl_objectHandle (TPM_RC_H + TPM_RC_2)
19 #   define RC_EvictControl_persistentHandle (TPM_RC_P + TPM_RC_1)
20
21 // Function prototype
22 TPM_RC
23 TPM2_EvictControl(EvictControl_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_EVICTCONTROL_FP_H_
26 #endif // CC_EvictControl

```

6.105 /tpm/include/private/prototypes/FieldUpgradeData_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_FieldUpgradeData // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_FIELDUPGRADEDATA_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_FIELDUPGRADEDATA_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPM2B_MAX_BUFFER fuData;
12 } FieldUpgradeData_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPMT_HA nextDigest;
18     TPMT_HA firstDigest;
19 } FieldUpgradeData_Out;
20
21 // Response code modifiers
22 #   define RC_FieldUpgradeData_fuData (TPM_RC_P + TPM_RC_1)
23
24 // Function prototype
25 TPM_RC
26 TPM2_FieldUpgradeData(FieldUpgradeData_In* in, FieldUpgradeData_Out* out);
27
28 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_FIELDUPGRADEDATA_FP_H_
29 #endif // CC_FieldUpgradeData

```


6.106 /tpm/include/private/prototypes/FieldUpgradeStart_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_FieldUpgradeStart  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_FIELDUPGRADESTART_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_FIELDUPGRADESTART_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_PLATFORM authorization;
12     TPMI_DH_OBJECT    keyHandle;
13     TPM2B_DIGEST      fuDigest;
14     TPMT_SIGNATURE    manifestSignature;
15 } FieldUpgradeStart_In;
16
17 // Response code modifiers
18 #  define RC_FieldUpgradeStart_authorization    (TPM_RC_H + TPM_RC_1)
19 #  define RC_FieldUpgradeStart_keyHandle        (TPM_RC_H + TPM_RC_2)
20 #  define RC_FieldUpgradeStart_fuDigest        (TPM_RC_P + TPM_RC_1)
21 #  define RC_FieldUpgradeStart_manifestSignature (TPM_RC_P + TPM_RC_2)
22
23 // Function prototype
24 TPM_RC
25 TPM2_FieldUpgradeStart(FieldUpgradeStart_In* in);
26
27 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_FIELDUPGRADESTART_FP_H_
28 #endif  // CC_FieldUpgradeStart

```

6.107 /tpm/include/private/prototypes/FirmwareRead_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_FirmwareRead  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_FIRMWAREREAD_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_FIRMWAREREAD_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     UINT32 sequenceNumber;
12 } FirmwareRead_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPM2B_MAX_BUFFER fuData;
18 } FirmwareRead_Out;
19
20 // Response code modifiers
21 #  define RC_FirmwareRead_sequenceNumber (TPM_RC_P + TPM_RC_1)
22
23 // Function prototype
24 TPM_RC
25 TPM2_FirmwareRead(FirmwareRead_In* in, FirmwareRead_Out* out);
26
27 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_FIRMWAREREAD_FP_H_
28 #endif  // CC_FirmwareRead

```

6.108 /tpm/include/private/prototypes/FlushContext_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_FlushContext  // Command must be enabled
4
5  #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_FLUSHCONTEXT_FP_H_
6  #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_FLUSHCONTEXT_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_CONTEXT flushHandle;
12 } FlushContext_In;
13
14 // Response code modifiers
15 #   define RC_FlushContext_flushHandle (TPM_RC_P + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_FlushContext(FlushContext_In* in);
20
21 #   endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_FLUSHCONTEXT_FP_H_
22 #endif  // CC_FlushContext

```

6.109 /tpm/include/private/prototypes/GetCapability_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_GetCapability  // Command must be enabled
4
5  #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETCAPABILITY_FP_H_
6  #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETCAPABILITY_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPM_CAP capability;
12     UINT32 property;
13     UINT32 propertyCount;
14 } GetCapability_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPMI_YES_NO        moreData;
20     TPMS_CAPABILITY_DATA capabilityData;
21 } GetCapability_Out;
22
23 // Response code modifiers
24 #   define RC_GetCapability_capability (TPM_RC_P + TPM_RC_1)
25 #   define RC_GetCapability_property (TPM_RC_P + TPM_RC_2)
26 #   define RC_GetCapability_propertyCount (TPM_RC_P + TPM_RC_3)
27
28 // Function prototype
29 TPM_RC
30 TPM2_GetCapability(GetCapability_In* in, GetCapability_Out* out);
31
32 #   endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETCAPABILITY_FP_H_
33 #endif  // CC_GetCapability

```

6.110 /tpm/include/private/prototypes/GetCommandAuditDigest_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_GetCommandAuditDigest  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETCOMMANDAUDITDIGEST_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETCOMMANDAUDITDIGEST_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_ENDORSEMENT privacyHandle;
12     TPMI_DH_OBJECT      signHandle;
13     TPM2B_DATA           qualifyingData;
14     TPMT_SIG_SCHEME      inScheme;
15 } GetCommandAuditDigest_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_ATTEST  auditInfo;
21     TPMT_SIGNATURE signature;
22 } GetCommandAuditDigest_Out;
23
24 // Response code modifiers
25 #  define RC_GetCommandAuditDigest_privacyHandle (TPM_RC_H + TPM_RC_1)
26 #  define RC_GetCommandAuditDigest_signHandle   (TPM_RC_H + TPM_RC_2)
27 #  define RC_GetCommandAuditDigest_qualifyingData (TPM_RC_P + TPM_RC_1)
28 #  define RC_GetCommandAuditDigest_inScheme      (TPM_RC_P + TPM_RC_2)
29
30 // Function prototype
31 TPM_RC
32 TPM2_GetCommandAuditDigest(GetCommandAuditDigest_In* in,
33                           GetCommandAuditDigest_Out* out);
34
35 #  endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETCOMMANDAUDITDIGEST_FP_H_
36 #endif  // CC_GetCommandAuditDigest

```

6.111 /tpm/include/private/prototypes/GetRandom_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_GetRandom  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETRANDOM_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETRANDOM_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     UINT16 bytesRequested;
12 } GetRandom_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPM2B_DIGEST randomBytes;
18 } GetRandom_Out;
19
20 // Response code modifiers
21 #  define RC_GetRandom_bytesRequested (TPM_RC_P + TPM_RC_1)
22
23 // Function prototype
24 TPM_RC

```

```

25 TPM2_GetRandom(GetRandom_In* in, GetRandom_Out* out);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETRANDOM_FP_H_
28 #endif // CC_GetRandom

```

6.112 /tpm/include/private/prototypes/GetSessionAuditDigest_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_GetSessionAuditDigest // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETSESSIONAUDITDIGEST_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETSESSIONAUDITDIGEST_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_ENDORSEMENT privacyAdminHandle;
12     TPMI_DH_OBJECT      signHandle;
13     TPMI_SH_HMAC         sessionHandle;
14     TPM2B_DATA           qualifyingData;
15     TPMT_SIG_SCHEME      inScheme;
16 } GetSessionAuditDigest_In;
17
18 // Output structure definition
19 typedef struct
20 {
21     TPM2B_ATTEST auditInfo;
22     TPMT_SIGNATURE signature;
23 } GetSessionAuditDigest_Out;
24
25 // Response code modifiers
26 #   define RC_GetSessionAuditDigest_privacyAdminHandle (TPM_RC_H + TPM_RC_1)
27 #   define RC_GetSessionAuditDigest_signHandle        (TPM_RC_H + TPM_RC_2)
28 #   define RC_GetSessionAuditDigest_sessionHandle     (TPM_RC_H + TPM_RC_3)
29 #   define RC_GetSessionAuditDigest_qualifyingData    (TPM_RC_P + TPM_RC_1)
30 #   define RC_GetSessionAuditDigest_inScheme          (TPM_RC_P + TPM_RC_2)
31
32 // Function prototype
33 TPM_RC
34 TPM2_GetSessionAuditDigest(GetSessionAuditDigest_In* in,
35                             GetSessionAuditDigest_Out* out);
36
37 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETSESSIONAUDITDIGEST_FP_H_
38 #endif // CC_GetSessionAuditDigest

```

6.113 /tpm/include/private/prototypes/GetTestResult_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_GetTestResult // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETTESTRESULT_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETTESTRESULT_FP_H_
7
8 // Output structure definition
9 typedef struct
10 {
11     TPM2B_MAX_BUFFER outData;
12     TPM_RC            testResult;
13 } GetTestResult_Out;
14
15 // Function prototype
16 TPM_RC

```

```

17 TPM2_GetTestResult(GetTestResult_Out* out);
18
19 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETTESTRESULT_FP_H_
20 #endif // CC_GetTestResult

```

6.114 /tpm/include/private/prototypes/GetTime_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_GetTime // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETTIME_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETTIME_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_ENDORSEMENT privacyAdminHandle;
12     TPMI_DH_OBJECT      signHandle;
13     TPM2B_DATA           qualifyingData;
14     TPMT_SIG_SCHEME      inScheme;
15 } GetTime_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_ATTEST  timeInfo;
21     TPMT_SIGNATURE signature;
22 } GetTime_Out;
23
24 // Response code modifiers
25 #   define RC_GetTime_privacyAdminHandle (TPM_RC_H + TPM_RC_1)
26 #   define RC_GetTime_signHandle        (TPM_RC_H + TPM_RC_2)
27 #   define RC_GetTime_qualifyingData    (TPM_RC_P + TPM_RC_1)
28 #   define RC_GetTime_inScheme          (TPM_RC_P + TPM_RC_2)
29
30 // Function prototype
31 TPM_RC
32 TPM2_GetTime(GetTime_In* in, GetTime_Out* out);
33
34 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_GETTIME_FP_H_
35 #endif // CC_GetTime

```

6.115 /tpm/include/private/prototypes/Handle_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 28, 2019 Time: 08:25:19PM
4 */
5
6 #ifndef HANDLE_FP_H_
7 #define HANDLE_FP_H_
8
9 /*** HandleGetType()
10 // This function returns the type of a handle which is the MSO of the handle.
11 TPM_HT
12 HandleGetType(TPM_HANDLE handle // IN: a handle to be checked
13 );
14
15 /*** NextPermanentHandle()
16 // This function returns the permanent handle that is equal to the input value or
17 // is the next higher value. If there is no handle with the input value and there
18 // is no next higher value, it returns 0:
19 TPM_HANDLE

```

```

20 NextPermanentHandle(TPM_HANDLE inHandle // IN: the handle to check
21 );
22
23 /*** PermanentCapGetHandles()
24 // This function returns a list of the permanent handles of PCR, started from
25 // 'handle'. If 'handle' is larger than the largest permanent handle, an empty list
26 // will be returned with 'more' set to NO.
27 // Return Type: TPMI_YES_NO
28 //     YES     if there are more handles available
29 //     NO     all the available handles has been returned
30 TPMI_YES_NO
31 PermanentCapGetHandles(TPM_HANDLE handle, // IN: start handle
32                        UINT32 count, // IN: count of returned handles
33                        TPML_HANDLE* handleList // OUT: list of handle
34 );
35
36 /*** PermanentCapGetOneHandle()
37 // This function returns whether a permanent handle exists.
38 BOOL PermanentCapGetOneHandle(TPM_HANDLE handle // IN: handle
39 );
40
41 /*** PermanentHandleGetPolicy()
42 // This function returns a list of the permanent handles of PCR, started from
43 // 'handle'. If 'handle' is larger than the largest permanent handle, an empty list
44 // will be returned with 'more' set to NO.
45 // Return Type: TPMI_YES_NO
46 //     YES     if there are more handles available
47 //     NO     all the available handles has been returned
48 TPMI_YES_NO
49 PermanentHandleGetPolicy(TPM_HANDLE handle, // IN: start handle
50                          UINT32 count, // IN: max count of returned handles
51                          TPML_TAGGED_POLICY* policyList // OUT: list of handle
52 );
53
54 /*** PermanentHandleGetOnePolicy()
55 // This function returns a permanent handle's policy, if present.
56 BOOL PermanentHandleGetOnePolicy(TPM_HANDLE handle, // IN: handle
57                                  TPMS_TAGGED_POLICY* policy // OUT: tagged policy
58 );
59
60 #endif // _HANDLE_FP_H_

```

6.116 /tpm/include/private/prototypes/HashSequenceStart_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_HashSequenceStart // Command must be enabled
4
5 # ifndef TPM_INCLUDE_PRIVATE_PROTOTYPES_HASHSEQUENCESTART_FP_H_
6 #   define TPM_INCLUDE_PRIVATE_PROTOTYPES_HASHSEQUENCESTART_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPM2B_AUTH auth;
12     TPMI_ALG_HASH hashAlg;
13 } HashSequenceStart_In;
14
15 // Output structure definition
16 typedef struct
17 {
18     TPMI_DH_OBJECT sequenceHandle;
19 } HashSequenceStart_Out;
20
21 // Response code modifiers

```



```

22 #   define RC_HashSequenceStart_auth      (TPM_RC_P + TPM_RC_1)
23 #   define RC_HashSequenceStart_hashAlg  (TPM_RC_P + TPM_RC_2)
24
25 // Function prototype
26 TPM_RC
27 TPM2_HashSequenceStart(HashSequenceStart_In* in, HashSequenceStart_Out* out);
28
29 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_HASHSEQUENCESTART_FP_H_
30 #endif // CC_HashSequenceStart

```

6.117 /tpm/include/private/prototypes/Hash_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Hash // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_HASH_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_HASH_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPM2B_MAX_BUFFER data;
12     TPMT_ALG_HASH hashAlg;
13     TPMT_RH_HIERARCHY hierarchy;
14 } Hash_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM2B_DIGEST outHash;
20     TPMT_TK_HASHCHECK validation;
21 } Hash_Out;
22
23 // Response code modifiers
24 #   define RC_Hash_data      (TPM_RC_P + TPM_RC_1)
25 #   define RC_Hash_hashAlg  (TPM_RC_P + TPM_RC_2)
26 #   define RC_Hash_hierarchy (TPM_RC_P + TPM_RC_3)
27
28 // Function prototype
29 TPM_RC
30 TPM2_Hash(Hash_In* in, Hash_Out* out);
31
32 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_HASH_FP_H_
33 #endif // CC_Hash

```

6.118 /tpm/include/private/prototypes/HierarchyChangeAuth_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_HierarchyChangeAuth // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_HIERARCHYCHANGEAUTH_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_HIERARCHYCHANGEAUTH_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMT_RH_HIERARCHY_AUTH authHandle;
12     TPM2B_AUTH newAuth;
13 } HierarchyChangeAuth_In;
14
15 // Response code modifiers
16 #   define RC_HierarchyChangeAuth_authHandle (TPM_RC_H + TPM_RC_1)

```

```

17 # define RC_HierarchyChangeAuth_newAuth (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_HierarchyChangeAuth(HierarchyChangeAuth_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_HIERARCHYCHANGEAUTH_FP_H_
24 #endif // CC_HierarchyChangeAuth

```

6.119 /tpm/include/private/prototypes/HierarchyControl_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_HierarchyControl // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_HIERARCHYCONTROL_FP_H_
6 # define _TPM_INCLUDE_PRIVATE_PROTOTYPES_HIERARCHYCONTROL_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_BASE_HIERARCHY authHandle;
12     TPMI_RH_ENABLES enable;
13     TPMI_YES_NO state;
14 } HierarchyControl_In;
15
16 // Response code modifiers
17 # define RC_HierarchyControl_authHandle (TPM_RC_H + TPM_RC_1)
18 # define RC_HierarchyControl_enable (TPM_RC_P + TPM_RC_1)
19 # define RC_HierarchyControl_state (TPM_RC_P + TPM_RC_2)
20
21 // Function prototype
22 TPM_RC
23 TPM2_HierarchyControl(HierarchyControl_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_HIERARCHYCONTROL_FP_H_
26 #endif // CC_HierarchyControl

```

6.120 /tpm/include/private/prototypes/Hierarchy_fp.h

```

1 /* (Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Apr 2, 2019 Time: 04:23:27PM
4  */
5
6 #ifndef _HIERARCHY_FP_H_
7 #define _HIERARCHY_FP_H_
8
9 /*** HierarchyPreInstall()
10 // This function performs the initialization functions for the hierarchy
11 // when the TPM is simulated. This function should not be called if the
12 // TPM is not in a manufacturing mode at the manufacturer, or in a simulated
13 // environment.
14 void HierarchyPreInstall_Init(void);
15
16 /*** HierarchyStartup()
17 // This function is called at TPM2_Startup() to initialize the hierarchy
18 // related values.
19 BOOL HierarchyStartup(STARTUP_TYPE type // IN: start up type
20 );
21
22 /*** HierarchyGetProof()
23 // This function derives the proof value associated with a hierarchy. It returns a
24 // buffer containing the proof value.

```

```

25 //
26 // Return Type: TPM_RC
27 // TPM_RC_FW_LIMITED The requested hierarchy is FW-limited, but the TPM
28 // does not support FW-limited objects or the TPM failed
29 // to derive the Firmware Secret.
30 // TPM_RC_SVN_LIMITED The requested hierarchy is SVN-limited, but the TPM
31 // does not support SVN-limited objects or the TPM failed
32 // to derive the Firmware SVN Secret for the requested
33 // SVN.
34 TPM_RC HierarchyGetProof(TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy constant
35 TPM2B_PROOF* proof // OUT: proof buffer
36 );
37
38 /*** HierarchyGetPrimarySeed()
39 // This function derives the primary seed of a hierarchy.
40 //
41 // Return Type: TPM_RC
42 // TPM_RC_FW_LIMITED The requested hierarchy is FW-limited, but the TPM
43 // does not support FW-limited objects or the TPM failed
44 // to derive the Firmware Secret.
45 // TPM_RC_SVN_LIMITED The requested hierarchy is SVN-limited, but the TPM
46 // does not support SVN-limited objects or the TPM failed
47 // to derive the Firmware SVN Secret for the requested
48 // SVN.
49 TPM_RC HierarchyGetPrimarySeed(TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy
50 TPM2B_SEED* seed // OUT: seed buffer
51 );
52
53 /*** ValidateHierarchy()
54 // This function ensures a given hierarchy is valid and enabled.
55 // Return Type: TPM_RC
56 // TPM_RC_HIERARCHY Hierarchy is disabled
57 // TPM_RC_FW_LIMITED The requested hierarchy is FW-limited, but the TPM
58 // does not support FW-limited objects.
59 // TPM_RC_SVN_LIMITED The requested hierarchy is SVN-limited, but the TPM
60 // does not support SVN-limited objects or the given SVN
61 // is greater than the TPM's current SVN.
62 // TPM_RC_VALUE Hierarchy is not valid
63 TPM_RC ValidateHierarchy(TPMI_RH_HIERARCHY hierarchy // IN: hierarchy
64 );
65
66 /*** HierarchyIsEnabled()
67 // This function checks to see if a hierarchy is enabled.
68 // NOTE: The TPM_RH_NULL hierarchy is always enabled.
69 // Return Type: BOOL
70 // TRUE(1) hierarchy is enabled
71 // FALSE(0) hierarchy is disabled
72 BOOL HierarchyIsEnabled(TPMI_RH_HIERARCHY hierarchy // IN: hierarchy
73 );
74
75 /*** HierarchyNormalizeHandle
76 // This function accepts a handle that may or may not be FW- or SVN-bound,
77 // and returns the base hierarchy to which the handle refers.
78 TPMI_RH_HIERARCHY HierarchyNormalizeHandle(TPMI_RH_HIERARCHY handle // IN
79 );
80
81 /*** HierarchyIsFirmwareLimited
82 // This function accepts a hierarchy handle and returns whether it is firmware-
83 // limited.
84 BOOL HierarchyIsFirmwareLimited(TPMI_RH_HIERARCHY handle // IN
85 );
86
87 /*** HierarchyIsSvnLimited
88 // This function accepts a hierarchy handle and returns whether it is SVN-
89 // limited.
90 BOOL HierarchyIsSvnLimited(TPMI_RH_HIERARCHY handle // IN

```

```

91 );
92
93 #endif // _HIERARCHY_FP_H_

```

6.121 /tpm/include/private/prototypes/HMAC_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_HMAC // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_HMAC_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_HMAC_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT handle;
12     TPM2B_MAX_BUFFER buffer;
13     TPMI_ALG_HASH hashAlg;
14 } HMAC_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM2B_DIGEST outHMAC;
20 } HMAC_Out;
21
22 // Response code modifiers
23 #   define RC_HMAC_handle (TPM_RC_H + TPM_RC_1)
24 #   define RC_HMAC_buffer (TPM_RC_P + TPM_RC_1)
25 #   define RC_HMAC_hashAlg (TPM_RC_P + TPM_RC_2)
26
27 // Function prototype
28 TPM_RC
29 TPM2_HMAC(HMAC_In* in, HMAC_Out* out);
30
31 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_HMAC_FP_H_
32 #endif // CC_HMAC

```

6.122 /tpm/include/private/prototypes/HMAC_Start_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_HMAC_Start // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_HMAC_START_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_HMAC_START_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT handle;
12     TPM2B_AUTH auth;
13     TPMI_ALG_HASH hashAlg;
14 } HMAC_Start_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPMI_DH_OBJECT sequenceHandle;
20 } HMAC_Start_Out;
21
22 // Response code modifiers
23 #   define RC_HMAC_Start_handle (TPM_RC_H + TPM_RC_1)

```

```

24 #   define RC_HMAC_Start_auth      (TPM_RC_P + TPM_RC_1)
25 #   define RC_HMAC_Start_hashAlg  (TPM_RC_P + TPM_RC_2)
26
27 // Function prototype
28 TPM_RC
29 TPM2_HMAC_Start(HMAC_Start_In* in, HMAC_Start_Out* out);
30
31 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_HMAC_START_FP_H_
32 #endif // CC_HMAC_Start

```

6.123 /tpm/include/private/prototypes/Import_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Import // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_IMPORT_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_IMPORT_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT      parentHandle;
12     TPM2B_DATA          encryptionKey;
13     TPM2B_PUBLIC        objectPublic;
14     TPM2B_PRIVATE       duplicate;
15     TPM2B_ENCRYPTED_SECRET inSymSeed;
16     TPMT_SYM_DEF_OBJECT symmetricAlg;
17 } Import_In;
18
19 // Output structure definition
20 typedef struct
21 {
22     TPM2B_PRIVATE outPrivate;
23 } Import_Out;
24
25 // Response code modifiers
26 #   define RC_Import_parentHandle (TPM_RC_H + TPM_RC_1)
27 #   define RC_Import_encryptionKey (TPM_RC_P + TPM_RC_1)
28 #   define RC_Import_objectPublic (TPM_RC_P + TPM_RC_2)
29 #   define RC_Import_duplicate (TPM_RC_P + TPM_RC_3)
30 #   define RC_Import_inSymSeed (TPM_RC_P + TPM_RC_4)
31 #   define RC_Import_symmetricAlg (TPM_RC_P + TPM_RC_5)
32
33 // Function prototype
34 TPM_RC
35 TPM2_Import(Import_In* in, Import_Out* out);
36
37 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_IMPORT_FP_H_
38 #endif // CC_Import

```

6.124 /tpm/include/private/prototypes/IncrementalSelfTest_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_IncrementalSelfTest // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_INCREMENTALSELFTEST_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_INCREMENTALSELFTEST_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPML_ALG toTest;

```

```

12 } IncrementalSelfTest_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPML_ALG toDoList;
18 } IncrementalSelfTest_Out;
19
20 // Response code modifiers
21 # define RC_IncrementalSelfTest_toTest (TPM_RC_P + TPM_RC_1)
22
23 // Function prototype
24 TPM_RC
25 TPM2_IncrementalSelfTest(IncrementalSelfTest_In* in, IncrementalSelfTest_Out* out);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_INCREMENTALSELFTEST_FP_H_
28 #endif // CC_IncrementalSelfTest

```

6.125 /tpm/include/private/prototypes/IOBuffers_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 28, 2019 Time: 08:25:19PM
4  */
5
6  #ifndef _IO_BUFFERS_FP_H_
7  #define _IO_BUFFERS_FP_H_
8
9  /*** MemoryIoBufferAllocationReset()
10 // This function is used to reset the allocation of buffers.
11 void MemoryIoBufferAllocationReset(void);
12
13 /*** MemoryIoBufferZero()
14 // Function zeros the action I/O buffer at the end of a command. Calling this is
15 // not mandatory for proper functionality.
16 void MemoryIoBufferZero(void);
17
18 /*** MemoryGetInBuffer()
19 // This function returns the address of the buffer into which the
20 // command parameters will be unmarshaled in preparation for calling
21 // the command actions.
22 BYTE* MemoryGetInBuffer(UINT32 size // Size, in bytes, required for the input
23 // unmarshaling
24 );
25
26 /*** MemoryGetOutBuffer()
27 // This function returns the address of the buffer into which the command
28 // action code places its output values.
29 BYTE* MemoryGetOutBuffer(UINT32 size // required size of the buffer
30 );
31
32 /*** IsLabelProperlyFormatted()
33 // This function checks that a label is a null-terminated string.
34 // NOTE: this function is here because there was no better place for it.
35 // Return Type: BOOL
36 // TRUE(1) string is null terminated
37 // FALSE(0) string is not null terminated
38 BOOL IsLabelProperlyFormatted(TPM2B* x);
39
40 #endif // _IO_BUFFERS_FP_H_

```

6.126 /tpm/include/private/prototypes/LoadExternal_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT

```



```

2
3 #if CC_LoadExternal // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_LOADEXTERNAL_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_LOADEXTERNAL_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPM2B_SENSITIVE    inPrivate;
12     TPM2B_PUBLIC       inPublic;
13     TPMI_RH_HIERARCHY hierarchy;
14 } LoadExternal_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM_HANDLE objectHandle;
20     TPM2B_NAME name;
21 } LoadExternal_Out;
22
23 // Response code modifiers
24 #   define RC_LoadExternal_inPrivate (TPM_RC_P + TPM_RC_1)
25 #   define RC_LoadExternal_inPublic (TPM_RC_P + TPM_RC_2)
26 #   define RC_LoadExternal_hierarchy (TPM_RC_P + TPM_RC_3)
27
28 // Function prototype
29 TPM_RC
30 TPM2_LoadExternal(LoadExternal_In* in, LoadExternal_Out* out);
31
32 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_LOADEXTERNAL_FP_H_
33 #endif // CC_LoadExternal

```

6.127 /tpm/include/private/prototypes/Load_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Load // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_LOAD_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_LOAD_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT parentHandle;
12     TPM2B_PRIVATE    inPrivate;
13     TPM2B_PUBLIC      inPublic;
14 } Load_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM_HANDLE objectHandle;
20     TPM2B_NAME name;
21 } Load_Out;
22
23 // Response code modifiers
24 #   define RC_Load_parentHandle (TPM_RC_H + TPM_RC_1)
25 #   define RC_Load_inPrivate    (TPM_RC_P + TPM_RC_1)
26 #   define RC_Load_inPublic     (TPM_RC_P + TPM_RC_2)
27
28 // Function prototype
29 TPM_RC
30 TPM2_Load(Load_In* in, Load_Out* out);

```

```

31
32 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_LOAD_FP_H_
33 #endif // CC_Load

```

6.128 /tpm/include/private/prototypes/Locality_fp.h

```

1  /*(Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Mar 28, 2019 Time: 08:25:19PM
4   */
5
6 #ifndef _LOCALITY_FP_H_
7 #define _LOCALITY_FP_H_
8
9 /** LocalityGetAttributes()
10  // This function will convert a locality expressed as an integer into
11  // TPMA_LOCALITY form.
12  //
13  // The function returns the locality attribute.
14  TPMA_LOCALITY
15  LocalityGetAttributes(UINT8 locality // IN: locality value
16  );
17
18 #endif // _LOCALITY_FP_H_

```

6.129 /tpm/include/private/prototypes/MAC_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_MAC // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_MAC_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_MAC_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT handle;
12     TPM2B_MAX_BUFFER buffer;
13     TPMI_ALG_MAC_SCHEME inScheme;
14 } MAC_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM2B_DIGEST outMAC;
20 } MAC_Out;
21
22 // Response code modifiers
23 #   define RC_MAC_handle (TPM_RC_H + TPM_RC_1)
24 #   define RC_MAC_buffer (TPM_RC_P + TPM_RC_1)
25 #   define RC_MAC_inScheme (TPM_RC_P + TPM_RC_2)
26
27 // Function prototype
28 TPM_RC
29 TPM2_MAC(MAC_In* in, MAC_Out* out);
30
31 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_MAC_FP_H_
32 #endif // CC_MAC

```

6.130 /tpm/include/private/prototypes/MAC_Start_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT

```

```

2
3 #if CC_MAC_Start // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_MAC_START_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_MAC_START_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT      handle;
12     TPM2B_AUTH          auth;
13     TPMI_ALG_MAC_SCHEME inScheme;
14 } MAC_Start_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPMI_DH_OBJECT sequenceHandle;
20 } MAC_Start_Out;
21
22 // Response code modifiers
23 #   define RC_MAC_Start_handle      (TPM_RC_H + TPM_RC_1)
24 #   define RC_MAC_Start_auth       (TPM_RC_P + TPM_RC_1)
25 #   define RC_MAC_Start_inScheme   (TPM_RC_P + TPM_RC_2)
26
27 // Function prototype
28 TPM_RC
29 TPM2_MAC_Start(MAC_Start_In* in, MAC_Start_Out* out);
30
31 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_MAC_START_FP_H_
32 #endif // CC_MAC_Start

```

6.131 /tpm/include/private/prototypes/MakeCredential_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_MakeCredential // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_MAKECREDENTIAL_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_MAKECREDENTIAL_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT handle;
12     TPM2B_DIGEST   credential;
13     TPM2B_NAME      objectName;
14 } MakeCredential_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM2B_ID_OBJECT credentialBlob;
20     TPM2B_ENCRYPTED_SECRET secret;
21 } MakeCredential_Out;
22
23 // Response code modifiers
24 #   define RC_MakeCredential_handle      (TPM_RC_H + TPM_RC_1)
25 #   define RC_MakeCredential_credential (TPM_RC_P + TPM_RC_1)
26 #   define RC_MakeCredential_objectName (TPM_RC_P + TPM_RC_2)
27
28 // Function prototype
29 TPM_RC
30 TPM2_MakeCredential(MakeCredential_In* in, MakeCredential_Out* out);
31

```

```

32 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_MAKECREDENTIAL_FP_H_
33 #endif // CC_MakeCredential

```

6.132 /tpm/include/private/prototypes/Marshal_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #ifndef _MARSHAL_FP_H_
4 #define _MARSHAL_FP_H_
5
6 // Table "Definition of Base Types" (Part 2: Structures)
7 // UINT8 definition
8 TPM_RC
9 UINT8_Unmarshal(UINT8* target, BYTE** buffer, INT32* size);
10 UINT16
11 UINT8_Marshal(UINT8* source, BYTE** buffer, INT32* size);
12
13 // BYTE definition
14 #if !USE_MARSHALING_DEFINES
15 TPM_RC
16 BYTE_Unmarshal(BYTE* target, BYTE** buffer, INT32* size);
17 #else // !USE_MARSHALING_DEFINES
18 # define BYTE_Unmarshal(target, buffer, size) \
19     UINT8_Unmarshal((UINT8*)(target), (buffer), (size))
20 #endif // !USE_MARSHALING_DEFINES
21 #if !USE_MARSHALING_DEFINES
22 UINT16
23 BYTE_Marshal(BYTE* source, BYTE** buffer, INT32* size);
24 #else // !USE_MARSHALING_DEFINES
25 # define BYTE_Marshal(source, buffer, size) \
26     UINT8_Marshal((UINT8*)(source), (buffer), (size))
27 #endif // !USE_MARSHALING_DEFINES
28
29 // INT8 definition
30 #if !USE_MARSHALING_DEFINES
31 TPM_RC
32 INT8_Unmarshal(INT8* target, BYTE** buffer, INT32* size);
33 #else // !USE_MARSHALING_DEFINES
34 # define INT8_Unmarshal(target, buffer, size) \
35     UINT8_Unmarshal((UINT8*)(target), (buffer), (size))
36 #endif // !USE_MARSHALING_DEFINES
37 #if !USE_MARSHALING_DEFINES
38 UINT16
39 INT8_Marshal(INT8* source, BYTE** buffer, INT32* size);
40 #else // !USE_MARSHALING_DEFINES
41 # define INT8_Marshal(source, buffer, size) \
42     UINT8_Marshal((UINT8*)(source), (buffer), (size))
43 #endif // !USE_MARSHALING_DEFINES
44
45 // UINT16 definition
46 TPM_RC
47 UINT16_Unmarshal(UINT16* target, BYTE** buffer, INT32* size);
48 UINT16
49 UINT16_Marshal(UINT16* source, BYTE** buffer, INT32* size);
50
51 // INT16 definition
52 #if !USE_MARSHALING_DEFINES
53 TPM_RC
54 INT16_Unmarshal(INT16* target, BYTE** buffer, INT32* size);
55 #else // !USE_MARSHALING_DEFINES
56 # define INT16_Unmarshal(target, buffer, size) \
57     UINT16_Unmarshal((UINT16*)(target), (buffer), (size))
58 #endif // !USE_MARSHALING_DEFINES
59 #if !USE_MARSHALING_DEFINES
60 UINT16

```

```

61  INT16_Marshal(INT16* source, BYTE** buffer, INT32* size);
62  #else // !USE_MARSHALING_DEFINES
63  # define INT16_Marshal(source, buffer, size) \
64      UINT16_Marshal((UINT16*)(source), (buffer), (size))
65  #endif // !USE_MARSHALING_DEFINES
66
67  //  UINT32 definition
68  TPM_RC
69  UINT32_Unmarshal(UINT32* target, BYTE** buffer, INT32* size);
70  UINT16
71  UINT32_Marshal(UINT32* source, BYTE** buffer, INT32* size);
72
73  //  INT32 definition
74  #if !USE_MARSHALING_DEFINES
75  TPM_RC
76  INT32_Unmarshal(INT32* target, BYTE** buffer, INT32* size);
77  #else // !USE_MARSHALING_DEFINES
78  # define INT32_Unmarshal(target, buffer, size) \
79      UINT32_Unmarshal((UINT32*)(target), (buffer), (size))
80  #endif // !USE_MARSHALING_DEFINES
81  #if !USE_MARSHALING_DEFINES
82  UINT16
83  INT32_Marshal(INT32* source, BYTE** buffer, INT32* size);
84  #else // !USE_MARSHALING_DEFINES
85  # define INT32_Marshal(source, buffer, size) \
86      UINT32_Marshal((UINT32*)(source), (buffer), (size))
87  #endif // !USE_MARSHALING_DEFINES
88
89  //  UINT64 definition
90  TPM_RC
91  UINT64_Unmarshal(UINT64* target, BYTE** buffer, INT32* size);
92  UINT16
93  UINT64_Marshal(UINT64* source, BYTE** buffer, INT32* size);
94
95  //  INT64 definition
96  #if !USE_MARSHALING_DEFINES
97  TPM_RC
98  INT64_Unmarshal(INT64* target, BYTE** buffer, INT32* size);
99  #else // !USE_MARSHALING_DEFINES
100 # define INT64_Unmarshal(target, buffer, size) \
101     UINT64_Unmarshal((UINT64*)(target), (buffer), (size))
102 #endif // !USE_MARSHALING_DEFINES
103 #if !USE_MARSHALING_DEFINES
104 UINT16
105 INT64_Marshal(INT64* source, BYTE** buffer, INT32* size);
106 #else // !USE_MARSHALING_DEFINES
107 # define INT64_Marshal(source, buffer, size) \
108     UINT64_Marshal((UINT64*)(source), (buffer), (size))
109 #endif // !USE_MARSHALING_DEFINES
110
111 // Table "Definition of Types for Documentation Clarity" (Part 2: Structures)
112 #if !USE_MARSHALING_DEFINES
113 TPM_RC
114 TPM_ALGORITHM_ID_Unmarshal(TPM_ALGORITHM_ID* target, BYTE** buffer, INT32* size);
115 #else // !USE_MARSHALING_DEFINES
116 # define TPM_ALGORITHM_ID_Unmarshal(target, buffer, size) \
117     UINT32_Unmarshal((UINT32*)(target), (buffer), (size))
118 #endif // !USE_MARSHALING_DEFINES
119 #if !USE_MARSHALING_DEFINES
120 UINT16
121 TPM_ALGORITHM_ID_Marshal(TPM_ALGORITHM_ID* source, BYTE** buffer, INT32* size);
122 #else // !USE_MARSHALING_DEFINES
123 # define TPM_ALGORITHM_ID_Marshal(source, buffer, size) \
124     UINT32_Marshal((UINT32*)(source), (buffer), (size))
125 #endif // !USE_MARSHALING_DEFINES
126 #if !USE_MARSHALING_DEFINES

```

```

127 TPM_RC
128 TPM_AUTHORIZATION_SIZE_Unmarshal(
129     TPM_AUTHORIZATION_SIZE* target, BYTE** buffer, INT32* size);
130 #else // !USE_MARSHALING_DEFINES
131 # define TPM_AUTHORIZATION_SIZE_Unmarshal(target, buffer, size) \
132     UINT32_Unmarshal((UINT32*)(target), (buffer), (size))
133 #endif // !USE_MARSHALING_DEFINES
134 #if !USE_MARSHALING_DEFINES
135     UINT16
136 TPM_AUTHORIZATION_SIZE_Marshal(
137     TPM_AUTHORIZATION_SIZE* source, BYTE** buffer, INT32* size);
138 #else // !USE_MARSHALING_DEFINES
139 # define TPM_AUTHORIZATION_SIZE_Marshal(source, buffer, size) \
140     UINT32_Marshal((UINT32*)(source), (buffer), (size))
141 #endif // !USE_MARSHALING_DEFINES
142 #if !USE_MARSHALING_DEFINES
143     TPM_RC
144 TPM_KEY_BITS_Unmarshal(TPM_KEY_BITS* target, BYTE** buffer, INT32* size);
145 #else // !USE_MARSHALING_DEFINES
146 # define TPM_KEY_BITS_Unmarshal(target, buffer, size) \
147     UINT16_Unmarshal((UINT16*)(target), (buffer), (size))
148 #endif // !USE_MARSHALING_DEFINES
149 #if !USE_MARSHALING_DEFINES
150     UINT16
151 TPM_KEY_BITS_Marshal(TPM_KEY_BITS* source, BYTE** buffer, INT32* size);
152 #else // !USE_MARSHALING_DEFINES
153 # define TPM_KEY_BITS_Marshal(source, buffer, size) \
154     UINT16_Marshal((UINT16*)(source), (buffer), (size))
155 #endif // !USE_MARSHALING_DEFINES
156 #if !USE_MARSHALING_DEFINES
157     TPM_RC
158 TPM_KEY_SIZE_Unmarshal(TPM_KEY_SIZE* target, BYTE** buffer, INT32* size);
159 #else // !USE_MARSHALING_DEFINES
160 # define TPM_KEY_SIZE_Unmarshal(target, buffer, size) \
161     UINT16_Unmarshal((UINT16*)(target), (buffer), (size))
162 #endif // !USE_MARSHALING_DEFINES
163 #if !USE_MARSHALING_DEFINES
164     UINT16
165 TPM_KEY_SIZE_Marshal(TPM_KEY_SIZE* source, BYTE** buffer, INT32* size);
166 #else // !USE_MARSHALING_DEFINES
167 # define TPM_KEY_SIZE_Marshal(source, buffer, size) \
168     UINT16_Marshal((UINT16*)(source), (buffer), (size))
169 #endif // !USE_MARSHALING_DEFINES
170 #if !USE_MARSHALING_DEFINES
171     TPM_RC
172 TPM_MODIFIER_INDICATOR_Unmarshal(
173     TPM_MODIFIER_INDICATOR* target, BYTE** buffer, INT32* size);
174 #else // !USE_MARSHALING_DEFINES
175 # define TPM_MODIFIER_INDICATOR_Unmarshal(target, buffer, size) \
176     UINT32_Unmarshal((UINT32*)(target), (buffer), (size))
177 #endif // !USE_MARSHALING_DEFINES
178 #if !USE_MARSHALING_DEFINES
179     UINT16
180 TPM_MODIFIER_INDICATOR_Marshal(
181     TPM_MODIFIER_INDICATOR* source, BYTE** buffer, INT32* size);
182 #else // !USE_MARSHALING_DEFINES
183 # define TPM_MODIFIER_INDICATOR_Marshal(source, buffer, size) \
184     UINT32_Marshal((UINT32*)(source), (buffer), (size))
185 #endif // !USE_MARSHALING_DEFINES
186 #if !USE_MARSHALING_DEFINES
187     TPM_RC
188 TPM_PARAMETER_SIZE_Unmarshal(TPM_PARAMETER_SIZE* target, BYTE** buffer, INT32* size);
189 #else // !USE_MARSHALING_DEFINES
190 # define TPM_PARAMETER_SIZE_Unmarshal(target, buffer, size) \
191     UINT32_Unmarshal((UINT32*)(target), (buffer), (size))
192 #endif // !USE_MARSHALING_DEFINES

```



```

193 #if !USE_MARSHALING_DEFINES
194 UINT16
195 TPM_PARAMETER_SIZE_Marshal(TPM_PARAMETER_SIZE* source, BYTE** buffer, INT32* size);
196 #else // !USE_MARSHALING_DEFINES
197 # define TPM_PARAMETER_SIZE_Marshal(source, buffer, size) \
198     UINT32_Marshal((UINT32*)(source), (buffer), (size))
199 #endif // !USE_MARSHALING_DEFINES
200
201 // Table "Definition of TPM_CONSTANTS32 Constants" (Part 2: Structures)
202 #if !USE_MARSHALING_DEFINES
203 UINT16
204 TPM_CONSTANTS32_Marshal(TPM_CONSTANTS32* source, BYTE** buffer, INT32* size);
205 #else // !USE_MARSHALING_DEFINES
206 # define TPM_CONSTANTS32_Marshal(source, buffer, size) \
207     UINT32_Marshal((UINT32*)(source), (buffer), (size))
208 #endif // !USE_MARSHALING_DEFINES
209
210 // Table "Definition of TPM_ALG_ID Constants" (Part 2: Structures)
211 #if !USE_MARSHALING_DEFINES
212 TPM_RC
213 TPM_ALG_ID_Unmarshal(TPM_ALG_ID* target, BYTE** buffer, INT32* size);
214 #else // !USE_MARSHALING_DEFINES
215 # define TPM_ALG_ID_Unmarshal(target, buffer, size) \
216     UINT16_Unmarshal((UINT16*)(target), (buffer), (size))
217 #endif // !USE_MARSHALING_DEFINES
218 #if !USE_MARSHALING_DEFINES
219 UINT16
220 TPM_ALG_ID_Marshal(TPM_ALG_ID* source, BYTE** buffer, INT32* size);
221 #else // !USE_MARSHALING_DEFINES
222 # define TPM_ALG_ID_Marshal(source, buffer, size) \
223     UINT16_Marshal((UINT16*)(source), (buffer), (size))
224 #endif // !USE_MARSHALING_DEFINES
225
226 // Table "Definition of TPM_ECC_CURVE Constants" (Part 2: Structures)
227 TPM_RC
228 TPM_ECC_CURVE_Unmarshal(TPM_ECC_CURVE* target, BYTE** buffer, INT32* size);
229 #if !USE_MARSHALING_DEFINES
230 UINT16
231 TPM_ECC_CURVE_Marshal(TPM_ECC_CURVE* source, BYTE** buffer, INT32* size);
232 #else // !USE_MARSHALING_DEFINES
233 # define TPM_ECC_CURVE_Marshal(source, buffer, size) \
234     UINT16_Marshal((UINT16*)(source), (buffer), (size))
235 #endif // !USE_MARSHALING_DEFINES
236
237 // Table "Definition of TPM_CC Constants" (Part 2: Structures)
238 #if !USE_MARSHALING_DEFINES
239 TPM_RC
240 TPM_CC_Unmarshal(TPM_CC* target, BYTE** buffer, INT32* size);
241 #else // !USE_MARSHALING_DEFINES
242 # define TPM_CC_Unmarshal(target, buffer, size) \
243     UINT32_Unmarshal((UINT32*)(target), (buffer), (size))
244 #endif // !USE_MARSHALING_DEFINES
245 #if !USE_MARSHALING_DEFINES
246 UINT16
247 TPM_CC_Marshal(TPM_CC* source, BYTE** buffer, INT32* size);
248 #else // !USE_MARSHALING_DEFINES
249 # define TPM_CC_Marshal(source, buffer, size) \
250     UINT32_Marshal((UINT32*)(source), (buffer), (size))
251 #endif // !USE_MARSHALING_DEFINES
252
253 // Table "Definition of TPM_RC Constants" (Part 2: Structures)
254 #if !USE_MARSHALING_DEFINES
255 UINT16
256 TPM_RC_Marshal(TPM_RC* source, BYTE** buffer, INT32* size);
257 #else // !USE_MARSHALING_DEFINES
258 # define TPM_RC_Marshal(source, buffer, size) \

```

```

259     UINT32_Marshal((UINT32*)(source), (buffer), (size))
260 #endif // !USE_MARSHALING_DEFINES
261
262 // Table "Definition of TPM_CLOCK_ADJUST Constants" (Part 2: Structures)
263 TPM_RC
264 TPM_CLOCK_ADJUST_Unmarshal(TPM_CLOCK_ADJUST* target, BYTE** buffer, INT32* size);
265
266 // Table "Definition of TPM_EO Constants" (Part 2: Structures)
267 TPM_RC
268 TPM_EO_Unmarshal(TPM_EO* target, BYTE** buffer, INT32* size);
269 #if !USE_MARSHALING_DEFINES
270 UINT16
271 TPM_EO_Marshal(TPM_EO* source, BYTE** buffer, INT32* size);
272 #else // !USE_MARSHALING_DEFINES
273 # define TPM_EO_Marshal(source, buffer, size) \
274     UINT16_Marshal((UINT16*)(source), (buffer), (size))
275 #endif // !USE_MARSHALING_DEFINES
276
277 // Table "Definition of TPM_ST Constants" (Part 2: Structures)
278 #if !USE_MARSHALING_DEFINES
279 TPM_RC
280 TPM_ST_Unmarshal(TPM_ST* target, BYTE** buffer, INT32* size);
281 #else // !USE_MARSHALING_DEFINES
282 # define TPM_ST_Unmarshal(target, buffer, size) \
283     UINT16_Unmarshal((UINT16*)(target), (buffer), (size))
284 #endif // !USE_MARSHALING_DEFINES
285 #if !USE_MARSHALING_DEFINES
286 UINT16
287 TPM_ST_Marshal(TPM_ST* source, BYTE** buffer, INT32* size);
288 #else // !USE_MARSHALING_DEFINES
289 # define TPM_ST_Marshal(source, buffer, size) \
290     UINT16_Marshal((UINT16*)(source), (buffer), (size))
291 #endif // !USE_MARSHALING_DEFINES
292
293 // Table "Definition of TPM_SU Constants" (Part 2: Structures)
294 TPM_RC
295 TPM_SU_Unmarshal(TPM_SU* target, BYTE** buffer, INT32* size);
296
297 // Table "Definition of TPM_SE Constants" (Part 2: Structures)
298 TPM_RC
299 TPM_SE_Unmarshal(TPM_SE* target, BYTE** buffer, INT32* size);
300
301 // Table "Definition of TPM_CAP Constants" (Part 2: Structures)
302 TPM_RC
303 TPM_CAP_Unmarshal(TPM_CAP* target, BYTE** buffer, INT32* size);
304 #if !USE_MARSHALING_DEFINES
305 UINT16
306 TPM_CAP_Marshal(TPM_CAP* source, BYTE** buffer, INT32* size);
307 #else // !USE_MARSHALING_DEFINES
308 # define TPM_CAP_Marshal(source, buffer, size) \
309     UINT32_Marshal((UINT32*)(source), (buffer), (size))
310 #endif // !USE_MARSHALING_DEFINES
311
312 // Table "Definition of TPM_PT Constants" (Part 2: Structures)
313 #if !USE_MARSHALING_DEFINES
314 TPM_RC
315 TPM_PT_Unmarshal(TPM_PT* target, BYTE** buffer, INT32* size);
316 #else // !USE_MARSHALING_DEFINES
317 # define TPM_PT_Unmarshal(target, buffer, size) \
318     UINT32_Unmarshal((UINT32*)(target), (buffer), (size))
319 #endif // !USE_MARSHALING_DEFINES
320 #if !USE_MARSHALING_DEFINES
321 UINT16
322 TPM_PT_Marshal(TPM_PT* source, BYTE** buffer, INT32* size);
323 #else // !USE_MARSHALING_DEFINES
324 # define TPM_PT_Marshal(source, buffer, size) \

```

```

325     UINT32_Marshal((UINT32*)(source), (buffer), (size))
326 #endif // !USE_MARSHALING_DEFINES
327
328 // Table "Definition of TPM_PT_PCR Constants" (Part 2: Structures)
329 #if !USE_MARSHALING_DEFINES
330 TPM_RC
331 TPM_PT_PCR_Unmarshal(TPM_PT_PCR* target, BYTE** buffer, INT32* size);
332 #else // !USE_MARSHALING_DEFINES
333 # define TPM_PT_PCR_Unmarshal(target, buffer, size) \
334     UINT32_Unmarshal((UINT32*)(target), (buffer), (size))
335 #endif // !USE_MARSHALING_DEFINES
336 #if !USE_MARSHALING_DEFINES
337 UINT16
338 TPM_PT_PCR_Marshal(TPM_PT_PCR* source, BYTE** buffer, INT32* size);
339 #else // !USE_MARSHALING_DEFINES
340 # define TPM_PT_PCR_Marshal(source, buffer, size) \
341     UINT32_Marshal((UINT32*)(source), (buffer), (size))
342 #endif // !USE_MARSHALING_DEFINES
343
344 // Table "Definition of TPM_PS Constants" (Part 2: Structures)
345 #if !USE_MARSHALING_DEFINES
346 UINT16
347 TPM_PS_Marshal(TPM_PS* source, BYTE** buffer, INT32* size);
348 #else // !USE_MARSHALING_DEFINES
349 # define TPM_PS_Marshal(source, buffer, size) \
350     UINT32_Marshal((UINT32*)(source), (buffer), (size))
351 #endif // !USE_MARSHALING_DEFINES
352
353 // Table "Definition of Types for Handles" (Part 2: Structures)
354 #if !USE_MARSHALING_DEFINES
355 TPM_RC
356 TPM_HANDLE_Unmarshal(TPM_HANDLE* target, BYTE** buffer, INT32* size);
357 #else // !USE_MARSHALING_DEFINES
358 # define TPM_HANDLE_Unmarshal(target, buffer, size) \
359     UINT32_Unmarshal((UINT32*)(target), (buffer), (size))
360 #endif // !USE_MARSHALING_DEFINES
361 #if !USE_MARSHALING_DEFINES
362 UINT16
363 TPM_HANDLE_Marshal(TPM_HANDLE* source, BYTE** buffer, INT32* size);
364 #else // !USE_MARSHALING_DEFINES
365 # define TPM_HANDLE_Marshal(source, buffer, size) \
366     UINT32_Marshal((UINT32*)(source), (buffer), (size))
367 #endif // !USE_MARSHALING_DEFINES
368
369 // Table "Definition of TPM_HT Constants" (Part 2: Structures)
370 #if !USE_MARSHALING_DEFINES
371 TPM_RC
372 TPM_HT_Unmarshal(TPM_HT* target, BYTE** buffer, INT32* size);
373 #else // !USE_MARSHALING_DEFINES
374 # define TPM_HT_Unmarshal(target, buffer, size) \
375     UINT8_Unmarshal((UINT8*)(target), (buffer), (size))
376 #endif // !USE_MARSHALING_DEFINES
377 #if !USE_MARSHALING_DEFINES
378 UINT16
379 TPM_HT_Marshal(TPM_HT* source, BYTE** buffer, INT32* size);
380 #else // !USE_MARSHALING_DEFINES
381 # define TPM_HT_Marshal(source, buffer, size) \
382     UINT8_Marshal((UINT8*)(source), (buffer), (size))
383 #endif // !USE_MARSHALING_DEFINES
384
385 // Table "Definition of TPM_RH Constants" (Part 2: Structures)
386 #if !USE_MARSHALING_DEFINES
387 TPM_RC
388 TPM_RH_Unmarshal(TPM_RH* target, BYTE** buffer, INT32* size);
389 #else // !USE_MARSHALING_DEFINES
390 # define TPM_RH_Unmarshal(target, buffer, size) \

```

```

391     TPM_HANDLE_Unmarshal((TPM_HANDLE*)(target), (buffer), (size))
392 #endif // !USE_MARSHALING_DEFINES
393 #if !USE_MARSHALING_DEFINES
394     UINT16
395     TPM_RH_Marshal(TPM_RH* source, BYTE** buffer, INT32* size);
396 #else // !USE_MARSHALING_DEFINES
397 # define TPM_RH_Marshal(source, buffer, size) \
398     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
399 #endif // !USE_MARSHALING_DEFINES
400
401 // Table "Definition of TPM_HC Constants" (Part 2: Structures)
402 #if !USE_MARSHALING_DEFINES
403     TPM_RC
404     TPM_HC_Unmarshal(TPM_HC* target, BYTE** buffer, INT32* size);
405 #else // !USE_MARSHALING_DEFINES
406 # define TPM_HC_Unmarshal(target, buffer, size) \
407     TPM_HANDLE_Unmarshal((TPM_HANDLE*)(target), (buffer), (size))
408 #endif // !USE_MARSHALING_DEFINES
409 #if !USE_MARSHALING_DEFINES
410     UINT16
411     TPM_HC_Marshal(TPM_HC* source, BYTE** buffer, INT32* size);
412 #else // !USE_MARSHALING_DEFINES
413 # define TPM_HC_Marshal(source, buffer, size) \
414     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
415 #endif // !USE_MARSHALING_DEFINES
416
417 // Table "Definition of TPMA_ALGORITHM Bits" (Part 2: Structures)
418     TPM_RC
419     TPMA_ALGORITHM_Unmarshal(TPMA_ALGORITHM* target, BYTE** buffer, INT32* size);
420 #if !USE_MARSHALING_DEFINES
421     UINT16
422     TPMA_ALGORITHM_Marshal(TPMA_ALGORITHM* source, BYTE** buffer, INT32* size);
423 #else // !USE_MARSHALING_DEFINES
424 # define TPMA_ALGORITHM_Marshal(source, buffer, size) \
425     UINT32_Marshal((UINT32*)(source), (buffer), (size))
426 #endif // !USE_MARSHALING_DEFINES
427
428 // Table "Definition of TPMA_OBJECT Bits" (Part 2: Structures)
429     TPM_RC
430     TPMA_OBJECT_Unmarshal(TPMA_OBJECT* target, BYTE** buffer, INT32* size);
431 #if !USE_MARSHALING_DEFINES
432     UINT16
433     TPMA_OBJECT_Marshal(TPMA_OBJECT* source, BYTE** buffer, INT32* size);
434 #else // !USE_MARSHALING_DEFINES
435 # define TPMA_OBJECT_Marshal(source, buffer, size) \
436     UINT32_Marshal((UINT32*)(source), (buffer), (size))
437 #endif // !USE_MARSHALING_DEFINES
438
439 // Table "Definition of TPMA_SESSION Bits" (Part 2: Structures)
440     TPM_RC
441     TPMA_SESSION_Unmarshal(TPMA_SESSION* target, BYTE** buffer, INT32* size);
442 #if !USE_MARSHALING_DEFINES
443     UINT16
444     TPMA_SESSION_Marshal(TPMA_SESSION* source, BYTE** buffer, INT32* size);
445 #else // !USE_MARSHALING_DEFINES
446 # define TPMA_SESSION_Marshal(source, buffer, size) \
447     UINT8_Marshal((UINT8*)(source), (buffer), (size))
448 #endif // !USE_MARSHALING_DEFINES
449
450 // Table "Definition of TPMA_LOCALITY Bits" (Part 2: Structures)
451 #if !USE_MARSHALING_DEFINES
452     TPM_RC
453     TPMA_LOCALITY_Unmarshal(TPMA_LOCALITY* target, BYTE** buffer, INT32* size);
454 #else // !USE_MARSHALING_DEFINES
455 # define TPMA_LOCALITY_Unmarshal(target, buffer, size) \
456     UINT8_Unmarshal((UINT8*)(target), (buffer), (size))

```

```

457 #endif // !USE_MARSHALING_DEFINES
458 #if !USE_MARSHALING_DEFINES
459 UINT16
460 TPMA_LOCALITY_Marshal(TPMA_LOCALITY* source, BYTE** buffer, INT32* size);
461 #else // !USE_MARSHALING_DEFINES
462 # define TPMA_LOCALITY_Marshal(source, buffer, size) \
463     UINT8_Marshal((UINT8*)(source), (buffer), (size))
464 #endif // !USE_MARSHALING_DEFINES
465
466 // Table "Definition of TPMA_PERMANENT Bits" (Part 2: Structures)
467 #if !USE_MARSHALING_DEFINES
468 UINT16
469 TPMA_PERMANENT_Marshal(TPMA_PERMANENT* source, BYTE** buffer, INT32* size);
470 #else // !USE_MARSHALING_DEFINES
471 # define TPMA_PERMANENT_Marshal(source, buffer, size) \
472     UINT32_Marshal((UINT32*)(source), (buffer), (size))
473 #endif // !USE_MARSHALING_DEFINES
474
475 // Table "Definition of TPMA_STARTUP_CLEAR Bits" (Part 2: Structures)
476 #if !USE_MARSHALING_DEFINES
477 UINT16
478 TPMA_STARTUP_CLEAR_Marshal(TPMA_STARTUP_CLEAR* source, BYTE** buffer, INT32* size);
479 #else // !USE_MARSHALING_DEFINES
480 # define TPMA_STARTUP_CLEAR_Marshal(source, buffer, size) \
481     UINT32_Marshal((UINT32*)(source), (buffer), (size))
482 #endif // !USE_MARSHALING_DEFINES
483
484 // Table "Definition of TPMA_MEMORY Bits" (Part 2: Structures)
485 #if !USE_MARSHALING_DEFINES
486 UINT16
487 TPMA_MEMORY_Marshal(TPMA_MEMORY* source, BYTE** buffer, INT32* size);
488 #else // !USE_MARSHALING_DEFINES
489 # define TPMA_MEMORY_Marshal(source, buffer, size) \
490     UINT32_Marshal((UINT32*)(source), (buffer), (size))
491 #endif // !USE_MARSHALING_DEFINES
492
493 // Table "Definition of TPMA_CC Bits" (Part 2: Structures)
494 #if !USE_MARSHALING_DEFINES
495 UINT16
496 TPMA_CC_Marshal(TPMA_CC* source, BYTE** buffer, INT32* size);
497 #else // !USE_MARSHALING_DEFINES
498 # define TPMA_CC_Marshal(source, buffer, size) \
499     UINT32_Marshal((UINT32*)(source), (buffer), (size))
500 #endif // !USE_MARSHALING_DEFINES
501
502 // Table "Definition of TPMA_MODES Bits" (Part 2: Structures)
503 #if !USE_MARSHALING_DEFINES
504 UINT16
505 TPMA_MODES_Marshal(TPMA_MODES* source, BYTE** buffer, INT32* size);
506 #else // !USE_MARSHALING_DEFINES
507 # define TPMA_MODES_Marshal(source, buffer, size) \
508     UINT32_Marshal((UINT32*)(source), (buffer), (size))
509 #endif // !USE_MARSHALING_DEFINES
510
511 // Table "Definition of TPMA_ACT Bits" (Part 2: Structures)
512 TPM_RC
513 TPMA_ACT_Unmarshal(TPMA_ACT* target, BYTE** buffer, INT32* size);
514 #if !USE_MARSHALING_DEFINES
515 UINT16
516 TPMA_ACT_Marshal(TPMA_ACT* source, BYTE** buffer, INT32* size);
517 #else // !USE_MARSHALING_DEFINES
518 # define TPMA_ACT_Marshal(source, buffer, size) \
519     UINT32_Marshal((UINT32*)(source), (buffer), (size))
520 #endif // !USE_MARSHALING_DEFINES
521
522 // Table "Definition of TPMI_YES_NO Type" (Part 2: Structures)

```



```

523 TPM_RC
524 TPMI_YES_NO_Unmarshal(TPMI_YES_NO* target, BYTE** buffer, INT32* size);
525 #if !USE_MARSHALING_DEFINES
526 UINT16
527 TPMI_YES_NO_Marshal(TPMI_YES_NO* source, BYTE** buffer, INT32* size);
528 #else // !USE_MARSHALING_DEFINES
529 # define TPMI_YES_NO_Marshal(source, buffer, size) \
530     BYTE_Marshal((BYTE*)(source), (buffer), (size))
531 #endif // !USE_MARSHALING_DEFINES
532
533 // Table "Definition of TPMI_DH_OBJECT Type" (Part 2: Structures)
534 TPM_RC
535 TPMI_DH_OBJECT_Unmarshal(
536     TPMI_DH_OBJECT* target, BYTE** buffer, INT32* size, BOOL flag);
537 #if !USE_MARSHALING_DEFINES
538 UINT16
539 TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT* source, BYTE** buffer, INT32* size);
540 #else // !USE_MARSHALING_DEFINES
541 # define TPMI_DH_OBJECT_Marshal(source, buffer, size) \
542     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
543 #endif // !USE_MARSHALING_DEFINES
544
545 // Table "Definition of TPMI_DH_PARENT Type" (Part 2: Structures)
546 TPM_RC
547 TPMI_DH_PARENT_Unmarshal(TPMI_DH_PARENT* target, BYTE** buffer, INT32* size);
548 #if !USE_MARSHALING_DEFINES
549 UINT16
550 TPMI_DH_PARENT_Marshal(TPMI_DH_PARENT* source, BYTE** buffer, INT32* size);
551 #else // !USE_MARSHALING_DEFINES
552 # define TPMI_DH_PARENT_Marshal(source, buffer, size) \
553     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
554 #endif // !USE_MARSHALING_DEFINES
555
556 // Table "Definition of TPMI_DH_PERSISTENT Type" (Part 2: Structures)
557 TPM_RC
558 TPMI_DH_PERSISTENT_Unmarshal(TPMI_DH_PERSISTENT* target, BYTE** buffer, INT32* size);
559 #if !USE_MARSHALING_DEFINES
560 UINT16
561 TPMI_DH_PERSISTENT_Marshal(TPMI_DH_PERSISTENT* source, BYTE** buffer, INT32* size);
562 #else // !USE_MARSHALING_DEFINES
563 # define TPMI_DH_PERSISTENT_Marshal(source, buffer, size) \
564     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
565 #endif // !USE_MARSHALING_DEFINES
566
567 // Table "Definition of TPMI_DH_ENTITY Type" (Part 2: Structures)
568 TPM_RC
569 TPMI_DH_ENTITY_Unmarshal(
570     TPMI_DH_ENTITY* target, BYTE** buffer, INT32* size, BOOL flag);
571
572 // Table "Definition of TPMI_DH_PCR Type" (Part 2: Structures)
573 TPM_RC
574 TPMI_DH_PCR_Unmarshal(TPMI_DH_PCR* target, BYTE** buffer, INT32* size, BOOL flag);
575
576 // Table "Definition of TPMI_SH_AUTH_SESSION Type" (Part 2: Structures)
577 TPM_RC
578 TPMI_SH_AUTH_SESSION_Unmarshal(
579     TPMI_SH_AUTH_SESSION* target, BYTE** buffer, INT32* size, BOOL flag);
580 #if !USE_MARSHALING_DEFINES
581 UINT16
582 TPMI_SH_AUTH_SESSION_Marshal(
583     TPMI_SH_AUTH_SESSION* source, BYTE** buffer, INT32* size);
584 #else // !USE_MARSHALING_DEFINES
585 # define TPMI_SH_AUTH_SESSION_Marshal(source, buffer, size) \
586     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
587 #endif // !USE_MARSHALING_DEFINES
588

```



```

589 // Table "Definition of TPMI_SH_HMAC Type" (Part 2: Structures)
590 TPM_RC
591 TPMI_SH_HMAC_Unmarshal(TPMI_SH_HMAC* target, BYTE** buffer, INT32* size);
592 #if !USE_MARSHALING_DEFINES
593 UINT16
594 TPMI_SH_HMAC_Marshal(TPMI_SH_HMAC* source, BYTE** buffer, INT32* size);
595 #else // !USE_MARSHALING_DEFINES
596 # define TPMI_SH_HMAC_Marshal(source, buffer, size) \
597     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
598 #endif // !USE_MARSHALING_DEFINES
599
600 // Table "Definition of TPMI_SH_POLICY Type" (Part 2: Structures)
601 TPM_RC
602 TPMI_SH_POLICY_Unmarshal(TPMI_SH_POLICY* target, BYTE** buffer, INT32* size);
603 #if !USE_MARSHALING_DEFINES
604 UINT16
605 TPMI_SH_POLICY_Marshal(TPMI_SH_POLICY* source, BYTE** buffer, INT32* size);
606 #else // !USE_MARSHALING_DEFINES
607 # define TPMI_SH_POLICY_Marshal(source, buffer, size) \
608     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
609 #endif // !USE_MARSHALING_DEFINES
610
611 // Table "Definition of TPMI_DH_CONTEXT Type" (Part 2: Structures)
612 TPM_RC
613 TPMI_DH_CONTEXT_Unmarshal(TPMI_DH_CONTEXT* target, BYTE** buffer, INT32* size);
614 #if !USE_MARSHALING_DEFINES
615 UINT16
616 TPMI_DH_CONTEXT_Marshal(TPMI_DH_CONTEXT* source, BYTE** buffer, INT32* size);
617 #else // !USE_MARSHALING_DEFINES
618 # define TPMI_DH_CONTEXT_Marshal(source, buffer, size) \
619     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
620 #endif // !USE_MARSHALING_DEFINES
621
622 // Table "Definition of TPMI_DH_SAVED Type" (Part 2: Structures)
623 TPM_RC
624 TPMI_DH_SAVED_Unmarshal(TPMI_DH_SAVED* target, BYTE** buffer, INT32* size);
625 #if !USE_MARSHALING_DEFINES
626 UINT16
627 TPMI_DH_SAVED_Marshal(TPMI_DH_SAVED* source, BYTE** buffer, INT32* size);
628 #else // !USE_MARSHALING_DEFINES
629 # define TPMI_DH_SAVED_Marshal(source, buffer, size) \
630     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
631 #endif // !USE_MARSHALING_DEFINES
632
633 // Table "Definition of TPMI_RH_HIERARCHY Type" (Part 2: Structures)
634 TPM_RC
635 TPMI_RH_HIERARCHY_Unmarshal(TPMI_RH_HIERARCHY* target, BYTE** buffer, INT32* size);
636 #if !USE_MARSHALING_DEFINES
637 UINT16
638 TPMI_RH_HIERARCHY_Marshal(TPMI_RH_HIERARCHY* source, BYTE** buffer, INT32* size);
639 #else // !USE_MARSHALING_DEFINES
640 # define TPMI_RH_HIERARCHY_Marshal(source, buffer, size) \
641     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
642 #endif // !USE_MARSHALING_DEFINES
643
644 // Table "Definition of TPMI_RH_ENABLES Type" (Part 2: Structures)
645 TPM_RC
646 TPMI_RH_ENABLES_Unmarshal(
647     TPMI_RH_ENABLES* target, BYTE** buffer, INT32* size, BOOL flag);
648 #if !USE_MARSHALING_DEFINES
649 UINT16
650 TPMI_RH_ENABLES_Marshal(TPMI_RH_ENABLES* source, BYTE** buffer, INT32* size);
651 #else // !USE_MARSHALING_DEFINES
652 # define TPMI_RH_ENABLES_Marshal(source, buffer, size) \
653     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
654 #endif // !USE_MARSHALING_DEFINES

```

```

655
656 // Table "Definition of TPMI_RH_HIERARCHY_AUTH Type" (Part 2: Structures)
657 TPM_RC
658 TPMI_RH_HIERARCHY_AUTH_Unmarshal(
659     TPMI_RH_HIERARCHY_AUTH* target, BYTE** buffer, INT32* size);
660
661 // Table "Definition of TPMI_RH_HIERARCHY_POLICY Type" (Part 2: Structures)
662 TPM_RC
663 TPMI_RH_HIERARCHY_POLICY_Unmarshal(
664     TPMI_RH_HIERARCHY_POLICY* target, BYTE** buffer, INT32* size);
665
666 // Table "Definition of TPMI_RH_BASE_HIERARCHY Type" (Part 2: Structures)
667 TPM_RC
668 TPMI_RH_BASE_HIERARCHY_Unmarshal(
669     TPMI_RH_BASE_HIERARCHY* target, BYTE** buffer, INT32* size);
670 #if !USE_MARSHALING_DEFINES
671 UINT16
672 TPMI_RH_BASE_HIERARCHY_Marshal(
673     TPMI_RH_BASE_HIERARCHY* source, BYTE** buffer, INT32* size);
674 #else // !USE_MARSHALING_DEFINES
675 # define TPMI_RH_BASE_HIERARCHY_Marshal(source, buffer, size) \
676     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
677 #endif // !USE_MARSHALING_DEFINES
678
679 // Table "Definition of TPMI_RH_PLATFORM Type" (Part 2: Structures)
680 TPM_RC
681 TPMI_RH_PLATFORM_Unmarshal(TPMI_RH_PLATFORM* target, BYTE** buffer, INT32* size);
682
683 // Table "Definition of TPMI_RH_OWNER Type" (Part 2: Structures)
684 TPM_RC
685 TPMI_RH_OWNER_Unmarshal(TPMI_RH_OWNER* target, BYTE** buffer, INT32* size, BOOL flag);
686
687 // Table "Definition of TPMI_RH_ENDORSEMENT Type" (Part 2: Structures)
688 TPM_RC
689 TPMI_RH_ENDORSEMENT_Unmarshal(
690     TPMI_RH_ENDORSEMENT* target, BYTE** buffer, INT32* size, BOOL flag);
691
692 // Table "Definition of TPMI_RH_PROVISION Type" (Part 2: Structures)
693 TPM_RC
694 TPMI_RH_PROVISION_Unmarshal(TPMI_RH_PROVISION* target, BYTE** buffer, INT32* size);
695
696 // Table "Definition of TPMI_RH_CLEAR Type" (Part 2: Structures)
697 TPM_RC
698 TPMI_RH_CLEAR_Unmarshal(TPMI_RH_CLEAR* target, BYTE** buffer, INT32* size);
699
700 // Table "Definition of TPMI_RH_NV_AUTH Type" (Part 2: Structures)
701 TPM_RC
702 TPMI_RH_NV_AUTH_Unmarshal(TPMI_RH_NV_AUTH* target, BYTE** buffer, INT32* size);
703
704 // Table "Definition of TPMI_RH_LOCKOUT Type" (Part 2: Structures)
705 TPM_RC
706 TPMI_RH_LOCKOUT_Unmarshal(TPMI_RH_LOCKOUT* target, BYTE** buffer, INT32* size);
707
708 // Table "Definition of TPMI_RH_NV_INDEX Type" (Part 2: Structures)
709 TPM_RC
710 TPMI_RH_NV_INDEX_Unmarshal(TPMI_RH_NV_INDEX* target, BYTE** buffer, INT32* size);
711 #if !USE_MARSHALING_DEFINES
712 UINT16
713 TPMI_RH_NV_INDEX_Marshal(TPMI_RH_NV_INDEX* source, BYTE** buffer, INT32* size);
714 #else // !USE_MARSHALING_DEFINES
715 # define TPMI_RH_NV_INDEX_Marshal(source, buffer, size) \
716     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
717 #endif // !USE_MARSHALING_DEFINES
718
719 // Table "Definition of TPMI_RH_NV_DEFINED_INDEX Type" (Part 2: Structures)
720 TPM_RC

```

```

721 TPMI_RH_NV_DEFINED_INDEX_Unmarshal(
722     TPMI_RH_NV_DEFINED_INDEX* target, BYTE** buffer, INT32* size);
723
724 // Table "Definition of TPMI_RH_NV_LEGACY_INDEX Type" (Part 2: Structures)
725 TPM_RC
726 TPMI_RH_NV_LEGACY_INDEX_Unmarshal(
727     TPMI_RH_NV_LEGACY_INDEX* target, BYTE** buffer, INT32* size);
728 #if !USE_MARSHALING_DEFINES
729 UINT16
730 TPMI_RH_NV_LEGACY_INDEX_Marshal(
731     TPMI_RH_NV_LEGACY_INDEX* source, BYTE** buffer, INT32* size);
732 #else // !USE_MARSHALING_DEFINES
733 # define TPMI_RH_NV_LEGACY_INDEX_Marshal(source, buffer, size) \
734     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
735 #endif // !USE_MARSHALING_DEFINES
736
737 // Table "Definition of TPMI_RH_NV_EXP_INDEX Type" (Part 2: Structures)
738 TPM_RC
739 TPMI_RH_NV_EXP_INDEX_Unmarshal(
740     TPMI_RH_NV_EXP_INDEX* target, BYTE** buffer, INT32* size);
741 #if !USE_MARSHALING_DEFINES
742 UINT16
743 TPMI_RH_NV_EXP_INDEX_Marshal(
744     TPMI_RH_NV_EXP_INDEX* source, BYTE** buffer, INT32* size);
745 #else // !USE_MARSHALING_DEFINES
746 # define TPMI_RH_NV_EXP_INDEX_Marshal(source, buffer, size) \
747     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
748 #endif // !USE_MARSHALING_DEFINES
749
750 // Table "Definition of TPMI_RH_AC Type" (Part 2: Structures)
751 TPM_RC
752 TPMI_RH_AC_Unmarshal(TPMI_RH_AC* target, BYTE** buffer, INT32* size);
753
754 // Table "Definition of TPMI_RH_ACT Type" (Part 2: Structures)
755 TPM_RC
756 TPMI_RH_ACT_Unmarshal(TPMI_RH_ACT* target, BYTE** buffer, INT32* size);
757 #if !USE_MARSHALING_DEFINES
758 UINT16
759 TPMI_RH_ACT_Marshal(TPMI_RH_ACT* source, BYTE** buffer, INT32* size);
760 #else // !USE_MARSHALING_DEFINES
761 # define TPMI_RH_ACT_Marshal(source, buffer, size) \
762     TPM_HANDLE_Marshal((TPM_HANDLE*)(source), (buffer), (size))
763 #endif // !USE_MARSHALING_DEFINES
764
765 // Table "Definition of TPMI_ALG_HASH Type" (Part 2: Structures)
766 TPM_RC
767 TPMI_ALG_HASH_Unmarshal(TPMI_ALG_HASH* target, BYTE** buffer, INT32* size, BOOL flag);
768 #if !USE_MARSHALING_DEFINES
769 UINT16
770 TPMI_ALG_HASH_Marshal(TPMI_ALG_HASH* source, BYTE** buffer, INT32* size);
771 #else // !USE_MARSHALING_DEFINES
772 # define TPMI_ALG_HASH_Marshal(source, buffer, size) \
773     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
774 #endif // !USE_MARSHALING_DEFINES
775
776 // Table "Definition of TPMI_ALG_ASYM Type" (Part 2: Structures)
777 TPM_RC
778 TPMI_ALG_ASYM_Unmarshal(TPMI_ALG_ASYM* target, BYTE** buffer, INT32* size, BOOL flag);
779 #if !USE_MARSHALING_DEFINES
780 UINT16
781 TPMI_ALG_ASYM_Marshal(TPMI_ALG_ASYM* source, BYTE** buffer, INT32* size);
782 #else // !USE_MARSHALING_DEFINES
783 # define TPMI_ALG_ASYM_Marshal(source, buffer, size) \
784     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
785 #endif // !USE_MARSHALING_DEFINES
786

```

```

787 // Table "Definition of TPMI_ALG_SYM Type" (Part 2: Structures)
788 TPM_RC
789 TPMI_ALG_SYM_Unmarshal(TPMI_ALG_SYM* target, BYTE** buffer, INT32* size, BOOL flag);
790 #if !USE_MARSHALING_DEFINES
791 UINT16
792 TPMI_ALG_SYM_Marshal(TPMI_ALG_SYM* source, BYTE** buffer, INT32* size);
793 #else // !USE_MARSHALING_DEFINES
794 # define TPMI_ALG_SYM_Marshal(source, buffer, size) \
795     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
796 #endif // !USE_MARSHALING_DEFINES
797
798 // Table "Definition of TPMI_ALG_SYM_OBJECT Type" (Part 2: Structures)
799 TPM_RC
800 TPMI_ALG_SYM_OBJECT_Unmarshal(
801     TPMI_ALG_SYM_OBJECT* target, BYTE** buffer, INT32* size, BOOL flag);
802 #if !USE_MARSHALING_DEFINES
803 UINT16
804 TPMI_ALG_SYM_OBJECT_Marshal(TPMI_ALG_SYM_OBJECT* source, BYTE** buffer, INT32* size);
805 #else // !USE_MARSHALING_DEFINES
806 # define TPMI_ALG_SYM_OBJECT_Marshal(source, buffer, size) \
807     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
808 #endif // !USE_MARSHALING_DEFINES
809
810 // Table "Definition of TPMI_ALG_SYM_MODE Type" (Part 2: Structures)
811 TPM_RC
812 TPMI_ALG_SYM_MODE_Unmarshal(
813     TPMI_ALG_SYM_MODE* target, BYTE** buffer, INT32* size, BOOL flag);
814 #if !USE_MARSHALING_DEFINES
815 UINT16
816 TPMI_ALG_SYM_MODE_Marshal(TPMI_ALG_SYM_MODE* source, BYTE** buffer, INT32* size);
817 #else // !USE_MARSHALING_DEFINES
818 # define TPMI_ALG_SYM_MODE_Marshal(source, buffer, size) \
819     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
820 #endif // !USE_MARSHALING_DEFINES
821
822 // Table "Definition of TPMI_ALG_KDF Type" (Part 2: Structures)
823 TPM_RC
824 TPMI_ALG_KDF_Unmarshal(TPMI_ALG_KDF* target, BYTE** buffer, INT32* size, BOOL flag);
825 #if !USE_MARSHALING_DEFINES
826 UINT16
827 TPMI_ALG_KDF_Marshal(TPMI_ALG_KDF* source, BYTE** buffer, INT32* size);
828 #else // !USE_MARSHALING_DEFINES
829 # define TPMI_ALG_KDF_Marshal(source, buffer, size) \
830     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
831 #endif // !USE_MARSHALING_DEFINES
832
833 // Table "Definition of TPMI_ALG_SIG_SCHEME Type" (Part 2: Structures)
834 TPM_RC
835 TPMI_ALG_SIG_SCHEME_Unmarshal(
836     TPMI_ALG_SIG_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
837 #if !USE_MARSHALING_DEFINES
838 UINT16
839 TPMI_ALG_SIG_SCHEME_Marshal(TPMI_ALG_SIG_SCHEME* source, BYTE** buffer, INT32* size);
840 #else // !USE_MARSHALING_DEFINES
841 # define TPMI_ALG_SIG_SCHEME_Marshal(source, buffer, size) \
842     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
843 #endif // !USE_MARSHALING_DEFINES
844
845 // Table "Definition of TPMI_ECC_KEY_EXCHANGE Type" (Part 2: Structures)
846 TPM_RC
847 TPMI_ECC_KEY_EXCHANGE_Unmarshal(
848     TPMI_ECC_KEY_EXCHANGE* target, BYTE** buffer, INT32* size, BOOL flag);
849 #if !USE_MARSHALING_DEFINES
850 UINT16
851 TPMI_ECC_KEY_EXCHANGE_Marshal(
852     TPMI_ECC_KEY_EXCHANGE* source, BYTE** buffer, INT32* size);

```

```

853 #else // !USE_MARSHALING_DEFINES
854 # define TPMI_ECC_KEY_EXCHANGE_Marshal(source, buffer, size) \
855     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
856 #endif // !USE_MARSHALING_DEFINES
857
858 // Table "Definition of TPMI_ST_COMMAND_TAG Type" (Part 2: Structures)
859 TPM_RC
860 TPMI_ST_COMMAND_TAG_Unmarshal(
861     TPMI_ST_COMMAND_TAG* target, BYTE** buffer, INT32* size);
862 #if !USE_MARSHALING_DEFINES
863 UINT16
864 TPMI_ST_COMMAND_TAG_Marshal(TPMI_ST_COMMAND_TAG* source, BYTE** buffer, INT32* size);
865 #else // !USE_MARSHALING_DEFINES
866 # define TPMI_ST_COMMAND_TAG_Marshal(source, buffer, size) \
867     TPM_ST_Marshal((TPM_ST*)(source), (buffer), (size))
868 #endif // !USE_MARSHALING_DEFINES
869
870 // Table "Definition of TPMI_ALG_MAC_SCHEME Type" (Part 2: Structures)
871 TPM_RC
872 TPMI_ALG_MAC_SCHEME_Unmarshal(
873     TPMI_ALG_MAC_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
874 #if !USE_MARSHALING_DEFINES
875 UINT16
876 TPMI_ALG_MAC_SCHEME_Marshal(TPMI_ALG_MAC_SCHEME* source, BYTE** buffer, INT32* size);
877 #else // !USE_MARSHALING_DEFINES
878 # define TPMI_ALG_MAC_SCHEME_Marshal(source, buffer, size) \
879     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
880 #endif // !USE_MARSHALING_DEFINES
881
882 // Table "Definition of TPMI_ALG_CIPHER_MODE Type" (Part 2: Structures)
883 TPM_RC
884 TPMI_ALG_CIPHER_MODE_Unmarshal(
885     TPMI_ALG_CIPHER_MODE* target, BYTE** buffer, INT32* size, BOOL flag);
886 #if !USE_MARSHALING_DEFINES
887 UINT16
888 TPMI_ALG_CIPHER_MODE_Marshal(
889     TPMI_ALG_CIPHER_MODE* source, BYTE** buffer, INT32* size);
890 #else // !USE_MARSHALING_DEFINES
891 # define TPMI_ALG_CIPHER_MODE_Marshal(source, buffer, size) \
892     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
893 #endif // !USE_MARSHALING_DEFINES
894
895 // Table "Definition of TPMS_EMPTY Structure" (Part 2: Structures)
896 TPM_RC
897 TPMS_EMPTY_Unmarshal(TPMS_EMPTY* target, BYTE** buffer, INT32* size);
898 UINT16
899 TPMS_EMPTY_Marshal(TPMS_EMPTY* source, BYTE** buffer, INT32* size);
900
901 // Table "Definition of TPMS_ALGORITHM_DESCRIPTION Structure" (Part 2: Structures)
902 UINT16
903 TPMS_ALGORITHM_DESCRIPTION_Marshal(
904     TPMS_ALGORITHM_DESCRIPTION* source, BYTE** buffer, INT32* size);
905
906 // Table "Definition of TPMU_HA Union" (Part 2: Structures)
907 TPM_RC
908 TPMU_HA_Unmarshal(TPMU_HA* target, BYTE** buffer, INT32* size, UINT32 selector);
909 UINT16
910 TPMU_HA_Marshal(TPMU_HA* source, BYTE** buffer, INT32* size, UINT32 selector);
911
912 // Table "Definition of TPMT_HA Structure" (Part 2: Structures)
913 TPM_RC
914 TPMT_HA_Unmarshal(TPMT_HA* target, BYTE** buffer, INT32* size, BOOL flag);
915 UINT16
916 TPMT_HA_Marshal(TPMT_HA* source, BYTE** buffer, INT32* size);
917
918 // Table "Definition of TPM2B_DIGEST Structure" (Part 2: Structures)

```



```

919 TPM_RC
920 TPM2B_DIGEST_Unmarshal(TPM2B_DIGEST* target, BYTE** buffer, INT32* size);
921 UINT16
922 TPM2B_DIGEST_Marshal(TPM2B_DIGEST* source, BYTE** buffer, INT32* size);
923
924 // Table "Definition of TPM2B_DATA Structure" (Part 2: Structures)
925 TPM_RC
926 TPM2B_DATA_Unmarshal(TPM2B_DATA* target, BYTE** buffer, INT32* size);
927 UINT16
928 TPM2B_DATA_Marshal(TPM2B_DATA* source, BYTE** buffer, INT32* size);
929
930 // Table "Definition of Types for TPM2B_NONCE" (Part 2: Structures)
931 #if !USE_MARSHALING_DEFINES
932 TPM_RC
933 TPM2B_NONCE_Unmarshal(TPM2B_NONCE* target, BYTE** buffer, INT32* size);
934 #else // !USE_MARSHALING_DEFINES
935 # define TPM2B_NONCE_Unmarshal(target, buffer, size) \
936     TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)(target), (buffer), (size))
937 #endif // !USE_MARSHALING_DEFINES
938 #if !USE_MARSHALING_DEFINES
939 UINT16
940 TPM2B_NONCE_Marshal(TPM2B_NONCE* source, BYTE** buffer, INT32* size);
941 #else // !USE_MARSHALING_DEFINES
942 # define TPM2B_NONCE_Marshal(source, buffer, size) \
943     TPM2B_DIGEST_Marshal((TPM2B_DIGEST*)(source), (buffer), (size))
944 #endif // !USE_MARSHALING_DEFINES
945
946 // Table "Definition of Types for TPM2B_AUTH" (Part 2: Structures)
947 #if !USE_MARSHALING_DEFINES
948 TPM_RC
949 TPM2B_AUTH_Unmarshal(TPM2B_AUTH* target, BYTE** buffer, INT32* size);
950 #else // !USE_MARSHALING_DEFINES
951 # define TPM2B_AUTH_Unmarshal(target, buffer, size) \
952     TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)(target), (buffer), (size))
953 #endif // !USE_MARSHALING_DEFINES
954 #if !USE_MARSHALING_DEFINES
955 UINT16
956 TPM2B_AUTH_Marshal(TPM2B_AUTH* source, BYTE** buffer, INT32* size);
957 #else // !USE_MARSHALING_DEFINES
958 # define TPM2B_AUTH_Marshal(source, buffer, size) \
959     TPM2B_DIGEST_Marshal((TPM2B_DIGEST*)(source), (buffer), (size))
960 #endif // !USE_MARSHALING_DEFINES
961
962 // Table "Definition of Types for TPM2B_OPERAND" (Part 2: Structures)
963 #if !USE_MARSHALING_DEFINES
964 TPM_RC
965 TPM2B_OPERAND_Unmarshal(TPM2B_OPERAND* target, BYTE** buffer, INT32* size);
966 #else // !USE_MARSHALING_DEFINES
967 # define TPM2B_OPERAND_Unmarshal(target, buffer, size) \
968     TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)(target), (buffer), (size))
969 #endif // !USE_MARSHALING_DEFINES
970 #if !USE_MARSHALING_DEFINES
971 UINT16
972 TPM2B_OPERAND_Marshal(TPM2B_OPERAND* source, BYTE** buffer, INT32* size);
973 #else // !USE_MARSHALING_DEFINES
974 # define TPM2B_OPERAND_Marshal(source, buffer, size) \
975     TPM2B_DIGEST_Marshal((TPM2B_DIGEST*)(source), (buffer), (size))
976 #endif // !USE_MARSHALING_DEFINES
977
978 // Table "Definition of TPM2B_EVENT Structure" (Part 2: Structures)
979 TPM_RC
980 TPM2B_EVENT_Unmarshal(TPM2B_EVENT* target, BYTE** buffer, INT32* size);
981 UINT16
982 TPM2B_EVENT_Marshal(TPM2B_EVENT* source, BYTE** buffer, INT32* size);
983
984 // Table "Definition of TPM2B_MAX_BUFFER Structure" (Part 2: Structures)

```



```
985 TPM_RC
986 TPM2B_MAX_BUFFER_Unmarshal(TPM2B_MAX_BUFFER* target, BYTE** buffer, INT32* size);
987 UINT16
988 TPM2B_MAX_BUFFER_Marshal(TPM2B_MAX_BUFFER* source, BYTE** buffer, INT32* size);
989
990 // Table "Definition of TPM2B_MAX_NV_BUFFER Structure" (Part 2: Structures)
991 TPM_RC
992 TPM2B_MAX_NV_BUFFER_Unmarshal(
993     TPM2B_MAX_NV_BUFFER* target, BYTE** buffer, INT32* size);
994 UINT16
995 TPM2B_MAX_NV_BUFFER_Marshal(TPM2B_MAX_NV_BUFFER* source, BYTE** buffer, INT32* size);
996
997 // Table "Definition of TPM2B_TIMEOUT Structure" (Part 2: Structures)
998 TPM_RC
999 TPM2B_TIMEOUT_Unmarshal(TPM2B_TIMEOUT* target, BYTE** buffer, INT32* size);
1000 UINT16
1001 TPM2B_TIMEOUT_Marshal(TPM2B_TIMEOUT* source, BYTE** buffer, INT32* size);
1002
1003 // Table "Definition of TPM2B_IV Structure" (Part 2: Structures)
1004 TPM_RC
1005 TPM2B_IV_Unmarshal(TPM2B_IV* target, BYTE** buffer, INT32* size);
1006 UINT16
1007 TPM2B_IV_Marshal(TPM2B_IV* source, BYTE** buffer, INT32* size);
1008
1009 // Table "Definition of TPM2B_VENDOR_PROPERTY Structure" (Part 2: Structures)
1010 TPM_RC
1011 TPM2B_VENDOR_PROPERTY_Unmarshal(
1012     TPM2B_VENDOR_PROPERTY* target, BYTE** buffer, INT32* size);
1013 UINT16
1014 TPM2B_VENDOR_PROPERTY_Marshal(
1015     TPM2B_VENDOR_PROPERTY* source, BYTE** buffer, INT32* size);
1016
1017 // Table "Definition of TPM2B_NAME Structure" (Part 2: Structures)
1018 TPM_RC
1019 TPM2B_NAME_Unmarshal(TPM2B_NAME* target, BYTE** buffer, INT32* size);
1020 UINT16
1021 TPM2B_NAME_Marshal(TPM2B_NAME* source, BYTE** buffer, INT32* size);
1022
1023 // Table "Definition of TPMS_PCR_SELECT Structure" (Part 2: Structures)
1024 TPM_RC
1025 TPMS_PCR_SELECT_Unmarshal(TPMS_PCR_SELECT* target, BYTE** buffer, INT32* size);
1026 UINT16
1027 TPMS_PCR_SELECT_Marshal(TPMS_PCR_SELECT* source, BYTE** buffer, INT32* size);
1028
1029 // Table "Definition of TPMS_PCR_SELECTION Structure" (Part 2: Structures)
1030 TPM_RC
1031 TPMS_PCR_SELECTION_Unmarshal(TPMS_PCR_SELECTION* target, BYTE** buffer, INT32* size);
1032 UINT16
1033 TPMS_PCR_SELECTION_Marshal(TPMS_PCR_SELECTION* source, BYTE** buffer, INT32* size);
1034
1035 // Table "Definition of TPMT_TK_CREATION Structure" (Part 2: Structures)
1036 TPM_RC
1037 TPMT_TK_CREATION_Unmarshal(TPMT_TK_CREATION* target, BYTE** buffer, INT32* size);
1038 UINT16
1039 TPMT_TK_CREATION_Marshal(TPMT_TK_CREATION* source, BYTE** buffer, INT32* size);
1040
1041 // Table "Definition of TPMT_TK_VERIFIED Structure" (Part 2: Structures)
1042 TPM_RC
1043 TPMT_TK_VERIFIED_Unmarshal(TPMT_TK_VERIFIED* target, BYTE** buffer, INT32* size);
1044 UINT16
1045 TPMT_TK_VERIFIED_Marshal(TPMT_TK_VERIFIED* source, BYTE** buffer, INT32* size);
1046
1047 // Table "Definition of TPMT_TK_AUTH Structure" (Part 2: Structures)
1048 TPM_RC
1049 TPMT_TK_AUTH_Unmarshal(TPMT_TK_AUTH* target, BYTE** buffer, INT32* size);
1050 UINT16
```

```
1051 TPMT_TK_AUTH_Marshal(TPMT_TK_AUTH* source, BYTE** buffer, INT32* size);
1052
1053 // Table "Definition of TPMT_TK_HASHCHECK Structure" (Part 2: Structures)
1054 TPM_RC
1055 TPMT_TK_HASHCHECK_Unmarshal(TPMT_TK_HASHCHECK* target, BYTE** buffer, INT32* size);
1056 UINT16
1057 TPMT_TK_HASHCHECK_Marshal(TPMT_TK_HASHCHECK* source, BYTE** buffer, INT32* size);
1058
1059 // Table "Definition of TPMS_ALG_PROPERTY Structure" (Part 2: Structures)
1060 UINT16
1061 TPMS_ALG_PROPERTY_Marshal(TPMS_ALG_PROPERTY* source, BYTE** buffer, INT32* size);
1062
1063 // Table "Definition of TPMS_TAGGED_PROPERTY Structure" (Part 2: Structures)
1064 UINT16
1065 TPMS_TAGGED_PROPERTY_Marshal(
1066     TPMS_TAGGED_PROPERTY* source, BYTE** buffer, INT32* size);
1067
1068 // Table "Definition of TPMS_TAGGED_PCR_SELECT Structure" (Part 2: Structures)
1069 UINT16
1070 TPMS_TAGGED_PCR_SELECT_Marshal(
1071     TPMS_TAGGED_PCR_SELECT* source, BYTE** buffer, INT32* size);
1072
1073 // Table "Definition of TPMS_TAGGED_POLICY Structure" (Part 2: Structures)
1074 UINT16
1075 TPMS_TAGGED_POLICY_Marshal(TPMS_TAGGED_POLICY* source, BYTE** buffer, INT32* size);
1076
1077 // Table "Definition of TPMS_ACT_DATA Structure" (Part 2: Structures)
1078 UINT16
1079 TPMS_ACT_DATA_Marshal(TPMS_ACT_DATA* source, BYTE** buffer, INT32* size);
1080
1081 // Table "Definition of TPML_CC Structure" (Part 2: Structures)
1082 TPM_RC
1083 TPML_CC_Unmarshal(TPML_CC* target, BYTE** buffer, INT32* size);
1084 UINT16
1085 TPML_CC_Marshal(TPML_CC* source, BYTE** buffer, INT32* size);
1086
1087 // Table "Definition of TPML_CCA Structure" (Part 2: Structures)
1088 UINT16
1089 TPML_CCA_Marshal(TPML_CCA* source, BYTE** buffer, INT32* size);
1090
1091 // Table "Definition of TPML_ALG Structure" (Part 2: Structures)
1092 TPM_RC
1093 TPML_ALG_Unmarshal(TPML_ALG* target, BYTE** buffer, INT32* size);
1094 UINT16
1095 TPML_ALG_Marshal(TPML_ALG* source, BYTE** buffer, INT32* size);
1096
1097 // Table "Definition of TPML_HANDLE Structure" (Part 2: Structures)
1098 UINT16
1099 TPML_HANDLE_Marshal(TPML_HANDLE* source, BYTE** buffer, INT32* size);
1100
1101 // Table "Definition of TPML_DIGEST Structure" (Part 2: Structures)
1102 TPM_RC
1103 TPML_DIGEST_Unmarshal(TPML_DIGEST* target, BYTE** buffer, INT32* size);
1104 UINT16
1105 TPML_DIGEST_Marshal(TPML_DIGEST* source, BYTE** buffer, INT32* size);
1106
1107 // Table "Definition of TPML_DIGEST_VALUES Structure" (Part 2: Structures)
1108 TPM_RC
1109 TPML_DIGEST_VALUES_Unmarshal(TPML_DIGEST_VALUES* target, BYTE** buffer, INT32* size);
1110 UINT16
1111 TPML_DIGEST_VALUES_Marshal(TPML_DIGEST_VALUES* source, BYTE** buffer, INT32* size);
1112
1113 // Table "Definition of TPML_PCR_SELECTION Structure" (Part 2: Structures)
1114 TPM_RC
1115 TPML_PCR_SELECTION_Unmarshal(TPML_PCR_SELECTION* target, BYTE** buffer, INT32* size);
1116 UINT16
```

```

1117 TPML_PCR_SELECTION_Marshal(TPML_PCR_SELECTION* source, BYTE** buffer, INT32* size);
1118
1119 // Table "Definition of TPML_ALG_PROPERTY Structure" (Part 2: Structures)
1120 UINT16
1121 TPML_ALG_PROPERTY_Marshal(TPML_ALG_PROPERTY* source, BYTE** buffer, INT32* size);
1122
1123 // Table "Definition of TPML_TAGGED_TPM_PROPERTY Structure" (Part 2: Structures)
1124 UINT16
1125 TPML_TAGGED_TPM_PROPERTY_Marshal(
1126     TPML_TAGGED_TPM_PROPERTY* source, BYTE** buffer, INT32* size);
1127
1128 // Table "Definition of TPML_TAGGED_PCR_PROPERTY Structure" (Part 2: Structures)
1129 UINT16
1130 TPML_TAGGED_PCR_PROPERTY_Marshal(
1131     TPML_TAGGED_PCR_PROPERTY* source, BYTE** buffer, INT32* size);
1132
1133 // Table "Definition of TPML_ECC_CURVE Structure" (Part 2: Structures)
1134 #if ALG_ECC
1135 UINT16
1136 TPML_ECC_CURVE_Marshal(TPML_ECC_CURVE* source, BYTE** buffer, INT32* size);
1137 #else // ALG_ECC
1138 #define TPML_ECC_CURVE_Marshal UNIMPLEMENTED_Marshal
1139 #endif // ALG_ECC
1140
1141 // Table "Definition of TPML_TAGGED_POLICY Structure" (Part 2: Structures)
1142 UINT16
1143 TPML_TAGGED_POLICY_Marshal(TPML_TAGGED_POLICY* source, BYTE** buffer, INT32* size);
1144
1145 // Table "Definition of TPML_ACT_DATA Structure" (Part 2: Structures)
1146 UINT16
1147 TPML_ACT_DATA_Marshal(TPML_ACT_DATA* source, BYTE** buffer, INT32* size);
1148
1149 // Table "Definition of TPML_VENDOR_PROPERTY Structure" (Part 2: Structures)
1150 TPM_RC
1151 TPML_VENDOR_PROPERTY_Unmarshal(
1152     TPML_VENDOR_PROPERTY* target, BYTE** buffer, INT32* size);
1153 UINT16
1154 TPML_VENDOR_PROPERTY_Marshal(
1155     TPML_VENDOR_PROPERTY* source, BYTE** buffer, INT32* size);
1156
1157 // Table "Definition of TPMU_CAPABILITIES Union" (Part 2: Structures)
1158 UINT16
1159 TPMU_CAPABILITIES_Marshal(
1160     TPMU_CAPABILITIES* source, BYTE** buffer, INT32* size, UINT32 selector);
1161
1162 // Table "Definition of TPMS_CAPABILITY_DATA Structure" (Part 2: Structures)
1163 UINT16
1164 TPMS_CAPABILITY_DATA_Marshal(
1165     TPMS_CAPABILITY_DATA* source, BYTE** buffer, INT32* size);
1166
1167 // Table "Definition of TPMU_SET_CAPABILITIES Structure" (Part 2: Structures)
1168 TPM_RC
1169 TPMU_SET_CAPABILITIES_Unmarshal(
1170     TPMU_SET_CAPABILITIES* target, BYTE** buffer, INT32* size, UINT32 selector);
1171
1172 // Table "Definition of TPMS_SET_CAPABILITY_DATA Structure" (Part 2: Structures)
1173 TPM_RC
1174 TPMS_SET_CAPABILITY_DATA_Unmarshal(
1175     TPMS_SET_CAPABILITY_DATA* target, BYTE** buffer, INT32* size);
1176
1177 // Table "Definition of TPM2B_SET_CAPABILITY_DATA Structure" (Part 2: Structures)
1178 TPM_RC
1179 TPM2B_SET_CAPABILITY_DATA_Unmarshal(
1180     TPM2B_SET_CAPABILITY_DATA* target, BYTE** buffer, INT32* size);
1181
1182 // Table "Definition of TPMS_CLOCK_INFO Structure" (Part 2: Structures)

```

```

1183 TPM_RC
1184 TPMS_CLOCK_INFO_Unmarshal(TPMS_CLOCK_INFO* target, BYTE** buffer, INT32* size);
1185 UINT16
1186 TPMS_CLOCK_INFO_Marshal(TPMS_CLOCK_INFO* source, BYTE** buffer, INT32* size);
1187
1188 // Table "Definition of TPMS_TIME_INFO Structure" (Part 2: Structures)
1189 TPM_RC
1190 TPMS_TIME_INFO_Unmarshal(TPMS_TIME_INFO* target, BYTE** buffer, INT32* size);
1191 UINT16
1192 TPMS_TIME_INFO_Marshal(TPMS_TIME_INFO* source, BYTE** buffer, INT32* size);
1193
1194 // Table "Definition of TPMS_TIME_ATTEST_INFO Structure" (Part 2: Structures)
1195 UINT16
1196 TPMS_TIME_ATTEST_INFO_Marshal(
1197     TPMS_TIME_ATTEST_INFO* source, BYTE** buffer, INT32* size);
1198
1199 // Table "Definition of TPMS_CERTIFY_INFO Structure" (Part 2: Structures)
1200 UINT16
1201 TPMS_CERTIFY_INFO_Marshal(TPMS_CERTIFY_INFO* source, BYTE** buffer, INT32* size);
1202
1203 // Table "Definition of TPMS_QUOTE_INFO Structure" (Part 2: Structures)
1204 UINT16
1205 TPMS_QUOTE_INFO_Marshal(TPMS_QUOTE_INFO* source, BYTE** buffer, INT32* size);
1206
1207 // Table "Definition of TPMS_COMMAND_AUDIT_INFO Structure" (Part 2: Structures)
1208 UINT16
1209 TPMS_COMMAND_AUDIT_INFO_Marshal(
1210     TPMS_COMMAND_AUDIT_INFO* source, BYTE** buffer, INT32* size);
1211
1212 // Table "Definition of TPMS_SESSION_AUDIT_INFO Structure" (Part 2: Structures)
1213 UINT16
1214 TPMS_SESSION_AUDIT_INFO_Marshal(
1215     TPMS_SESSION_AUDIT_INFO* source, BYTE** buffer, INT32* size);
1216
1217 // Table "Definition of TPMS_CREATION_INFO Structure" (Part 2: Structures)
1218 UINT16
1219 TPMS_CREATION_INFO_Marshal(TPMS_CREATION_INFO* source, BYTE** buffer, INT32* size);
1220
1221 // Table "Definition of TPMS_NV_CERTIFY_INFO Structure" (Part 2: Structures)
1222 UINT16
1223 TPMS_NV_CERTIFY_INFO_Marshal(
1224     TPMS_NV_CERTIFY_INFO* source, BYTE** buffer, INT32* size);
1225
1226 // Table "Definition of TPMS_NV_DIGEST_CERTIFY_INFO Structure" (Part 2: Structures)
1227 UINT16
1228 TPMS_NV_DIGEST_CERTIFY_INFO_Marshal(
1229     TPMS_NV_DIGEST_CERTIFY_INFO* source, BYTE** buffer, INT32* size);
1230
1231 // Table "Definition of TPMI_ST_ATTEST Type" (Part 2: Structures)
1232 #if !USE_MARSHALING_DEFINES
1233 UINT16
1234 TPMI_ST_ATTEST_Marshal(TPMI_ST_ATTEST* source, BYTE** buffer, INT32* size);
1235 #else // !USE_MARSHALING_DEFINES
1236 #define TPMI_ST_ATTEST_Marshal(source, buffer, size) \
1237     TPM_ST_Marshal((TPM_ST*)(source), (buffer), (size))
1238 #endif // !USE_MARSHALING_DEFINES
1239
1240 // Table "Definition of TPMU_ATTEST Union" (Part 2: Structures)
1241 UINT16
1242 TPMU_ATTEST_Marshal(TPMU_ATTEST* source, BYTE** buffer, INT32* size, UINT32 selector);
1243
1244 // Table "Definition of TPMS_ATTEST Structure" (Part 2: Structures)
1245 UINT16
1246 TPMS_ATTEST_Marshal(TPMS_ATTEST* source, BYTE** buffer, INT32* size);
1247
1248 // Table "Definition of TPM2B_ATTEST Structure" (Part 2: Structures)

```

```

1249  UINT16
1250  TPM2B_ATTEST_Marshal(TPM2B_ATTEST* source, BYTE** buffer, INT32* size);
1251
1252  // Table "Definition of TPMS_AUTH_COMMAND Structure" (Part 2: Structures)
1253  TPM_RC
1254  TPMS_AUTH_COMMAND_Unmarshal(TPMS_AUTH_COMMAND* target, BYTE** buffer, INT32* size);
1255
1256  // Table "Definition of TPMS_AUTH_RESPONSE Structure" (Part 2: Structures)
1257  UINT16
1258  TPMS_AUTH_RESPONSE_Marshal(TPMS_AUTH_RESPONSE* source, BYTE** buffer, INT32* size);
1259
1260  // Table "Definition of TPMI_AES_KEY_BITS Type" (Part 2: Structures)
1261  TPM_RC
1262  TPMI_AES_KEY_BITS_Unmarshal(TPMI_AES_KEY_BITS* target, BYTE** buffer, INT32* size);
1263  #if !USE_MARSHALING_DEFINES
1264  UINT16
1265  TPMI_AES_KEY_BITS_Marshal(TPMI_AES_KEY_BITS* source, BYTE** buffer, INT32* size);
1266  #else // !USE_MARSHALING_DEFINES
1267  # define TPMI_AES_KEY_BITS_Marshal(source, buffer, size) \
1268      TPM_KEY_BITS_Marshal((TPM_KEY_BITS*)(source), (buffer), (size))
1269  #endif // !USE_MARSHALING_DEFINES
1270  // Table "Definition of TPMI_SM4_KEY_BITS Type" (Part 2: Structures)
1271  TPM_RC
1272  TPMI_SM4_KEY_BITS_Unmarshal(TPMI_SM4_KEY_BITS* target, BYTE** buffer, INT32* size);
1273  #if !USE_MARSHALING_DEFINES
1274  UINT16
1275  TPMI_SM4_KEY_BITS_Marshal(TPMI_SM4_KEY_BITS* source, BYTE** buffer, INT32* size);
1276  #else // !USE_MARSHALING_DEFINES
1277  # define TPMI_SM4_KEY_BITS_Marshal(source, buffer, size) \
1278      TPM_KEY_BITS_Marshal((TPM_KEY_BITS*)(source), (buffer), (size))
1279  #endif // !USE_MARSHALING_DEFINES
1280
1281  // Table "Definition of TPMI_CAMELLIA_KEY_BITS Type" (Part 2: Structures)
1282  TPM_RC
1283  TPMI_CAMELLIA_KEY_BITS_Unmarshal(
1284      TPMI_CAMELLIA_KEY_BITS* target, BYTE** buffer, INT32* size);
1285  #if !USE_MARSHALING_DEFINES
1286  UINT16
1287  TPMI_CAMELLIA_KEY_BITS_Marshal(
1288      TPMI_CAMELLIA_KEY_BITS* source, BYTE** buffer, INT32* size);
1289  #else // !USE_MARSHALING_DEFINES
1290  # define TPMI_CAMELLIA_KEY_BITS_Marshal(source, buffer, size) \
1291      TPM_KEY_BITS_Marshal((TPM_KEY_BITS*)(source), (buffer), (size))
1292  #endif // !USE_MARSHALING_DEFINES
1293
1294  // Table "Definition of TPMU_SYM_KEY_BITS Union" (Part 2: Structures)
1295  TPM_RC
1296  TPMU_SYM_KEY_BITS_Unmarshal(
1297      TPMU_SYM_KEY_BITS* target, BYTE** buffer, INT32* size, UINT32 selector);
1298  UINT16
1299  TPMU_SYM_KEY_BITS_Marshal(
1300      TPMU_SYM_KEY_BITS* source, BYTE** buffer, INT32* size, UINT32 selector);
1301
1302  // Table "Definition of TPMU_SYM_MODE Union" (Part 2: Structures)
1303  TPM_RC
1304  TPMU_SYM_MODE_Unmarshal(
1305      TPMU_SYM_MODE* target, BYTE** buffer, INT32* size, UINT32 selector);
1306  UINT16
1307  TPMU_SYM_MODE_Marshal(
1308      TPMU_SYM_MODE* source, BYTE** buffer, INT32* size, UINT32 selector);
1309
1310  // Table "Definition of TPMT_SYM_DEF Structure" (Part 2: Structures)
1311  TPM_RC
1312  TPMT_SYM_DEF_Unmarshal(TPMT_SYM_DEF* target, BYTE** buffer, INT32* size, BOOL flag);
1313  UINT16
1314  TPMT_SYM_DEF_Marshal(TPMT_SYM_DEF* source, BYTE** buffer, INT32* size);

```



```

1315
1316 // Table "Definition of TPMT_SYM_DEF_OBJECT Structure" (Part 2: Structures)
1317 TPM_RC
1318 TPMT_SYM_DEF_OBJECT_Unmarshal(
1319     TPMT_SYM_DEF_OBJECT* target, BYTE** buffer, INT32* size, BOOL flag);
1320 UINT16
1321 TPMT_SYM_DEF_OBJECT_Marshal(TPMT_SYM_DEF_OBJECT* source, BYTE** buffer, INT32* size);
1322
1323 // Table "Definition of TPM2B_SYM_KEY Structure" (Part 2: Structures)
1324 TPM_RC
1325 TPM2B_SYM_KEY_Unmarshal(TPM2B_SYM_KEY* target, BYTE** buffer, INT32* size);
1326 UINT16
1327 TPM2B_SYM_KEY_Marshal(TPM2B_SYM_KEY* source, BYTE** buffer, INT32* size);
1328
1329 // Table "Definition of TPMS_SYMCIPHER_PARMS Structure" (Part 2: Structures)
1330 TPM_RC
1331 TPMS_SYMCIPHER_PARMS_Unmarshal(
1332     TPMS_SYMCIPHER_PARMS* target, BYTE** buffer, INT32* size);
1333 UINT16
1334 TPMS_SYMCIPHER_PARMS_Marshal(
1335     TPMS_SYMCIPHER_PARMS* source, BYTE** buffer, INT32* size);
1336
1337 // Table "Definition of TPM2B_LABEL Structure" (Part 2: Structures)
1338 TPM_RC
1339 TPM2B_LABEL_Unmarshal(TPM2B_LABEL* target, BYTE** buffer, INT32* size);
1340 UINT16
1341 TPM2B_LABEL_Marshal(TPM2B_LABEL* source, BYTE** buffer, INT32* size);
1342
1343 // Table "Definition of TPMS_DERIVE Structure" (Part 2: Structures)
1344 TPM_RC
1345 TPMS_DERIVE_Unmarshal(TPMS_DERIVE* target, BYTE** buffer, INT32* size);
1346 UINT16
1347 TPMS_DERIVE_Marshal(TPMS_DERIVE* source, BYTE** buffer, INT32* size);
1348
1349 // Table "Definition of TPM2B_DERIVE Structure" (Part 2: Structures)
1350 TPM_RC
1351 TPM2B_DERIVE_Unmarshal(TPM2B_DERIVE* target, BYTE** buffer, INT32* size);
1352 UINT16
1353 TPM2B_DERIVE_Marshal(TPM2B_DERIVE* source, BYTE** buffer, INT32* size);
1354
1355 // Table "Definition of TPM2B_SENSITIVE_DATA Structure" (Part 2: Structures)
1356 TPM_RC
1357 TPM2B_SENSITIVE_DATA_Unmarshal(
1358     TPM2B_SENSITIVE_DATA* target, BYTE** buffer, INT32* size);
1359 UINT16
1360 TPM2B_SENSITIVE_DATA_Marshal(
1361     TPM2B_SENSITIVE_DATA* source, BYTE** buffer, INT32* size);
1362
1363 // Table "Definition of TPMS_SENSITIVE_CREATE Structure" (Part 2: Structures)
1364 TPM_RC
1365 TPMS_SENSITIVE_CREATE_Unmarshal(
1366     TPMS_SENSITIVE_CREATE* target, BYTE** buffer, INT32* size);
1367
1368 // Table "Definition of TPM2B_SENSITIVE_CREATE Structure" (Part 2: Structures)
1369 TPM_RC
1370 TPM2B_SENSITIVE_CREATE_Unmarshal(
1371     TPM2B_SENSITIVE_CREATE* target, BYTE** buffer, INT32* size);
1372
1373 // Table "Definition of TPMS_SCHEME_HASH Structure" (Part 2: Structures)
1374 TPM_RC
1375 TPMS_SCHEME_HASH_Unmarshal(TPMS_SCHEME_HASH* target, BYTE** buffer, INT32* size);
1376 UINT16
1377 TPMS_SCHEME_HASH_Marshal(TPMS_SCHEME_HASH* source, BYTE** buffer, INT32* size);
1378
1379 // Table "Definition of TPMS_SCHEME_ECDSA Structure" (Part 2: Structures)
1380 TPM_RC

```



```

1381 TPMS_SCHEME_ECDA_A_Unmarshal(TPMS_SCHEME_ECDA_A* target, BYTE** buffer, INT32* size);
1382 UINT16
1383 TPMS_SCHEME_ECDA_A_Marshal(TPMS_SCHEME_ECDA_A* source, BYTE** buffer, INT32* size);
1384
1385 // Table "Definition of TPMI_ALG_KEYEDHASH_SCHEME Type" (Part 2: Structures)
1386 TPM_RC
1387 TPMI_ALG_KEYEDHASH_SCHEME_Unmarshal(
1388     TPMI_ALG_KEYEDHASH_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
1389 #if !USE_MARSHALING_DEFINES
1390 UINT16
1391 TPMI_ALG_KEYEDHASH_SCHEME_Marshal(
1392     TPMI_ALG_KEYEDHASH_SCHEME* source, BYTE** buffer, INT32* size);
1393 #else // !USE_MARSHALING_DEFINES
1394 # define TPMI_ALG_KEYEDHASH_SCHEME_Marshal(source, buffer, size) \
1395     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
1396 #endif // !USE_MARSHALING_DEFINES
1397
1398 // Table "Definition of Types for HMAC_SIG_SCHEME" (Part 2: Structures)
1399 #if !USE_MARSHALING_DEFINES
1400 TPM_RC
1401 TPMS_SCHEME_HMAC_Unmarshal(TPMS_SCHEME_HMAC* target, BYTE** buffer, INT32* size);
1402 #else // !USE_MARSHALING_DEFINES
1403 # define TPMS_SCHEME_HMAC_Unmarshal(target, buffer, size) \
1404     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1405 #endif // !USE_MARSHALING_DEFINES
1406 #if !USE_MARSHALING_DEFINES
1407 UINT16
1408 TPMS_SCHEME_HMAC_Marshal(TPMS_SCHEME_HMAC* source, BYTE** buffer, INT32* size);
1409 #else // !USE_MARSHALING_DEFINES
1410 # define TPMS_SCHEME_HMAC_Marshal(source, buffer, size) \
1411     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1412 #endif // !USE_MARSHALING_DEFINES
1413
1414 // Table "Definition of TPMS_SCHEME_XOR Structure" (Part 2: Structures)
1415 TPM_RC
1416 TPMS_SCHEME_XOR_Unmarshal(TPMS_SCHEME_XOR* target, BYTE** buffer, INT32* size);
1417 UINT16
1418 TPMS_SCHEME_XOR_Marshal(TPMS_SCHEME_XOR* source, BYTE** buffer, INT32* size);
1419
1420 // Table "Definition of TPMU_SCHEME_KEYEDHASH Union" (Part 2: Structures)
1421 TPM_RC
1422 TPMU_SCHEME_KEYEDHASH_Unmarshal(
1423     TPMU_SCHEME_KEYEDHASH* target, BYTE** buffer, INT32* size, UINT32 selector);
1424 UINT16
1425 TPMU_SCHEME_KEYEDHASH_Marshal(
1426     TPMU_SCHEME_KEYEDHASH* source, BYTE** buffer, INT32* size, UINT32 selector);
1427
1428 // Table "Definition of TPMT_KEYEDHASH_SCHEME Structure" (Part 2: Structures)
1429 TPM_RC
1430 TPMT_KEYEDHASH_SCHEME_Unmarshal(
1431     TPMT_KEYEDHASH_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
1432 UINT16
1433 TPMT_KEYEDHASH_SCHEME_Marshal(
1434     TPMT_KEYEDHASH_SCHEME* source, BYTE** buffer, INT32* size);
1435
1436 // Table "Definition of Types for RSA Signature Schemes" (Part 2: Structures)
1437 #if !USE_MARSHALING_DEFINES
1438 TPM_RC
1439 TPMS_SIG_SCHEME_RSASSA_Unmarshal(
1440     TPMS_SIG_SCHEME_RSASSA* target, BYTE** buffer, INT32* size);
1441 #else // !USE_MARSHALING_DEFINES
1442 # define TPMS_SIG_SCHEME_RSASSA_Unmarshal(target, buffer, size) \
1443     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1444 #endif // !USE_MARSHALING_DEFINES
1445 #if !USE_MARSHALING_DEFINES
1446 UINT16

```

```

1447 TPMS_SIG_SCHEME_RSASSA_Marshal(
1448     TPMS_SIG_SCHEME_RSASSA* source, BYTE** buffer, INT32* size);
1449 #else // !USE_MARSHALING_DEFINES
1450 # define TPMS_SIG_SCHEME_RSASSA_Marshal(source, buffer, size) \
1451     TPMS_SIG_SCHEME_HASH_Marshal((TPMS_SIG_SCHEME_HASH*)(source), (buffer), (size))
1452 #endif // !USE_MARSHALING_DEFINES
1453 #if !USE_MARSHALING_DEFINES
1454 TPM_RC
1455 TPMS_SIG_SCHEME_RSAPSS_Unmarshal(
1456     TPMS_SIG_SCHEME_RSAPSS* target, BYTE** buffer, INT32* size);
1457 #else // !USE_MARSHALING_DEFINES
1458 # define TPMS_SIG_SCHEME_RSAPSS_Unmarshal(target, buffer, size) \
1459     TPMS_SIG_SCHEME_HASH_Unmarshal((TPMS_SIG_SCHEME_HASH*)(target), (buffer), (size))
1460 #endif // !USE_MARSHALING_DEFINES
1461 #if !USE_MARSHALING_DEFINES
1462 UINT16
1463 TPMS_SIG_SCHEME_RSAPSS_Marshal(
1464     TPMS_SIG_SCHEME_RSAPSS* source, BYTE** buffer, INT32* size);
1465 #else // !USE_MARSHALING_DEFINES
1466 # define TPMS_SIG_SCHEME_RSAPSS_Marshal(source, buffer, size) \
1467     TPMS_SIG_SCHEME_HASH_Marshal((TPMS_SIG_SCHEME_HASH*)(source), (buffer), (size))
1468 #endif // !USE_MARSHALING_DEFINES
1469
1470 // Table "Definition of Types for ECC Signature Schemes" (Part 2: Structures)
1471 #if !USE_MARSHALING_DEFINES
1472 TPM_RC
1473 TPMS_SIG_SCHEME_ECDSA_Unmarshal(
1474     TPMS_SIG_SCHEME_ECDSA* target, BYTE** buffer, INT32* size);
1475 #else // !USE_MARSHALING_DEFINES
1476 # define TPMS_SIG_SCHEME_ECDSA_Unmarshal(target, buffer, size) \
1477     TPMS_SIG_SCHEME_HASH_Unmarshal((TPMS_SIG_SCHEME_HASH*)(target), (buffer), (size))
1478 #endif // !USE_MARSHALING_DEFINES
1479 #if !USE_MARSHALING_DEFINES
1480 UINT16
1481 TPMS_SIG_SCHEME_ECDSA_Marshal(
1482     TPMS_SIG_SCHEME_ECDSA* source, BYTE** buffer, INT32* size);
1483 #else // !USE_MARSHALING_DEFINES
1484 # define TPMS_SIG_SCHEME_ECDSA_Marshal(source, buffer, size) \
1485     TPMS_SIG_SCHEME_HASH_Marshal((TPMS_SIG_SCHEME_HASH*)(source), (buffer), (size))
1486 #endif // !USE_MARSHALING_DEFINES
1487 #if !USE_MARSHALING_DEFINES
1488 TPM_RC
1489 TPMS_SIG_SCHEME_ECDSA_Unmarshal(
1490     TPMS_SIG_SCHEME_ECDSA* target, BYTE** buffer, INT32* size);
1491 #else // !USE_MARSHALING_DEFINES
1492 # define TPMS_SIG_SCHEME_ECDSA_Unmarshal(target, buffer, size) \
1493     TPMS_SIG_SCHEME_ECDSA_Unmarshal((TPMS_SIG_SCHEME_ECDSA*)(target), (buffer), (size))
1494 #endif // !USE_MARSHALING_DEFINES
1495 #if !USE_MARSHALING_DEFINES
1496 UINT16
1497 TPMS_SIG_SCHEME_ECDSA_Marshal(
1498     TPMS_SIG_SCHEME_ECDSA* source, BYTE** buffer, INT32* size);
1499 #else // !USE_MARSHALING_DEFINES
1500 # define TPMS_SIG_SCHEME_ECDSA_Marshal(source, buffer, size) \
1501     TPMS_SIG_SCHEME_ECDSA_Marshal((TPMS_SIG_SCHEME_ECDSA*)(source), (buffer), (size))
1502 #endif // !USE_MARSHALING_DEFINES
1503 #if !USE_MARSHALING_DEFINES
1504 TPM_RC
1505 TPMS_SIG_SCHEME_SM2_Unmarshal(
1506     TPMS_SIG_SCHEME_SM2* target, BYTE** buffer, INT32* size);
1507 #else // !USE_MARSHALING_DEFINES
1508 # define TPMS_SIG_SCHEME_SM2_Unmarshal(target, buffer, size) \
1509     TPMS_SIG_SCHEME_HASH_Unmarshal((TPMS_SIG_SCHEME_HASH*)(target), (buffer), (size))
1510 #endif // !USE_MARSHALING_DEFINES
1511 #if !USE_MARSHALING_DEFINES
1512 UINT16

```

```

1513 TPMS_SIG_SCHEME_SM2_Marshal(TPMS_SIG_SCHEME_SM2* source, BYTE** buffer, INT32* size);
1514 #else // !USE_MARSHALING_DEFINES
1515 # define TPMS_SIG_SCHEME_SM2_Marshal(source, buffer, size) \
1516     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1517 #endif // !USE_MARSHALING_DEFINES
1518 #if !USE_MARSHALING_DEFINES
1519 TPM_RC
1520 TPMS_SIG_SCHEME_ECSCNORR_Unmarshal(
1521     TPMS_SIG_SCHEME_ECSCNORR* target, BYTE** buffer, INT32* size);
1522 #else // !USE_MARSHALING_DEFINES
1523 # define TPMS_SIG_SCHEME_ECSCNORR_Unmarshal(target, buffer, size) \
1524     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1525 #endif // !USE_MARSHALING_DEFINES
1526 #if !USE_MARSHALING_DEFINES
1527 UINT16
1528 TPMS_SIG_SCHEME_ECSCNORR_Marshal(
1529     TPMS_SIG_SCHEME_ECSCNORR* source, BYTE** buffer, INT32* size);
1530 #else // !USE_MARSHALING_DEFINES
1531 # define TPMS_SIG_SCHEME_ECSCNORR_Marshal(source, buffer, size) \
1532     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1533 #endif // !USE_MARSHALING_DEFINES
1534 #if !USE_MARSHALING_DEFINES
1535 TPM_RC
1536 TPMS_SIG_SCHEME_EDDSA_Unmarshal(
1537     TPMS_SIG_SCHEME_EDDSA* target, BYTE** buffer, INT32* size);
1538 #else // !USE_MARSHALING_DEFINES
1539 # define TPMS_SIG_SCHEME_EDDSA_Unmarshal(target, buffer, size) \
1540     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1541 #endif // !USE_MARSHALING_DEFINES
1542 #if !USE_MARSHALING_DEFINES
1543 UINT16
1544 TPMS_SIG_SCHEME_EDDSA_Marshal(
1545     TPMS_SIG_SCHEME_EDDSA* source, BYTE** buffer, INT32* size);
1546 #else // !USE_MARSHALING_DEFINES
1547 # define TPMS_SIG_SCHEME_EDDSA_Marshal(source, buffer, size) \
1548     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1549 #endif // !USE_MARSHALING_DEFINES
1550 #if !USE_MARSHALING_DEFINES
1551 TPM_RC
1552 TPMS_SIG_SCHEME_EDDSA_PH_Unmarshal(
1553     TPMS_SIG_SCHEME_EDDSA_PH* target, BYTE** buffer, INT32* size);
1554 #else // !USE_MARSHALING_DEFINES
1555 # define TPMS_SIG_SCHEME_EDDSA_PH_Unmarshal(target, buffer, size) \
1556     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1557 #endif // !USE_MARSHALING_DEFINES
1558 #if !USE_MARSHALING_DEFINES
1559 UINT16
1560 TPMS_SIG_SCHEME_EDDSA_PH_Marshal(
1561     TPMS_SIG_SCHEME_EDDSA_PH* source, BYTE** buffer, INT32* size);
1562 #else // !USE_MARSHALING_DEFINES
1563 # define TPMS_SIG_SCHEME_EDDSA_PH_Marshal(source, buffer, size) \
1564     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1565 #endif // !USE_MARSHALING_DEFINES
1566
1567 // Table "Definition of TPMU_SIG_SCHEME Union" (Part 2: Structures)
1568 TPM_RC
1569 TPMU_SIG_SCHEME_Unmarshal(
1570     TPMU_SIG_SCHEME* target, BYTE** buffer, INT32* size, UINT32 selector);
1571 UINT16
1572 TPMU_SIG_SCHEME_Marshal(
1573     TPMU_SIG_SCHEME* source, BYTE** buffer, INT32* size, UINT32 selector);
1574
1575 // Table "Definition of TPMT_SIG_SCHEME Structure" (Part 2: Structures)
1576 TPM_RC
1577 TPMT_SIG_SCHEME_Unmarshal(
1578     TPMT_SIG_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);

```

```

1579 UINT16
1580 TPMT_SIG_SCHEME_Marshal(TPMT_SIG_SCHEME* source, BYTE** buffer, INT32* size);
1581
1582 // Table "Definition of Types for Encryption Schemes" (Part 2: Structures)
1583 #if !USE_MARSHALING_DEFINES
1584 TPM_RC
1585 TPMS_ENC_SCHEME_RSAES_Unmarshal(
1586     TPMS_ENC_SCHEME_RSAES* target, BYTE** buffer, INT32* size);
1587 #else // !USE_MARSHALING_DEFINES
1588 # define TPMS_ENC_SCHEME_RSAES_Unmarshal(target, buffer, size) \
1589     TPMS_EMPTY_Unmarshal((TPMS_EMPTY*)(target), (buffer), (size))
1590 #endif // !USE_MARSHALING_DEFINES
1591 #if !USE_MARSHALING_DEFINES
1592 UINT16
1593 TPMS_ENC_SCHEME_RSAES_Marshal(
1594     TPMS_ENC_SCHEME_RSAES* source, BYTE** buffer, INT32* size);
1595 #else // !USE_MARSHALING_DEFINES
1596 # define TPMS_ENC_SCHEME_RSAES_Marshal(source, buffer, size) \
1597     TPMS_EMPTY_Marshal((TPMS_EMPTY*)(source), (buffer), (size))
1598 #endif // !USE_MARSHALING_DEFINES
1599 #if !USE_MARSHALING_DEFINES
1600 TPM_RC
1601 TPMS_ENC_SCHEME_OAEP_Unmarshal(
1602     TPMS_ENC_SCHEME_OAEP* target, BYTE** buffer, INT32* size);
1603 #else // !USE_MARSHALING_DEFINES
1604 # define TPMS_ENC_SCHEME_OAEP_Unmarshal(target, buffer, size) \
1605     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1606 #endif // !USE_MARSHALING_DEFINES
1607 #if !USE_MARSHALING_DEFINES
1608 UINT16
1609 TPMS_ENC_SCHEME_OAEP_Marshal(
1610     TPMS_ENC_SCHEME_OAEP* source, BYTE** buffer, INT32* size);
1611 #else // !USE_MARSHALING_DEFINES
1612 # define TPMS_ENC_SCHEME_OAEP_Marshal(source, buffer, size) \
1613     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1614 #endif // !USE_MARSHALING_DEFINES
1615
1616 // Table "Definition of Types for ECC Key Exchange" (Part 2: Structures)
1617 #if !USE_MARSHALING_DEFINES
1618 TPM_RC
1619 TPMS_KEY_SCHEME_ECDH_Unmarshal(
1620     TPMS_KEY_SCHEME_ECDH* target, BYTE** buffer, INT32* size);
1621 #else // !USE_MARSHALING_DEFINES
1622 # define TPMS_KEY_SCHEME_ECDH_Unmarshal(target, buffer, size) \
1623     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1624 #endif // !USE_MARSHALING_DEFINES
1625 #if !USE_MARSHALING_DEFINES
1626 UINT16
1627 TPMS_KEY_SCHEME_ECDH_Marshal(
1628     TPMS_KEY_SCHEME_ECDH* source, BYTE** buffer, INT32* size);
1629 #else // !USE_MARSHALING_DEFINES
1630 # define TPMS_KEY_SCHEME_ECDH_Marshal(source, buffer, size) \
1631     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1632 #endif // !USE_MARSHALING_DEFINES
1633 #if !USE_MARSHALING_DEFINES
1634 TPM_RC
1635 TPMS_KEY_SCHEME_SM2_Unmarshal(
1636     TPMS_KEY_SCHEME_SM2* target, BYTE** buffer, INT32* size);
1637 #else // !USE_MARSHALING_DEFINES
1638 # define TPMS_KEY_SCHEME_SM2_Unmarshal(target, buffer, size) \
1639     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1640 #endif // !USE_MARSHALING_DEFINES
1641 #if !USE_MARSHALING_DEFINES
1642 UINT16
1643 TPMS_KEY_SCHEME_SM2_Marshal(TPMS_KEY_SCHEME_SM2* source, BYTE** buffer, INT32* size);
1644 #else // !USE_MARSHALING_DEFINES

```



```

1645 # define TPMS_KEY_SCHEME_SM2_Marshal(source, buffer, size) \
1646     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1647 #endif // !USE_MARSHALING_DEFINES
1648 #if !USE_MARSHALING_DEFINES
1649 TPM_RC
1650 TPMS_KEY_SCHEME_ECMQV_Unmarshal(
1651     TPMS_KEY_SCHEME_ECMQV* target, BYTE** buffer, INT32* size);
1652 #else // !USE_MARSHALING_DEFINES
1653 # define TPMS_KEY_SCHEME_ECMQV_Unmarshal(target, buffer, size) \
1654     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1655 #endif // !USE_MARSHALING_DEFINES
1656 #if !USE_MARSHALING_DEFINES
1657 UINT16
1658 TPMS_KEY_SCHEME_ECMQV_Marshal(
1659     TPMS_KEY_SCHEME_ECMQV* source, BYTE** buffer, INT32* size);
1660 #else // !USE_MARSHALING_DEFINES
1661 # define TPMS_KEY_SCHEME_ECMQV_Marshal(source, buffer, size) \
1662     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1663 #endif // !USE_MARSHALING_DEFINES
1664
1665 // Table "Definition of Types for KDF Schemes" (Part 2: Structures)
1666 #if !USE_MARSHALING_DEFINES
1667 TPM_RC
1668 TPMS_KDF_SCHEME_MGF1_Unmarshal(
1669     TPMS_KDF_SCHEME_MGF1* target, BYTE** buffer, INT32* size);
1670 #else // !USE_MARSHALING_DEFINES
1671 # define TPMS_KDF_SCHEME_MGF1_Unmarshal(target, buffer, size) \
1672     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1673 #endif // !USE_MARSHALING_DEFINES
1674 #if !USE_MARSHALING_DEFINES
1675 UINT16
1676 TPMS_KDF_SCHEME_MGF1_Marshal(
1677     TPMS_KDF_SCHEME_MGF1* source, BYTE** buffer, INT32* size);
1678 #else // !USE_MARSHALING_DEFINES
1679 # define TPMS_KDF_SCHEME_MGF1_Marshal(source, buffer, size) \
1680     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1681 #endif // !USE_MARSHALING_DEFINES
1682 #if !USE_MARSHALING_DEFINES
1683 TPM_RC
1684 TPMS_KDF_SCHEME_KDF1_SP800_56A_Unmarshal(
1685     TPMS_KDF_SCHEME_KDF1_SP800_56A* target, BYTE** buffer, INT32* size);
1686 #else // !USE_MARSHALING_DEFINES
1687 # define TPMS_KDF_SCHEME_KDF1_SP800_56A_Unmarshal(target, buffer, size) \
1688     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1689 #endif // !USE_MARSHALING_DEFINES
1690 #if !USE_MARSHALING_DEFINES
1691 UINT16
1692 TPMS_KDF_SCHEME_KDF1_SP800_56A_Marshal(
1693     TPMS_KDF_SCHEME_KDF1_SP800_56A* source, BYTE** buffer, INT32* size);
1694 #else // !USE_MARSHALING_DEFINES
1695 # define TPMS_KDF_SCHEME_KDF1_SP800_56A_Marshal(source, buffer, size) \
1696     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1697 #endif // !USE_MARSHALING_DEFINES
1698 #if !USE_MARSHALING_DEFINES
1699 TPM_RC
1700 TPMS_KDF_SCHEME_KDF2_Unmarshal(
1701     TPMS_KDF_SCHEME_KDF2* target, BYTE** buffer, INT32* size);
1702 #else // !USE_MARSHALING_DEFINES
1703 # define TPMS_KDF_SCHEME_KDF2_Unmarshal(target, buffer, size) \
1704     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1705 #endif // !USE_MARSHALING_DEFINES
1706 #if !USE_MARSHALING_DEFINES
1707 UINT16
1708 TPMS_KDF_SCHEME_KDF2_Marshal(
1709     TPMS_KDF_SCHEME_KDF2* source, BYTE** buffer, INT32* size);
1710 #else // !USE_MARSHALING_DEFINES

```

```

1711 # define TPMS_KDF_SCHEME_KDF2_Marshal(source, buffer, size) \
1712     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1713 #endif // !USE_MARSHALING_DEFINES
1714 #if !USE_MARSHALING_DEFINES
1715 TPM_RC
1716 TPMS_KDF_SCHEME_KDF1_SP800_108_Unmarshal(
1717     TPMS_KDF_SCHEME_KDF1_SP800_108* target, BYTE** buffer, INT32* size);
1718 #else // !USE_MARSHALING_DEFINES
1719 # define TPMS_KDF_SCHEME_KDF1_SP800_108_Unmarshal(target, buffer, size) \
1720     TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)(target), (buffer), (size))
1721 #endif // !USE_MARSHALING_DEFINES
1722 #if !USE_MARSHALING_DEFINES
1723 UINT16
1724 TPMS_KDF_SCHEME_KDF1_SP800_108_Marshal(
1725     TPMS_KDF_SCHEME_KDF1_SP800_108* source, BYTE** buffer, INT32* size);
1726 #else // !USE_MARSHALING_DEFINES
1727 # define TPMS_KDF_SCHEME_KDF1_SP800_108_Marshal(source, buffer, size) \
1728     TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)(source), (buffer), (size))
1729 #endif // !USE_MARSHALING_DEFINES
1730
1731 // Table "Definition of TPMU_KDF_SCHEME Union" (Part 2: Structures)
1732 TPM_RC
1733 TPMU_KDF_SCHEME_Unmarshal(
1734     TPMU_KDF_SCHEME* target, BYTE** buffer, INT32* size, UINT32 selector);
1735 UINT16
1736 TPMU_KDF_SCHEME_Marshal(
1737     TPMU_KDF_SCHEME* source, BYTE** buffer, INT32* size, UINT32 selector);
1738
1739 // Table "Definition of TPMT_KDF_SCHEME Structure" (Part 2: Structures)
1740 TPM_RC
1741 TPMT_KDF_SCHEME_Unmarshal(
1742     TPMT_KDF_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
1743 UINT16
1744 TPMT_KDF_SCHEME_Marshal(TPMT_KDF_SCHEME* source, BYTE** buffer, INT32* size);
1745
1746 // Table "Definition of TPMI_ALG_ASYM_SCHEME Type" (Part 2: Structures)
1747 TPM_RC
1748 TPMI_ALG_ASYM_SCHEME_Unmarshal(
1749     TPMI_ALG_ASYM_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
1750 #if !USE_MARSHALING_DEFINES
1751 UINT16
1752 TPMI_ALG_ASYM_SCHEME_Marshal(
1753     TPMI_ALG_ASYM_SCHEME* source, BYTE** buffer, INT32* size);
1754 #else // !USE_MARSHALING_DEFINES
1755 # define TPMI_ALG_ASYM_SCHEME_Marshal(source, buffer, size) \
1756     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
1757 #endif // !USE_MARSHALING_DEFINES
1758
1759 // Table "Definition of TPMU_ASYM_SCHEME Union" (Part 2: Structures)
1760 TPM_RC
1761 TPMU_ASYM_SCHEME_Unmarshal(
1762     TPMU_ASYM_SCHEME* target, BYTE** buffer, INT32* size, UINT32 selector);
1763 UINT16
1764 TPMU_ASYM_SCHEME_Marshal(
1765     TPMU_ASYM_SCHEME* source, BYTE** buffer, INT32* size, UINT32 selector);
1766
1767 // Table "Definition of TPMI_ALG_RSA_SCHEME Type" (Part 2: Structures)
1768 TPM_RC
1769 TPMI_ALG_RSA_SCHEME_Unmarshal(
1770     TPMI_ALG_RSA_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
1771 #if !USE_MARSHALING_DEFINES
1772 UINT16
1773 TPMI_ALG_RSA_SCHEME_Marshal(TPMI_ALG_RSA_SCHEME* source, BYTE** buffer, INT32* size);
1774 #else // !USE_MARSHALING_DEFINES
1775 # define TPMI_ALG_RSA_SCHEME_Marshal(source, buffer, size) \
1776     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))

```



```

1777 #endif // !USE_MARSHALING_DEFINES
1778
1779 // Table "Definition of TPMT_RSA_SCHEME Structure" (Part 2: Structures)
1780 TPM_RC
1781 TPMT_RSA_SCHEME_Unmarshal(
1782     TPMT_RSA_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
1783 UINT16
1784 TPMT_RSA_SCHEME_Marshal(TPMT_RSA_SCHEME* source, BYTE** buffer, INT32* size);
1785
1786 // Table "Definition of TPMI_ALG_RSA_DECRYPT Type" (Part 2: Structures)
1787 TPM_RC
1788 TPMI_ALG_RSA_DECRYPT_Unmarshal(
1789     TPMI_ALG_RSA_DECRYPT* target, BYTE** buffer, INT32* size, BOOL flag);
1790 #if !USE_MARSHALING_DEFINES
1791 UINT16
1792 TPMI_ALG_RSA_DECRYPT_Marshal(
1793     TPMI_ALG_RSA_DECRYPT* source, BYTE** buffer, INT32* size);
1794 #else // !USE_MARSHALING_DEFINES
1795 # define TPMI_ALG_RSA_DECRYPT_Marshal(source, buffer, size) \
1796     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
1797 #endif // !USE_MARSHALING_DEFINES
1798
1799 // Table "Definition of TPMT_RSA_DECRYPT Structure" (Part 2: Structures)
1800 TPM_RC
1801 TPMT_RSA_DECRYPT_Unmarshal(
1802     TPMT_RSA_DECRYPT* target, BYTE** buffer, INT32* size, BOOL flag);
1803 UINT16
1804 TPMT_RSA_DECRYPT_Marshal(TPMT_RSA_DECRYPT* source, BYTE** buffer, INT32* size);
1805
1806 // Table "Definition of TPM2B_PUBLIC_KEY_RSA Structure" (Part 2: Structures)
1807 TPM_RC
1808 TPM2B_PUBLIC_KEY_RSA_Unmarshal(
1809     TPM2B_PUBLIC_KEY_RSA* target, BYTE** buffer, INT32* size);
1810 UINT16
1811 TPM2B_PUBLIC_KEY_RSA_Marshal(
1812     TPM2B_PUBLIC_KEY_RSA* source, BYTE** buffer, INT32* size);
1813
1814 // Table "Definition of TPMI_RSA_KEY_BITS Type" (Part 2: Structures)
1815 TPM_RC
1816 TPMI_RSA_KEY_BITS_Unmarshal(TPMI_RSA_KEY_BITS* target, BYTE** buffer, INT32* size);
1817 #if !USE_MARSHALING_DEFINES
1818 UINT16
1819 TPMI_RSA_KEY_BITS_Marshal(TPMI_RSA_KEY_BITS* source, BYTE** buffer, INT32* size);
1820 #else // !USE_MARSHALING_DEFINES
1821 # define TPMI_RSA_KEY_BITS_Marshal(source, buffer, size) \
1822     TPM_KEY_BITS_Marshal((TPM_KEY_BITS*)(source), (buffer), (size))
1823 #endif // !USE_MARSHALING_DEFINES
1824
1825 // Table "Definition of TPM2B_PRIVATE_KEY_RSA Structure" (Part 2: Structures)
1826 TPM_RC
1827 TPM2B_PRIVATE_KEY_RSA_Unmarshal(
1828     TPM2B_PRIVATE_KEY_RSA* target, BYTE** buffer, INT32* size);
1829 UINT16
1830 TPM2B_PRIVATE_KEY_RSA_Marshal(
1831     TPM2B_PRIVATE_KEY_RSA* source, BYTE** buffer, INT32* size);
1832
1833 // Table "Definition of TPM2B_ECC_PARAMETER Structure" (Part 2: Structures)
1834 TPM_RC
1835 TPM2B_ECC_PARAMETER_Unmarshal(
1836     TPM2B_ECC_PARAMETER* target, BYTE** buffer, INT32* size);
1837 UINT16
1838 TPM2B_ECC_PARAMETER_Marshal(TPM2B_ECC_PARAMETER* source, BYTE** buffer, INT32* size);
1839
1840 // Table "Definition of TPMS_ECC_POINT Structure" (Part 2: Structures)
1841 TPM_RC
1842 TPMS_ECC_POINT_Unmarshal(TPMS_ECC_POINT* target, BYTE** buffer, INT32* size);

```

```

1843 UINT16
1844 TPMS_ECC_POINT_Marshal(TPMS_ECC_POINT* source, BYTE** buffer, INT32* size);
1845
1846 // Table "Definition of TPM2B_ECC_POINT Structure" (Part 2: Structures)
1847 TPM_RC
1848 TPM2B_ECC_POINT_Unmarshal(TPM2B_ECC_POINT* target, BYTE** buffer, INT32* size);
1849 UINT16
1850 TPM2B_ECC_POINT_Marshal(TPM2B_ECC_POINT* source, BYTE** buffer, INT32* size);
1851
1852 // Table "Definition of TPMI_ALG_ECC_SCHEME Type" (Part 2: Structures)
1853 TPM_RC
1854 TPMI_ALG_ECC_SCHEME_Unmarshal(
1855     TPMI_ALG_ECC_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
1856 #if !USE_MARSHALING_DEFINES
1857 UINT16
1858 TPMI_ALG_ECC_SCHEME_Marshal(TPMI_ALG_ECC_SCHEME* source, BYTE** buffer, INT32* size);
1859 #else // !USE_MARSHALING_DEFINES
1860 # define TPMI_ALG_ECC_SCHEME_Marshal(source, buffer, size) \
1861     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
1862 #endif // !USE_MARSHALING_DEFINES
1863
1864 // Table "Definition of TPMI_ECC_CURVE Type" (Part 2: Structures)
1865 TPM_RC
1866 TPMI_ECC_CURVE_Unmarshal(
1867     TPMI_ECC_CURVE* target, BYTE** buffer, INT32* size, BOOL flag);
1868 #if !USE_MARSHALING_DEFINES
1869 UINT16
1870 TPMI_ECC_CURVE_Marshal(TPMI_ECC_CURVE* source, BYTE** buffer, INT32* size);
1871 #else // !USE_MARSHALING_DEFINES
1872 # define TPMI_ECC_CURVE_Marshal(source, buffer, size) \
1873     TPM_ECC_CURVE_Marshal((TPM_ECC_CURVE*)(source), (buffer), (size))
1874 #endif // !USE_MARSHALING_DEFINES
1875
1876 // Table "Definition of TPMT_ECC_SCHEME Structure" (Part 2: Structures)
1877 TPM_RC
1878 TPMT_ECC_SCHEME_Unmarshal(
1879     TPMT_ECC_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag);
1880 UINT16
1881 TPMT_ECC_SCHEME_Marshal(TPMT_ECC_SCHEME* source, BYTE** buffer, INT32* size);
1882
1883 // Table "Definition of TPMS_ALGORITHM_DETAIL_ECC Structure" (Part 2: Structures)
1884 UINT16
1885 TPMS_ALGORITHM_DETAIL_ECC_Marshal(
1886     TPMS_ALGORITHM_DETAIL_ECC* source, BYTE** buffer, INT32* size);
1887
1888 // Table "Definition of TPMS_SIGNATURE_RSA Structure" (Part 2: Structures)
1889 TPM_RC
1890 TPMS_SIGNATURE_RSA_Unmarshal(TPMS_SIGNATURE_RSA* target, BYTE** buffer, INT32* size);
1891 UINT16
1892 TPMS_SIGNATURE_RSA_Marshal(TPMS_SIGNATURE_RSA* source, BYTE** buffer, INT32* size);
1893
1894 // Table "Definition of Types for Signature" (Part 2: Structures)
1895 #if !USE_MARSHALING_DEFINES
1896 TPM_RC
1897 TPMS_SIGNATURE_RSASSA_Unmarshal(
1898     TPMS_SIGNATURE_RSASSA* target, BYTE** buffer, INT32* size);
1899 #else // !USE_MARSHALING_DEFINES
1900 # define TPMS_SIGNATURE_RSASSA_Unmarshal(target, buffer, size) \
1901     TPMS_SIGNATURE_RSA_Unmarshal((TPMS_SIGNATURE_RSA*)(target), (buffer), (size))
1902 #endif // !USE_MARSHALING_DEFINES
1903 #if !USE_MARSHALING_DEFINES
1904 UINT16
1905 TPMS_SIGNATURE_RSASSA_Marshal(
1906     TPMS_SIGNATURE_RSASSA* source, BYTE** buffer, INT32* size);
1907 #else // !USE_MARSHALING_DEFINES
1908 # define TPMS_SIGNATURE_RSASSA_Marshal(source, buffer, size) \

```

```

1909     TPMS_SIGNATURE_RSA_Marshal((TPMS_SIGNATURE_RSA*)(source), (buffer), (size))
1910 #endif // !USE_MARSHALING_DEFINES
1911 #if !USE_MARSHALING_DEFINES
1912 TPM_RC
1913 TPMS_SIGNATURE_RSAPSS_Unmarshal(
1914     TPMS_SIGNATURE_RSAPSS* target, BYTE** buffer, INT32* size);
1915 #else // !USE_MARSHALING_DEFINES
1916 # define TPMS_SIGNATURE_RSAPSS_Unmarshal(target, buffer, size) \
1917     TPMS_SIGNATURE_RSA_Unmarshal((TPMS_SIGNATURE_RSA*)(target), (buffer), (size))
1918 #endif // !USE_MARSHALING_DEFINES
1919 #if !USE_MARSHALING_DEFINES
1920 UINT16
1921 TPMS_SIGNATURE_RSAPSS_Marshal(
1922     TPMS_SIGNATURE_RSAPSS* source, BYTE** buffer, INT32* size);
1923 #else // !USE_MARSHALING_DEFINES
1924 # define TPMS_SIGNATURE_RSAPSS_Marshal(source, buffer, size) \
1925     TPMS_SIGNATURE_RSA_Marshal((TPMS_SIGNATURE_RSA*)(source), (buffer), (size))
1926 #endif // !USE_MARSHALING_DEFINES
1927
1928 // Table "Definition of TPMS_SIGNATURE_ECC Structure" (Part 2: Structures)
1929 TPM_RC
1930 TPMS_SIGNATURE_ECC_Unmarshal(TPMS_SIGNATURE_ECC* target, BYTE** buffer, INT32* size);
1931 UINT16
1932 TPMS_SIGNATURE_ECC_Marshal(TPMS_SIGNATURE_ECC* source, BYTE** buffer, INT32* size);
1933
1934 // Table "Definition of Types for TPMS_SIGNATURE_ECC" (Part 2: Structures)
1935 #if !USE_MARSHALING_DEFINES
1936 TPM_RC
1937 TPMS_SIGNATURE_ECDSA_Unmarshal(
1938     TPMS_SIGNATURE_ECDSA* target, BYTE** buffer, INT32* size);
1939 #else // !USE_MARSHALING_DEFINES
1940 # define TPMS_SIGNATURE_ECDSA_Unmarshal(target, buffer, size) \
1941     TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)(target), (buffer), (size))
1942 #endif // !USE_MARSHALING_DEFINES
1943 #if !USE_MARSHALING_DEFINES
1944 UINT16
1945 TPMS_SIGNATURE_ECDSA_Marshal(
1946     TPMS_SIGNATURE_ECDSA* source, BYTE** buffer, INT32* size);
1947 #else // !USE_MARSHALING_DEFINES
1948 # define TPMS_SIGNATURE_ECDSA_Marshal(source, buffer, size) \
1949     TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)(source), (buffer), (size))
1950 #endif // !USE_MARSHALING_DEFINES
1951 #if !USE_MARSHALING_DEFINES
1952 TPM_RC
1953 TPMS_SIGNATURE_ECDSA_Unmarshal(
1954     TPMS_SIGNATURE_ECDSA* target, BYTE** buffer, INT32* size);
1955 #else // !USE_MARSHALING_DEFINES
1956 # define TPMS_SIGNATURE_ECDSA_Unmarshal(target, buffer, size) \
1957     TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)(target), (buffer), (size))
1958 #endif // !USE_MARSHALING_DEFINES
1959 #if !USE_MARSHALING_DEFINES
1960 UINT16
1961 TPMS_SIGNATURE_ECDSA_Marshal(
1962     TPMS_SIGNATURE_ECDSA* source, BYTE** buffer, INT32* size);
1963 #else // !USE_MARSHALING_DEFINES
1964 # define TPMS_SIGNATURE_ECDSA_Marshal(source, buffer, size) \
1965     TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)(source), (buffer), (size))
1966 #endif // !USE_MARSHALING_DEFINES
1967 #if !USE_MARSHALING_DEFINES
1968 TPM_RC
1969 TPMS_SIGNATURE_SM2_Unmarshal(TPMS_SIGNATURE_SM2* target, BYTE** buffer, INT32* size);
1970 #else // !USE_MARSHALING_DEFINES
1971 # define TPMS_SIGNATURE_SM2_Unmarshal(target, buffer, size) \
1972     TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)(target), (buffer), (size))
1973 #endif // !USE_MARSHALING_DEFINES
1974 #if !USE_MARSHALING_DEFINES

```

```

1975 UINT16
1976 TPMS_SIGNATURE_SM2_Marshal(TPMS_SIGNATURE_SM2* source, BYTE** buffer, INT32* size);
1977 #else // !USE_MARSHALING_DEFINES
1978 # define TPMS_SIGNATURE_SM2_Marshal(source, buffer, size) \
1979     TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)(source), (buffer), (size))
1980 #endif // !USE_MARSHALING_DEFINES
1981 #if !USE_MARSHALING_DEFINES
1982 TPM_RC
1983 TPMS_SIGNATURE_ECSCNORR_Unmarshal(
1984     TPMS_SIGNATURE_ECSCNORR* target, BYTE** buffer, INT32* size);
1985 #else // !USE_MARSHALING_DEFINES
1986 # define TPMS_SIGNATURE_ECSCNORR_Unmarshal(target, buffer, size) \
1987     TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)(target), (buffer), (size))
1988 #endif // !USE_MARSHALING_DEFINES
1989 #if !USE_MARSHALING_DEFINES
1990 UINT16
1991 TPMS_SIGNATURE_ECSCNORR_Marshal(
1992     TPMS_SIGNATURE_ECSCNORR* source, BYTE** buffer, INT32* size);
1993 #else // !USE_MARSHALING_DEFINES
1994 # define TPMS_SIGNATURE_ECSCNORR_Marshal(source, buffer, size) \
1995     TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)(source), (buffer), (size))
1996 #endif // !USE_MARSHALING_DEFINES
1997 #if !USE_MARSHALING_DEFINES
1998 TPM_RC
1999 TPMS_SIGNATURE_EDDSA_Unmarshal(
2000     TPMS_SIGNATURE_EDDSA* target, BYTE** buffer, INT32* size);
2001 #else // !USE_MARSHALING_DEFINES
2002 # define TPMS_SIGNATURE_EDDSA_Unmarshal(target, buffer, size) \
2003     TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)(target), (buffer), (size))
2004 #endif // !USE_MARSHALING_DEFINES
2005 #if !USE_MARSHALING_DEFINES
2006 UINT16
2007 TPMS_SIGNATURE_EDDSA_Marshal(
2008     TPMS_SIGNATURE_EDDSA* source, BYTE** buffer, INT32* size);
2009 #else // !USE_MARSHALING_DEFINES
2010 # define TPMS_SIGNATURE_EDDSA_Marshal(source, buffer, size) \
2011     TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)(source), (buffer), (size))
2012 #endif // !USE_MARSHALING_DEFINES
2013 #if !USE_MARSHALING_DEFINES
2014 TPM_RC
2015 TPMS_SIGNATURE_EDDSA_PH_Unmarshal(
2016     TPMS_SIGNATURE_EDDSA_PH* target, BYTE** buffer, INT32* size);
2017 #else // !USE_MARSHALING_DEFINES
2018 # define TPMS_SIGNATURE_EDDSA_PH_Unmarshal(target, buffer, size) \
2019     TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)(target), (buffer), (size))
2020 #endif // !USE_MARSHALING_DEFINES
2021 #if !USE_MARSHALING_DEFINES
2022 UINT16
2023 TPMS_SIGNATURE_EDDSA_PH_Marshal(
2024     TPMS_SIGNATURE_EDDSA_PH* source, BYTE** buffer, INT32* size);
2025 #else // !USE_MARSHALING_DEFINES
2026 # define TPMS_SIGNATURE_EDDSA_PH_Marshal(source, buffer, size) \
2027     TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)(source), (buffer), (size))
2028 #endif // !USE_MARSHALING_DEFINES
2029
2030 // Table "Definition of TPMU_SIGNATURE Union" (Part 2: Structures)
2031 TPM_RC
2032 TPMU_SIGNATURE_Unmarshal(
2033     TPMU_SIGNATURE* target, BYTE** buffer, INT32* size, UINT32 selector);
2034 UINT16
2035 TPMU_SIGNATURE_Marshal(
2036     TPMU_SIGNATURE* source, BYTE** buffer, INT32* size, UINT32 selector);
2037
2038 // Table "Definition of TPMT_SIGNATURE Structure" (Part 2: Structures)
2039 TPM_RC
2040 TPMT_SIGNATURE_Unmarshal(

```



```

2041     TPMT_SIGNATURE* target, BYTE** buffer, INT32* size, BOOL flag);
2042 UINT16
2043 TPMT_SIGNATURE_Marshal(TPMT_SIGNATURE* source, BYTE** buffer, INT32* size);
2044
2045 // Table "Definition of TPMU_ENCRYPTED_SECRET Union" (Part 2: Structures)
2046 TPM_RC
2047 TPMU_ENCRYPTED_SECRET_Unmarshal(
2048     TPMU_ENCRYPTED_SECRET* target, BYTE** buffer, INT32* size, UINT32 selector);
2049 UINT16
2050 TPMU_ENCRYPTED_SECRET_Marshal(
2051     TPMU_ENCRYPTED_SECRET* source, BYTE** buffer, INT32* size, UINT32 selector);
2052
2053 // Table "Definition of TPM2B_ENCRYPTED_SECRET Structure" (Part 2: Structures)
2054 TPM_RC
2055 TPM2B_ENCRYPTED_SECRET_Unmarshal(
2056     TPM2B_ENCRYPTED_SECRET* target, BYTE** buffer, INT32* size);
2057 UINT16
2058 TPM2B_ENCRYPTED_SECRET_Marshal(
2059     TPM2B_ENCRYPTED_SECRET* source, BYTE** buffer, INT32* size);
2060
2061 // Table "Definition of TPMI_ALG_PUBLIC Type" (Part 2: Structures)
2062 TPM_RC
2063 TPMI_ALG_PUBLIC_Unmarshal(TPMI_ALG_PUBLIC* target, BYTE** buffer, INT32* size);
2064 #if !USE_MARSHALING_DEFINES
2065 UINT16
2066 TPMI_ALG_PUBLIC_Marshal(TPMI_ALG_PUBLIC* source, BYTE** buffer, INT32* size);
2067 #else // !USE_MARSHALING_DEFINES
2068 # define TPMI_ALG_PUBLIC_Marshal(source, buffer, size) \
2069     TPM_ALG_ID_Marshal((TPM_ALG_ID*)(source), (buffer), (size))
2070 #endif // !USE_MARSHALING_DEFINES
2071
2072 // Table "Definition of TPMU_PUBLIC_ID Union" (Part 2: Structures)
2073 TPM_RC
2074 TPMU_PUBLIC_ID_Unmarshal(
2075     TPMU_PUBLIC_ID* target, BYTE** buffer, INT32* size, UINT32 selector);
2076 UINT16
2077 TPMU_PUBLIC_ID_Marshal(
2078     TPMU_PUBLIC_ID* source, BYTE** buffer, INT32* size, UINT32 selector);
2079
2080 // Table "Definition of TPMS_KEYEDHASH_PARMS Structure" (Part 2: Structures)
2081 TPM_RC
2082 TPMS_KEYEDHASH_PARMS_Unmarshal(
2083     TPMS_KEYEDHASH_PARMS* target, BYTE** buffer, INT32* size);
2084 UINT16
2085 TPMS_KEYEDHASH_PARMS_Marshal(
2086     TPMS_KEYEDHASH_PARMS* source, BYTE** buffer, INT32* size);
2087
2088 // Table "Definition of TPMS_RSA_PARMS Structure" (Part 2: Structures)
2089 TPM_RC
2090 TPMS_RSA_PARMS_Unmarshal(TPMS_RSA_PARMS* target, BYTE** buffer, INT32* size);
2091 UINT16
2092 TPMS_RSA_PARMS_Marshal(TPMS_RSA_PARMS* source, BYTE** buffer, INT32* size);
2093
2094 // Table "Definition of TPMS_ECC_PARMS Structure" (Part 2: Structures)
2095 TPM_RC
2096 TPMS_ECC_PARMS_Unmarshal(TPMS_ECC_PARMS* target, BYTE** buffer, INT32* size);
2097 UINT16
2098 TPMS_ECC_PARMS_Marshal(TPMS_ECC_PARMS* source, BYTE** buffer, INT32* size);
2099
2100 // Table "Definition of TPMU_PUBLIC_PARMS Union" (Part 2: Structures)
2101 TPM_RC
2102 TPMU_PUBLIC_PARMS_Unmarshal(
2103     TPMU_PUBLIC_PARMS* target, BYTE** buffer, INT32* size, UINT32 selector);
2104 UINT16
2105 TPMU_PUBLIC_PARMS_Marshal(
2106     TPMU_PUBLIC_PARMS* source, BYTE** buffer, INT32* size, UINT32 selector);

```

```
2107
2108 // Table "Definition of TPMT_PUBLIC_PARMS Structure" (Part 2: Structures)
2109 TPM_RC
2110 TPMT_PUBLIC_PARMS_Unmarshal(TPMT_PUBLIC_PARMS* target, BYTE** buffer, INT32* size);
2111 UINT16
2112 TPMT_PUBLIC_PARMS_Marshal(TPMT_PUBLIC_PARMS* source, BYTE** buffer, INT32* size);
2113
2114 // Table "Definition of TPMT_PUBLIC Structure" (Part 2: Structures)
2115 TPM_RC
2116 TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC* target, BYTE** buffer, INT32* size, BOOL flag);
2117 UINT16
2118 TPMT_PUBLIC_Marshal(TPMT_PUBLIC* source, BYTE** buffer, INT32* size);
2119
2120 // Table "Definition of TPM2B_PUBLIC Structure" (Part 2: Structures)
2121 TPM_RC
2122 TPM2B_PUBLIC_Unmarshal(TPM2B_PUBLIC* target, BYTE** buffer, INT32* size, BOOL flag);
2123 UINT16
2124 TPM2B_PUBLIC_Marshal(TPM2B_PUBLIC* source, BYTE** buffer, INT32* size);
2125
2126 // Table "Definition of TPM2B_TEMPLATE Structure" (Part 2: Structures)
2127 TPM_RC
2128 TPM2B_TEMPLATE_Unmarshal(TPM2B_TEMPLATE* target, BYTE** buffer, INT32* size);
2129 UINT16
2130 TPM2B_TEMPLATE_Marshal(TPM2B_TEMPLATE* source, BYTE** buffer, INT32* size);
2131
2132 // Table "Definition of TPM2B_PRIVATE_VENDOR_SPECIFIC Structure" (Part 2: Structures)
2133 TPM_RC
2134 TPM2B_PRIVATE_VENDOR_SPECIFIC_Unmarshal(
2135     TPM2B_PRIVATE_VENDOR_SPECIFIC* target, BYTE** buffer, INT32* size);
2136 UINT16
2137 TPM2B_PRIVATE_VENDOR_SPECIFIC_Marshal(
2138     TPM2B_PRIVATE_VENDOR_SPECIFIC* source, BYTE** buffer, INT32* size);
2139
2140 // Table "Definition of TPMU_SENSITIVE_COMPOSITE Union" (Part 2: Structures)
2141 TPM_RC
2142 TPMU_SENSITIVE_COMPOSITE_Unmarshal(
2143     TPMU_SENSITIVE_COMPOSITE* target, BYTE** buffer, INT32* size, UINT32 selector);
2144 UINT16
2145 TPMU_SENSITIVE_COMPOSITE_Marshal(
2146     TPMU_SENSITIVE_COMPOSITE* source, BYTE** buffer, INT32* size, UINT32 selector);
2147
2148 // Table "Definition of TPMT_SENSITIVE Structure" (Part 2: Structures)
2149 TPM_RC
2150 TPMT_SENSITIVE_Unmarshal(TPMT_SENSITIVE* target, BYTE** buffer, INT32* size);
2151 UINT16
2152 TPMT_SENSITIVE_Marshal(TPMT_SENSITIVE* source, BYTE** buffer, INT32* size);
2153
2154 // Table "Definition of TPM2B_SENSITIVE Structure" (Part 2: Structures)
2155 TPM_RC
2156 TPM2B_SENSITIVE_Unmarshal(TPM2B_SENSITIVE* target, BYTE** buffer, INT32* size);
2157 UINT16
2158 TPM2B_SENSITIVE_Marshal(TPM2B_SENSITIVE* source, BYTE** buffer, INT32* size);
2159
2160 // Table "Definition of TPM2B_PRIVATE Structure" (Part 2: Structures)
2161 TPM_RC
2162 TPM2B_PRIVATE_Unmarshal(TPM2B_PRIVATE* target, BYTE** buffer, INT32* size);
2163 UINT16
2164 TPM2B_PRIVATE_Marshal(TPM2B_PRIVATE* source, BYTE** buffer, INT32* size);
2165
2166 // Table "Definition of TPM2B_ID_OBJECT Structure" (Part 2: Structures)
2167 TPM_RC
2168 TPM2B_ID_OBJECT_Unmarshal(TPM2B_ID_OBJECT* target, BYTE** buffer, INT32* size);
2169 UINT16
2170 TPM2B_ID_OBJECT_Marshal(TPM2B_ID_OBJECT* source, BYTE** buffer, INT32* size);
2171
2172 // Table "Definition of TPMS_NV_PIN_COUNTER_PARAMETERS Structure" (Part 2: Structures)
```



```

2173 TPM_RC
2174 TPMS_NV_PIN_COUNTER_PARAMETERS_Unmarshal(
2175     TPMS_NV_PIN_COUNTER_PARAMETERS* target, BYTE** buffer, INT32* size);
2176 UINT16
2177 TPMS_NV_PIN_COUNTER_PARAMETERS_Marshal(
2178     TPMS_NV_PIN_COUNTER_PARAMETERS* source, BYTE** buffer, INT32* size);
2179
2180 // Table "Definition of TPMA_NV Bits" (Part 2: Structures)
2181 TPM_RC
2182 TPMA_NV_Unmarshal(TPMA_NV* target, BYTE** buffer, INT32* size);
2183 #if !USE_MARSHALING_DEFINES
2184 UINT16
2185 TPMA_NV_Marshal(TPMA_NV* source, BYTE** buffer, INT32* size);
2186 #else // !USE_MARSHALING_DEFINES
2187 # define TPMA_NV_Marshal(source, buffer, size) \
2188     UINT32_Marshal((UINT32*)(source), (buffer), (size))
2189 #endif // !USE_MARSHALING_DEFINES
2190
2191 // Table "Definition of TPMA_NV_EXP Bits" (Part 2: Structures)
2192 TPM_RC
2193 TPMA_NV_EXP_Unmarshal(TPMA_NV_EXP* target, BYTE** buffer, INT32* size);
2194 #if !USE_MARSHALING_DEFINES
2195 UINT16
2196 TPMA_NV_EXP_Marshal(TPMA_NV_EXP* source, BYTE** buffer, INT32* size);
2197 #else // !USE_MARSHALING_DEFINES
2198 # define TPMA_NV_EXP_Marshal(source, buffer, size) \
2199     UINT64_Marshal((UINT64*)(source), (buffer), (size))
2200 #endif // !USE_MARSHALING_DEFINES
2201
2202 // Table "Definition of TPMS_NV_PUBLIC Structure" (Part 2: Structures)
2203 TPM_RC
2204 TPMS_NV_PUBLIC_Unmarshal(TPMS_NV_PUBLIC* target, BYTE** buffer, INT32* size);
2205 UINT16
2206 TPMS_NV_PUBLIC_Marshal(TPMS_NV_PUBLIC* source, BYTE** buffer, INT32* size);
2207
2208 // Table "Definition of TPM2B_NV_PUBLIC Structure" (Part 2: Structures)
2209 TPM_RC
2210 TPM2B_NV_PUBLIC_Unmarshal(TPM2B_NV_PUBLIC* target, BYTE** buffer, INT32* size);
2211 UINT16
2212 TPM2B_NV_PUBLIC_Marshal(TPM2B_NV_PUBLIC* source, BYTE** buffer, INT32* size);
2213
2214 // Table "Definition of TPMS_NV_PUBLIC_EXP_ATTR Structure" (Part 2: Structures)
2215 TPM_RC
2216 TPMS_NV_PUBLIC_EXP_ATTR_Unmarshal(
2217     TPMS_NV_PUBLIC_EXP_ATTR* target, BYTE** buffer, INT32* size);
2218 UINT16
2219 TPMS_NV_PUBLIC_EXP_ATTR_Marshal(
2220     TPMS_NV_PUBLIC_EXP_ATTR* source, BYTE** buffer, INT32* size);
2221
2222 // Table "Definition of TPMU_NV_PUBLIC_2 Union" (Part 2: Structures)
2223 TPM_RC
2224 TPMU_NV_PUBLIC_2_Unmarshal(
2225     TPMU_NV_PUBLIC_2* target, BYTE** buffer, INT32* size, UINT32 selector);
2226 UINT16
2227 TPMU_NV_PUBLIC_2_Marshal(
2228     TPMU_NV_PUBLIC_2* source, BYTE** buffer, INT32* size, UINT32 selector);
2229
2230 // Table "Definition of TPMT_NV_PUBLIC_2 Structure" (Part 2: Structures)
2231 TPM_RC
2232 TPMT_NV_PUBLIC_2_Unmarshal(TPMT_NV_PUBLIC_2* target, BYTE** buffer, INT32* size);
2233 UINT16
2234 TPMT_NV_PUBLIC_2_Marshal(TPMT_NV_PUBLIC_2* source, BYTE** buffer, INT32* size);
2235
2236 // Table "Definition of TPM2B_NV_PUBLIC_2 Structure" (Part 2: Structures)
2237 TPM_RC
2238 TPM2B_NV_PUBLIC_2_Unmarshal(TPM2B_NV_PUBLIC_2* target, BYTE** buffer, INT32* size);

```

```

2239 UINT16
2240 TPM2B_NV_PUBLIC_2_Marshal(TPM2B_NV_PUBLIC_2* source, BYTE** buffer, INT32* size);
2241
2242 // Table "Definition of TPM2B_CONTEXT_SENSITIVE Structure" (Part 2: Structures)
2243 TPM_RC
2244 TPM2B_CONTEXT_SENSITIVE_Unmarshal(
2245     TPM2B_CONTEXT_SENSITIVE* target, BYTE** buffer, INT32* size);
2246 UINT16
2247 TPM2B_CONTEXT_SENSITIVE_Marshal(
2248     TPM2B_CONTEXT_SENSITIVE* source, BYTE** buffer, INT32* size);
2249
2250 // Table "Definition of TPMS_CONTEXT_DATA Structure" (Part 2: Structures)
2251 TPM_RC
2252 TPMS_CONTEXT_DATA_Unmarshal(TPMS_CONTEXT_DATA* target, BYTE** buffer, INT32* size);
2253 UINT16
2254 TPMS_CONTEXT_DATA_Marshal(TPMS_CONTEXT_DATA* source, BYTE** buffer, INT32* size);
2255
2256 // Table "Definition of TPM2B_CONTEXT_DATA Structure" (Part 2: Structures)
2257 TPM_RC
2258 TPM2B_CONTEXT_DATA_Unmarshal(TPM2B_CONTEXT_DATA* target, BYTE** buffer, INT32* size);
2259 UINT16
2260 TPM2B_CONTEXT_DATA_Marshal(TPM2B_CONTEXT_DATA* source, BYTE** buffer, INT32* size);
2261
2262 // Table "Definition of TPMS_CONTEXT Structure" (Part 2: Structures)
2263 TPM_RC
2264 TPMS_CONTEXT_Unmarshal(TPMS_CONTEXT* target, BYTE** buffer, INT32* size);
2265 UINT16
2266 TPMS_CONTEXT_Marshal(TPMS_CONTEXT* source, BYTE** buffer, INT32* size);
2267
2268 // Table "Definition of TPMS_CREATION_DATA Structure" (Part 2: Structures)
2269 UINT16
2270 TPMS_CREATION_DATA_Marshal(TPMS_CREATION_DATA* source, BYTE** buffer, INT32* size);
2271
2272 // Table "Definition of TPM2B_CREATION_DATA Structure" (Part 2: Structures)
2273 UINT16
2274 TPM2B_CREATION_DATA_Marshal(TPM2B_CREATION_DATA* source, BYTE** buffer, INT32* size);
2275
2276 // Table "Definition of TPM_AT Constants" (Part 2: Structures)
2277 TPM_RC
2278 TPM_AT_Unmarshal(TPM_AT* target, BYTE** buffer, INT32* size);
2279 #if !USE_MARSHALING_DEFINES
2280 UINT16
2281 TPM_AT_Marshal(TPM_AT* source, BYTE** buffer, INT32* size);
2282 #else // !USE_MARSHALING_DEFINES
2283 # define TPM_AT_Marshal(source, buffer, size) \
2284     UINT32_Marshal((UINT32*)(source), (buffer), (size))
2285 #endif // !USE_MARSHALING_DEFINES
2286
2287 // Table "Definition of TPM_AE Constants" (Part 2: Structures)
2288 #if !USE_MARSHALING_DEFINES
2289 UINT16
2290 TPM_AE_Marshal(TPM_AE* source, BYTE** buffer, INT32* size);
2291 #else // !USE_MARSHALING_DEFINES
2292 # define TPM_AE_Marshal(source, buffer, size) \
2293     UINT32_Marshal((UINT32*)(source), (buffer), (size))
2294 #endif // !USE_MARSHALING_DEFINES
2295
2296 // Table "Definition of TPMS_AC_OUTPUT Structure" (Part 2: Structures)
2297 UINT16
2298 TPMS_AC_OUTPUT_Marshal(TPMS_AC_OUTPUT* source, BYTE** buffer, INT32* size);
2299
2300 // Table "Definition of TPML_AC_CAPABILITIES Structure" (Part 2: Structures)
2301 UINT16
2302 TPML_AC_CAPABILITIES_Marshal(
2303     TPML_AC_CAPABILITIES* source, BYTE** buffer, INT32* size);
2304

```

```
2305 // For structures that unmarshals/marshals an array, the code calls an
2306 // un/marshaling function to process the array of the defined type.
2307 // This section contains the functions that perform that operation
2308 // Array Unmarshal/Marshal for BYTE
2309 TPM_RC
2310 BYTE_Array_Unmarshal(BYTE* target, BYTE** buffer, INT32* size, INT32 count);
2311 UINT16
2312 BYTE_Array_Marshal(BYTE* source, BYTE** buffer, INT32* size, INT32 count);
2313
2314 // Array Unmarshal and Marshal for TPM_ALG_ID
2315 TPM_RC
2316 TPM_ALG_ID_Array_Unmarshal(
2317     TPM_ALG_ID* target, BYTE** buffer, INT32* size, INT32 count);
2318 UINT16
2319 TPM_ALG_ID_Array_Marshal(TPM_ALG_ID* source, BYTE** buffer, INT32* size, INT32 count);
2320
2321 // Array Unmarshal and Marshal for TPM_CC
2322 TPM_RC
2323 TPM_CC_Array_Unmarshal(TPM_CC* target, BYTE** buffer, INT32* size, INT32 count);
2324 UINT16
2325 TPM_CC_Array_Marshal(TPM_CC* source, BYTE** buffer, INT32* size, INT32 count);
2326
2327 // Array Marshal for TPM_ECC_CURVE
2328 #if ALG_ECC
2329 UINT16
2330 TPM_ECC_CURVE_Array_Marshal(
2331     TPM_ECC_CURVE* source, BYTE** buffer, INT32* size, INT32 count);
2332 #else // ALG_ECC
2333 # define TPM_ECC_CURVE_Array_Marshal UNIMPLEMENTED_Marshal
2334 #endif // ALG_ECC
2335
2336 // Array Marshal for TPM_HANDLE
2337 UINT16
2338 TPM_HANDLE_Array_Marshal(TPM_HANDLE* source, BYTE** buffer, INT32* size, INT32 count);
2339
2340 // Array Unmarshal and Marshal for TPM2B_DIGEST
2341 TPM_RC
2342 TPM2B_DIGEST_Array_Unmarshal(
2343     TPM2B_DIGEST* target, BYTE** buffer, INT32* size, INT32 count);
2344 UINT16
2345 TPM2B_DIGEST_Array_Marshal(
2346     TPM2B_DIGEST* source, BYTE** buffer, INT32* size, INT32 count);
2347
2348 // Array Unmarshal and Marshal for TPM2B_VENDOR_PROPERTY
2349 TPM_RC
2350 TPM2B_VENDOR_PROPERTY_Array_Unmarshal(
2351     TPM2B_VENDOR_PROPERTY* target, BYTE** buffer, INT32* size, INT32 count);
2352 UINT16
2353 TPM2B_VENDOR_PROPERTY_Array_Marshal(
2354     TPM2B_VENDOR_PROPERTY* source, BYTE** buffer, INT32* size, INT32 count);
2355
2356 // Array Marshal for TPMA_CC
2357 UINT16
2358 TPMA_CC_Array_Marshal(TPMA_CC* source, BYTE** buffer, INT32* size, INT32 count);
2359
2360 // Array Marshal for TPMS_AC_OUTPUT
2361 UINT16
2362 TPMS_AC_OUTPUT_Array_Marshal(
2363     TPMS_AC_OUTPUT* source, BYTE** buffer, INT32* size, INT32 count);
2364
2365 // Array Marshal for TPMS_ACT_DATA
2366 UINT16
2367 TPMS_ACT_DATA_Array_Marshal(
2368     TPMS_ACT_DATA* source, BYTE** buffer, INT32* size, INT32 count);
2369
2370 // Array Marshal for TPMS_ALG_PROPERTY
```

```

2371 UINT16
2372 TPMS_ALG_PROPERTY_Array_Marshal(
2373     TPMS_ALG_PROPERTY* source, BYTE** buffer, INT32* size, INT32 count);
2374
2375 // Array Unmarshal and Marshal for TPMS_PCR_SELECTION
2376 TPM_RC
2377 TPMS_PCR_SELECTION_Array_Unmarshal(
2378     TPMS_PCR_SELECTION* target, BYTE** buffer, INT32* size, INT32 count);
2379
2380 UINT16
2381 TPMS_PCR_SELECTION_Array_Marshal(
2382     TPMS_PCR_SELECTION* source, BYTE** buffer, INT32* size, INT32 count);
2383
2384 // Array Marshal for TPMS_TAGGED_PCR_SELECT
2385 UINT16
2386 TPMS_TAGGED_PCR_SELECT_Array_Marshal(
2387     TPMS_TAGGED_PCR_SELECT* source, BYTE** buffer, INT32* size, INT32 count);
2388
2389 // Array Marshal for TPMS_TAGGED_POLICY
2390 UINT16
2391 TPMS_TAGGED_POLICY_Array_Marshal(
2392     TPMS_TAGGED_POLICY* source, BYTE** buffer, INT32* size, INT32 count);
2393
2394 // Array Marshal for TPMS_TAGGED_PROPERTY
2395 UINT16
2396 TPMS_TAGGED_PROPERTY_Array_Marshal(
2397     TPMS_TAGGED_PROPERTY* source, BYTE** buffer, INT32* size, INT32 count);
2398
2399 // Array Unmarshal and Marshal for TPMT_HA
2400 TPM_RC
2401 TPMT_HA_Array_Unmarshal(
2402     TPMT_HA* target, BYTE** buffer, INT32* size, BOOL flag, INT32 count);
2403
2404 TPMT_HA_Array_Marshal(TPMT_HA* source, BYTE** buffer, INT32* size, INT32 count);
2405 #endif // _MARSHAL_FP_H

```

6.133 /tpm/include/private/prototypes/MathOnByteBuffers_fp.h

```

1  /* (Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Mar 28, 2019 Time: 08:25:19PM
4   */
5
6  #ifndef MATH_ON_BYTE_BUFFERS_FP_H
7  #define MATH_ON_BYTE_BUFFERS_FP_H
8
9  /*** UnsignedCmpB
10 // This function compare two unsigned values. The values are byte-aligned,
11 // big-endian numbers (e.g, a hash).
12 // Return Type: int
13 //      1      if (a > b)
14 //      0      if (a = b)
15 //     -1      if (a < b)
16 LIB_EXPORT int UnsignedCompareB(UINT32 aSize, // IN: size of a
17                                const BYTE* a, // IN: a
18                                UINT32 bSize, // IN: size of b
19                                const BYTE* b // IN: b
20 );
21
22 /*** SignedCompareB()
23 // Compare two signed integers:
24 // Return Type: int
25 //      1      if a > b
26 //      0      if a = b
27 //     -1      if a < b
28 int SignedCompareB(const UINT32 aSize, // IN: size of a

```

```

29         const BYTE* a,          // IN: a buffer
30         const UINT32 bSize,      // IN: size of b
31         const BYTE* b           // IN: b buffer
32     );
33
34     /*** ModExpB
35     // This function is used to do modular exponentiation in support of RSA.
36     // The most typical uses are: 'c' = 'm'^'e' mod 'n' (RSA encrypt) and
37     // 'm' = 'c'^'d' mod 'n' (RSA decrypt). When doing decryption, the 'e' parameter
38     // of the function will contain the private exponent 'd' instead of the public
39     // exponent 'e'.
40     //
41     // If the results will not fit in the provided buffer,
42     // an error is returned (CRYPT_ERROR_UNDERFLOW). If the results is smaller
43     // than the buffer, the results is de-normalized.
44     //
45     // This version is intended for use with RSA and requires that 'm' be
46     // less than 'n'.
47     //
48     // Return Type: TPM_RC
49     //     TPM_RC_SIZE          number to exponentiate is larger than the modulus
50     //     TPM_RC_NO_RESULT     result will not fit into the provided buffer
51     //
52     TPM_RC
53     ModExpB(UINT32 cSize, // IN: the size of the output buffer. It will
54                // need to be the same size as the modulus
55                BYTE* c,  // OUT: the buffer to receive the results
56                // (c->size must be set to the maximum size
57                // for the returned value)
58                const UINT32 mSize,
59                const BYTE* m, // IN: number to exponentiate
60                const UINT32 eSize,
61                const BYTE* e, // IN: power
62                const UINT32 nSize,
63                const BYTE* n // IN: modulus
64    );
65
66     /*** DivideB()
67     // Divide an integer ('n') by an integer ('d') producing a quotient ('q') and
68     // a remainder ('r'). If 'q' or 'r' is not needed, then the pointer to them
69     // may be set to NULL.
70     //
71     // Return Type: TPM_RC
72     //     TPM_RC_NO_RESULT     'q' or 'r' is too small to receive the result
73     //
74     LIB_EXPORT TPM_RC DivideB(const TPM2B* n, // IN: numerator
75                               const TPM2B* d, // IN: denominator
76                               TPM2B* q, // OUT: quotient
77                               TPM2B* r // OUT: remainder
78    );
79
80     /*** AdjustNumberB()
81     // Remove/add leading zeros from a number in a TPM2B. Will try to make the number
82     // by adding or removing leading zeros. If the number is larger than the requested
83     // size, it will make the number as small as possible. Setting 'requestedSize' to
84     // zero is equivalent to requesting that the number be normalized.
85     UINT16
86     AdjustNumberB(TPM2B* num, UINT16 requestedSize);
87
88     /*** ShiftLeft()
89     // This function shifts a byte buffer (a TPM2B) one byte to the left. That is,
90     // the most significant bit of the most significant byte is lost.
91     TPM2B* ShiftLeft(TPM2B* value // IN/OUT: value to shift and shifted value out
92    );
93
94     #endif // _MATH_ON_BYTE_BUFFERS_FP_H_

```


6.134 /tpm/include/private/prototypes/Memory_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Apr 7, 2019 Time: 06:58:58PM
4  */
5
6  #ifndef _MEMORY_FP_H_
7  #define _MEMORY_FP_H_
8
9  /*** MemoryCopy()
10 // This is an alias for memmove. This is used in place of memcpy because
11 // some of the moves may overlap and rather than try to make sure that
12 // memmove is used when necessary, it is always used.
13 void MemoryCopy(void* dest, const void* src, int sSize);
14
15 /*** MemoryEqual()
16 // This function indicates if two buffers have the same values in the indicated
17 // number of bytes.
18 // Return Type: BOOL
19 //     TRUE(1)           all octets are the same
20 //     FALSE(0)          all octets are not the same
21 BOOL MemoryEqual(const void* buffer1, // IN: compare buffer1
22                 const void* buffer2, // IN: compare buffer2
23                 unsigned int size    // IN: size of bytes being compared
24 );
25
26 /*** MemoryCopy2B()
27 // This function copies a TPM2B. This can be used when the TPM2B types are
28 // the same or different.
29 //
30 // This function returns the number of octets in the data buffer of the TPM2B.
31 LIB_EXPORT INT16 MemoryCopy2B(TPM2B* dest, // OUT: receiving TPM2B
32                               const TPM2B* source, // IN: source TPM2B
33                               unsigned int dSize // IN: size of the receiving buffer
34 );
35
36 /*** MemoryConcat2B()
37 // This function will concatenate the buffer contents of a TPM2B to an
38 // the buffer contents of another TPM2B and adjust the size accordingly
39 // ('a' := ('a' | 'b')).
40 void MemoryConcat2B(
41     TPM2B* aInOut, // IN/OUT: destination 2B
42     TPM2B* bIn,    // IN: second 2B
43     unsigned int aMaxSize // IN: The size of aInOut.buffer (max values for
44                          // aInOut.size)
45 );
46
47 /*** MemoryEqual2B()
48 // This function will compare two TPM2B structures. To be equal, they
49 // need to be the same size and the buffer contexts need to be the same
50 // in all octets.
51 // Return Type: BOOL
52 //     TRUE(1)           size and buffer contents are the same
53 //     FALSE(0)          size or buffer contents are not the same
54 BOOL MemoryEqual2B(const TPM2B* aIn, // IN: compare value
55                   const TPM2B* bIn  // IN: compare value
56 );
57
58 /*** MemorySet()
59 // This function will set all the octets in the specified memory range to
60 // the specified octet value.
61 // Note: A previous version had an additional parameter (dSize) that was
62 // intended to make sure that the destination would not be overrun. The
63 // problem is that, in use, all that was happening was that the value of
64 // size was used for dSize so there was no benefit in the extra parameter.

```



```

65 void MemorySet(void* dest, int value, size_t size);
66
67 /*** MemoryPad2B()
68 // Function to pad a TPM2B with zeros and adjust the size.
69 void MemoryPad2B(TPM2B* b, UINT16 newSize);
70
71 /*** Uint16ToByteArray()
72 // Function to write an integer to a byte array
73 void Uint16ToByteArray(UINT16 i, BYTE* a);
74
75 /*** Uint32ToByteArray()
76 // Function to write an integer to a byte array
77 void Uint32ToByteArray(UINT32 i, BYTE* a);
78
79 /*** Uint64ToByteArray()
80 // Function to write an integer to a byte array
81 void Uint64ToByteArray(UINT64 i, BYTE* a);
82
83 /*** ByteArrayToUint8()
84 // Function to write a UINT8 to a byte array. This is included for completeness
85 // and to allow certain macro expansions
86 UINT8
87 ByteArrayToUint8(BYTE* a);
88
89 /*** ByteArrayToUint16()
90 // Function to write an integer to a byte array
91 UINT16
92 ByteArrayToUint16(BYTE* a);
93
94 /*** ByteArrayToUint32()
95 // Function to write an integer to a byte array
96 UINT32
97 ByteArrayToUint32(BYTE* a);
98
99 /*** ByteArrayToUint64()
100 // Function to write an integer to a byte array
101 UINT64
102 ByteArrayToUint64(BYTE* a);
103
104 #endif // _MEMORY_FP_H_

```

6.135 /tpm/include/private/prototypes/NvDynamic_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 7, 2020 Time: 07:15:54PM
4  */
5
6  #ifndef _NV_DYNAMIC_FP_H_
7  #define _NV_DYNAMIC_FP_H_
8
9  /*** NvWriteNvListEnd()
10 // Function to write the list terminator.
11 NV_REF
12 NvWriteNvListEnd(NV_REF end);
13
14 /*** NvUpdateIndexOrderlyData()
15 // This function is used to cause an update of the orderly data to the NV backing
16 // store.
17 void NvUpdateIndexOrderlyData(void);
18
19 /*** NvReadIndex()
20 // This function is used to read the NV Index NV_INDEX. This is used so that the
21 // index information can be compressed and only this function would be needed
22 // to decompress it. Mostly, compression would only be able to save the space

```

```

23 // needed by the policy.
24 void NvReadNvIndexInfo(NV_REF ref, // IN: points to NV where index is located
25 NV_INDEX* nvIndex // OUT: place to receive index data
26 );
27
28 /*** NvReadObject()
29 // This function is used to read a persistent object. This is used so that the
30 // object information can be compressed and only this function would be needed
31 // to uncompress it.
32 void NvReadObject(NV_REF ref, // IN: points to NV where index is located
33 OBJECT* object // OUT: place to receive the object data
34 );
35
36 /*** NvIndexIsDefined()
37 // See if an index is already defined
38 BOOL NvIndexIsDefined(TPM_HANDLE nvHandle // IN: Index to look for
39 );
40
41 /*** NvIsPlatformPersistentHandle()
42 // This function indicates if a handle references a persistent object in the
43 // range belonging to the platform.
44 // Return Type: BOOL
45 // TRUE(1) handle references a platform persistent object
46 // FALSE(0) and may reference an owner persistent object either
47 // handle does not reference platform persistent object
48 BOOL NvIsPlatformPersistentHandle(TPM_HANDLE handle // IN: handle
49 );
50
51 /*** NvIsOwnerPersistentHandle()
52 // This function indicates if a handle references a persistent object in the
53 // range belonging to the owner.
54 // Return Type: BOOL
55 // TRUE(1) handle is owner persistent handle
56 // FALSE(0) handle is not owner persistent handle and may not be
57 // a persistent handle at all
58 BOOL NvIsOwnerPersistentHandle(TPM_HANDLE handle // IN: handle
59 );
60
61 /*** NvIndexIsAccessible()
62 //
63 // This function validates that a handle references a defined NV Index and
64 // that the Index is currently accessible.
65 // Return Type: TPM_RC
66 // TPM_RC_HANDLE the handle points to an undefined NV Index
67 // If shEnable is CLEAR, this would include an index
68 // created using ownerAuth. If phEnableNV is CLEAR,
69 // this would include an index created using
70 // platformAuth
71 // TPM_RC_NV_READLOCKED Index is present but locked for reading and command
72 // does not write to the index
73 // TPM_RC_NV_WRITELOCKED Index is present but locked for writing and command
74 // writes to the index
75 TPM_RC
76 NvIndexIsAccessible(TPM_HANDLE handle // IN: handle
77 );
78
79 /*** NvGetEvictObject()
80 // This function is used to dereference an evict object handle and get a pointer
81 // to the object.
82 // Return Type: TPM_RC
83 // TPM_RC_HANDLE the handle does not point to an existing
84 // persistent object
85 TPM_RC
86 NvGetEvictObject(TPM_HANDLE handle, // IN: handle
87 OBJECT* object // OUT: object data
88 );

```

```

89
90  /*** NvIndexCacheInit()
91  // Function to initialize the Index cache
92  void NvIndexCacheInit(void);
93
94  /*** NvGetIndexData()
95  // This function is used to access the data in an NV Index. The data is returned
96  // as a byte sequence.
97  //
98  // This function requires that the NV Index be defined, and that the
99  // required data is within the data range. It also requires that TPMA_NV_WRITTEN
100 // of the Index is SET.
101 void NvGetIndexData(NV_INDEX* nvIndex, // IN: the in RAM index descriptor
102                    NV_REF locator, // IN: where the data is located
103                    UINT32 offset, // IN: offset of NV data
104                    UINT16 size, // IN: number of octets of NV data to read
105                    void* data // OUT: data buffer
106 );
107
108 /*** NvHashIndexData()
109 // This function adds Index data to a hash. It does this in parts to avoid large stack
110 // buffers.
111 void NvHashIndexData(HASH_STATE* hashState, // IN: Initialized hash state
112                     NV_INDEX* nvIndex, // IN: Index
113                     NV_REF locator, // IN: where the data is located
114                     UINT32 offset, // IN: starting offset
115                     UINT16 size // IN: amount to hash
116 );
117
118 /*** NvGetUINT64Data()
119 // Get data in integer format of a bit or counter NV Index.
120 //
121 // This function requires that the NV Index is defined and that the NV Index
122 // previously has been written.
123 UINT64
124 NvGetUINT64Data(NV_INDEX* nvIndex, // IN: the in RAM index descriptor
125                 NV_REF locator // IN: where index exists in NV
126 );
127
128 /*** NvWriteIndexAttributes()
129 // This function is used to write just the attributes of an index.
130 // Return type: TPM_RC
131 // TPM_RC_NV_RATE NV is rate limiting so retry
132 // TPM_RC_NV_UNAVAILABLE NV is not available
133 TPM_RC
134 NvWriteIndexAttributes(TPM_HANDLE handle,
135                        NV_REF locator, // IN: location of the index
136                        TPMA_NV attributes // IN: attributes to write
137 );
138
139 /*** NvWriteIndexAuth()
140 // This function is used to write the authValue of an index. It is used by
141 // TPM2_NV_ChangeAuth()
142 // Return type: TPM_RC
143 // TPM_RC_NV_RATE NV is rate limiting so retry
144 // TPM_RC_NV_UNAVAILABLE NV is not available
145 TPM_RC
146 NvWriteIndexAuth(NV_REF locator, // IN: location of the index
147                  TPM2B_AUTH* authValue // IN: the authValue to write
148 );
149
150 /*** NvGetIndexInfo()
151 // This function loads the nvIndex Info into the NV cache and returns a pointer
152 // to the NV_INDEX. If the returned value is zero, the index was not found.
153 // The 'locator' parameter, if not NULL, will be set to the offset in NV of the
154 // Index (the location of the handle of the Index).

```

```

155 //
156 // This function will set the index cache. If the index is orderly, the attributes
157 // from RAM are substituted for the attributes in the cached index
158 NV_INDEX* NvGetIndexInfo(TPM_HANDLE nvHandle, // IN: the index handle
159                          NV_REF* locator // OUT: location of the index
160 );
161
162 /*** NvWriteIndexData()
163 // This function is used to write NV index data. It is intended to be used to
164 // update the data associated with the default index.
165 //
166 // This function requires that the NV Index is defined, and the data is
167 // within the defined data range for the index.
168 //
169 // Index data is only written due to a command that modifies the data in a single
170 // index. There is no case where changes are made to multiple indexes data at the
171 // same time. Multiple attributes may be change but not multiple index data. This
172 // is important because we will normally be handling the index for which we have
173 // the cached pointer values.
174 // Return type: TPM_RC
175 // TPM_RC_NV_RATE NV is rate limiting so retry
176 // TPM_RC_NV_UNAVAILABLE NV is not available
177 TPM_RC
178 NvWriteIndexData(NV_INDEX* nvIndex, // IN: the description of the index
179                 UINT32 offset, // IN: offset of NV data
180                 UINT32 size, // IN: size of NV data
181                 void* data // IN: data buffer
182 );
183
184 /*** NvWriteUINT64Data()
185 // This function to write back a UINT64 value. The various UINT64 values (bits,
186 // counters, and PINs) are kept in canonical format but manipulate in native
187 // format. This takes a native format value converts it and saves it back as
188 // in canonical format.
189 //
190 // This function will return the value from NV or RAM depending on the type of the
191 // index (orderly or not)
192 //
193 TPM_RC
194 NvWriteUINT64Data(NV_INDEX* nvIndex, // IN: the description of the index
195                  UINT64 intValue // IN: the value to write
196 );
197
198 /*** NvGetNameByIndexHandle()
199 // This function is used to compute the Name of an NV Index referenced by handle.
200 //
201 // The 'name' buffer receives the bytes of the Name and the return value
202 // is the number of octets in the Name.
203 //
204 // This function requires that the NV Index is defined.
205 TPM2B_NAME* NvGetNameByIndexHandle(
206     TPMI_RH_NV_INDEX handle, // IN: handle of the index
207     TPM2B_NAME* name // OUT: name of the index
208 );
209
210 /*** NvDefineIndex()
211 // This function is used to assign NV memory to an NV Index.
212 //
213 // Return Type: TPM_RC
214 // TPM_RC_NV_SPACE insufficient NV space
215 TPM_RC
216 NvDefineIndex(TPMS_NV_PUBLIC* publicArea, // IN: A template for an area to create.
217              TPM2B_AUTH* authValue // IN: The initial authorization value
218 );
219
220 /*** NvAddEvictObject()

```

```

221 // This function is used to assign NV memory to a persistent object.
222 // Return Type: TPM_RC
223 //     TPM_RC_NV_HANDLE      the requested handle is already in use
224 //     TPM_RC_NV_SPACE      insufficient NV space
225 TPM_RC
226 NvAddEvictObject(TPMI_DH_OBJECT evictHandle, // IN: new evict handle
227                 OBJECT* object // IN: object to be added
228 );
229
230 /*** NvDeleteIndex()
231 // This function is used to delete an NV Index.
232 // Return Type: TPM_RC
233 //     TPM_RC_NV_UNAVAILABLE NV is not accessible
234 //     TPM_RC_NV_RATE       NV is rate limiting
235 TPM_RC
236 NvDeleteIndex(NV_INDEX* nvIndex, // IN: an in RAM index descriptor
237              NV_REF entityAddr // IN: location in NV
238 );
239
240 TPM_RC
241 NvDeleteEvict(TPM_HANDLE handle // IN: handle of entity to be deleted
242 );
243
244 /*** NvFlushHierarchy()
245 // This function will delete persistent objects belonging to the indicated hierarchy.
246 // If the storage hierarchy is selected, the function will also delete any
247 // NV Index defined using ownerAuth.
248 // Return Type: TPM_RC
249 //     TPM_RC_NV_RATE NV is unavailable because of rate limit
250 //     TPM_RC_NV_UNAVAILABLE NV is inaccessible
251 TPM_RC
252 NvFlushHierarchy(TPMI_RH_HIERARCHY hierarchy // IN: hierarchy to be flushed.
253 );
254
255 /*** NvSetGlobalLock()
256 // This function is used to SET the TPMA_NV_WRITELOCKED attribute for all
257 // NV indexes that have TPMA_NV_GLOBALLOCK SET. This function is use by
258 // TPM2_NV_GlobalWriteLock().
259 // Return Type: TPM_RC
260 //     TPM_RC_NV_RATE NV is unavailable because of rate limit
261 //     TPM_RC_NV_UNAVAILABLE NV is inaccessible
262 TPM_RC
263 NvSetGlobalLock(void);
264
265 /*** NvCapGetPersistent()
266 // This function is used to get a list of handles of the persistent objects,
267 // starting at 'handle'.
268 //
269 // 'Handle' must be in valid persistent object handle range, but does not
270 // have to reference an existing persistent object.
271 // Return Type: TPMI_YES_NO
272 //     YES if there are more handles available
273 //     NO all the available handles has been returned
274 TPMI_YES_NO
275 NvCapGetPersistent(TPMI_DH_OBJECT handle, // IN: start handle
276                  UINT32 count, // IN: maximum number of returned handles
277                  TPML_HANDLE* handleList // OUT: list of handle
278 );
279
280 /*** NvCapGetOnePersistent()
281 // This function returns whether a given persistent handle exists.
282 //
283 // 'Handle' must be in valid persistent object handle range.
284 BOOL NvCapGetOnePersistent(TPMI_DH_OBJECT handle // IN: handle
285 );
286

```



```

287  /*** NvCapGetIndex()
288  // This function returns a list of handles of NV indexes, starting from 'handle'.
289  // 'Handle' must be in the range of NV indexes, but does not have to reference
290  // an existing NV Index.
291  // Return Type: TPML_YES_NO
292  //     YES      if there are more handles to report
293  //     NO       all the available handles has been reported
294  TPML_YES_NO
295  NvCapGetIndex(TPML_DH_OBJECT handle,      // IN: start handle
296               UINT32 count,              // IN: max number of returned handles
297               TPML_HANDLE* handleList    // OUT: list of handle
298  );
299
300  /*** NvCapGetOneIndex()
301  // This function whether an NV index exists.
302  BOOL NvCapGetOneIndex(TPML_DH_OBJECT handle); // IN: start handle
303
304  /*** NvCapGetIndexNumber()
305  // This function returns the count of NV Indexes currently defined.
306  UINT32
307  NvCapGetIndexNumber(void);
308
309  /*** NvCapGetPersistentNumber()
310  // Function returns the count of persistent objects currently in NV memory.
311  UINT32
312  NvCapGetPersistentNumber(void);
313
314  /*** NvCapGetPersistentAvail()
315  // This function returns an estimate of the number of additional persistent
316  // objects that could be loaded into NV memory.
317  UINT32
318  NvCapGetPersistentAvail(void);
319
320  /*** NvCapGetCounterNumber()
321  // Get the number of defined NV Indexes that are counter indexes.
322  UINT32
323  NvCapGetCounterNumber(void);
324
325  /*** NvEntityStartup()
326  // This function is called at TPM_Startup(). If the startup completes
327  // a TPM Resume cycle, no action is taken. If the startup is a TPM Reset
328  // or a TPM Restart, then this function will:
329  // a) clear read/write lock;
330  // b) reset NV Index data that has TPMA_NV_CLEAR_STCLEAR SET; and
331  // c) set the lower bits in orderly counters to 1 for a non-orderly startup
332  //
333  // It is a prerequisite that NV be available for writing before this
334  // function is called.
335  BOOL NvEntityStartup(STARTUP_TYPE type // IN: start up type
336  );
337
338  /*** NvCapGetCounterAvail()
339  // This function returns an estimate of the number of additional counter type
340  // NV indexes that can be defined.
341  UINT32
342  NvCapGetCounterAvail(void);
343
344  /*** NvFindHandle()
345  // this function returns the offset in NV memory of the entity associated
346  // with the input handle. A value of zero indicates that handle does not
347  // exist reference an existing persistent object or defined NV Index.
348  NV_REF
349  NvFindHandle(TPML_HANDLE handle);
350
351  /*** NvReadMaxCount()
352  // This function returns the max NV counter value.

```



```

353 //
354 UINT64
355 NvReadMaxCount(void) ;
356
357 /*** NvUpdateMaxCount()
358 // This function updates the max counter value to NV memory. This is just staging
359 // for the actual write that will occur when the NV index memory is modified.
360 //
361 void NvUpdateMaxCount(UINT64 count) ;
362
363 /*** NvSetMaxCount()
364 // This function is used at NV initialization time to set the initial value of
365 // the maximum counter.
366 void NvSetMaxCount(UINT64 value) ;
367
368 /*** NvGetMaxCount()
369 // Function to get the NV max counter value from the end-of-list marker
370 UINT64
371 NvGetMaxCount(void) ;
372
373 #endif // _NV_DYNAMIC_FP_H_

```

6.136 /tpm/include/private/prototypes/NvReserved_fp.h

```

1  /* (Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Apr 2, 2019 Time: 04:23:27PM
4   */
5
6  #ifndef _NV_RESERVED_FP_H_
7  #define _NV_RESERVED_FP_H_
8
9  /*** NvCheckState()
10 // Function to check the NV state by accessing the platform-specific function
11 // to get the NV state. The result state is registered in s_NvIsAvailable
12 // that will be reported by NvIsAvailable.
13 //
14 // This function is called at the beginning of ExecuteCommand before any potential
15 // check of g_NvStatus.
16 void NvCheckState(void) ;
17
18 /*** NvCommit
19 // This is a wrapper for the platform function to commit pending NV writes.
20 BOOL NvCommit(void) ;
21
22 /*** NvPowerOn()
23 // This function is called at _TPM_Init to initialize the NV environment.
24 // Return Type: BOOL
25 // TRUE(1) all NV was initialized
26 // FALSE(0) the NV containing saved state had an error and
27 // TPM2_Startup(CLEAR) is required
28 BOOL NvPowerOn(void) ;
29
30 /*** NvManufacture()
31 // This function initializes the NV system at pre-install time.
32 //
33 // This function should only be called in a manufacturing environment or in a
34 // simulation.
35 //
36 // The layout of NV memory space is an implementation choice.
37 void NvManufacture(void) ;
38
39 /*** NvRead()
40 // This function is used to move reserved data from NV memory to RAM.
41 void NvRead(void* outBuffer, // OUT: buffer to receive data

```

```

42         UINT32 nvOffset,    // IN: offset in NV of value
43         UINT32 size        // IN: size of the value to read
44     );
45
46     /*** NvWrite()
47     // This function is used to post reserved data for writing to NV memory. Before
48     // the TPM completes the operation, the value will be written.
49     BOOL NvWrite(UINT32 nvOffset, // IN: location in NV to receive data
50                 UINT32 size,     // IN: size of the data to move
51                 void* inBuffer  // IN: location containing data to write
52     );
53
54     /*** NvUpdatePersistent()
55     // This function is used to update a value in the PERSISTENT_DATA structure and
56     // commits the value to NV.
57     void NvUpdatePersistent(
58         UINT32 offset, // IN: location in PERMANENT_DATA to be updated
59         UINT32 size,   // IN: size of the value
60         void* buffer   // IN: the new data
61     );
62
63     /*** NvClearPersistent()
64     // This function is used to clear a persistent data entry and commit it to NV
65     void NvClearPersistent(UINT32 offset, // IN: the offset in the PERMANENT_DATA
66                           // structure to be cleared (zeroed)
67                           UINT32 size    // IN: number of bytes to clear
68     );
69
70     /*** NvReadPersistent()
71     // This function reads persistent data to the RAM copy of the 'gp' structure.
72     void NvReadPersistent(void);
73
74     #endif // _NV_RESERVED_FP_H_

```

6.137 /tpm/include/private/prototypes/NV_Certify_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_NV_Certify // Command must be enabled
4
5  # ifndef TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_CERTIFY_FP_H_
6  #     define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_CERTIFY_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT    signHandle;
12     TPMI_RH_NV_AUTH   authHandle;
13     TPMI_RH_NV_INDEX  nvIndex;
14     TPM2B_DATA        qualifyingData;
15     TPMT_SIG_SCHEME   inScheme;
16     UINT16            size;
17     UINT16            offset;
18 } NV_Certify_In;
19
20 // Output structure definition
21 typedef struct
22 {
23     TPM2B_ATTEST    certifyInfo;
24     TPMT_SIGNATURE  signature;
25 } NV_Certify_Out;
26
27 // Response code modifiers
28 #     define RC_NV_Certify_signHandle    (TPM_RC_H + TPM_RC_1)
29 #     define RC_NV_Certify_authHandle    (TPM_RC_H + TPM_RC_2)

```

```

30 #   define RC_NV_Certify_nvIndex      (TPM_RC_H + TPM_RC_3)
31 #   define RC_NV_Certify_qualifyingData (TPM_RC_P + TPM_RC_1)
32 #   define RC_NV_Certify_inScheme     (TPM_RC_P + TPM_RC_2)
33 #   define RC_NV_Certify_size         (TPM_RC_P + TPM_RC_3)
34 #   define RC_NV_Certify_offset       (TPM_RC_P + TPM_RC_4)
35
36 // Function prototype
37 TPM_RC
38 TPM2_NV_Certify(NV_Certify_In* in, NV_Certify_Out* out);
39
40 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_CERTIFY_FP_H_
41 #endif // CC_NV_Certify

```

6.138 /tpm/include/private/prototypes/NV_ChangeAuth_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_ChangeAuth // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_CHANGEAUTH_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_CHANGEAUTH_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_NV_INDEX nvIndex;
12     TPM2B_AUTH        newAuth;
13 } NV_ChangeAuth_In;
14
15 // Response code modifiers
16 #   define RC_NV_ChangeAuth_nvIndex (TPM_RC_H + TPM_RC_1)
17 #   define RC_NV_ChangeAuth_newAuth (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_NV_ChangeAuth(NV_ChangeAuth_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_CHANGEAUTH_FP_H_
24 #endif // CC_NV_ChangeAuth

```

6.139 /tpm/include/private/prototypes/NV_DefineSpace2_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_DefineSpace2 // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_DEFINESPACE2_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_DEFINESPACE2_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PROVISION authHandle;
12     TPM2B_AUTH        auth;
13     TPM2B_NV_PUBLIC_2 publicInfo;
14 } NV_DefineSpace2_In;
15
16 // Response code modifiers
17 #   define RC_NV_DefineSpace2_authHandle (TPM_RC_H + TPM_RC_1)
18 #   define RC_NV_DefineSpace2_auth      (TPM_RC_P + TPM_RC_1)
19 #   define RC_NV_DefineSpace2_publicInfo (TPM_RC_P + TPM_RC_2)
20
21 // Function prototype
22 TPM_RC

```

```

23 TPM2_NV_DefineSpace2(NV_DefineSpace2_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_DEFINESPACE2_FP_H_
26 #endif // CC_NV_DefineSpace2

```

6.140 /tpm/include/private/prototypes/NV_DefineSpace_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_DefineSpace // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_DEFINESPACE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_DEFINESPACE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PROVISION authHandle;
12     TPM2B_AUTH auth;
13     TPM2B_NV_PUBLIC publicInfo;
14 } NV_DefineSpace_In;
15
16 // Response code modifiers
17 #   define RC_NV_DefineSpace_authHandle (TPM_RC_H + TPM_RC_1)
18 #   define RC_NV_DefineSpace_auth (TPM_RC_P + TPM_RC_1)
19 #   define RC_NV_DefineSpace_publicInfo (TPM_RC_P + TPM_RC_2)
20
21 // Function prototype
22 TPM_RC
23 TPM2_NV_DefineSpace(NV_DefineSpace_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_DEFINESPACE_FP_H_
26 #endif // CC_NV_DefineSpace

```

6.141 /tpm/include/private/prototypes/NV_Extend_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_Extend // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_EXTEND_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_EXTEND_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_NV_AUTH authHandle;
12     TPMI_RH_NV_INDEX nvIndex;
13     TPM2B_MAX_NV_BUFFER data;
14 } NV_Extend_In;
15
16 // Response code modifiers
17 #   define RC_NV_Extend_authHandle (TPM_RC_H + TPM_RC_1)
18 #   define RC_NV_Extend_nvIndex (TPM_RC_H + TPM_RC_2)
19 #   define RC_NV_Extend_data (TPM_RC_P + TPM_RC_1)
20
21 // Function prototype
22 TPM_RC
23 TPM2_NV_Extend(NV_Extend_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_EXTEND_FP_H_
26 #endif // CC_NV_Extend

```

6.142 /tpm/include/private/prototypes/NV_GlobalWriteLock_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_NV_GlobalWriteLock  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_GLOBALWRITELOCK_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_GLOBALWRITELOCK_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_PROVISION authHandle;
12 } NV_GlobalWriteLock_In;
13
14 // Response code modifiers
15 #  define RC_NV_GlobalWriteLock_authHandle (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_NV_GlobalWriteLock(NV_GlobalWriteLock_In* in);
20
21 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_GLOBALWRITELOCK_FP_H_
22 #endif  // CC_NV_GlobalWriteLock

```

6.143 /tpm/include/private/prototypes/NV_Increment_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_NV_Increment  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_INCREMENT_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_INCREMENT_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_NV_AUTH authHandle;
12     TPMI_RH_NV_INDEX nvIndex;
13 } NV_Increment_In;
14
15 // Response code modifiers
16 #  define RC_NV_Increment_authHandle (TPM_RC_H + TPM_RC_1)
17 #  define RC_NV_Increment_nvIndex (TPM_RC_H + TPM_RC_2)
18
19 // Function prototype
20 TPM_RC
21 TPM2_NV_Increment(NV_Increment_In* in);
22
23 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_INCREMENT_FP_H_
24 #endif  // CC_NV_Increment

```

6.144 /tpm/include/private/prototypes/NV_ReadLock_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_NV_ReadLock  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READLOCK_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READLOCK_FP_H_
7
8  // Input structure definition
9  typedef struct

```

```

10 {
11     TPMI_RH_NV_AUTH authHandle;
12     TPMI_RH_NV_INDEX nvIndex;
13 } NV_ReadLock_In;
14
15 // Response code modifiers
16 # define RC_NV_ReadLock_authHandle (TPM_RC_H + TPM_RC_1)
17 # define RC_NV_ReadLock_nvIndex (TPM_RC_H + TPM_RC_2)
18
19 // Function prototype
20 TPM_RC
21 TPM2_NV_ReadLock(NV_ReadLock_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READLOCK_FP_H_
24 #endif // CC_NV_ReadLock

```

6.145 /tpm/include/private/prototypes/NV_ReadPublic2_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_ReadPublic2 // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READPUBLIC2_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READPUBLIC2_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_NV_INDEX nvIndex;
12 } NV_ReadPublic2_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPM2B_NV_PUBLIC_2 nvPublic;
18     TPM2B_NAME nvName;
19 } NV_ReadPublic2_Out;
20
21 // Response code modifiers
22 # define RC_NV_ReadPublic2_nvIndex (TPM_RC_H + TPM_RC_1)
23
24 // Function prototype
25 TPM_RC
26 TPM2_NV_ReadPublic2(NV_ReadPublic2_In* in, NV_ReadPublic2_Out* out);
27
28 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READPUBLIC2_FP_H_
29 #endif // CC_NV_ReadPublic2

```

6.146 /tpm/include/private/prototypes/NV_ReadPublic_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_ReadPublic // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READPUBLIC_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READPUBLIC_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_NV_INDEX nvIndex;
12 } NV_ReadPublic_In;
13
14 // Output structure definition

```



```

15 typedef struct
16 {
17     TPM2B_NV_PUBLIC nvPublic;
18     TPM2B_NAME      nvName;
19 } NV_ReadPublic_Out;
20
21 // Response code modifiers
22 # define RC_NV_ReadPublic_nvIndex (TPM_RC_H + TPM_RC_1)
23
24 // Function prototype
25 TPM_RC
26 TPM2_NV_ReadPublic(NV_ReadPublic_In* in, NV_ReadPublic_Out* out);
27
28 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READPUBLIC_FP_H_
29 #endif // CC_NV_ReadPublic

```

6.147 /tpm/include/private/prototypes/NV_Read_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_Read // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READ_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READ_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_NV_AUTH authHandle;
12     TPMI_RH_NV_INDEX nvIndex;
13     UINT16           size;
14     UINT16           offset;
15 } NV_Read_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_MAX_NV_BUFFER data;
21 } NV_Read_Out;
22
23 // Response code modifiers
24 # define RC_NV_Read_authHandle (TPM_RC_H + TPM_RC_1)
25 # define RC_NV_Read_nvIndex (TPM_RC_H + TPM_RC_2)
26 # define RC_NV_Read_size (TPM_RC_P + TPM_RC_1)
27 # define RC_NV_Read_offset (TPM_RC_P + TPM_RC_2)
28
29 // Function prototype
30 TPM_RC
31 TPM2_NV_Read(NV_Read_In* in, NV_Read_Out* out);
32
33 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_READ_FP_H_
34 #endif // CC_NV_Read

```

6.148 /tpm/include/private/prototypes/NV_SetBits_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_SetBits // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_SETBITS_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_SETBITS_FP_H_
7
8 // Input structure definition
9 typedef struct

```

```

10 {
11     TPMI_RH_NV_AUTH  authHandle;
12     TPMI_RH_NV_INDEX nvIndex;
13     UINT64           bits;
14 } NV_SetBits_In;
15
16 // Response code modifiers
17 # define RC_NV_SetBits_authHandle (TPM_RC_H + TPM_RC_1)
18 # define RC_NV_SetBits_nvIndex    (TPM_RC_H + TPM_RC_2)
19 # define RC_NV_SetBits_bits      (TPM_RC_P + TPM_RC_1)
20
21 // Function prototype
22 TPM_RC
23 TPM2_NV_SetBits(NV_SetBits_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_SETBITS_FP_H_
26 #endif // CC_NV_SetBits

```

6.149 /tpm/include/private/prototypes/NV_spt_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 28, 2019 Time: 08:25:18PM
4  */
5
6  #ifndef _NV_SPT_FP_H_
7  #define _NV_SPT_FP_H_
8
9  /*** NvReadAccessChecks()
10 // Common routine for validating a read
11 // Used by TPM2_NV_Read, TPM2_NV_ReadLock and TPM2_PolicyNV
12 // Return Type: TPM_RC
13 //     TPM_RC_NV_AUTHORIZATION    authHandle is not allowed to authorize read
14 //                                of the index
15 //     TPM_RC_NV_LOCKED          Read locked
16 //     TPM_RC_NV_UNINITIALIZED   Try to read an uninitialized index
17 //
18 TPM_RC
19 NvReadAccessChecks(TPM_HANDLE authHandle, // IN: the handle that provided the
20 //                                // authorization
21                    TPM_HANDLE nvHandle,  // IN: the handle of the NV index to be read
22                    TPMA_NV attributes    // IN: the attributes of 'nvHandle'
23 );
24
25 /*** NvWriteAccessChecks()
26 // Common routine for validating a write
27 // Used by TPM2_NV_Write, TPM2_NV_Increment, TPM2_SetBits, and TPM2_NV_WriteLock
28 // Return Type: TPM_RC
29 //     TPM_RC_NV_AUTHORIZATION    Authorization fails
30 //     TPM_RC_NV_LOCKED          Write locked
31 //
32 TPM_RC
33 NvWriteAccessChecks(
34     TPM_HANDLE authHandle, // IN: the handle that provided the
35     //                                // authorization
36     TPM_HANDLE nvHandle,   // IN: the handle of the NV index to be written
37     TPMA_NV attributes     // IN: the attributes of 'nvHandle'
38 );
39
40 /*** NvClearOrderly()
41 // This function is used to cause gp.orderlyState to be cleared to the
42 // non-orderly state.
43 TPM_RC
44 NvClearOrderly(void);
45

```

```

46  /*** NvIsPinPassIndex()
47  // Function to check to see if an NV index is a PIN Pass Index
48  // Return Type: BOOL
49  //     TRUE(1)         is pin pass
50  //     FALSE(0)        is not pin pass
51  BOOL NvIsPinPassIndex(TPM_HANDLE index // IN: Handle to check
52  );
53
54  /*** NvGetIndexName()
55  // This function computes the Name of an index
56  // The 'name' buffer receives the bytes of the Name and the return value
57  // is the number of octets in the Name.
58  //
59  // This function requires that the NV Index is defined.
60  TPM2B_NAME* NvGetIndexName(
61      NV_INDEX* nvIndex, // IN: the index over which the name is to be
62                          // computed
63      TPM2B_NAME* name    // OUT: name of the index
64  );
65
66  /*** NvPublic2FromNvPublic()
67  // This function converts a legacy-form NV public (TPMS_NV_PUBLIC) into the
68  // generalized TPMT_NV_PUBLIC_2 tagged-union representation.
69  TPM_RC NvPublic2FromNvPublic(
70      TPMS_NV_PUBLIC* nvPublic, // IN: the source S-form NV public area
71      TPMT_NV_PUBLIC_2* nvPublic2 // OUT: the T-form NV public area to populate
72  );
73
74  /*** NvPublicFromNvPublic2()
75  // This function converts a tagged-union NV public (TPMT_NV_PUBLIC_2) into the
76  // legacy TPMS_NV_PUBLIC representation. This is a lossy conversion: any
77  // bits in the extended area of the attributes are lost, and the Name cannot be
78  // computed based on it.
79  TPM_RC NvPublicFromNvPublic2(
80      TPMT_NV_PUBLIC_2* nvPublic2, // IN: the source T-form NV public area
81      TPMS_NV_PUBLIC* nvPublic    // OUT: the S-form NV public area to populate
82  );
83
84  /*** NvDefineSpace()
85  // This function combines the common functionality of TPM2_NV_DefineSpace and
86  // TPM2_NV_DefineSpace2.
87  TPM_RC NvDefineSpace(TPMI_RH_PROVISION authHandle,
88                      TPM2B_AUTH* auth,
89                      TPMS_NV_PUBLIC* publicInfo,
90                      TPM_RC blameAuthHandle,
91                      TPM_RC blameAuth,
92                      TPM_RC blamePublic);
93
94  #endif // _NV_SPT_FP_H_

```

6.150 /tpm/include/private/prototypes/NV_UndefineSpaceSpecial_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_NV_UndefineSpaceSpecial // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_UNDEFINESPACESPECIAL_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_UNDEFINESPACESPECIAL_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_NV_DEFINED_INDEX nvIndex;
12     TPMI_RH_PLATFORM platform;
13 } NV_UndefineSpaceSpecial_In;

```

```

14
15 // Response code modifiers
16 # define RC_NV_UndefineSpaceSpecial_nvIndex (TPM_RC_H + TPM_RC_1)
17 # define RC_NV_UndefineSpaceSpecial_platform (TPM_RC_H + TPM_RC_2)
18
19 // Function prototype
20 TPM_RC
21 TPM2_NV_UndefineSpaceSpecial(NV_UndefineSpaceSpecial_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_UNDEFINESPACESPECIAL_FP_H_
24 #endif // CC_NV_UndefineSpaceSpecial

```

6.151 /tpm/include/private/prototypes/NV_UndefineSpace_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_UndefineSpace // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_UNDEFINESPACE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_UNDEFINESPACE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PROVISION authHandle;
12     TPMI_RH_NV_DEFINED_INDEX nvIndex;
13 } NV_UndefineSpace_In;
14
15 // Response code modifiers
16 # define RC_NV_UndefineSpace_authHandle (TPM_RC_H + TPM_RC_1)
17 # define RC_NV_UndefineSpace_nvIndex (TPM_RC_H + TPM_RC_2)
18
19 // Function prototype
20 TPM_RC
21 TPM2_NV_UndefineSpace(NV_UndefineSpace_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_UNDEFINESPACE_FP_H_
24 #endif // CC_NV_UndefineSpace

```

6.152 /tpm/include/private/prototypes/NV_WriteLock_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_NV_WriteLock // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_WRITELOCK_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_WRITELOCK_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_NV_AUTH authHandle;
12     TPMI_RH_NV_INDEX nvIndex;
13 } NV_WriteLock_In;
14
15 // Response code modifiers
16 # define RC_NV_WriteLock_authHandle (TPM_RC_H + TPM_RC_1)
17 # define RC_NV_WriteLock_nvIndex (TPM_RC_H + TPM_RC_2)
18
19 // Function prototype
20 TPM_RC
21 TPM2_NV_WriteLock(NV_WriteLock_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_WRITELOCK_FP_H_

```

```
24 #endif    // CC_NV_WriteLock
```

6.153 /tpm/include/private/prototypes/NV_Write_fp.h

```
1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_NV_Write    // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_WRITE_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_WRITE_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_NV_AUTH    authHandle;
12     TPMI_RH_NV_INDEX    nvIndex;
13     TPM2B_MAX_NV_BUFFER data;
14     UINT16            offset;
15 } NV_Write_In;
16
17 // Response code modifiers
18 #  define RC_NV_Write_authHandle (TPM_RC_H + TPM_RC_1)
19 #  define RC_NV_Write_nvIndex    (TPM_RC_H + TPM_RC_2)
20 #  define RC_NV_Write_data      (TPM_RC_P + TPM_RC_1)
21 #  define RC_NV_Write_offset    (TPM_RC_P + TPM_RC_2)
22
23 // Function prototype
24 TPM_RC
25 TPM2_NV_Write(NV_Write_In* in);
26
27 #  endif    // _TPM_INCLUDE_PRIVATE_PROTOTYPES_NV_WRITE_FP_H_
28 #endif    // CC_NV_Write
```

6.154 /tpm/include/private/prototypes/ObjectChangeAuth_fp.h

```
1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_ObjectChangeAuth    // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_OBJECTCHANGEAUTH_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_OBJECTCHANGEAUTH_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT objectHandle;
12     TPMI_DH_OBJECT parentHandle;
13     TPM2B_AUTH    newAuth;
14 } ObjectChangeAuth_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM2B_PRIVATE outPrivate;
20 } ObjectChangeAuth_Out;
21
22 // Response code modifiers
23 #  define RC_ObjectChangeAuth_objectHandle (TPM_RC_H + TPM_RC_1)
24 #  define RC_ObjectChangeAuth_parentHandle (TPM_RC_H + TPM_RC_2)
25 #  define RC_ObjectChangeAuth_newAuth     (TPM_RC_P + TPM_RC_1)
26
27 // Function prototype
28 TPM_RC
29 TPM2_ObjectChangeAuth(ObjectChangeAuth_In* in, ObjectChangeAuth_Out* out);
```

```

30
31 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_OBJECTCHANGEAUTH_FP_H_
32 #endif // CC_ObjectChangeAuth

```

6.155 /tpm/include/private/prototypes/Object_fp.h

```

1  /* (Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Mar 4, 2020 Time: 02:36:44PM
4   */
5
6 #ifndef _OBJECT_FP_H_
7 #define _OBJECT_FP_H_
8
9 /*** ObjectFlush()
10 // This function marks an object slot as available.
11 // Since there is no checking of the input parameters, it should be used
12 // judiciously.
13 // Note: This could be converted to a macro.
14 void ObjectFlush(OBJECT* object);
15
16 /*** ObjectSetInUse()
17 // This access function sets the occupied attribute of an object slot.
18 void ObjectSetInUse(OBJECT* object);
19
20 /*** ObjectStartup()
21 // This function is called at TPM2_Startup() to initialize the object subsystem.
22 BOOL ObjectStartup(void);
23
24 /*** ObjectCleanupEvict()
25 //
26 // In this implementation, a persistent object is moved from NV into an object slot
27 // for processing. It is flushed after command execution. This function is called
28 // from ExecuteCommand().
29 void ObjectCleanupEvict(void);
30
31 /*** IsObjectPresent()
32 // This function checks to see if a transient handle references a loaded
33 // object. This routine should not be called if the handle is not a
34 // transient handle. The function validates that the handle is in the
35 // implementation-dependent allowed in range for loaded transient objects.
36 // Return Type: BOOL
37 // TRUE(1) handle references a loaded object
38 // FALSE(0) handle is not an object handle, or it does not
39 // reference to a loaded object
40 BOOL IsObjectPresent(TPMI_DH_OBJECT handle // IN: handle to be checked
41 );
42
43 /*** ObjectIsSequence()
44 // This function is used to check if the object is a sequence object. This function
45 // should not be called if the handle does not reference a loaded object.
46 // Return Type: BOOL
47 // TRUE(1) object is an HMAC, hash, or event sequence object
48 // FALSE(0) object is not an HMAC, hash, or event sequence object
49 BOOL ObjectIsSequence(OBJECT* object // IN: handle to be checked
50 );
51
52 /*** HandleToObject()
53 // This function is used to find the object structure associated with a handle.
54 //
55 // This function requires that 'handle' references a loaded object or a permanent
56 // handle.
57 OBJECT* HandleToObject(TPMI_DH_OBJECT handle // IN: handle of the object
58 );
59

```



```

60  /*** GetQualifiedName()
61  // This function returns the Qualified Name of the object. In this implementation,
62  // the Qualified Name is computed when the object is loaded and is saved in the
63  // internal representation of the object. The alternative would be to retain the
64  // Name of the parent and compute the QN when needed. This would take the same
65  // amount of space so it is not recommended that the alternate be used.
66  //
67  // This function requires that 'handle' references a loaded object.
68  void GetQualifiedName(TPMI_DH_OBJECT handle,      // IN: handle of the object
69                      TPM2B_NAME* qualifiedName    // OUT: qualified name of the object
70  );
71
72  /*** GetHierarchy()
73  // This function returns the handle of the hierarchy to which a handle belongs.
74  //
75  // This function requires that 'handle' references a loaded object.
76  TPMI_RH_HIERARCHY
77  GetHierarchy(TPMI_DH_OBJECT handle // IN :object handle
78  );
79
80  /*** FindEmptyObjectSlot()
81  // This function finds an open object slot, if any. It will clear the attributes
82  // but will not set the occupied attribute. This is so that a slot may be used
83  // and discarded if everything does not go as planned.
84  // Return Type: OBJECT *
85  //     NULL      no open slot found
86  //     != NULL   pointer to available slot
87  OBJECT* FindEmptyObjectSlot(TPMI_DH_OBJECT* handle // OUT: (optional)
88  );
89
90  /*** ObjectAllocateSlot()
91  // This function is used to allocate a slot in internal object array.
92  OBJECT* ObjectAllocateSlot(TPMI_DH_OBJECT* handle // OUT: handle of allocated object
93  );
94
95  /*** ObjectSetLoadedAttributes()
96  // This function sets the internal attributes for a loaded object. It is called to
97  // finalize the OBJECT attributes (not the TPMA_OBJECT attributes) for a loaded
98  // object.
99  void ObjectSetLoadedAttributes(OBJECT* object, // IN: object attributes to finalize
100                              TPM_HANDLE parentHandle // IN: the parent handle
101  );
102
103  /*** ObjectLoad()
104  // Common function to load a non-primary object (i.e., either an Ordinary Object,
105  // or an External Object). A loaded object has its public area validated
106  // (unless its 'nameAlg' is TPM_ALG_NULL). If a sensitive part is loaded, it is
107  // verified to be correct and if both public and sensitive parts are loaded, then
108  // the cryptographic binding between the objects is validated. This function does
109  // not cause the allocated slot to be marked as in use.
110  TPM_RC
111  ObjectLoad(OBJECT* object,      // IN: pointer to object slot
112            // object
113            OBJECT* parent,      // IN: (optional) the parent object
114            TPMT_PUBLIC* publicArea, // IN: public area to be installed in the object
115            TPMT_SENSITIVE* sensitive, // IN: (optional) sensitive area to be
116            // installed in the object
117            TPM_RC blamePublic,   // IN: parameter number to associate with the
118            // publicArea errors
119            TPM_RC blameSensitive, // IN: parameter number to associate with the
120            // sensitive area errors
121            TPM2B_NAME* name      // IN: (optional)
122  );
123
124  #if CC_HMAC_Start || CC_MAC_Start
125  /*** ObjectCreateHMACSequence()

```

```

126 // This function creates an internal HMAC sequence object.
127 // Return Type: TPM_RC
128 //     TPM_RC_OBJECT_MEMORY      if there is no free slot for an object
129 TPM_RC
130 ObjectCreateHMACSequence(
131     TPMI_ALG_HASH    hashAlg,    // IN: hash algorithm
132     OBJECT*          keyObject,   // IN: the object containing the HMAC key
133     TPM2B_AUTH*       auth,       // IN: authValue
134     TPMI_DH_OBJECT*   newHandle   // OUT: HMAC sequence object handle
135 );
136 #endif
137
138 /*** ObjectCreateHashSequence()
139 // This function creates a hash sequence object.
140 // Return Type: TPM_RC
141 //     TPM_RC_OBJECT_MEMORY      if there is no free slot for an object
142 TPM_RC
143 ObjectCreateHashSequence(TPMI_ALG_HASH    hashAlg,    // IN: hash algorithm
144     TPM2B_AUTH*       auth,       // IN: authValue
145     TPMI_DH_OBJECT*   newHandle   // OUT: sequence object handle
146 );
147
148 /*** ObjectCreateEventSequence()
149 // This function creates an event sequence object.
150 // Return Type: TPM_RC
151 //     TPM_RC_OBJECT_MEMORY      if there is no free slot for an object
152 TPM_RC
153 ObjectCreateEventSequence(TPM2B_AUTH*       auth,       // IN: authValue
154     TPMI_DH_OBJECT*   newHandle   // OUT: sequence object handle
155 );
156
157 /*** ObjectTerminateEvent()
158 // This function is called to close out the event sequence and clean up the hash
159 // context states.
160 void ObjectTerminateEvent(void);
161
162 /*** ObjectContextLoad()
163 // This function loads an object from a saved object context.
164 // Return Type: OBJECT *
165 //     NULL      if there is no free slot for an object
166 //     != NULL   points to the loaded object
167 OBJECT* ObjectContextLoad(
168     ANY_OBJECT_BUFFER* object, // IN: pointer to object structure in saved
169     // context
170     TPMI_DH_OBJECT* handle     // OUT: object handle
171 );
172
173 /*** FlushObject()
174 // This function frees an object slot.
175 //
176 // This function requires that the object is loaded.
177 void FlushObject(TPMI_DH_OBJECT handle // IN: handle to be freed
178 );
179
180 /*** ObjectFlushHierarchy()
181 // This function is called to flush all the loaded transient objects associated
182 // with a hierarchy when the hierarchy is disabled.
183 void ObjectFlushHierarchy(TPMI_RH_HIERARCHY hierarchy // IN: hierarchy to be flush
184 );
185
186 /*** ObjectLoadEvict()
187 // This function loads a persistent object into a transient object slot.
188 //
189 // This function requires that 'handle' is associated with a persistent object.
190 // Return Type: TPM_RC
191 //     TPM_RC_HANDLE      the persistent object does not exist

```

```

192 // or the associated hierarchy is disabled.
193 // TPM_RC_OBJECT_MEMORY no object slot
194 TPM_RC
195 ObjectLoadEvict(TPM_HANDLE* handle, // IN:OUT: evict object handle. If success, it
196 // will be replace by the loaded object handle
197 COMMAND_INDEX commandIndex // IN: the command being processed
198 );
199
200 /*** ObjectComputeName()
201 // This does the name computation from a public area (can be marshaled or not).
202 TPM2B_NAME* ObjectComputeName(UINT32 size, // IN: the size of the area to digest
203 BYTE* publicArea, // IN: the public area to digest
204 TPM_ALG_ID nameAlg, // IN: the hash algorithm to use
205 TPM2B_NAME* name // OUT: Computed name
206 );
207
208 /*** PublicMarshalAndComputeName()
209 // This function computes the Name of an object from its public area.
210 TPM2B_NAME* PublicMarshalAndComputeName(
211 TPMT_PUBLIC* publicArea, // IN: public area of an object
212 TPM2B_NAME* name // OUT: name of the object
213 );
214
215 /*** ComputeQualifiedName()
216 // This function computes the qualified name of an object.
217 void ComputeQualifiedName(
218 TPM_HANDLE parentHandle, // IN: parent's handle
219 TPM_ALG_ID nameAlg, // IN: name hash
220 TPM2B_NAME* name, // IN: name of the object
221 TPM2B_NAME* qualifiedName // OUT: qualified name of the object
222 );
223
224 /*** ObjectIsStorage()
225 // This function determines if an object has the attributes associated
226 // with a parent. A parent is an asymmetric or symmetric block cipher key
227 // that has its 'restricted' and 'decrypt' attributes SET, and 'sign' CLEAR.
228 // Return Type: BOOL
229 // TRUE(1) object is a storage key
230 // FALSE(0) object is not a storage key
231 BOOL ObjectIsStorage(TPMI_DH_OBJECT handle // IN: object handle
232 );
233
234 /*** ObjectCapGetLoaded()
235 // This function returns a list of handles of loaded object, starting from
236 // 'handle'. 'Handle' must be in the range of valid transient object handles,
237 // but does not have to be the handle of a loaded transient object.
238 // Return Type: TPMI_YES_NO
239 // YES if there are more handles available
240 // NO all the available handles has been returned
241 TPMI_YES_NO
242 ObjectCapGetLoaded(TPMI_DH_OBJECT handle, // IN: start handle
243 UINT32 count, // IN: count of returned handles
244 TPML_HANDLE* handleList // OUT: list of handle
245 );
246
247 /*** ObjectCapGetOneLoaded()
248 // This function returns whether a handle is loaded.
249 BOOL ObjectCapGetOneLoaded(TPMI_DH_OBJECT handle // IN: handle
250 );
251
252 /*** ObjectCapGetTransientAvail()
253 // This function returns an estimate of the number of additional transient
254 // objects that could be loaded into the TPM.
255 UINT32
256 ObjectCapGetTransientAvail(void);
257

```

```

258  /*** ObjectGetPublicAttributes()
259  // Returns the attributes associated with an object handles.
260  TPMA_OBJECT
261  ObjectGetPublicAttributes(TPM_HANDLE handle);
262
263  OBJECT_ATTRIBUTES
264  ObjectGetProperties(TPM_HANDLE handle);
265
266  #endif // _OBJECT_FP_H

```

6.156 /tpm/include/private/prototypes/Object_spt_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 7, 2020 Time: 07:06:44PM
4  */
5
6  #ifndef OBJECT_SPT_FP_H
7  #define OBJECT_SPT_FP_H
8
9  /*** AdjustAuthSize()
10 // This function will validate that the input authValue is no larger than the
11 // digestSize for the nameAlg. It will then pad with zeros to the size of the
12 // digest.
13 BOOL AdjustAuthSize(TPM2B_AUTH* auth, // IN/OUT: value to adjust
14 TPMI_ALG_HASH nameAlg // IN:
15 );
16
17 /*** AreAttributesForParent()
18 // This function is called by create, load, and import functions.
19 //
20 // Note: The 'isParent' attribute is SET when an object is loaded and it has
21 // attributes that are suitable for a parent object.
22 // Return Type: BOOL
23 // TRUE(1) properties are those of a parent
24 // FALSE(0) properties are not those of a parent
25 BOOL ObjectIsParent(OBJECT* parentObject // IN: parent handle
26 );
27
28 /*** CreateChecks()
29 // Attribute checks that are unique to creation.
30 // If parentObject is not NULL, then this function checks the object's
31 // attributes as an Ordinary or Derived Object with the given parent.
32 // If parentObject is NULL, and primaryHandle is not 0, then this function
33 // checks the object's attributes as a Primary Object in the given hierarchy.
34 // If parentObject is NULL, and primaryHandle is 0, then this function checks
35 // the object's attributes as an External Object.
36 // Return Type: TPM_RC
37 // TPM_RC_ATTRIBUTES sensitiveDataOrigin is not consistent with the
38 // object type
39 // other returns from PublicAttributesValidation()
40 TPM_RC
41 CreateChecks(OBJECT* parentObject,
42 TPMI_RH_HIERARCHY primaryHierarchy,
43 TPMT_PUBLIC* publicArea,
44 UINT16 sensitiveDataSize);
45
46 /*** SchemeChecks
47 // This function is called by TPM2_LoadExternal() and PublicAttributesValidation().
48 // This function validates the schemes in the public area of an object.
49 // Return Type: TPM_RC
50 // TPM_RC_HASH non-duplicable storage key and its parent have different
51 // name algorithm
52 // TPM_RC_KDF incorrect KDF specified for decrypting keyed hash object
53 // TPM_RC_KEY invalid key size values in an asymmetric key public area

```

```

54 //      TPM_RCS_SCHEME      inconsistent attributes 'decrypt', 'sign', 'restricted'
55 //                          and key's scheme ID; or hash algorithm is inconsistent
56 //                          with the scheme ID for keyed hash object
57 //      TPM_RC_SYMMETRIC    a storage key with no symmetric algorithm specified; or
58 //                          non-storage key with symmetric algorithm different from
59 //                          TPM_ALG_NULL
60 TPM_RC
61 SchemeChecks(OBJECT*      parentObject, // IN: parent (null if primary seed)
62              TPMT_PUBLIC* publicArea   // IN: public area of the object
63 );
64
65 /*** PublicAttributesValidation()
66 // This function validates the values in the public area of an object.
67 // This function is used in the processing of TPM2_Create, TPM2_CreatePrimary,
68 // TPM2_CreateLoaded(), TPM2_Load(), TPM2_Import(), and TPM2_LoadExternal().
69 // For TPM2_Import() this is only used if the new parent has fixedTPM SET. For
70 // TPM2_LoadExternal(), this is not used for a public-only key.
71 // If parentObject is not NULL, then primaryHandle is not used.
72 // Return Type: TPM_RC
73 //      TPM_RC_ATTRIBUTES   'fixedTPM', 'fixedParent', or 'encryptedDuplication'
74 //                          attributes are inconsistent between themselves or with
75 //                          those of the parent object;
76 //                          inconsistent 'restricted', 'decrypt' and 'sign'
77 //                          attributes;
78 //                          attempt to inject sensitive data for an asymmetric key;
79 //                          attempt to create a symmetric cipher key that is not
80 //                          a decryption key
81 //      TPM_RC_HASH         nameAlg is TPM_ALG_NULL
82 //      TPM_RC_SIZE         'authPolicy' size does not match digest size of the name
83 //                          algorithm in 'publicArea'
84 //      other               returns from SchemeChecks()
85 TPM_RC
86 PublicAttributesValidation(
87     // IN: input parent object (if ordinary or derived object; NULL otherwise)
88     OBJECT* parentObject,
89     // IN: hierarchy (if primary object; 0 otherwise)
90     TPMI_RH_HIERARCHY primaryHierarchy,
91     // IN: public area of the object
92     TPMT_PUBLIC* publicArea);
93
94 /*** FillInCreationData()
95 // Fill in creation data for an object.
96 // Return Type: void
97 void FillInCreationData(
98     TPMI_DH_OBJECT      parentHandle, // IN: handle of parent
99     TPMI_ALG_HASH        nameHashAlg, // IN: name hash algorithm
100     TPML_PCR_SELECTION* creationPCR,  // IN: PCR selection
101     TPM2B_DATA*          outsideData,  // IN: outside data
102     TPM2B_CREATION_DATA* outCreation,  // OUT: creation data for output
103     TPM2B_DIGEST*        creationDigest // OUT: creation digest
104 );
105
106 /*** GetSeedForKDF()
107 // Get a seed for KDF. The KDF for encryption and HMAC key use the same seed.
108 const TPM2B* GetSeedForKDF(OBJECT* protector // IN: the protector handle
109 );
110
111 /*** ProduceOuterWrap()
112 // This function produce outer wrap for a buffer containing the sensitive data.
113 // It requires the sensitive data being marshaled to the outerBuffer, with the
114 // leading bytes reserved for integrity hash. If iv is used, iv space should
115 // be reserved at the beginning of the buffer. It assumes the sensitive data
116 // starts at address (outerBuffer + integrity size @).
117 // This function:
118 // a) adds IV before sensitive area if required;
119 // b) encrypts sensitive data with IV or a NULL IV as required;

```



```

120 // c) adds HMAC integrity at the beginning of the buffer; and
121 // d) returns the total size of blob with outer wrap.
122 UINT16
123 ProduceOuterWrap(OBJECT* protector, // IN: The handle of the object that provides
124 // protection. For object, it is parent
125 // handle. For credential, it is the handle
126 // of encrypt object.
127 TPM2B* name, // IN: the name of the object
128 TPM_ALG_ID hashAlg, // IN: hash algorithm for outer wrap
129 TPM2B* seed, // IN: an external seed may be provided for
130 // duplication blob. For non duplication
131 // blob, this parameter should be NULL
132 BOOL useIV, // IN: indicate if an IV is used
133 UINT16 dataSize, // IN: the size of sensitive data, excluding the
134 // leading integrity buffer size or the
135 // optional iv size
136 BYTE* outerBuffer // IN/OUT: outer buffer with sensitive data in
137 // it
138 );
139
140 /*** UnwrapOuter()
141 // This function remove the outer wrap of a blob containing sensitive data
142 // This function:
143 // a) checks integrity of outer blob; and
144 // b) decrypts the outer blob.
145 //
146 // Return Type: TPM_RC
147 // TPM_RC_INSUFFICIENT error during sensitive data unmarshaling
148 // TPM_RC_INTEGRITY sensitive data integrity is broken
149 // TPM_RC_SIZE error during sensitive data unmarshaling
150 // TPM_RC_VALUE IV size for CFB does not match the encryption
151 // algorithm block size
152 TPM_RC
153 UnwrapOuter(OBJECT* protector, // IN: The object that provides
154 // protection. For object, it is parent
155 // handle. For credential, it is the
156 // encrypt object.
157 TPM2B* name, // IN: the name of the object
158 TPM_ALG_ID hashAlg, // IN: hash algorithm for outer wrap
159 TPM2B* seed, // IN: an external seed may be provided for
160 // duplication blob. For non duplication
161 // blob, this parameter should be NULL.
162 BOOL useIV, // IN: indicates if an IV is used
163 UINT16 dataSize, // IN: size of sensitive data in outerBuffer,
164 // including the leading integrity buffer
165 // size, and an optional iv area
166 BYTE* outerBuffer // IN/OUT: sensitive data
167 );
168
169 /*** SensitiveToPrivate()
170 // This function prepare the private blob for off the chip storage
171 // This function:
172 // a) marshals TPM2B_SENSITIVE structure into the buffer of TPM2B_PRIVATE
173 // b) applies encryption to the sensitive area; and
174 // c) applies outer integrity computation.
175 void SensitiveToPrivate(
176 TPMT_SENSITIVE* sensitive, // IN: sensitive structure
177 TPM2B_NAME* name, // IN: the name of the object
178 OBJECT* parent, // IN: The parent object
179 TPM_ALG_ID nameAlg, // IN: hash algorithm in public area. This
180 // parameter is used when parentHandle is
181 // NULL, in which case the object is
182 // temporary.
183 TPM2B_PRIVATE* outPrivate // OUT: output private structure
184 );
185

```



```

186  /*** PrivateToSensitive()
187  // Unwrap a input private area. Check the integrity, decrypt and retrieve data
188  // to a sensitive structure.
189  // This function:
190  // a) checks the integrity HMAC of the input private area;
191  // b) decrypts the private buffer; and
192  // c) unmarshals TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE.
193  // Return Type: TPM_RC
194  //     TPM_RCS_INTEGRITY      if the private area integrity is bad
195  //     TPM_RC_SENSITIVE       unmarshal errors while unmarshaling TPMS_ENCRYPT
196  //                           from input private
197  //     TPM_RCS_SIZE           error during sensitive data unmarshaling
198  //     TPM_RC_VALUE           outer wrapper does not have an iV of the correct
199  //                           size
200  TPM_RC
201  PrivateToSensitive(TPM2B*      inPrivate, // IN: input private structure
202                    TPM2B*      name,      // IN: the name of the object
203                    OBJECT*      parent,    // IN: parent object
204                    TPM_ALG_ID   nameAlg,   // IN: hash algorithm in public area. It is
205                    //             passed separately because we only pass
206                    //             name, rather than the whole public area
207                    //             of the object. This parameter is used in
208                    //             the following two cases: 1. primary
209                    //             objects. 2. duplication blob with inner
210                    //             wrap. In other cases, this parameter
211                    //             will be ignored
212                    TPMT_SENSITIVE* sensitive // OUT: sensitive structure
213  );
214
215  /*** SensitiveToDuplicate()
216  // This function prepare the duplication blob from the sensitive area.
217  // This function:
218  // a) marshals TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE;
219  // b) applies inner wrap to the sensitive area if required; and
220  // c) applies outer wrap if required.
221  void SensitiveToDuplicate(
222      TPMT_SENSITIVE* sensitive, // IN: sensitive structure
223      TPM2B*          name,      // IN: the name of the object
224      OBJECT*          parent,    // IN: The new parent object
225      TPM_ALG_ID       nameAlg,   // IN: hash algorithm in public area. It
226      //               is passed separately because we
227      //               only pass name, rather than the
228      //               whole public area of the object.
229      TPM2B* seed,             // IN: the external seed. If external
230      //               seed is provided with size of 0,
231      //               no outer wrap should be applied
232      //               to duplication blob.
233      TPMT_SYM_DEF_OBJECT* symDef, // IN: Symmetric key definition. If the
234      //               symmetric key algorithm is NULL,
235      //               no inner wrap should be applied.
236      TPM2B_DATA* innerSymKey,     // IN/OUT: a symmetric key may be
237      //               provided to encrypt the inner
238      //               wrap of a duplication blob. May
239      //               be generated here if needed.
240      TPM2B_PRIVATE* outPrivate // OUT: output private structure
241  );
242
243  /*** DuplicateToSensitive()
244  // Unwrap a duplication blob. Check the integrity, decrypt and retrieve data
245  // to a sensitive structure.
246  // This function:
247  // a) checks the integrity HMAC of the input private area;
248  // b) decrypts the private buffer; and
249  // c) unmarshals TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE.
250  //
251  // Return Type: TPM_RC

```

```

252 //      TPM_RC_INSUFFICIENT      unmarshaling sensitive data from 'inPrivate' failed
253 //      TPM_RC_INTEGRITY          'inPrivate' data integrity is broken
254 //      TPM_RC_SIZE               unmarshaling sensitive data from 'inPrivate' failed
255 TPM_RC
256 DuplicateToSensitive(
257     TPM2B*      inPrivate,        // IN: input private structure
258     TPM2B*      name,             // IN: the name of the object
259     OBJECT*     parent,           // IN: the parent
260     TPM_ALG_ID  nameAlg,          // IN: hash algorithm in public area.
261     TPM2B*      seed,             // IN: an external seed may be provided.
262                                     // If external seed is provided with
263                                     // size of 0, no outer wrap is
264                                     // applied
265     TPMT_SYM_DEF_OBJECT* symDef,  // IN: Symmetric key definition. If the
266                                     // symmetric key algorithm is NULL,
267                                     // no inner wrap is applied
268     TPM2B* innerSymKey,           // IN: a symmetric key may be provided
269                                     // to decrypt the inner wrap of a
270                                     // duplication blob.
271     TPMT_SENSITIVE* sensitive    // OUT: sensitive structure
272 );
273
274 /*** SecretToCredential()
275 // This function prepare the credential blob from a secret (a TPM2B_DIGEST)
276 // This function:
277 // a) marshals TPM2B_DIGEST structure into the buffer of TPM2B_ID_OBJECT;
278 // b) encrypts the private buffer, excluding the leading integrity HMAC area;
279 // c) computes integrity HMAC and append to the beginning of the buffer; and
280 // d) sets the total size of TPM2B_ID_OBJECT buffer.
281 void SecretToCredential(TPM2B_DIGEST* secret, // IN: secret information
282     TPM2B*      name, // IN: the name of the object
283     TPM2B*      seed, // IN: an external seed.
284     OBJECT*     protector, // IN: the protector
285     TPM2B_ID_OBJECT* outIDObject // OUT: output credential
286 );
287
288 /*** CredentialToSecret()
289 // Unwrap a credential. Check the integrity, decrypt and retrieve data
290 // to a TPM2B_DIGEST structure.
291 // This function:
292 // a) checks the integrity HMAC of the input credential area;
293 // b) decrypts the credential buffer; and
294 // c) unmarshals TPM2B_DIGEST structure into the buffer of TPM2B_DIGEST.
295 //
296 // Return Type: TPM_RC
297 //      TPM_RC_INSUFFICIENT      error during credential unmarshaling
298 //      TPM_RC_INTEGRITY          credential integrity is broken
299 //      TPM_RC_SIZE               error during credential unmarshaling
300 //      TPM_RC_VALUE              IV size does not match the encryption algorithm
301 //                                 block size
302 TPM_RC
303 CredentialToSecret(TPM2B* inIDObject, // IN: input credential blob
304     TPM2B*      name, // IN: the name of the object
305     TPM2B*      seed, // IN: an external seed.
306     OBJECT*     protector, // IN: the protector
307     TPM2B_DIGEST* secret // OUT: secret information
308 );
309
310 /*** MemoryRemoveTrailingZeros()
311 // This function is used to adjust the length of an authorization value.
312 // It adjusts the size of the TPM2B so that it does not include octets
313 // at the end of the buffer that contain zero.
314 //
315 // This function returns the number of non-zero octets in the buffer.
316 UINT16
317 MemoryRemoveTrailingZeros(TPM2B_AUTH* auth // IN/OUT: value to adjust

```

```

318 );
319
320 /*** SetLabelAndContext()
321 // This function sets the label and context for a derived key. It is possible
322 // that 'label' or 'context' can end up being an Empty Buffer.
323 TPM_RC
324 SetLabelAndContext(TPMS_DERIVE* labelContext,          // IN/OUT: the recovered label and
325                  // context
326                  TPM2B_SENSITIVE_DATA* sensitive // IN: the sensitive data
327 );
328
329 /*** UnmarshalToPublic()
330 // Support function to unmarshal the template. This is used because the
331 // Input may be a TPMT_TEMPLATE and that structure does not have the same
332 // size as a TPMT_PUBLIC because of the difference between the 'unique' and
333 // 'seed' fields.
334 //
335 // If 'derive' is not NULL, then the 'seed' field is assumed to contain
336 // a 'label' and 'context' that are unmarshaled into 'derive'.
337 TPM_RC
338 UnmarshalToPublic(TPMT_PUBLIC* tOut, // OUT: output
339                  TPM2B_TEMPLATE* tIn, // IN:
340                  BOOL derivation, // IN: indicates if this is for a derivation
341                  TPMS_DERIVE* labelContext // OUT: label and context if derivation
342 );
343
344 /*** ObjectSetExternal()
345 // Set the external attributes for an object.
346 void ObjectSetExternal(OBJECT* object);
347
348 #endif // _OBJECT_SPT_FP_H_

```

6.157 /tpm/include/private/prototypes/PCR_Allocate_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PCR_Allocate // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_ALLOCATE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_ALLOCATE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PLATFORM authHandle;
12     TPML_PCR_SELECTION pcrAllocation;
13 } PCR_Allocate_In;
14
15 // Output structure definition
16 typedef struct
17 {
18     TPMI_YES_NO allocationSuccess;
19     UINT32 maxPCR;
20     UINT32 sizeNeeded;
21     UINT32 sizeAvailable;
22 } PCR_Allocate_Out;
23
24 // Response code modifiers
25 #   define RC_PCR_Allocate_authHandle (TPM_RC_H + TPM_RC_1)
26 #   define RC_PCR_Allocate_pcrAllocation (TPM_RC_P + TPM_RC_1)
27
28 // Function prototype
29 TPM_RC
30 TPM2_PCR_Allocate(PCR_Allocate_In* in, PCR_Allocate_Out* out);
31

```

```

32 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_ALLOCATE_FP_H_
33 #endif // CC_PCR_Allocate

```

6.158 /tpm/include/private/prototypes/PCR_Event_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PCR_Event // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_EVENT_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_EVENT_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_PCR pcrHandle;
12     TPM2B_EVENT eventData;
13 } PCR_Event_In;
14
15 // Output structure definition
16 typedef struct
17 {
18     TPML_DIGEST_VALUES digests;
19 } PCR_Event_Out;
20
21 // Response code modifiers
22 #   define RC_PCR_Event_pcrHandle (TPM_RC_H + TPM_RC_1)
23 #   define RC_PCR_Event_eventData (TPM_RC_P + TPM_RC_1)
24
25 // Function prototype
26 TPM_RC
27 TPM2_PCR_Event(PCR_Event_In* in, PCR_Event_Out* out);
28
29 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_EVENT_FP_H_
30 #endif // CC_PCR_Event

```

6.159 /tpm/include/private/prototypes/PCR_Extend_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PCR_Extend // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_EXTEND_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_EXTEND_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_PCR pcrHandle;
12     TPML_DIGEST_VALUES digests;
13 } PCR_Extend_In;
14
15 // Response code modifiers
16 #   define RC_PCR_Extend_pcrHandle (TPM_RC_H + TPM_RC_1)
17 #   define RC_PCR_Extend_digests (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PCR_Extend(PCR_Extend_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_EXTEND_FP_H_
24 #endif // CC_PCR_Extend

```

6.160 /tpm/include/private/prototypes/PCR_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar  4, 2020   Time: 02:36:44PM
4  */
5
6  #ifndef _PCR_FP_H_
7  #define _PCR_FP_H_
8
9  /*** PCRBelongsAuthGroup()
10 // This function indicates if a PCR belongs to a group that requires an authValue
11 // in order to modify the PCR.  If it does, 'groupIndex' is set to value of
12 // the group index.  This feature of PCR is decided by the platform specification.
13 //
14 // Return Type: BOOL
15 //     TRUE(1)      PCR belongs an authorization group
16 //     FALSE(0)     PCR does not belong an authorization group
17 BOOL PCRBelongsAuthGroup(TPMI_DH_PCR handle,    // IN: handle of PCR
18                          UINT32* groupIndex // OUT: group index if PCR belongs a
19                          // group that allows authValue.  If PCR
20                          // does not belong to an authorization
21                          // group, the value in this parameter is
22                          // invalid
23 );
24
25 /*** PCRBelongsPolicyGroup()
26 // This function indicates if a PCR belongs to a group that requires a policy
27 // authorization in order to modify the PCR.  If it does, 'groupIndex' is set
28 // to value of the group index.  This feature of PCR is decided by the platform
29 // specification.
30 //
31 // Return Type: BOOL
32 //     TRUE(1)      PCR belongs to a policy group
33 //     FALSE(0)     PCR does not belong to a policy group
34 BOOL PCRBelongsPolicyGroup(
35     TPMI_DH_PCR handle,    // IN: handle of PCR
36     UINT32* groupIndex // OUT: group index if PCR belongs a group that
37                       // allows policy.  If PCR does not belong to
38                       // a policy group, the value in this
39                       // parameter is invalid
40 );
41
42 /*** PCRPolicyIsAvailable()
43 // This function indicates if a policy is available for a PCR.
44 //
45 // Return Type: BOOL
46 //     TRUE(1)      the PCR may be authorized by policy
47 //     FALSE(0)     the PCR does not allow policy
48 BOOL PCRPolicyIsAvailable(TPMI_DH_PCR handle // IN: PCR handle
49 );
50
51 /*** PCRGetAuthValue()
52 // This function is used to access the authValue of a PCR.  If PCR does not
53 // belong to an authValue group, an EmptyAuth will be returned.
54 TPM2B_AUTH* PCRGetAuthValue(TPMI_DH_PCR handle // IN: PCR handle
55 );
56
57 /*** PCRGetAuthPolicy()
58 // This function is used to access the authorization policy of a PCR. It sets
59 // 'policy' to the authorization policy and returns the hash algorithm for policy
60 // If the PCR does not allow a policy, TPM_ALG_NULL is returned.
61 TPMI_ALG_HASH
62 PCRGetAuthPolicy(TPMI_DH_PCR handle, // IN: PCR handle
63                 TPM2B_DIGEST* policy // OUT: policy of PCR
64 );

```



```

65
66 /*** PCRManufacture()
67 // This function is used to initialize the policies when a TPM is manufactured.
68 // This function would only be called in a manufacturing environment or in
69 // a TPM simulator.
70 void PCRManufacture(void);
71
72 /*** PcrIsAllocated()
73 // This function indicates if a PCR number for the particular hash algorithm
74 // is allocated.
75 // Return Type: BOOL
76 //     TRUE(1)         PCR is allocated
77 //     FALSE(0)        PCR is not allocated
78 BOOL PcrIsAllocated(UINT32 pcr, // IN: The number of the PCR
79                     TPMI_ALG_HASH hashAlg // IN: The PCR algorithm
80 );
81
82 /*** PcrDrtm()
83 // This function does the DRTM and H-CRTM processing it is called from
84 // _TPM_Hash_End.
85 void PcrDrtm(const TPMI_DH_PCR pcrHandle, // IN: the index of the PCR to be
86             // modified
87             const TPMI_ALG_HASH hash, // IN: the bank identifier
88             const TPM2B_DIGEST* digest // IN: the digest to modify the PCR
89 );
90
91 /*** PCR_ClearAuth()
92 // This function is used to reset the PCR authorization values. It is called
93 // on TPM2_Startup(CLEAR) and TPM2_Clear().
94 void PCR_ClearAuth(void);
95
96 /*** PCRStartup()
97 // This function initializes the PCR subsystem at TPM2_Startup().
98 BOOL PCRStartup(STARTUP_TYPE type, // IN: startup type
99                BYTE locality // IN: startup locality
100 );
101
102 /*** PCRStateSave()
103 // This function is used to save the PCR values that will be restored on TPM Resume.
104 void PCRStateSave(TPM_SU type // IN: startup type
105 );
106
107 /*** PCRIStateSaved()
108 // This function indicates if the selected PCR is a PCR that is state saved
109 // on TPM2_Shutdown(STATE). The return value is based on PCR attributes.
110 // Return Type: BOOL
111 //     TRUE(1)         PCR is state saved
112 //     FALSE(0)        PCR is not state saved
113 BOOL PCRIStateSaved(TPMI_DH_PCR handle // IN: PCR handle to be extended
114 );
115
116 /*** PCRIResetAllowed()
117 // This function indicates if a PCR may be reset by the current command locality.
118 // The return value is based on PCR attributes, and not the PCR allocation.
119 // Return Type: BOOL
120 //     TRUE(1)         TPM2_PCR_Reset is allowed
121 //     FALSE(0)        TPM2_PCR_Reset is not allowed
122 BOOL PCRIResetAllowed(TPMI_DH_PCR handle // IN: PCR handle to be extended
123 );
124
125 /*** PCRChanged()
126 // This function checks a PCR handle to see if the attributes for the PCR are set
127 // so that any change to the PCR causes an increment of the pcrCounter. If it does,
128 // then the function increments the counter. Will also bump the counter if the
129 // handle is zero which means that PCR 0 can not be in the TCB group. Bump on zero
130 // is used by TPM2_Clear().

```



```

131 void PCRChanged(TPM_HANDLE pcrHandle // IN: the handle of the PCR that changed.
132 );
133
134 /*** PCRIsExtendAllowed()
135 // This function indicates a PCR may be extended at the current command locality.
136 // The return value is based on PCR attributes, and not the PCR allocation.
137 // Return Type: BOOL
138 //     TRUE(1)         extend is allowed
139 //     FALSE(0)        extend is not allowed
140 BOOL PCRIsExtendAllowed(TPMI_DH_PCR handle // IN: PCR handle to be extended
141 );
142
143 /*** PCRExtend()
144 // This function is used to extend a PCR in a specific bank.
145 void PCRExtend(TPMI_DH_PCR handle, // IN: PCR handle to be extended
146               TPMI_ALG_HASH hash, // IN: hash algorithm of PCR
147               UINT32 size, // IN: size of data to be extended
148               BYTE* data // IN: data to be extended
149 );
150
151 /*** PCRComputeCurrentDigest()
152 // This function computes the digest of the selected PCR.
153 //
154 // As a side-effect, 'selection' is modified so that only the implemented PCR
155 // will have their bits still set.
156 void PCRComputeCurrentDigest(
157     TPMI_ALG_HASH hashAlg, // IN: hash algorithm to compute digest
158     TPML_PCR_SELECTION* selection, // IN/OUT: PCR selection (filtered on
159                                     // output)
160     TPM2B_DIGEST* digest // OUT: digest
161 );
162
163 /*** PCRRead()
164 // This function is used to read a list of selected PCR. If the requested PCR
165 // number exceeds the maximum number that can be output, the 'selection' is
166 // adjusted to reflect the actual output PCR.
167 void PCRRead(TPML_PCR_SELECTION* selection, // IN/OUT: PCR selection (filtered on
168                                     // output)
169               TPML_DIGEST* digest, // OUT: digest
170               UINT32* pcrCounter // OUT: the current value of PCR generation
171                                     // number
172 );
173
174 /*** PCRAllocate()
175 // This function is used to change the PCR allocation.
176 // Return Type: TPM_RC
177 //     TPM_RC_NO_RESULT allocate failed
178 //     TPM_RC_PCR        improper allocation
179 TPM_RC
180 PCRAllocate(TPML_PCR_SELECTION* allocate, // IN: required allocation
181             UINT32* maxPCR, // OUT: Maximum number of PCR
182             UINT32* sizeNeeded, // OUT: required space
183             UINT32* sizeAvailable // OUT: available space
184 );
185
186 /*** PCRSetValue()
187 // This function is used to set the designated PCR in all banks to an initial value.
188 // The initial value is signed and will be sign extended into the entire PCR.
189 //
190 void PCRSetValue(TPM_HANDLE handle, // IN: the handle of the PCR to set
191                 INT8 initialValue // IN: the value to set
192 );
193
194 /*** PCRResetDynamics
195 // This function is used to reset a dynamic PCR to 0. This function is used in
196 // DRTM sequence.

```

```

197 void PCRResetDynamics(void);
198
199 /*** PCRCapGetAllocation()
200 // This function is used to get the current allocation of PCR banks.
201 // Return Type: TPMI_YES_NO
202 //     YES         if the return count is 0
203 //     NO          if the return count is not 0
204 TPMI_YES_NO
205 PCRCapGetAllocation(UINT32 count, // IN: count of return
206                     TPML_PCR_SELECTION* pcrSelection // OUT: PCR allocation list
207 );
208
209 /*** PCRCapGetProperties()
210 // This function returns a list of PCR properties starting at 'property'.
211 // Return Type: TPMI_YES_NO
212 //     YES         if no more property is available
213 //     NO          if there are more properties not reported
214 TPMI_YES_NO
215 PCRCapGetProperties(TPM_PT_PCR property, // IN: the starting PCR property
216                   UINT32 count, // IN: count of returned properties
217                   TPML_TAGGED_PCR_PROPERTY* select // OUT: PCR select
218 );
219
220 /*** PCRGetProperty()
221 // This function returns the selected PCR property.
222 // Return Type: BOOL
223 //     TRUE(1)      the property type is implemented
224 //     FALSE(0)     the property type is not implemented
225 BOOL PCRGetProperty(TPM_PT_PCR property, TPMS_TAGGED_PCR_SELECT* select);
226
227 /*** PCRCapGetHandles()
228 // This function is used to get a list of handles of PCR, started from 'handle'.
229 // If 'handle' exceeds the maximum PCR handle range, an empty list will be
230 // returned and the return value will be NO.
231 // Return Type: TPMI_YES_NO
232 //     YES         if there are more handles available
233 //     NO          all the available handles has been returned
234 TPMI_YES_NO
235 PCRCapGetHandles(TPMI_DH_PCR handle, // IN: start handle
236                 UINT32 count, // IN: count of returned handles
237                 TPML_HANDLE* handleList // OUT: list of handle
238 );
239
240 /*** PCRCapGetOneHandle()
241 // This function is used to check whether a PCR handle exists.
242 BOOL PCRCapGetOneHandle(TPMI_DH_PCR handle // IN: handle
243 );
244
245 #endif // _PCR_FP_H_

```

6.161 /tpm/include/private/prototypes/PCR_Read_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PCR_Read // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_READ_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_READ_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPML_PCR_SELECTION pcrSelectionIn;
12 } PCR_Read_In;
13

```

```

14 // Output structure definition
15 typedef struct
16 {
17     UINT32                pcrUpdateCounter;
18     TPML_PCR_SELECTION    pcrSelectionOut;
19     TPML_DIGEST           pcrValues;
20 } PCR_Read_Out;
21
22 // Response code modifiers
23 # define RC_PCR_Read_pcrSelectionIn (TPM_RC_P + TPM_RC_1)
24
25 // Function prototype
26 TPM_RC
27 TPM2_PCR_Read(PCR_Read_In* in, PCR_Read_Out* out);
28
29 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_READ_FP_H_
30 #endif // CC_PCR_Read

```

6.162 /tpm/include/private/prototypes/PCR_Reset_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PCR_Reset // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_RESET_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_RESET_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_PCR pcrHandle;
12 } PCR_Reset_In;
13
14 // Response code modifiers
15 # define RC_PCR_Reset_pcrHandle (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_PCR_Reset(PCR_Reset_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_RESET_FP_H_
22 #endif // CC_PCR_Reset

```

6.163 /tpm/include/private/prototypes/PCR_SetAuthPolicy_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PCR_SetAuthPolicy // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_SETAUTHPOLICY_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_SETAUTHPOLICY_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PLATFORM authHandle;
12     TPM2B_DIGEST      authPolicy;
13     TPMI_ALG_HASH      hashAlg;
14     TPMI_DH_PCR        pcrNum;
15 } PCR_SetAuthPolicy_In;
16
17 // Response code modifiers
18 # define RC_PCR_SetAuthPolicy_authHandle (TPM_RC_H + TPM_RC_1)
19 # define RC_PCR_SetAuthPolicy_authPolicy (TPM_RC_P + TPM_RC_1)

```

```

20 # define RC_PCR_SetAuthPolicy_hashAlg (TPM_RC_P + TPM_RC_2)
21 # define RC_PCR_SetAuthPolicy_pcrNum (TPM_RC_P + TPM_RC_3)
22
23 // Function prototype
24 TPM_RC
25 TPM2_PCR_SetAuthPolicy(PCR_SetAuthPolicy_In* in);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_SETAUTHPOLICY_FP_H_
28 #endif // CC_PCR_SetAuthPolicy

```

6.164 /tpm/include/private/prototypes/PCR_SetAuthValue_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PCR_SetAuthValue // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_SETAUTHVALUE_FP_H_
6 # define _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_SETAUTHVALUE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_PCR pcrHandle;
12     TPM2B_DIGEST auth;
13 } PCR_SetAuthValue_In;
14
15 // Response code modifiers
16 # define RC_PCR_SetAuthValue_pcrHandle (TPM_RC_H + TPM_RC_1)
17 # define RC_PCR_SetAuthValue_auth (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PCR_SetAuthValue(PCR_SetAuthValue_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_PCR_SETAUTHVALUE_FP_H_
24 #endif // CC_PCR_SetAuthValue

```

6.165 /tpm/include/private/prototypes/PolicyAuthorizeNV_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyAuthorizeNV // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYAUTHORIZENV_FP_H_
6 # define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYAUTHORIZENV_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_NV_AUTH authHandle;
12     TPMI_RH_NV_INDEX nvIndex;
13     TPMI_SH_POLICY policySession;
14 } PolicyAuthorizeNV_In;
15
16 // Response code modifiers
17 # define RC_PolicyAuthorizeNV_authHandle (TPM_RC_H + TPM_RC_1)
18 # define RC_PolicyAuthorizeNV_nvIndex (TPM_RC_H + TPM_RC_2)
19 # define RC_PolicyAuthorizeNV_policySession (TPM_RC_H + TPM_RC_3)
20
21 // Function prototype
22 TPM_RC
23 TPM2_PolicyAuthorizeNV(PolicyAuthorizeNV_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYAUTHORIZENV_FP_H_

```

```
26 #endif // CC_PolicyAuthorizeNV
```

6.166 /tpm/include/private/prototypes/PolicyAuthorize_fp.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyAuthorize // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYAUTHORIZE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYAUTHORIZE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY    policySession;
12     TPM2B_DIGEST      approvedPolicy;
13     TPM2B_NONCE       policyRef;
14     TPM2B_NAME        keySign;
15     TPMT_TK_VERIFIED  checkTicket;
16 } PolicyAuthorize_In;
17
18 // Response code modifiers
19 #   define RC_PolicyAuthorize_policySession (TPM_RC_H + TPM_RC_1)
20 #   define RC_PolicyAuthorize_approvedPolicy (TPM_RC_P + TPM_RC_1)
21 #   define RC_PolicyAuthorize_policyRef (TPM_RC_P + TPM_RC_2)
22 #   define RC_PolicyAuthorize_keySign (TPM_RC_P + TPM_RC_3)
23 #   define RC_PolicyAuthorize_checkTicket (TPM_RC_P + TPM_RC_4)
24
25 // Function prototype
26 TPM_RC
27 TPM2_PolicyAuthorize(PolicyAuthorize_In* in);
28
29 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYAUTHORIZE_FP_H_
30 #endif // CC_PolicyAuthorize
```

6.167 /tpm/include/private/prototypes/PolicyAuthValue_fp.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyAuthValue // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYAUTHVALUE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYAUTHVALUE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12 } PolicyAuthValue_In;
13
14 // Response code modifiers
15 #   define RC_PolicyAuthValue_policySession (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_PolicyAuthValue(PolicyAuthValue_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYAUTHVALUE_FP_H_
22 #endif // CC_PolicyAuthValue
```

6.168 /tpm/include/private/prototypes/PolicyCapability_fp.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
```

```

2
3 #if CC_PolicyCapability // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCAPABILITY_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCAPABILITY_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPM2B_OPERAND operandB;
13     UINT16 offset;
14     TPM_EO operation;
15     TPM_CAP capability;
16     UINT32 property;
17 } PolicyCapability_In;
18
19 // Response code modifiers
20 #   define RC_PolicyCapability_policySession (TPM_RC_H + TPM_RC_1)
21 #   define RC_PolicyCapability_operandB (TPM_RC_P + TPM_RC_1)
22 #   define RC_PolicyCapability_offset (TPM_RC_P + TPM_RC_2)
23 #   define RC_PolicyCapability_operation (TPM_RC_P + TPM_RC_3)
24 #   define RC_PolicyCapability_capability (TPM_RC_P + TPM_RC_4)
25 #   define RC_PolicyCapability_property (TPM_RC_P + TPM_RC_5)
26
27 // Function prototype
28 TPM_RC
29 TPM2_PolicyCapability(PolicyCapability_In* in);
30
31 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCAPABILITY_FP_H_
32 #endif // CC_PolicyCapability

```

6.169 /tpm/include/private/prototypes/PolicyCommandCode_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyCommandCode // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCOMMANDCODE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCOMMANDCODE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPM_CC code;
13 } PolicyCommandCode_In;
14
15 // Response code modifiers
16 #   define RC_PolicyCommandCode_policySession (TPM_RC_H + TPM_RC_1)
17 #   define RC_PolicyCommandCode_code (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PolicyCommandCode(PolicyCommandCode_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCOMMANDCODE_FP_H_
24 #endif // CC_PolicyCommandCode

```

6.170 /tpm/include/private/prototypes/PolicyCounterTimer_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyCounterTimer // Command must be enabled

```



```

4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCOUNTERTIMER_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCOUNTERTIMER_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPM2B_OPERAND operandB;
13     UINT16 offset;
14     TPM_EO operation;
15 } PolicyCounterTimer_In;
16
17 // Response code modifiers
18 #   define RC_PolicyCounterTimer_policySession (TPM_RC_H + TPM_RC_1)
19 #   define RC_PolicyCounterTimer_operandB (TPM_RC_P + TPM_RC_1)
20 #   define RC_PolicyCounterTimer_offset (TPM_RC_P + TPM_RC_2)
21 #   define RC_PolicyCounterTimer_operation (TPM_RC_P + TPM_RC_3)
22
23 // Function prototype
24 TPM_RC
25 TPM2_PolicyCounterTimer(PolicyCounterTimer_In* in);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCOUNTERTIMER_FP_H_
28 #endif // CC_PolicyCounterTimer

```

6.171 /tpm/include/private/prototypes/PolicyCpHash_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyCpHash // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCPHASH_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCPHASH_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPM2B_DIGEST cpHashA;
13 } PolicyCpHash_In;
14
15 // Response code modifiers
16 #   define RC_PolicyCpHash_policySession (TPM_RC_H + TPM_RC_1)
17 #   define RC_PolicyCpHash_cpHashA (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PolicyCpHash(PolicyCpHash_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYCPHASH_FP_H_
24 #endif // CC_PolicyCpHash

```

6.172 /tpm/include/private/prototypes/PolicyDuplicationSelect_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyDuplicationSelect // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYDUPLICATIONSELECT_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYDUPLICATIONSELECT_FP_H_
7
8 // Input structure definition
9 typedef struct

```

```

10 {
11     TPMI_SH_POLICY policySession;
12     TPM2B_NAME      objectName;
13     TPM2B_NAME      newParentName;
14     TPMI_YES_NO     includeObject;
15 } PolicyDuplicationSelect_In;
16
17 // Response code modifiers
18 # define RC_PolicyDuplicationSelect_policySession (TPM_RC_H + TPM_RC_1)
19 # define RC_PolicyDuplicationSelect_objectName     (TPM_RC_P + TPM_RC_1)
20 # define RC_PolicyDuplicationSelect_newParentName  (TPM_RC_P + TPM_RC_2)
21 # define RC_PolicyDuplicationSelect_includeObject  (TPM_RC_P + TPM_RC_3)
22
23 // Function prototype
24 TPM_RC
25 TPM2_PolicyDuplicationSelect(PolicyDuplicationSelect_In* in);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYDUPLICATIONSELECT_FP_H_
28 #endif // CC_PolicyDuplicationSelect

```

6.173 /tpm/include/private/prototypes/PolicyGetDigest_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyGetDigest // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYGETDIGEST_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYGETDIGEST_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12 } PolicyGetDigest_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPM2B_DIGEST policyDigest;
18 } PolicyGetDigest_Out;
19
20 // Response code modifiers
21 # define RC_PolicyGetDigest_policySession (TPM_RC_H + TPM_RC_1)
22
23 // Function prototype
24 TPM_RC
25 TPM2_PolicyGetDigest(PolicyGetDigest_In* in, PolicyGetDigest_Out* out);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYGETDIGEST_FP_H_
28 #endif // CC_PolicyGetDigest

```

6.174 /tpm/include/private/prototypes/PolicyLocality_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyLocality // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYLOCALITY_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYLOCALITY_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;

```

```

12     TPMA_LOCALITY locality;
13 } PolicyLocality_In;
14
15 // Response code modifiers
16 #   define RC_PolicyLocality_policySession (TPM_RC_H + TPM_RC_1)
17 #   define RC_PolicyLocality_locality      (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PolicyLocality(PolicyLocality_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYLOCALITY_FP_H_
24 #endif // CC_PolicyLocality

```

6.175 /tpm/include/private/prototypes/PolicyNameHash_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyNameHash // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYNAMEHASH_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYNAMEHASH_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPM2B_DIGEST nameHash;
13 } PolicyNameHash_In;
14
15 // Response code modifiers
16 #   define RC_PolicyNameHash_policySession (TPM_RC_H + TPM_RC_1)
17 #   define RC_PolicyNameHash_nameHash      (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PolicyNameHash(PolicyNameHash_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYNAMEHASH_FP_H_
24 #endif // CC_PolicyNameHash

```

6.176 /tpm/include/private/prototypes/PolicyNvWritten_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyNvWritten // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYNVWRITTEN_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYNVWRITTEN_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPMI_YES_NO writtenSet;
13 } PolicyNvWritten_In;
14
15 // Response code modifiers
16 #   define RC_PolicyNvWritten_policySession (TPM_RC_H + TPM_RC_1)
17 #   define RC_PolicyNvWritten_writtenSet    (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PolicyNvWritten(PolicyNvWritten_In* in);

```

```

22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYNVWRITTEN_FP_H_
24 #endif // CC_PolicyNvWritten

```

6.177 /tpm/include/private/prototypes/PolicyNV_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyNV // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYNV_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYNV_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_NV_AUTH authHandle;
12     TPMI_RH_NV_INDEX nvIndex;
13     TPMI_SH_POLICY policySession;
14     TPM2B_OPERAND operandB;
15     UINT16 offset;
16     TPM_EO operation;
17 } PolicyNV_In;
18
19 // Response code modifiers
20 #   define RC_PolicyNV_authHandle (TPM_RC_H + TPM_RC_1)
21 #   define RC_PolicyNV_nvIndex (TPM_RC_H + TPM_RC_2)
22 #   define RC_PolicyNV_policySession (TPM_RC_H + TPM_RC_3)
23 #   define RC_PolicyNV_operandB (TPM_RC_P + TPM_RC_1)
24 #   define RC_PolicyNV_offset (TPM_RC_P + TPM_RC_2)
25 #   define RC_PolicyNV_operation (TPM_RC_P + TPM_RC_3)
26
27 // Function prototype
28 TPM_RC
29 TPM2_PolicyNV(PolicyNV_In* in);
30
31 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYNV_FP_H_
32 #endif // CC_PolicyNV

```

6.178 /tpm/include/private/prototypes/PolicyOR_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyOR // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYOR_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYOR_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPML_DIGEST pHashList;
13 } PolicyOR_In;
14
15 // Response code modifiers
16 #   define RC_PolicyOR_policySession (TPM_RC_H + TPM_RC_1)
17 #   define RC_PolicyOR_pHashList (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PolicyOR(PolicyOR_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYOR_FP_H_

```

```
24 #endif // CC_PolicyOR
```

6.179 /tpm/include/private/prototypes/PolicyParameters_fp.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyParameters // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPARAMETERS_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPARAMETERS_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPM2B_DIGEST pHash;
13 } PolicyParameters_In;
14
15 // Response code modifiers
16 #   define RC_PolicyParameters_policySession (TPM_RC_H + TPM_RC_1)
17 #   define RC_PolicyParameters_pHash (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PolicyParameters(PolicyParameters_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPARAMETERS_FP_H_
24 #endif // CC_PolicyParameters
```

6.180 /tpm/include/private/prototypes/PolicyPassword_fp.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyPassword // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPASSWORD_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPASSWORD_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12 } PolicyPassword_In;
13
14 // Response code modifiers
15 #   define RC_PolicyPassword_policySession (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_PolicyPassword(PolicyPassword_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPASSWORD_FP_H_
22 #endif // CC_PolicyPassword
```

6.181 /tpm/include/private/prototypes/PolicyPCR_fp.h

```
1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyPCR // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPCR_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPCR_FP_H_
7
```

```

8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY    policySession;
12     TPM2B_DIGEST      pcrDigest;
13     TPML_PCR_SELECTION pcrs;
14 } PolicyPCR_In;
15
16 // Response code modifiers
17 # define RC_PolicyPCR_policySession (TPM_RC_H + TPM_RC_1)
18 # define RC_PolicyPCR_pcrDigest     (TPM_RC_P + TPM_RC_1)
19 # define RC_PolicyPCR_pcrs          (TPM_RC_P + TPM_RC_2)
20
21 // Function prototype
22 TPM_RC
23 TPM2_PolicyPCR(PolicyPCR_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPCR_FP_H_
26 #endif // CC_PolicyPCR

```

6.182 /tpm/include/private/prototypes/PolicyPhysicalPresence_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyPhysicalPresence // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPHYSICALPRESENCE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPHYSICALPRESENCE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12 } PolicyPhysicalPresence_In;
13
14 // Response code modifiers
15 # define RC_PolicyPhysicalPresence_policySession (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_PolicyPhysicalPresence(PolicyPhysicalPresence_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYPHYSICALPRESENCE_FP_H_
22 #endif // CC_PolicyPhysicalPresence

```

6.183 /tpm/include/private/prototypes/PolicyRestart_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyRestart // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYRESTART_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYRESTART_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY sessionHandle;
12 } PolicyRestart_In;
13
14 // Response code modifiers
15 # define RC_PolicyRestart_sessionHandle (TPM_RC_H + TPM_RC_1)
16
17 // Function prototype

```



```

18 TPM_RC
19 TPM2_PolicyRestart(PolicyRestart_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYRESTART_FP_H_
22 #endif // CC_PolicyRestart

```

6.184 /tpm/include/private/prototypes/PolicySecret_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicySecret // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYSECRET_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYSECRET_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_ENTITY authHandle;
12     TPMI_SH_POLICY policySession;
13     TPM2B_NONCE nonceTPM;
14     TPM2B_DIGEST cpHashA;
15     TPM2B_NONCE policyRef;
16     INT32 expiration;
17 } PolicySecret_In;
18
19 // Output structure definition
20 typedef struct
21 {
22     TPM2B_TIMEOUT timeout;
23     TPMT_TK_AUTH policyTicket;
24 } PolicySecret_Out;
25
26 // Response code modifiers
27 # define RC_PolicySecret_authHandle (TPM_RC_H + TPM_RC_1)
28 # define RC_PolicySecret_policySession (TPM_RC_H + TPM_RC_2)
29 # define RC_PolicySecret_nonceTPM (TPM_RC_P + TPM_RC_1)
30 # define RC_PolicySecret_cpHashA (TPM_RC_P + TPM_RC_2)
31 # define RC_PolicySecret_policyRef (TPM_RC_P + TPM_RC_3)
32 # define RC_PolicySecret_expiration (TPM_RC_P + TPM_RC_4)
33
34 // Function prototype
35 TPM_RC
36 TPM2_PolicySecret(PolicySecret_In* in, PolicySecret_Out* out);
37
38 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYSECRET_FP_H_
39 #endif // CC_PolicySecret

```

6.185 /tpm/include/private/prototypes/PolicySigned_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicySigned // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYSigned_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYSigned_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT authObject;
12     TPMI_SH_POLICY policySession;
13     TPM2B_NONCE nonceTPM;
14     TPM2B_DIGEST cpHashA;

```

```

15     TPM2B_NONCE    policyRef;
16     INT32          expiration;
17     TPMT_SIGNATURE auth;
18 } PolicySigned_In;
19
20 // Output structure definition
21 typedef struct
22 {
23     TPM2B_TIMEOUT timeout;
24     TPMT_TK_AUTH  policyTicket;
25 } PolicySigned_Out;
26
27 // Response code modifiers
28 #   define RC_PolicySigned_authObject      (TPM_RC_H + TPM_RC_1)
29 #   define RC_PolicySigned_policySession  (TPM_RC_H + TPM_RC_2)
30 #   define RC_PolicySigned_nonceTPM      (TPM_RC_P + TPM_RC_1)
31 #   define RC_PolicySigned_cpHashA       (TPM_RC_P + TPM_RC_2)
32 #   define RC_PolicySigned_policyRef     (TPM_RC_P + TPM_RC_3)
33 #   define RC_PolicySigned_expiration    (TPM_RC_P + TPM_RC_4)
34 #   define RC_PolicySigned_auth         (TPM_RC_P + TPM_RC_5)
35
36 // Function prototype
37 TPM_RC
38 TPM2_PolicySigned(PolicySigned_In* in, PolicySigned_Out* out);
39
40 #   endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICY_SIGNED_FP_H_
41 #endif // CC_PolicySigned

```

6.186 /tpm/include/private/prototypes/PolicyTemplate_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyTemplate // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYTEMPLATE_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYTEMPLATE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPM2B_DIGEST  templateHash;
13 } PolicyTemplate_In;
14
15 // Response code modifiers
16 #   define RC_PolicyTemplate_policySession (TPM_RC_H + TPM_RC_1)
17 #   define RC_PolicyTemplate_templateHash  (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_PolicyTemplate(PolicyTemplate_In* in);
22
23 #   endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYTEMPLATE_FP_H_
24 #endif // CC_PolicyTemplate

```

6.187 /tpm/include/private/prototypes/PolicyTicket_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_PolicyTicket // Command must be enabled
4
5 #   ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYTICKET_FP_H_
6 #       define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYTICKET_FP_H_
7

```

```

8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPM2B_TIMEOUT timeout;
13     TPM2B_DIGEST cpHashA;
14     TPM2B_NONCE policyRef;
15     TPM2B_NAME authName;
16     TPMT_TK_AUTH ticket;
17 } PolicyTicket_In;
18
19 // Response code modifiers
20 # define RC_PolicyTicket_policySession (TPM_RC_H + TPM_RC_1)
21 # define RC_PolicyTicket_timeout (TPM_RC_P + TPM_RC_1)
22 # define RC_PolicyTicket_cpHashA (TPM_RC_P + TPM_RC_2)
23 # define RC_PolicyTicket_policyRef (TPM_RC_P + TPM_RC_3)
24 # define RC_PolicyTicket_authName (TPM_RC_P + TPM_RC_4)
25 # define RC_PolicyTicket_ticket (TPM_RC_P + TPM_RC_5)
26
27 // Function prototype
28 TPM_RC
29 TPM2_PolicyTicket(PolicyTicket_In* in);
30
31 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICYTICKET_FP_H_
32 #endif // CC_PolicyTicket

```

6.188 /tpm/include/private/prototypes/Policy_AC_SendSelect_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Policy_AC_SendSelect // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICY_AC_SENDSELECT_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICY_AC_SENDSELECT_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_SH_POLICY policySession;
12     TPM2B_NAME objectName;
13     TPM2B_NAME authHandleName;
14     TPM2B_NAME acName;
15     TPMI_YES_NO includeObject;
16 } Policy_AC_SendSelect_In;
17
18 // Response code modifiers
19 # define RC_Policy_AC_SendSelect_policySession (TPM_RC_H + TPM_RC_1)
20 # define RC_Policy_AC_SendSelect_objectName (TPM_RC_P + TPM_RC_1)
21 # define RC_Policy_AC_SendSelect_authHandleName (TPM_RC_P + TPM_RC_2)
22 # define RC_Policy_AC_SendSelect_acName (TPM_RC_P + TPM_RC_3)
23 # define RC_Policy_AC_SendSelect_includeObject (TPM_RC_P + TPM_RC_4)
24
25 // Function prototype
26 TPM_RC
27 TPM2_Policy_AC_SendSelect(Policy_AC_SendSelect_In* in);
28
29 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_POLICY_AC_SENDSELECT_FP_H_
30 #endif // CC_Policy_AC_SendSelect

```

6.189 /tpm/include/private/prototypes/Policy_spt_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 4, 2020 Time: 02:36:44PM

```

```

4  */
5
6  #ifndef _POLICY_SPT_FP_H_
7  #define _POLICY_SPT_FP_H_
8
9  /** Functions
10 /** PolicyParameterChecks()
11 // This function validates the common parameters of TPM2_PolicySinged()
12 // and TPM2_PolicySecret(). The common parameters are 'nonceTPM',
13 // 'expiration', and 'cpHashA'.
14 TPM_RC
15 PolicyParameterChecks(SESSION* session,
16                       UINT64 authTimeout,
17                       TPM2B_DIGEST* cpHashA,
18                       TPM2B_NONCE* nonce,
19                       TPM_RC blameNonce,
20                       TPM_RC blameCpHash,
21                       TPM_RC blameExpiration);
22
23 /** PolicyContextUpdate()
24 // Update policy hash
25 // Update the policyDigest in policy session by extending policyRef and
26 // objectName to it. This will also update the cpHash if it is present.
27 //
28 // Return Type: void
29 void PolicyContextUpdate(
30     TPM_CC commandCode, // IN: command code
31     TPM2B_NAME* name, // IN: name of entity
32     TPM2B_NONCE* ref, // IN: the reference data
33     TPM2B_DIGEST* cpHash, // IN: the cpHash (optional)
34     UINT64 policyTimeout, // IN: the timeout value for the policy
35     SESSION* session // IN/OUT: policy session to be updated
36 );
37
38 /** ComputeAuthTimeout()
39 // This function is used to determine what the authorization timeout value for
40 // the session should be.
41 UINT64
42 ComputeAuthTimeout(SESSION* session, // IN: the session containing the time
43                   // values
44                   INT32 expiration, // IN: either the number of seconds from
45                   // the start of the session or the
46                   // time in g_timer;
47                   TPM2B_NONCE* nonce // IN: indicator of the time base
48 );
49
50 /** PolicyDigestClear()
51 // Function to reset the policyDigest of a session
52 void PolicyDigestClear(SESSION* session);
53
54 /** PolicySptCheckCondition()
55 // Checks to see if the condition in the policy is satisfied.
56 BOOL PolicySptCheckCondition(TPM_EO operation, BYTE* opA, BYTE* opB, UINT16 size);
57
58 #endif // _POLICY_SPT_FP_H_

```

6.190 /tpm/include/private/prototypes/Power_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Apr 2, 2019 Time: 11:00:49AM
4  */
5
6  #ifndef _POWER_FP_H_
7  #define _POWER_FP_H_

```

```

8
9  /*** TPMInit()
10 // This function is used to process a power on event.
11 void TPMInit(void);
12
13 /*** TPMRegisterStartup()
14 // This function registers the fact that the TPM has been initialized
15 // (a TPM2_Startup() has completed successfully).
16 BOOL TPMRegisterStartup(void);
17
18 /*** TPMIsStarted()
19 // Indicates if the TPM has been initialized (a TPM2_Startup() has completed
20 // successfully after a _TPM_Init).
21 // Return Type: BOOL
22 //     TRUE(1)       TPM has been initialized
23 //     FALSE(0)      TPM has not been initialized
24 BOOL TPMIsStarted(void);
25
26 #endif // _POWER_FP_H_

```

6.191 /tpm/include/private/prototypes/PP_Commands_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_PP_Commands // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_PP_COMMANDS_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_PP_COMMANDS_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_RH_PLATFORM auth;
12     TPML_CC          setList;
13     TPML_CC          clearList;
14 } PP_Commands_In;
15
16 // Response code modifiers
17 #   define RC_PP_Commands_auth      (TPM_RC_H + TPM_RC_1)
18 #   define RC_PP_Commands_setList   (TPM_RC_P + TPM_RC_1)
19 #   define RC_PP_Commands_clearList (TPM_RC_P + TPM_RC_2)
20
21 // Function prototype
22 TPM_RC
23 TPM2_PP_Commands(PP_Commands_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_PP_COMMANDS_FP_H_
26 #endif // CC_PP_Commands

```

6.192 /tpm/include/private/prototypes/PP_fp.h

```

1  /* (Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Mar 28, 2019 Time: 08:25:19PM
4   */
5
6  #ifndef _PP_FP_H_
7  #define _PP_FP_H_
8
9  /*** PhysicalPresencePreInstall_Init()
10 // This function is used to initialize the array of commands that always require
11 // confirmation with physical presence. The array is an array of bits that
12 // has a correspondence with the command code.
13 //

```

```

14 // This command should only ever be executable in a manufacturing setting or in
15 // a simulation.
16 //
17 // When set, these cannot be cleared.
18 //
19 void PhysicalPresencePreInstall_Init(void);
20
21 /*** PhysicalPresenceCommandSet()
22 // This function is used to set the indicator that a command requires
23 // PP confirmation.
24 void PhysicalPresenceCommandSet(TPM_CC commandCode // IN: command code
25 );
26
27 /*** PhysicalPresenceCommandClear()
28 // This function is used to clear the indicator that a command requires PP
29 // confirmation.
30 void PhysicalPresenceCommandClear(TPM_CC commandCode // IN: command code
31 );
32
33 /*** PhysicalPresenceIsRequired()
34 // This function indicates if PP confirmation is required for a command.
35 // Return Type: BOOL
36 //     TRUE(1)      physical presence is required
37 //     FALSE(0)     physical presence is not required
38 BOOL PhysicalPresenceIsRequired(COMMAND_INDEX commandIndex // IN: command index
39 );
40
41 /*** PhysicalPresenceCapGetCCList()
42 // This function returns a list of commands that require PP confirmation. The
43 // list starts from the first implemented command that has a command code that
44 // the same or greater than 'commandCode'.
45 // Return Type: TPML_YES_NO
46 //     YES         if there are more command codes available
47 //     NO          all the available command codes have been returned
48 TPML_YES_NO
49 PhysicalPresenceCapGetCCList(TPM_CC commandCode, // IN: start command code
50                             UINT32 count, // IN: count of returned TPM_CC
51                             TPML_CC* commandList // OUT: list of TPM_CC
52 );
53
54 /*** PhysicalPresenceCapGetOneCC()
55 // This function returns true if the command requires Physical Presence.
56 BOOL PhysicalPresenceCapGetOneCC(TPM_CC commandCode // IN: command code
57 );
58
59 #endif // _PP_FP_H_

```

6.193 /tpm/include/private/prototypes/PropertyCap_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 28, 2019 Time: 08:25:19PM
4 */
5
6 #ifndef PROPERTY_CAP_FP_H_
7 #define PROPERTY_CAP_FP_H_
8
9 /*** TPMCapGetProperties()
10 // This function is used to get the TPM_PT values. The search of properties will
11 // start at 'property' and continue until 'propertyList' has as many values as
12 // will fit, or the last property has been reported, or the list has as many
13 // values as requested in 'count'.
14 // Return Type: TPML_YES_NO
15 //     YES         more properties are available
16 //     NO          no more properties to be reported

```



```

17 TPMI_YES_NO
18 TPMCapGetProperties(TPM_PT property, // IN: the starting TPM property
19                    UINT32 count,    // IN: maximum number of returned
20                                // properties
21                    TPML_TAGGED_TPM_PROPERTY* propertyList // OUT: property list
22 );
23
24 /*** TPMCapGetOneProperty()
25 // This function returns a single TPM property, if present.
26 BOOL TPMCapGetOneProperty(TPM_PT pt, // IN: the TPM property
27                           TPMS_TAGGED_PROPERTY* property // OUT: tagged property
28 );
29
30 #endif // _PROPERTY_CAP_FP_H_

```

6.194 /tpm/include/private/prototypes/Quote_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Quote // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_QUOTE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_QUOTE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT    signHandle;
12     TPM2B_DATA        qualifyingData;
13     TPMT_SIG_SCHEME   inScheme;
14     TPML_PCR_SELECTION PCRselect;
15 } Quote_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_ATTEST    quoted;
21     TPMT_SIGNATURE  signature;
22 } Quote_Out;
23
24 // Response code modifiers
25 #   define RC_Quote_signHandle      (TPM_RC_H + TPM_RC_1)
26 #   define RC_Quote_qualifyingData (TPM_RC_P + TPM_RC_1)
27 #   define RC_Quote_inScheme       (TPM_RC_P + TPM_RC_2)
28 #   define RC_Quote_PCRselect      (TPM_RC_P + TPM_RC_3)
29
30 // Function prototype
31 TPM_RC
32 TPM2_Quote(Quote_In* in, Quote_Out* out);
33
34 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_QUOTE_FP_H_
35 #endif // CC_Quote

```

6.195 /tpm/include/private/prototypes/ReadClock_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ReadClock // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_READCLOCK_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_READCLOCK_FP_H_
7
8 // Output structure definition
9 typedef struct

```

```

10 {
11     TPMS_TIME_INFO currentTime;
12 } ReadClock_Out;
13
14 // Function prototype
15 TPM_RC
16 TPM2_ReadClock(ReadClock_Out* out);
17
18 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_READCLOCK_FP_H_
19 #endif // CC_ReadClock

```

6.196 /tpm/include/private/prototypes/ReadPublic_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_ReadPublic // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_READPUBLIC_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_READPUBLIC_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT objectHandle;
12 } ReadPublic_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPM2B_PUBLIC outPublic;
18     TPM2B_NAME name;
19     TPM2B_NAME qualifiedName;
20 } ReadPublic_Out;
21
22 // Response code modifiers
23 #   define RC_ReadPublic_objectHandle (TPM_RC_H + TPM_RC_1)
24
25 // Function prototype
26 TPM_RC
27 TPM2_ReadPublic(ReadPublic_In* in, ReadPublic_Out* out);
28
29 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_READPUBLIC_FP_H_
30 #endif // CC_ReadPublic

```

6.197 /tpm/include/private/prototypes/ResponseCodeProcessing_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 28, 2019 Time: 08:25:19PM
4 */
5
6 #ifndef _RESPONSE_CODE_PROCESSING_FP_H_
7 #define _RESPONSE_CODE_PROCESSING_FP_H_
8
9 /** RcSafeAddToResult()
10 // Adds a modifier to a response code as long as the response code allows a modifier
11 // and no modifier has already been added.
12 TPM_RC
13 RcSafeAddToResult(TPM_RC responseCode, TPM_RC modifier);
14
15 #endif // _RESPONSE_CODE_PROCESSING_FP_H_

```

6.198 /tpm/include/private/prototypes/Response_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 28, 2019   Time: 08:25:19PM
4  */
5
6  #ifndef _RESPONSE_FP_H_
7  #define _RESPONSE_FP_H_
8
9  /** BuildResponseHeader()
10 // Adds the response header to the response. It will update command->parameterSize
11 // to indicate the total size of the response.
12 void BuildResponseHeader(COMMAND* command, // IN: main control structure
13                          BYTE*   buffer,  // OUT: the output buffer
14                          TPM_RC   result  // IN: the response code
15 );
16
17 #endif // _RESPONSE_FP_H_

```

6.199 /tpm/include/private/prototypes/Rewrap_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_Rewrap // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_REWRAP_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_REWRAP_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT    oldParent;
12     TPMI_DH_OBJECT    newParent;
13     TPM2B_PRIVATE      inDuplicate;
14     TPM2B_NAME         name;
15     TPM2B_ENCRYPTED_SECRET inSymSeed;
16 } Rewrap_In;
17
18 // Output structure definition
19 typedef struct
20 {
21     TPM2B_PRIVATE      outDuplicate;
22     TPM2B_ENCRYPTED_SECRET outSymSeed;
23 } Rewrap_Out;
24
25 // Response code modifiers
26 #   define RC_Rewrap_oldParent    (TPM_RC_H + TPM_RC_1)
27 #   define RC_Rewrap_newParent    (TPM_RC_H + TPM_RC_2)
28 #   define RC_Rewrap_inDuplicate  (TPM_RC_P + TPM_RC_1)
29 #   define RC_Rewrap_name         (TPM_RC_P + TPM_RC_2)
30 #   define RC_Rewrap_inSymSeed    (TPM_RC_P + TPM_RC_3)
31
32 // Function prototype
33 TPM_RC
34 TPM2_Rewrap(Rewrap_In* in, Rewrap_Out* out);
35
36 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_REWRAP_FP_H_
37 #endif // CC_Rewrap

```

6.200 /tpm/include/private/prototypes/RsaKeyCache_fp.h

```

1  /* (Auto-generated)

```

```

2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 28, 2019 Time: 08:25:19PM
4  */
5
6  #ifndef _RSA_KEY_CACHE_FP_H_
7  #define _RSA_KEY_CACHE_FP_H_
8
9  #if USE_RSA_KEY_CACHE
10
11  /*** RsaKeyCacheControl()
12  // Used to enable and disable the RSA key cache.
13  LIB_EXPORT void RsaKeyCacheControl(int state);
14
15  /*** GetCachedRsaKey()
16  // Return Type: BOOL
17  // TRUE(1) key loaded
18  // FALSE(0) key not loaded
19  BOOL GetCachedRsaKey(TPMT_PUBLIC* publicArea,
20                      TPMT_SENSITIVE* sensitive,
21                      RAND_STATE* rand // IN: if not NULL, the deterministic
22                                      // RNG state
23  );
24  #endif // defined SIMULATION && defined USE_RSA_KEY_CACHE
25
26  #endif // _RSA_KEY_CACHE_FP_H_

```

6.201 /tpm/include/private/prototypes/RSA_Decrypt_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_RSA_Decrypt // Command must be enabled
4
5  # if _ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_RSA_DECRYPT_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_RSA_DECRYPT_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT keyHandle;
12     TPM2B_PUBLIC_KEY_RSA cipherText;
13     TPMT_RSA_DECRYPT inScheme;
14     TPM2B_DATA label;
15 } RSA_Decrypt_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_PUBLIC_KEY_RSA message;
21 } RSA_Decrypt_Out;
22
23 // Response code modifiers
24 #   define RC_RSA_Decrypt_keyHandle (TPM_RC_H + TPM_RC_1)
25 #   define RC_RSA_Decrypt_cipherText (TPM_RC_P + TPM_RC_1)
26 #   define RC_RSA_Decrypt_inScheme (TPM_RC_P + TPM_RC_2)
27 #   define RC_RSA_Decrypt_label (TPM_RC_P + TPM_RC_3)
28
29 // Function prototype
30 TPM_RC
31 TPM2_RSA_Decrypt(RSA_Decrypt_In* in, RSA_Decrypt_Out* out);
32
33 #   endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_RSA_DECRYPT_FP_H_
34 #endif // CC_RSA_Decrypt

```

6.202 /tpm/include/private/prototypes/RSA_Encrypt_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_RSA_Encrypt  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_RSA_ENCRYPT_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_RSA_ENCRYPT_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT      keyHandle;
12     TPM2B_PUBLIC_KEY_RSA message;
13     TPMT_RSA_DECRYPT     inScheme;
14     TPM2B_DATA          label;
15 } RSA_Encrypt_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPM2B_PUBLIC_KEY_RSA outData;
21 } RSA_Encrypt_Out;
22
23 // Response code modifiers
24 #  define RC_RSA_Encrypt_keyHandle (TPM_RC_H + TPM_RC_1)
25 #  define RC_RSA_Encrypt_message (TPM_RC_P + TPM_RC_1)
26 #  define RC_RSA_Encrypt_inScheme (TPM_RC_P + TPM_RC_2)
27 #  define RC_RSA_Encrypt_label (TPM_RC_P + TPM_RC_3)
28
29 // Function prototype
30 TPM_RC
31 TPM2_RSA_Encrypt(RSA_Encrypt_In* in, RSA_Encrypt_Out* out);
32
33 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_RSA_ENCRYPT_FP_H_
34 #endif  // CC_RSA_Encrypt

```

6.203 /tpm/include/private/prototypes/SelfTest_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_SelfTest  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_SELFTEST_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_SELFTEST_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_YES_NO fullTest;
12 } SelfTest_In;
13
14 // Response code modifiers
15 #  define RC_SelfTest_fullTest (TPM_RC_P + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_SelfTest(SelfTest_In* in);
20
21 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_SELFTEST_FP_H_
22 #endif  // CC_SelfTest

```

6.204 /tpm/include/private/prototypes/SequenceComplete_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_SequenceComplete  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_SEQUENCECOMPLETE_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_SEQUENCECOMPLETE_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT    sequenceHandle;
12     TPM2B_MAX_BUFFER  buffer;
13     TPMI_RH_HIERARCHY hierarchy;
14 } SequenceComplete_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPM2B_DIGEST       result;
20     TPMT_TK_HASHCHECK  validation;
21 } SequenceComplete_Out;
22
23 // Response code modifiers
24 #  define RC_SequenceComplete_sequenceHandle (TPM_RC_H + TPM_RC_1)
25 #  define RC_SequenceComplete_buffer         (TPM_RC_P + TPM_RC_1)
26 #  define RC_SequenceComplete_hierarchy      (TPM_RC_P + TPM_RC_2)
27
28 // Function prototype
29 TPM_RC
30 TPM2_SequenceComplete(SequenceComplete_In* in, SequenceComplete_Out* out);
31
32 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_SEQUENCECOMPLETE_FP_H_
33 #endif  // CC_SequenceComplete

```

6.205 /tpm/include/private/prototypes/SequenceUpdate_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_SequenceUpdate  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_SEQUENCEUPDATE_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_SEQUENCEUPDATE_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT    sequenceHandle;
12     TPM2B_MAX_BUFFER  buffer;
13 } SequenceUpdate_In;
14
15 // Response code modifiers
16 #  define RC_SequenceUpdate_sequenceHandle (TPM_RC_H + TPM_RC_1)
17 #  define RC_SequenceUpdate_buffer         (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_SequenceUpdate(SequenceUpdate_In* in);
22
23 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_SEQUENCEUPDATE_FP_H_
24 #endif  // CC_SequenceUpdate

```


6.206 /tpm/include/private/prototypes/SessionProcess_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar  7, 2020   Time: 07:17:48PM
4  */
5
6  #ifndef _SESSION_PROCESS_FP_H_
7  #define _SESSION_PROCESS_FP_H_
8
9  /*** IsDAExempted()
10 // This function indicates if a handle is exempted from DA logic.
11 // A handle is exempted if it is:
12 //   a) a primary seed handle;
13 //   b) an object with noDA bit SET;
14 //   c) an NV Index with TPMA_NV_NO_DA bit SET; or
15 //   d) a PCR handle.
16 //
17 // Return Type: BOOL
18 //   TRUE(1)      handle is exempted from DA logic
19 //   FALSE(0)     handle is not exempted from DA logic
20 BOOL IsDAExempted(TPM_HANDLE handle // IN: entity handle
21 );
22
23 /*** ClearCpRphashes()
24 void ClearCpRphashes(COMMAND* command);
25
26 /*** CompareNameHash()
27 // This function computes the name hash and compares it to the nameHash in the
28 // session data, returning true if they are equal.
29 BOOL CompareNameHash(COMMAND* command, // IN: main parsing structure
30                     SESSION* session // IN: session structure with nameHash
31 );
32
33 /*** CompareParametersHash()
34 // This function computes the parameters hash and compares it to the pHash in
35 // the session data, returning true if they are equal.
36 BOOL CompareParametersHash(COMMAND* command, // IN: main parsing structure
37                           SESSION* session // IN: session structure with pHash
38 );
39
40 /*** ParseSessionBuffer()
41 // This function is the entry function for command session processing.
42 // It iterates sessions in session area and reports if the required authorization
43 // has been properly provided. It also processes audit session and passes the
44 // information of encryption sessions to parameter encryption module.
45 //
46 // Return Type: TPM_RC
47 //   various      parsing failure or authorization failure
48 //
49 TPM_RC
50 ParseSessionBuffer(COMMAND* command // IN: the structure that contains
51 );
52
53 /*** CheckAuthNoSession()
54 // Function to process a command with no session associated.
55 // The function makes sure all the handles in the command require no authorization.
56 //
57 // Return Type: TPM_RC
58 //   TPM_RC_AUTH_MISSING      failure - one or more handles require
59 //                             authorization
60 TPM_RC
61 CheckAuthNoSession(COMMAND* command // IN: command parsing structure
62 );
63
64 /*** BuildResponseSession()

```

```

65 // Function to build Session buffer in a response. The authorization data is added
66 // to the end of command->responseBuffer. The size of the authorization area is
67 // accumulated in command->authSize.
68 // When this is called, command->responseBuffer is pointing at the next location
69 // in the response buffer to be filled. This is where the authorization sessions
70 // will go, if any. command->parameterSize is the number of bytes that have been
71 // marshaled as parameters in the output buffer.
72 TPM_RC
73 BuildResponseSession(COMMAND* command // IN: structure that has relevant command
74 // information
75 );
76
77 /*** SessionRemoveAssociationToHandle()
78 // This function deals with the case where an entity associated with an authorization
79 // is deleted during command processing. The primary use of this is to support
80 // UndefineSpaceSpecial().
81 void SessionRemoveAssociationToHandle(TPM_HANDLE handle);
82
83 #endif // _SESSION_PROCESS_FP_H_

```

6.207 /tpm/include/private/prototypes/Session_fp.h

```

1  /*(Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar  4, 2020   Time: 02:36:44PM
4  */
5
6  #ifndef _SESSION_FP_H_
7  #define _SESSION_FP_H_
8
9  /*** Startup Function -- SessionStartup()
10 // This function initializes the session subsystem on TPM2_Startup().
11 BOOL SessionStartup(STARTUP_TYPE type);
12
13 /*** SessionIsLoaded()
14 // This function test a session handle references a loaded session. The handle
15 // must have previously been checked to make sure that it is a valid handle for
16 // an authorization session.
17 // NOTE: A PWAP authorization does not have a session.
18 //
19 // Return Type: BOOL
20 //     TRUE(1)      session is loaded
21 //     FALSE(0)     session is not loaded
22 //
23 BOOL SessionIsLoaded(TPM_HANDLE handle // IN: session handle
24 );
25
26 /*** SessionIsSaved()
27 // This function test a session handle references a saved session. The handle
28 // must have previously been checked to make sure that it is a valid handle for
29 // an authorization session.
30 // NOTE: An password authorization does not have a session.
31 //
32 // This function requires that the handle be a valid session handle.
33 //
34 // Return Type: BOOL
35 //     TRUE(1)      session is saved
36 //     FALSE(0)     session is not saved
37 //
38 BOOL SessionIsSaved(TPM_HANDLE handle // IN: session handle
39 );
40
41 /*** SequenceNumberForSavedContextIsValid()
42 // This function validates that the sequence number and handle value within a
43 // saved context are valid.

```

```

44  BOOL SequenceNumberForSavedContextIsValid(
45      TPMS_CONTEXT* context // IN: pointer to a context structure to be
46                          // validated
47  );
48
49  /*** SessionPCRValueIsCurrent()
50  //
51  // This function is used to check if PCR values have been updated since the
52  // last time they were checked in a policy session.
53  //
54  // This function requires the session is loaded.
55  // Return Type: BOOL
56  //     TRUE(1)          PCR value is current
57  //     FALSE(0)         PCR value is not current
58  BOOL SessionPCRValueIsCurrent(SESSION* session // IN: session structure
59  );
60
61  /*** SessionGet()
62  // This function returns a pointer to the session object associated with a
63  // session handle.
64  //
65  // The function requires that the session is loaded.
66  SESSION* SessionGet(TPM_HANDLE handle // IN: session handle
67  );
68
69  /*** SessionCreate()
70  //
71  // This function does the detailed work for starting an authorization session.
72  // This is done in a support routine rather than in the action code because
73  // the session management may differ in implementations. This implementation
74  // uses a fixed memory allocation to hold sessions and a fixed allocation
75  // to hold the contextID for the saved contexts.
76  //
77  // Return Type: TPM_RC
78  //     TPM_RC_CONTEXT_GAP          need to recycle sessions
79  //     TPM_RC_SESSION_HANDLE       active session space is full
80  //     TPM_RC_SESSION_MEMORY       loaded session space is full
81  TPM_RC
82  SessionCreate(TPM_SE      sessionType, // IN: the session type
83               TPMI_ALG_HASH authHash,  // IN: the hash algorithm
84               TPM2B_NONCE* nonceCaller, // IN: initial nonceCaller
85               TPMT_SYM_DEF* symmetric,   // IN: the symmetric algorithm
86               TPMI_DH_ENTITY bind,       // IN: the bind object
87               TPM2B_DATA* seed,          // IN: seed data
88               TPM_HANDLE* sessionHandle, // OUT: the session handle
89               TPM2B_NONCE* nonceTpm      // OUT: the session nonce
90  );
91
92  /*** SessionContextSave()
93  // This function is called when a session context is to be saved. The
94  // contextID of the saved session is returned. If no contextID can be
95  // assigned, then the routine returns TPM_RC_CONTEXT_GAP.
96  // If the function completes normally, the session slot will be freed.
97  //
98  // This function requires that 'handle' references a loaded session.
99  // Otherwise, it should not be called at the first place.
100 //
101 // Return Type: TPM_RC
102 //     TPM_RC_CONTEXT_GAP          a contextID could not be assigned
103 //     TPM_RC_TOO_MANY_CONTEXTS    the counter maxed out
104 //
105 TPM_RC
106 SessionContextSave(TPM_HANDLE handle, // IN: session handle
107                   CONTEXT_COUNTER* contextID // OUT: assigned contextID
108  );
109

```

```

110  /*** SessionContextLoad()
111  // This function is used to load a session from saved context. The session
112  // handle must be for a saved context.
113  //
114  // If the gap is at a maximum, then the only session that can be loaded is
115  // the oldest session, otherwise TPM_RC_CONTEXT_GAP is returned.
116  //
117  // This function requires that 'handle' references a valid saved session.
118  //
119  // Return Type: TPM_RC
120  //     TPM_RC_SESSION_MEMORY    no free session slots
121  //     TPM_RC_CONTEXT_GAP       the gap count is maximum and this
122  //                               is not the oldest saved context
123  //
124  TPM_RC
125  SessionContextLoad(SESSION_BUF* session, // IN: session structure from saved context
126                    TPM_HANDLE* handle    // IN/OUT: session handle
127  );
128
129  /*** SessionFlush()
130  // This function is used to flush a session referenced by its handle. If the
131  // session associated with 'handle' is loaded, the session array entry is
132  // marked as available.
133  //
134  // This function requires that 'handle' be a valid active session.
135  //
136  void SessionFlush(TPM_HANDLE handle // IN: loaded or saved session handle
137  );
138
139  /*** SessionComputeBoundEntity()
140  // This function computes the binding value for a session. The binding value
141  // for a reserved handle is the handle itself. For all the other entities,
142  // the authValue at the time of binding is included to prevent squatting.
143  // For those values, the Name and the authValue are concatenated
144  // into the bind buffer. If they will not both fit, they will be overlapped
145  // by XORing bytes. If XOR is required, the bind value will be full.
146  void SessionComputeBoundEntity(TPMI_DH_ENTITY entityHandle, // IN: handle of entity
147                                TPM2B_NAME* bind             // OUT: binding value
148  );
149
150  /*** SessionSetStartTime()
151  // This function is used to initialize the session timing
152  void SessionSetStartTime(SESSION* session // IN: the session to update
153  );
154
155  /*** SessionResetPolicyData()
156  // This function is used to reset the policy data without changing the nonce
157  // or the start time of the session.
158  void SessionResetPolicyData(SESSION* session // IN: the session to reset
159  );
160
161  /*** SessionCapGetLoaded()
162  // This function returns a list of handles of loaded session, started
163  // from input 'handle'
164  //
165  // 'Handle' must be in valid loaded session handle range, but does not
166  // have to point to a loaded session.
167  // Return Type: TPMI_YES_NO
168  //     YES      if there are more handles available
169  //     NO       all the available handles has been returned
170  TPMI_YES_NO
171  SessionCapGetLoaded(TPMI_SH_POLICY handle, // IN: start handle
172                    UINT32 count,           // IN: count of returned handles
173                    TPML_HANDLE* handleList // OUT: list of handle
174  );
175

```

```

176  /*** SessionCapGetOneLoaded()
177  // This function returns whether a session handle exists and is loaded.
178  BOOL SessionCapGetOneLoaded(TPMI_SH_POLICY handle // IN: handle
179  );
180
181  /*** SessionCapGetSaved()
182  // This function returns a list of handles for saved session, starting at
183  // 'handle'.
184  //
185  // 'Handle' must be in a valid handle range, but does not have to point to a
186  // saved session
187  //
188  // Return Type: TPMI_YES_NO
189  //     YES      if there are more handles available
190  //     NO       all the available handles has been returned
191  TPMI_YES_NO
192  SessionCapGetSaved(TPMI_SH_HMAC handle, // IN: start handle
193                    UINT32 count, // IN: count of returned handles
194                    TPML_HANDLE* handleList // OUT: list of handle
195  );
196
197  /*** SessionCapGetOneSaved()
198  // This function returns whether a session handle exists and is saved.
199  BOOL SessionCapGetOneSaved(TPMI_SH_HMAC handle // IN: handle
200  );
201
202  /*** SessionCapGetLoadedNumber()
203  // This function return the number of authorization sessions currently
204  // loaded into TPM RAM.
205  UINT32
206  SessionCapGetLoadedNumber(void);
207
208  /*** SessionCapGetLoadedAvail()
209  // This function returns the number of additional authorization sessions, of
210  // any type, that could be loaded into TPM RAM.
211  // NOTE: In other implementations, this number may just be an estimate. The only
212  // requirement for the estimate is, if it is one or more, then at least one
213  // session must be loadable.
214  UINT32
215  SessionCapGetLoadedAvail(void);
216
217  /*** SessionCapGetActiveNumber()
218  // This function returns the number of active authorization sessions currently
219  // being tracked by the TPM.
220  UINT32
221  SessionCapGetActiveNumber(void);
222
223  /*** SessionCapGetActiveAvail()
224  // This function returns the number of additional authorization sessions, of any
225  // type, that could be created. This not the number of slots for sessions, but
226  // the number of additional sessions that the TPM is capable of tracking.
227  UINT32
228  SessionCapGetActiveAvail(void);
229
230  #endif // _SESSION_FP_H_

```

6.208 /tpm/include/private/prototypes/SetAlgorithmSet_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_SetAlgorithmSet // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETALGORITHMSET_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETALGORITHMSET_FP_H_
7

```

```

8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PLATFORM authHandle;
12     UINT32            algorithmSet;
13 } SetAlgorithmSet_In;
14
15 // Response code modifiers
16 # define RC_SetAlgorithmSet_authHandle (TPM_RC_H + TPM_RC_1)
17 # define RC_SetAlgorithmSet_algorithmSet (TPM_RC_P + TPM_RC_1)
18
19 // Function prototype
20 TPM_RC
21 TPM2_SetAlgorithmSet(SetAlgorithmSet_In* in);
22
23 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETALGORITHMSET_FP_H_
24 #endif // CC_SetAlgorithmSet

```

6.209 /tpm/include/private/prototypes/SetCapability_fp.h

```

1 #if CC_SetCapability // Command must be enabled
2
3 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETCAPABILITY_FP_H_
4 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETCAPABILITY_FP_H_
5
6 // Input structure definition
7 typedef struct
8 {
9     TPMI_RH_HIERARCHY authHandle;
10    TPM2B_SET_CAPABILITY_DATA setCapabilityData;
11 } SetCapability_In;
12
13 // Response code modifiers
14 # define SetCapability_authHandle (TPM_RC_H + TPM_RC_1)
15 # define SetCapability_setCapabilityData (TPM_RC_P + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC TPM2_SetCapability(SetCapability_In* in);
19
20 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETCAPABILITY_FP_H_
21 #endif // CC_SetCapability

```

6.210 /tpm/include/private/prototypes/SetCommandCodeAuditStatus_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_SetCommandCodeAuditStatus // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETCOMMANDCODEAUDITSTATUS_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETCOMMANDCODEAUDITSTATUS_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_PROVISION auth;
12     TPMI_ALG_HASH      auditAlg;
13     TPML_CC            setList;
14     TPML_CC            clearList;
15 } SetCommandCodeAuditStatus_In;
16
17 // Response code modifiers
18 # define RC_SetCommandCodeAuditStatus_auth (TPM_RC_H + TPM_RC_1)
19 # define RC_SetCommandCodeAuditStatus_auditAlg (TPM_RC_P + TPM_RC_1)
20 # define RC_SetCommandCodeAuditStatus_setList (TPM_RC_P + TPM_RC_2)

```



```

21 # define RC_SetCommandCodeAuditStatus_clearList (TPM_RC_P + TPM_RC_3)
22
23 // Function prototype
24 TPM_RC
25 TPM2_SetCommandCodeAuditStatus(SetCommandCodeAuditStatus_In* in);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETCOMMANDCODEAUDITSTATUS_FP_H_
28 #endif // CC_SetCommandCodeAuditStatus

```

6.211 /tpm/include/private/prototypes/SetPrimaryPolicy_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_SetPrimaryPolicy // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETPRIMARYPOLICY_FP_H_
6 # define _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETPRIMARYPOLICY_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_RH_HIERARCHY_POLICY authHandle;
12     TPM2B_DIGEST authPolicy;
13     TPMI_ALG_HASH hashAlg;
14 } SetPrimaryPolicy_In;
15
16 // Response code modifiers
17 # define RC_SetPrimaryPolicy_authHandle (TPM_RC_H + TPM_RC_1)
18 # define RC_SetPrimaryPolicy_authPolicy (TPM_RC_P + TPM_RC_1)
19 # define RC_SetPrimaryPolicy_hashAlg (TPM_RC_P + TPM_RC_2)
20
21 // Function prototype
22 TPM_RC
23 TPM2_SetPrimaryPolicy(SetPrimaryPolicy_In* in);
24
25 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_SETPRIMARYPOLICY_FP_H_
26 #endif // CC_SetPrimaryPolicy

```

6.212 /tpm/include/private/prototypes/Shutdown_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Shutdown // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_SHUTDOWN_FP_H_
6 # define _TPM_INCLUDE_PRIVATE_PROTOTYPES_SHUTDOWN_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPM_SU shutdownType;
12 } Shutdown_In;
13
14 // Response code modifiers
15 # define RC_Shutdown_shutdownType (TPM_RC_P + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_Shutdown(Shutdown_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_SHUTDOWN_FP_H_
22 #endif // CC_Shutdown

```

6.213 /tpm/include/private/prototypes/Sign_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_Sign  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_SIGN_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_SIGN_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT    keyHandle;
12     TPM2B_DIGEST      digest;
13     TPMT_SIG_SCHEME   inScheme;
14     TPMT_TK_HASHCHECK validation;
15 } Sign_In;
16
17 // Output structure definition
18 typedef struct
19 {
20     TPMT_SIGNATURE signature;
21 } Sign_Out;
22
23 // Response code modifiers
24 #  define RC_Sign_keyHandle    (TPM_RC_H + TPM_RC_1)
25 #  define RC_Sign_digest      (TPM_RC_P + TPM_RC_1)
26 #  define RC_Sign_inScheme    (TPM_RC_P + TPM_RC_2)
27 #  define RC_Sign_validation  (TPM_RC_P + TPM_RC_3)
28
29 // Function prototype
30 TPM_RC
31 TPM2_Sign(Sign_In* in, Sign_Out* out);
32
33 #  endif  // _TPM_INCLUDE_PRIVATE_PROTOTYPES_SIGN_FP_H_
34 #endif  // CC_Sign

```

6.214 /tpm/include/private/prototypes/StartAuthSession_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_StartAuthSession  // Command must be enabled
4
5  #  ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_STARTAUTHSESSION_FP_H_
6  #    define _TPM_INCLUDE_PRIVATE_PROTOTYPES_STARTAUTHSESSION_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT    tpmKey;
12     TPMI_DH_ENTITY    bind;
13     TPM2B_NONCE        nonceCaller;
14     TPM2B_ENCRYPTED_SECRET encryptedSalt;
15     TPM_SE             sessionType;
16     TPMT_SYM_DEF        symmetric;
17     TPMI_ALG_HASH      authHash;
18 } StartAuthSession_In;
19
20 // Output structure definition
21 typedef struct
22 {
23     TPMI_SH_AUTH_SESSION sessionHandle;
24     TPM2B_NONCE          nonceTPM;
25 } StartAuthSession_Out;
26

```

```

27 // Response code modifiers
28 # define RC_StartAuthSession_tpmKey (TPM_RC_H + TPM_RC_1)
29 # define RC_StartAuthSession_bind (TPM_RC_H + TPM_RC_2)
30 # define RC_StartAuthSession_nonceCaller (TPM_RC_P + TPM_RC_1)
31 # define RC_StartAuthSession_encryptedSalt (TPM_RC_P + TPM_RC_2)
32 # define RC_StartAuthSession_sessionType (TPM_RC_P + TPM_RC_3)
33 # define RC_StartAuthSession_symmetric (TPM_RC_P + TPM_RC_4)
34 # define RC_StartAuthSession_authHash (TPM_RC_P + TPM_RC_5)
35
36 // Function prototype
37 TPM_RC
38 TPM2_StartAuthSession(StartAuthSession_In* in, StartAuthSession_Out* out);
39
40 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_STARTAUTHSESSION_FP_H_
41 #endif // CC_StartAuthSession

```

6.215 /tpm/include/private/prototypes/Startup_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_Startup // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_STARTUP_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_STARTUP_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPM_SU startupType;
12 } Startup_In;
13
14 // Response code modifiers
15 # define RC_Startup_startupType (TPM_RC_P + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_Startup(Startup_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_STARTUP_FP_H_
22 #endif // CC_Startup

```

6.216 /tpm/include/private/prototypes/StirRandom_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_StirRandom // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_STIRRANDOM_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_STIRRANDOM_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPM2B_SENSITIVE_DATA inData;
12 } StirRandom_In;
13
14 // Response code modifiers
15 # define RC_StirRandom_inData (TPM_RC_P + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_StirRandom(StirRandom_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_STIRRANDOM_FP_H_

```

```
22 #endif // CC_StirRandom
```

6.217 /tpm/include/private/prototypes/TableDrivenMarshal_fp.h

```
1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 4, 2020 Time: 02:36:44PM
4  */
5
6  #ifndef _TABLE_DRIVEN_MARSHAL_FP_H_
7  #define _TABLE_DRIVEN_MARSHAL_FP_H_
8
9  #if TABLE_DRIVEN_MARSHAL
10
11  /***UnmarshalUnion()
12  TPM_RC
13  UnmarshalUnion(UINT16  typeIndex, // IN: the thing to unmarshal
14                  void*   target,   // IN: where the data goes to
15                  UINT8**  buffer,   // IN/OUT: the data source buffer
16                  INT32*   size,     // IN/OUT: the remaining size
17                  UINT32   selector);
18
19  /*** MarshalUnion()
20  UINT16
21  MarshalUnion(UINT16  typeIndex, // IN: the thing to marshal
22               void*   source,    // IN: where the data comes from
23               UINT8**  buffer,    // IN/OUT: the data source buffer
24               INT32*   size,      // IN/OUT: the remaining size
25               UINT32   selector  // IN: the union selector
26  );
27
28  TPM_RC
29  UnmarshalInteger(int    iSize, // IN: Number of bytes in the integer
30                  void*   target, // OUT: receives the integer
31                  UINT8**  buffer, // IN/OUT: source of the data
32                  INT32*   size,   // IN/OUT: amount of data available
33                  UINT32*  value  // OUT: optional copy of 'target'
34  );
35
36  /*** Unmarshal()
37  // This is the function that performs unmarshaling of different numbered types. Each
38  // TPM type has a number. The number is used to lookup the address of the data
39  // structure that describes how to unmarshal that data type.
40  //
41  TPM_RC
42  Unmarshal(UINT16  typeIndex, // IN: the thing to marshal
43            void*   target,    // IN: where the data goes from
44            UINT8**  buffer,    // IN/OUT: the data source buffer
45            INT32*   size      // IN/OUT: the remaining size
46  );
47
48  /*** Marshal()
49  // This is the function that drives marshaling of output. Because there is no
50  // validation of the output, there is a lot less code.
51  UINT16 Marshal(UINT16  typeIndex, // IN: the thing to marshal
52                void*   source,    // IN: where the data comes from
53                UINT8**  buffer,    // IN/OUT: the data source buffer
54                INT32*   size      // IN/OUT: the remaining size
55  );
56 #endif // TABLE_DRIVEN_MARSHAL
57
58 #endif // _TABLE_DRIVEN_MARSHAL_FP_H_
```

6.218 /tpm/include/private/prototypes/TestParms_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_TestParms // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_TESTPARMS_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_TESTPARMS_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMT_PUBLIC_PARMS parameters;
12 } TestParms_In;
13
14 // Response code modifiers
15 #   define RC_TestParms_parameters (TPM_RC_P + TPM_RC_1)
16
17 // Function prototype
18 TPM_RC
19 TPM2_TestParms(TestParms_In* in);
20
21 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_TESTPARMS_FP_H_
22 #endif // CC_TestParms

```

6.219 /tpm/include/private/prototypes/Ticket_fp.h

```

1  /*(Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Mar 28, 2019 Time: 08:25:19PM
4   */
5
6  #ifndef _TICKET_FP_H_
7  #define _TICKET_FP_H_
8
9  /*** TicketIsSafe()
10 // This function indicates if producing a ticket is safe.
11 // It checks if the leading bytes of an input buffer is TPM_GENERATED_VALUE
12 // or its substring of canonical form. If so, it is not safe to produce ticket
13 // for an input buffer claiming to be TPM generated buffer
14 // Return Type: BOOL
15 //     TRUE(1)         safe to produce ticket
16 //     FALSE(0)        not safe to produce ticket
17 BOOL TicketIsSafe(TPM2B* buffer);
18
19 /*** TicketComputeVerified()
20 // This function creates a TPMT_TK_VERIFIED ticket.
21 TPM_RC TicketComputeVerified(
22     TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy constant for ticket
23     TPM2B_DIGEST* digest,       // IN: digest
24     TPM2B_NAME* keyName,        // IN: name of key that signed the values
25     TPMT_TK_VERIFIED* ticket    // OUT: verified ticket
26 );
27
28 /*** TicketComputeAuth()
29 // This function creates a TPMT_TK_AUTH ticket.
30 TPM_RC TicketComputeAuth(
31     TPM_ST type,                // IN: the type of ticket.
32     TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy constant for ticket
33     UINT64 timeout,             // IN: timeout
34     BOOL expiresOnReset,        // IN: flag to indicate if ticket expires on
35                                     // TPM Reset
36     TPM2B_DIGEST* cpHashA,      // IN: input cpHashA
37     TPM2B_NONCE* policyRef,     // IN: input policyRef
38     TPM2B_NAME* entityName,     // IN: name of entity

```

```

39     TPMT_TK_AUTH* ticket                // OUT: Created ticket
40 );
41
42 /** TicketComputeHashCheck()
43 // This function creates a TPMT_TK_HASHCHECK ticket.
44 TPM_RC TicketComputeHashCheck(
45     TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy constant for ticket
46     TPM_ALG_ID hashAlg, // IN: the hash algorithm for 'digest'
47     TPM2B_DIGEST* digest, // IN: input digest
48     TPMT_TK_HASHCHECK* ticket // OUT: Created ticket
49 );
50
51 /** TicketComputeCreation()
52 // This function creates a TPMT_TK_CREATION ticket.
53 TPM_RC TicketComputeCreation(TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy for ticket
54     TPM2B_NAME* name, // IN: object name
55     TPM2B_DIGEST* creation, // IN: creation hash
56     TPMT_TK_CREATION* ticket // OUT: created ticket
57 );
58
59 #endif // _TICKET_FP_H_

```

6.220 /tpm/include/private/prototypes/Time_fp.h

```

1  /* (Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Apr 2, 2019 Time: 04:23:27PM
4   */
5
6  #ifndef _TIME_FP_H_
7  #define _TIME_FP_H_
8
9  /** TimePowerOn()
10 // This function initialize time info at _TPM_Init().
11 //
12 // This function is called at _TPM_Init() so that the TPM time can start counting
13 // as soon as the TPM comes out of reset and doesn't have to wait until
14 // TPM2_Startup() in order to begin the new time epoch. This could be significant
15 // for systems that could get powered up but not run any TPM commands for some
16 // period of time.
17 //
18 void TimePowerOn(void);
19
20 /** TimeStartup()
21 // This function updates the resetCount and restartCount components of
22 // TPMS_CLOCK_INFO structure at TPM2_Startup().
23 //
24 // This function will deal with the deferred creation of a new epoch.
25 // TimeUpdateToCurrent() will not start a new epoch even if one is due when
26 // TPM2_Startup() has not been run. This is because the state of NV is not known
27 // until startup completes. When Startup is done, then it will create the epoch
28 // nonce to complete the initializations by calling this function.
29 BOOL TimeStartup(STARTUP_TYPE type // IN: start up type
30 );
31
32 /** TimeClockUpdate()
33 // This function updates go.clock. If 'newTime' requires an update of NV, then
34 // NV is checked for availability. If it is not available or is rate limiting, then
35 // go.clock is not updated and the function returns an error. If 'newTime' would
36 // not cause an NV write, then go.clock is updated. If an NV write occurs, then
37 // go.safe is SET.
38 void TimeClockUpdate(UINT64 newTime // IN: New time value in mS.
39 );
40
41 /** TimeUpdate()

```



```

42 // This function is used to update the time and clock values. If the TPM
43 // has run TPM2_Startup(), this function is called at the start of each command.
44 // If the TPM has not run TPM2_Startup(), this is called from TPM2_Startup() to
45 // get the clock values initialized. It is not called on command entry because, in
46 // this implementation, the go structure is not read from NV until TPM2_Startup().
47 // The reason for this is that the initialization code (_TPM_Init()) may run before
48 // NV is accessible.
49 void TimeUpdate(void);
50
51 /*** TimeUpdateToCurrent()
52 // This function updates the 'Time' and 'Clock' in the global
53 // TPMS_TIME_INFO structure.
54 //
55 // In this implementation, 'Time' and 'Clock' are updated at the beginning
56 // of each command and the values are unchanged for the duration of the
57 // command.
58 //
59 // Because 'Clock' updates may require a write to NV memory, 'Time' and 'Clock'
60 // are not allowed to advance if NV is not available. When clock is not advancing,
61 // any function that uses 'Clock' will fail and return TPM_RC_NV_UNAVAILABLE or
62 // TPM_RC_NV_RATE.
63 //
64 // This implementation does not do rate limiting. If the implementation does do
65 // rate limiting, then the 'Clock' update should not be inhibited even when doing
66 // rate limiting.
67 void TimeUpdateToCurrent(void);
68
69 /*** TimeSetAdjustRate()
70 // This function is used to perform rate adjustment on 'Time' and 'Clock'.
71 void TimeSetAdjustRate(TPM_CLOCK_ADJUST adjust // IN: adjust constant
72 );
73
74 /*** TimeGetMarshaled()
75 // This function is used to access TPMS_TIME_INFO in canonical form.
76 // The function collects the time information and marshals it into 'dataBuffer'
77 // and returns the marshaled size
78 UINT16
79 TimeGetMarshaled(TIME_INFO* dataBuffer // OUT: result buffer
80 );
81
82 /*** TimeFillInfo
83 // This function gathers information to fill in a TPMS_CLOCK_INFO structure.
84 void TimeFillInfo(TPMS_CLOCK_INFO* clockInfo);
85
86 #endif // _TIME_FP_H

```

6.221 /tpm/include/private/prototypes/TpmASN1_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Aug 30, 2019 Time: 02:11:54PM
4 */
5
6 #ifndef _TPM_ASN1_FP_H
7 #define _TPM_ASN1_FP_H
8
9 /*** ASN1UnmarshalContextInitialize()
10 // Function does standard initialization of a context.
11 // Return Type: BOOL
12 // TRUE(1) success
13 // FALSE(0) failure
14 BOOL ASN1UnmarshalContextInitialize(
15     ASN1UnmarshalContext* ctx, INT16 size, BYTE* buffer);
16
17 /***ASN1DecodeLength()

```

```

18 // This function extracts the length of an element from 'buffer' starting at 'offset'.
19 // Return Type: UINT16
20 //     >=0      the extracted length
21 //     <0      an error
22 INT16
23 ASN1DecodeLength(ASN1UnmarshalContext* ctx);
24
25 /***ASN1NextTag()
26 // This function extracts the next type from 'buffer' starting at 'offset'.
27 // It advances 'offset' as it parses the type and the length of the type. It returns
28 // the length of the type. On return, the 'length' octets starting at 'offset' are the
29 // octets of the type.
30 // Return Type: UINT
31 //     >=0      the number of octets in 'type'
32 //     <0      an error
33 INT16
34 ASN1NextTag(ASN1UnmarshalContext* ctx);
35
36 /*** ASN1GetBitStringValue()
37 // Try to parse a bit string of up to 32 bits from a value that is expected to be
38 // a bit string. The bit string is left justified so that the MSb of the input is
39 // the MSb of the returned value.
40 // If there is a general parsing error, the context->size is set to -1.
41 // Return Type: BOOL
42 //     TRUE(1)    success
43 //     FALSE(0)   failure
44 BOOL ASN1GetBitStringValue(ASN1UnmarshalContext* ctx, UINT32* val);
45
46 /*** ASN1InitialializeMarshalContext()
47 // This creates a structure for handling marshaling of an ASN.1 formatted data
48 // structure.
49 void ASN1InitialializeMarshalContext(
50     ASN1MarshalContext* ctx, INT16 length, BYTE* buffer);
51
52 /*** ASN1StartMarshalContext()
53 // This starts a new constructed element. It is constructed on 'top' of the value
54 // that was previously placed in the structure.
55 void ASN1StartMarshalContext(ASN1MarshalContext* ctx);
56
57 /*** ASN1EndMarshalContext()
58 // This function restores the end pointer for an encapsulating structure.
59 // Return Type: INT16
60 //     > 0      the size of the encapsulated structure that was just ended
61 //     <= 0     an error
62 INT16
63 ASN1EndMarshalContext(ASN1MarshalContext* ctx);
64
65 /***ASN1EndEncapsulation()
66 // This function puts a tag and length in the buffer. In this function, an embedded
67 // BIT_STRING is assumed to be a collection of octets. To indicate that all bits
68 // are used, a byte of zero is prepended. If a raw bit-string is needed, a new
69 // function like ASN1PushInteger() would be needed.
70 // Return Type: INT16
71 //     > 0      number of octets in the encapsulation
72 //     == 0     failure
73 UINT16
74 ASN1EndEncapsulation(ASN1MarshalContext* ctx, BYTE tag);
75
76 /*** ASN1PushByte()
77 BOOL ASN1PushByte(ASN1MarshalContext* ctx, BYTE b);
78
79 /*** ASN1PushBytes()
80 // Push some raw bytes onto the buffer. 'count' cannot be zero.
81 // Return Type: INT16
82 //     > 0      count bytes
83 //     == 0     failure unless count was zero

```

```

84  INT16
85  ASN1PushBytes(ASN1MarshalContext* ctx, INT16 count, const BYTE* buffer);
86
87  /*** ASN1PushNull()
88  // Return Type: INT16
89  //     > 0          count bytes
90  //     == 0          failure unless count was zero
91  INT16
92  ASN1PushNull(ASN1MarshalContext* ctx);
93
94  /*** ASN1PushLength()
95  // Push a length value. This will only handle length values that fit in an INT16.
96  // Return Type: UINT16
97  //     > 0          number of bytes added
98  //     == 0          failure
99  INT16
100  ASN1PushLength(ASN1MarshalContext* ctx, INT16 len);
101
102  /*** ASN1PushTagAndLength()
103  // Return Type: INT16
104  //     > 0          number of bytes added
105  //     == 0          failure
106  INT16
107  ASN1PushTagAndLength(ASN1MarshalContext* ctx, BYTE tag, INT16 length);
108
109  /*** ASN1PushTaggedOctetString()
110  // This function will push a random octet string.
111  // Return Type: INT16
112  //     > 0          number of bytes added
113  //     == 0          failure
114  INT16
115  ASN1PushTaggedOctetString(
116      ASN1MarshalContext* ctx, INT16 size, const BYTE* string, BYTE tag);
117
118  /*** ASN1PushUINT()
119  // This function pushes an native-endian integer value. This just changes a
120  // native-endian integer into a big-endian byte string and calls ASN1PushInteger().
121  // That function will remove leading zeros and make sure that the number is positive.
122  // Return Type: INT16
123  //     > 0          count bytes
124  //     == 0          failure unless count was zero
125  INT16
126  ASN1PushUINT(ASN1MarshalContext* ctx, UINT32 integer);
127
128  /*** ASN1PushInteger
129  // Push a big-endian integer on the end of the buffer
130  // Return Type: UINT16
131  //     > 0          the number of bytes marshaled for the integer
132  //     == 0          failure
133  INT16
134  ASN1PushInteger(ASN1MarshalContext* ctx,          // IN/OUT: buffer context
135                  INT16 iLen,                      // IN: octets of the integer
136                  BYTE* integer                    // IN: big-endian integer
137  );
138
139  /*** ASN1PushOID()
140  // This function is used to add an OID. An OID is 0x06 followed by a byte of size
141  // followed by size bytes. This is used to avoid having to do anything special in the
142  // definition of an OID.
143  // Return Type: UINT16
144  //     > 0          the number of bytes marshaled for the integer
145  //     == 0          failure
146  INT16
147  ASN1PushOID(ASN1MarshalContext* ctx, const BYTE* OID);
148
149  #endif // _TPM_ASN1_FP_H_

```

6.222 /tpm/include/private/prototypes/TpmEcc_Signature_ECDAF_fp.h

```

1  #ifndef _TPMECC_SIGNATURE_ECDAF_FP_H
2  #define _TPMECC_SIGNATURE_ECDAF_FP_H
3  #if ALG_ECC && ALG_ECDAF
4
5  /*** TpmEcc_SignEcdaf()
6  //
7  // This function performs 's' = 'r' + 'T' * 'd' mod 'q' where
8  // 1) 'r' is a random, or pseudo-random value created in the commit phase
9  // 2) 'nonceK' is a TPM-generated, random value 0 < 'nonceK' < 'n'
10 // 3) 'T' is mod 'q' of "Hash"('nonceK' || 'digest'), and
11 // 4) 'd' is a private key.
12 //
13 // The signature is the tuple ('nonceK', 's')
14 //
15 // Regrettably, the parameters in this function kind of collide with the parameter
16 // names used in ECSCNORR making for a lot of confusion.
17 // Return Type: TPM_RC
18 //     TPM_RC_SCHEME      unsupported hash algorithm
19 //     TPM_RC_NO_RESULT   cannot get values from random number generator
20 TPM_RC TpmEcc_SignEcdaf(
21     TPM2B_ECC_PARAMETER* nonceK, // OUT: 'nonce' component of the signature
22     Crypt_Int* bnS, // OUT: 's' component of the signature
23     const Crypt_EccCurve* E, // IN: the curve used in signing
24     Crypt_Int* bnD, // IN: the private key
25     const TPM2B_DIGEST* digest, // IN: the value to sign (mod 'q')
26     TPMT_ECC_SCHEME* scheme, // IN: signing scheme (contains the
27                             // commit count value).
28     OBJECT* eccKey, // IN: The signing key
29     RAND_STATE* rand // IN: a random number state
30 );
31
32 #endif // ALG_ECC && ALG_ECDAF
33 #endif // _TPMECC_SIGNATURE_ECDAF_FP_H

```

6.223 /tpm/include/private/prototypes/TpmEcc_Signature_ECDSA_fp.h

```

1  #ifndef _TPMECC_SIGNATURE_ECDSA_FP_H
2  #define _TPMECC_SIGNATURE_ECDSA_FP_H
3  #if ALG_ECC && ALG_ECDSA
4
5  /*** TpmEcc_SignEcdsa()
6  // This function implements the ECDSA signing algorithm. The method is described
7  // in the comments below.
8  TPM_RC
9  TpmEcc_SignEcdsa(Crypt_Int* bnR, // OUT: 'r' component of the signature
10                  Crypt_Int* bnS, // OUT: 's' component of the signature
11                  const Crypt_EccCurve* E, // IN: the curve used in the signature
12                                          // process
13                  Crypt_Int* bnD, // IN: private signing key
14                  const TPM2B_DIGEST* digest, // IN: the digest to sign
15                  RAND_STATE* rand // IN: used in debug of signing
16 );
17
18 /*** TpmEcc_ValidateSignatureEcdsa()
19 // This function validates an ECDSA signature. rIn and sIn should have been checked
20 // to make sure that they are in the range 0 < 'v' < 'n'
21 // Return Type: TPM_RC
22 //     TPM_RC_SIGNATURE      signature not valid
23 TPM_RC
24 TpmEcc_ValidateSignatureEcdsa(
25     Crypt_Int* bnR, // IN: 'r' component of the signature
26     Crypt_Int* bnS, // IN: 's' component of the signature
27     const Crypt_EccCurve* E, // IN: the curve used in the signature

```

```

28                                     // process
29     const Crypt_Point* ecQ, // IN: the public point of the key
30     const TPM2B_DIGEST* digest // IN: the digest that was signed
31 );
32
33 #endif // ALG_ECC && ALG_ECDSA
34 #endif // _TPMECC_SIGNATURE_ECDSA_FP_H_

```

6.224 /tpm/include/private/prototypes/TpmEcc_Signature_Schnorr_fp.h

```

1  #ifndef _TPMECC_SIGNATURE_SCHNORR_FP_H_
2  #define _TPMECC_SIGNATURE_SCHNORR_FP_H_
3
4  #if ALG_ECC && ALG_EC Schnorr
5  TPM_RC TpmEcc_SignEcSchnorr(
6      Crypt_Int* bnR, // OUT: 'r' component of the signature
7      Crypt_Int* bnS, // OUT: 's' component of the signature
8      const Crypt_EccCurve* E, // IN: the curve used in signing
9      Crypt_Int* bnD, // IN: the signing key
10     const TPM2B_DIGEST* digest, // IN: the digest to sign
11     TPM_ALG_ID hashAlg, // IN: signing scheme (contains a hash)
12     RAND_STATE* rand // IN: non-NULL when testing
13 );
14
15 /*** TpmEcc_ValidateSignatureEcSchnorr()
16 // This function is used to validate an EC Schnorr signature.
17 // Return Type: TPM_RC
18 // TPM_RC_SIGNATURE signature not valid
19 TPM_RC TpmEcc_ValidateSignatureEcSchnorr(
20     Crypt_Int* bnR, // IN: 'r' component of the signature
21     Crypt_Int* bnS, // IN: 's' component of the signature
22     TPM_ALG_ID hashAlg, // IN: hash algorithm of the signature
23     const Crypt_EccCurve* E, // IN: the curve used in the signature
24     // process
25     Crypt_Point* ecQ, // IN: the public point of the key
26     const TPM2B_DIGEST* digest // IN: the digest that was signed
27 );
28
29 #endif // ALG_ECC && ALG_EC Schnorr
30 #endif // _TPMECC_SIGNATURE_SCHNORR_FP_H_

```

6.225 /tpm/include/private/prototypes/TpmEcc_Signature_SM2_fp.h

```

1  #ifndef _TPMECC_SIGNATURE_SM2_FP_H_
2  #define _TPMECC_SIGNATURE_SM2_FP_H_
3
4  #if ALG_ECC && ALG_SM2
5  /*** TpmEcc_SignEcSm2()
6 // This function signs a digest using the method defined in SM2 Part 2. The method
7 // in the standard will add a header to the message to be signed that is a hash of
8 // the values that define the key. This then hashed with the message to produce a
9 // digest ('e'). This function signs 'e'.
10 // Return Type: TPM_RC
11 // TPM_RC_VALUE bad curve
12 TPM_RC TpmEcc_SignEcSm2(Crypt_Int* bnR, // OUT: 'r' component of the signature
13     Crypt_Int* bnS, // OUT: 's' component of the signature
14     const Crypt_EccCurve* E, // IN: the curve used in signing
15     Crypt_Int* bnD, // IN: the private key
16     const TPM2B_DIGEST* digest, // IN: the digest to sign
17     RAND_STATE* rand // IN: random number generator (mostly for
18     // debug)
19 );
20
21 /*** TpmEcc_ValidateSignatureEcSm2()

```



```

22 // This function is used to validate an SM2 signature.
23 // Return Type: TPM_RC
24 //     TPM_RC_SIGNATURE      signature not valid
25 TPM_RC TpmEcc_ValidateSignatureEcSm2(
26     Crypt_Int*    bnR, // IN: 'r' component of the signature
27     Crypt_Int*    bnS, // IN: 's' component of the signature
28     const Crypt_EccCurve* E, // IN: the curve used in the signature
29                             // process
30     Crypt_Point*   ecQ, // IN: the public point of the key
31     const TPM2B_DIGEST* digest // IN: the digest that was signed
32 );
33
34 #endif // ALG_ECC && ALG_SM2
35 #endif // _TPMECC_SIGNATURE_SM2_FP_H_

```

6.226 /tpm/include/private/prototypes/TpmEcc_Signature_Util_fp.h

```

1 // functions shared by multiple signature algorithms
2 #ifndef _TPMECC_SIGNATURE_UTIL_FP_H_
3 #define _TPMECC_SIGNATURE_UTIL_FP_H_
4
5 #if ALG_ECC
6 /*** TpmEcc_SchnorrCalculateS()
7 // This contains the Schnorr signature (S) computation. It is used by both ECDSA and
8 // Schnorr signing. The result is computed as: ['s' = 'k' + 'r' * 'd' (mod 'n')]
9 // where
10 // 1) 's' is the signature
11 // 2) 'k' is a random value
12 // 3) 'r' is the value to sign
13 // 4) 'd' is the private EC key
14 // 5) 'n' is the order of the curve
15 // Return Type: TPM_RC
16 //     TPM_RC_NO_RESULT      the result of the operation was zero or 'r' (mod 'n')
17 //                           is zero
18 TPM_RC TpmEcc_SchnorrCalculateS(
19     Crypt_Int*    bnS, // OUT: 's' component of the signature
20     const Crypt_Int* bnK, // IN: a random value
21     Crypt_Int*    bnR, // IN: the signature 'r' value
22     const Crypt_Int* bnD, // IN: the private key
23     const Crypt_Int* bnN // IN: the order of the curve
24 );
25
26 #endif // ALG_ECC
27 #endif // _TPMECC_SIGNATURE_UTIL_FP_H_

```

6.227 /tpm/include/private/prototypes/TpmEcc_Util_fp.h

```

1 #ifndef _TPMECC_UTIL_FP_H_
2 #define _TPMECC_UTIL_FP_H_
3
4 #if ALG_ECC
5
6 /*** TpmEcc_PointFrom2B()
7 // Function to create a Crypt_Point structure from a 2B point.
8 // This function doesn't take an Crypt_EccCurve for legacy reasons -
9 // this should probably be changed.
10 // returns NULL if the input value is invalid or doesn't fit.
11 LIB_EXPORT Crypt_Point* TpmEcc_PointFrom2B(
12     Crypt_Point*   ecP, // OUT: the preallocated point structure
13     TPMS_ECC_POINT* p // IN: the number to convert
14 );
15
16 /*** TpmEcc_PointTo2B()
17 // This function converts a Crypt_Point into a TPMS_ECC_POINT. A TPMS_ECC_POINT

```



```

18 // contains two TPM2B_ECC_PARAMETER values. The maximum size of the parameters
19 // is dependent on the maximum EC key size used in an implementation.
20 // The presumption is that the TPMS_ECC_POINT is large enough to hold 2 TPM2B
21 // values, each as large as a MAX_ECC_PARAMETER_BYTES
22 LIB_EXPORT BOOL TpmEcc_PointTo2B(
23     TPMS_ECC_POINT* p, // OUT: the converted 2B structure
24     const Crypt_Point* ecP, // IN: the values to be converted
25     const Crypt_EccCurve* E // IN: curve descriptor for the point
26 );
27
28 #endif // ALG_ECC
29 #endif // _TPMECC_UTIL_FP_H_

```

6.228 /tpm/include/private/prototypes/TpmMath_Debug_fp.h

```

1 //
2 // debug and test utilities. Not expected to be compiled into final products
3 #ifndef TPMMATH_DEBUG_FP_H_
4 #define TPMMATH_DEBUG_FP_H_
5
6 #if ALG_ECC || ALG_RSA
7
8 /*** TpmEccDebug_HexEqual()
9 // This function compares a bignum value to a hex string.
10 // using TpmEcc namespace because code assumes the max size
11 // is correct for ECC.
12 // Return Type: BOOL
13 // TRUE(1) values equal
14 // FALSE(0) values not equal
15 BOOL TpmMath_Debug_HexEqual(const Crypt_Int* bn, //IN: big number value
16                             const char* c //IN: character string number
17 );
18
19 LIB_EXPORT Crypt_Int* TpmMath_Debug_FromHex(
20     Crypt_Int* bn, // OUT:
21     const unsigned char* hex, // IN:
22     size_t maxsizeHex // IN: maximum size of hex
23 );
24
25 #endif // ALG_ECC or ALG_RSA
26 #endif // _TPMMATH_DEBUG_FP_H_

```

6.229 /tpm/include/private/prototypes/TpmMath_Util_fp.h

```

1 #ifndef TPM_MATH_FP_H_
2 #define TPM_MATH_FP_H_
3
4 /*** TpmMath_IntFrom2B()
5 // Convert an TPM2B to a Crypt_Int.
6 // If the input value does not exist, or the output does not exist, or the input
7 // will not fit into the output the function returns NULL
8 LIB_EXPORT Crypt_Int* TpmMath_IntFrom2B(Crypt_Int* value, // OUT:
9                                         const TPM2B* a2B // IN: number to convert
10 );
11
12 /*** TpmMath_IntTo2B()
13 //
14 // Function to convert a Crypt_Int to TPM2B. The TPM2B bytes are
15 // always in big-endian ordering (most significant byte first). If 'size' is
16 // non-zero and less than required by 'value' then an error is returned. If
17 // 'size' is non-zero and larger than 'value', the result buffer is padded
18 // with zeros. If 'size' is zero, then the TPM2B is assumed to be large enough
19 // for the data and a2b->size will be adjusted accordingly.
20 LIB_EXPORT BOOL TpmMath_IntTo2B(

```

```

21     const Crypt_Int* value, // IN: value to convert
22     TPM2B*          a2B,    // OUT: buffer for output
23     NUMBYTES        size    // IN: Size of output buffer - see comments.
24 );
25
26 /*** TpmMath_GetRandomBits()
27 // This function gets random bits for use in various places.
28 //
29 // One consequence of the generation scheme is that, if the number of bits requested
30 // is not a multiple of 8, then the high-order bits are set to zero. This would come
31 // into play when generating a 521-bit ECC key. A 66-byte (528-bit) value is
32 // generated and the high order 7 bits are masked off (CLEAR).
33 // In this situation, the highest order byte is the first byte (big-endian/TPM2B
34 // format)
35 // Return Type: BOOL
36 //     TRUE(1)      success
37 //     FALSE(0)     failure
38 LIB_EXPORT BOOL TpmMath_GetRandomBits(
39     BYTE*      pBuffer, // OUT: buffer to set
40     size_t     bits,    // IN: number of bits to generate (see remarks)
41     RAND_STATE* rand     // IN: random engine
42 );
43
44 /*** TpmMath_GetRandomInteger
45 // This function generates a random integer with the requested number of bits.
46 // Except for size, no range checking is performed.
47 // The maximum size that can be created is LARGEST_NUMBER + 64 bits.
48 // if either more bits, or the Crypt_Int* is too small to contain the requested bits
49 // the TPM enters failure mode and this function returns FALSE.
50 LIB_EXPORT BOOL TpmMath_GetRandomInteger(Crypt_Int* bn, // OUT: integer buffer to set
51     size_t     bits, // IN: size of output,
52     RAND_STATE* rand // IN: random engine
53 );
54
55 /*** TpmMath_GetRandomInRange()
56 // This function is used to generate a random number r in the range 1 <= r < limit.
57 // The function gets a random number of bits that is the size of limit. There is some
58 // some probability that the returned number is going to be greater than or equal
59 // to the limit. If it is, try again. There is no more than 50% chance that the
60 // next number is also greater, so try again. We keep trying until we get a
61 // value that meets the criteria. Since limit is very often a number with a LOT of
62 // high order ones, this rarely would need a second try.
63 // Return Type: BOOL
64 //     TRUE(1)      success
65 //     FALSE(0)     failure ('limit' is too small)
66 LIB_EXPORT BOOL TpmMath_GetRandomInRange(
67     Crypt_Int* dest, // OUT: integer buffer to set
68     const Crypt_Int* limit, // IN: limit (see remarks)
69     RAND_STATE* rand // IN: random engine
70 );
71 #endif // _TPM_MATH_FP_H_

```

6.230 /tpm/include/private/prototypes/TpmSizeChecks_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Oct 24, 2019   Time: 11:37:07AM
4  */
5
6  #ifndef _TPM_SIZE_CHECKS_FP_H_
7  #define _TPM_SIZE_CHECKS_FP_H_
8
9  #if RUNTIME_SIZE_CHECKS
10

```

```

11  /** TpmSizeChecks()
12  // This function is used during the development process to make sure that the
13  // vendor-specific values result in a consistent implementation. When possible,
14  // the code contains "#if" to do compile-time checks. However, in some cases, the
15  // values require the use of "sizeof()" and that can't be used in an #if.
16  BOOL TpmSizeChecks(void);
17  #endif // RUNTIME_SIZE_CHECKS
18
19  #endif // _TPM_SIZE_CHECKS_FP_H_

```

6.231 /tpm/include/private/prototypes/Unseal_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_Unseal // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_UNSEAL_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_UNSEAL_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPMI_DH_OBJECT itemHandle;
12 } Unseal_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPM2B_SENSITIVE_DATA outData;
18 } Unseal_Out;
19
20 // Response code modifiers
21 #   define RC_Unseal_itemHandle (TPM_RC_H + TPM_RC_1)
22
23 // Function prototype
24 TPM_RC
25 TPM2_Unseal(Unseal_In* in, Unseal_Out* out);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_UNSEAL_FP_H_
28 #endif // CC_Unseal

```

6.232 /tpm/include/private/prototypes/Vendor_TCG_Test_fp.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #if CC_Vendor_TCG_Test // Command must be enabled
4
5  # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_VENDOR_TCG_TEST_FP_H_
6  #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_VENDOR_TCG_TEST_FP_H_
7
8  // Input structure definition
9  typedef struct
10 {
11     TPM2B_DATA inputData;
12 } Vendor_TCG_Test_In;
13
14 // Output structure definition
15 typedef struct
16 {
17     TPM2B_DATA outputData;
18 } Vendor_TCG_Test_Out;
19
20 // Response code modifiers
21 #   define RC_Vendor_TCG_Test_inputData (TPM_RC_P + TPM_RC_1)

```

```

22
23 // Function prototype
24 TPM_RC
25 TPM2_Vendor_TCG_Test(Vendor_TCG_Test_In* in, Vendor_TCG_Test_Out* out);
26
27 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_VENDOR_TCG_TEST_FP_H_
28 #endif // CC_Vendor_TCG_Test

```

6.233 /tpm/include/private/prototypes/VerifySignature_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #if CC_VerifySignature // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_VERIFYSIGNATURE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_VERIFYSIGNATURE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT keyHandle;
12     TPM2B_DIGEST digest;
13     TPMT_SIGNATURE signature;
14 } VerifySignature_In;
15
16 // Output structure definition
17 typedef struct
18 {
19     TPMT_TK_VERIFIED validation;
20 } VerifySignature_Out;
21
22 // Response code modifiers
23 #   define RC_VerifySignature_keyHandle (TPM_RC_H + TPM_RC_1)
24 #   define RC_VerifySignature_digest (TPM_RC_P + TPM_RC_1)
25 #   define RC_VerifySignature_signature (TPM_RC_P + TPM_RC_2)
26
27 // Function prototype
28 TPM_RC
29 TPM2_VerifySignature(VerifySignature_In* in, VerifySignature_Out* out);
30
31 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_VERIFYSIGNATURE_FP_H_
32 #endif // CC_VerifySignature

```

6.234 /tpm/include/private/prototypes/X509_ECC_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Apr 2, 2019 Time: 11:00:49AM
4 */
5
6 #ifndef _X509_ECC_FP_H_
7 #define _X509_ECC_FP_H_
8
9 /*** X509PushPoint()
10 // This seems like it might be used more than once so...
11 // Return Type: INT16
12 // > 0 number of bytes added
13 // == 0 failure
14 INT16
15 X509PushPoint(ASN1MarshalContext* ctx, TPMS_ECC_POINT* p);
16
17 /*** X509AddSigningAlgorithmECC()
18 // This creates the signing algorithm data.
19 // Return Type: INT16

```

```

20 //      > 0          number of bytes added
21 //      == 0         failure
22 INT16
23 X509AddSigningAlgorithmECC(
24     OBJECT* signKey, TPMT_SIG_SCHEME* scheme, ASN1MarshalContext* ctx);
25
26 /*** X509AddPublicECC()
27 // This function will add the publicKey description to the DER data. If ctx is
28 // NULL, then no data is transferred and this function will indicate if the TPM
29 // has the values for DER-encoding of the public key.
30 // Return Type: INT16
31 //      > 0          number of bytes added
32 //      == 0         failure
33 INT16
34 X509AddPublicECC(OBJECT* object, ASN1MarshalContext* ctx);
35
36 #endif // _X509_ECC_FP_H_

```

6.235 /tpm/include/private/prototypes/X509_RSA_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Apr 2, 2019 Time: 11:00:49AM
4  */
5
6  #ifndef _X509_RSA_FP_H_
7  #define _X509_RSA_FP_H_
8
9  #if ALG_RSA
10
11  /*** X509AddSigningAlgorithmRSA()
12  // This creates the signing algorithm data.
13  // Return Type: INT16
14  //      > 0          number of bytes added
15  //      == 0         failure
16  INT16
17  X509AddSigningAlgorithmRSA(
18     OBJECT* signKey, TPMT_SIG_SCHEME* scheme, ASN1MarshalContext* ctx);
19
20  /*** X509AddPublicRSA()
21  // This function will add the publicKey description to the DER data. If fillPtr is
22  // NULL, then no data is transferred and this function will indicate if the TPM
23  // has the values for DER-encoding of the public key.
24  // Return Type: INT16
25  //      > 0          number of bytes added
26  //      == 0         failure
27  INT16
28  X509AddPublicRSA(OBJECT* object, ASN1MarshalContext* ctx);
29  #endif // ALG_RSA
30
31 #endif // _X509_RSA_FP_H_

```

6.236 /tpm/include/private/prototypes/X509_spt_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Nov 14, 2019 Time: 05:57:02PM
4  */
5
6  #ifndef _X509_SPT_FP_H_
7  #define _X509_SPT_FP_H_
8
9  /*** X509FindExtensionByOID()
10 // This will search a list of X509 extensions to find an extension with the

```

```

11 // requested OID. If the extension is found, the output context ('ctx') is set up
12 // to point to the OID in the extension.
13 // Return Type: BOOL
14 //     TRUE(1)          success
15 //     FALSE(0)         failure (could be catastrophic)
16 BOOL X509FindExtensionByOID(ASN1UnmarshalContext* ctxIn, // IN: the context to search
17                             ASN1UnmarshalContext* ctx, // OUT: the extension context
18                             const BYTE*          OID   // IN: oid to search for
19 );
20
21 /*** X509GetExtensionBits()
22 // This function will extract a bit field from an extension. If the extension doesn't
23 // contain a bit string, it will fail.
24 // Return Type: BOOL
25 //     TRUE(1)          success
26 //     FALSE(0)         failure
27 UINT32
28 X509GetExtensionBits(ASN1UnmarshalContext* ctx, UINT32* value);
29
30 /***X509ProcessExtensions()
31 // This function is used to process the TPMA_OBJECT and KeyUsage extensions. It is not
32 // in the CertifyX509.c code because it makes the code harder to follow.
33 // Return Type: TPM_RC
34 //     TPM_RC_ATTRIBUTES      the attributes of object are not consistent with
35 //                             the extension setting
36 //     TPM_RC_VALUE           problem parsing the extensions
37 TPM_RC
38 X509ProcessExtensions(
39     OBJECT* object, // IN: The object with the attributes to
40                     // check
41     stringRef* extension // IN: The start and length of the extensions
42 );
43
44 /*** X509AddSigningAlgorithm()
45 // This creates the signing algorithm data.
46 // Return Type: INT16
47 //     > 0          number of octets added
48 //     <= 0         failure
49 INT16
50 X509AddSigningAlgorithm(
51     ASN1MarshalContext* ctx, OBJECT* signKey, TPMT_SIG_SCHEME* scheme);
52
53 /*** X509AddPublicKey()
54 // This function will add the publicKey description to the DER data. If fillPtr is
55 // NULL, then no data is transferred and this function will indicate if the TPM
56 // has the values for DER-encoding of the public key.
57 // Return Type: INT16
58 //     > 0          number of octets added
59 //     == 0         failure
60 INT16
61 X509AddPublicKey(ASN1MarshalContext* ctx, OBJECT* object);
62
63 /*** X509PushAlgorithmIdentifierSequence()
64 // The function adds the algorithm identifier sequence.
65 // Return Type: INT16
66 //     > 0          number of bytes added
67 //     == 0         failure
68 INT16
69 X509PushAlgorithmIdentifierSequence(ASN1MarshalContext* ctx, const BYTE* OID);
70
71 #endif // _X509_SPT_FP_H_

```

6.237 /tpm/include/private/prototypes/ZGen_2Phase_fp.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT

```



```

2
3 #if CC_ZGen_2Phase // Command must be enabled
4
5 # ifndef _TPM_INCLUDE_PRIVATE_PROTOTYPES_ZGEN_2PHASE_FP_H_
6 #   define _TPM_INCLUDE_PRIVATE_PROTOTYPES_ZGEN_2PHASE_FP_H_
7
8 // Input structure definition
9 typedef struct
10 {
11     TPMI_DH_OBJECT      keyA;
12     TPM2B_ECC_POINT     inQsB;
13     TPM2B_ECC_POINT     inQeB;
14     TPMI_ECC_KEY_EXCHANGE inScheme;
15     UINT16              counter;
16 } ZGen_2Phase_In;
17
18 // Output structure definition
19 typedef struct
20 {
21     TPM2B_ECC_POINT outZ1;
22     TPM2B_ECC_POINT outZ2;
23 } ZGen_2Phase_Out;
24
25 // Response code modifiers
26 #   define RC_ZGen_2Phase_keyA      (TPM_RC_H + TPM_RC_1)
27 #   define RC_ZGen_2Phase_inQsB    (TPM_RC_P + TPM_RC_1)
28 #   define RC_ZGen_2Phase_inQeB    (TPM_RC_P + TPM_RC_2)
29 #   define RC_ZGen_2Phase_inScheme (TPM_RC_P + TPM_RC_3)
30 #   define RC_ZGen_2Phase_counter  (TPM_RC_P + TPM_RC_4)
31
32 // Function prototype
33 TPM_RC
34 TPM2_ZGen_2Phase(ZGen_2Phase_In* in, ZGen_2Phase_Out* out);
35
36 # endif // _TPM_INCLUDE_PRIVATE_PROTOTYPES_ZGEN_2PHASE_FP_H_
37 #endif // CC_ZGen_2Phase

```

6.238 /tpm/include/public/ACT.h

```

1 #ifndef _ACT_H_
2 #define _ACT_H_
3
4 #include <TpmConfiguration/TpmProfile.h>
5
6 #if ACT_SUPPORT
7     != (RH_ACT_0 | RH_ACT_1 | RH_ACT_2 | RH_ACT_3 | RH_ACT_4 | RH_ACT_5 | RH_ACT_6 \
8         | RH_ACT_7 | RH_ACT_8 | RH_ACT_9 | RH_ACT_A | RH_ACT_B | RH_ACT_C | RH_ACT_D \
9         | RH_ACT_E | RH_ACT_F)
10 # error "If ACT_SUPPORT == NO, no ACTs can be enabled"
11 #endif // (ACT_SUPPORT != ...)
12
13 #if !(defined RH_ACT_0) || (RH_ACT_0 != YES)
14 # undef RH_ACT_0
15 # define RH_ACT_0 NO
16 # define IF_ACT_0_IMPLEMENTED(op)
17 #else
18 # define IF_ACT_0_IMPLEMENTED(op) op(0)
19 #endif
20 #if !(defined RH_ACT_1) || (RH_ACT_1 != YES)
21 # undef RH_ACT_1
22 # define RH_ACT_1 NO
23 # define IF_ACT_1_IMPLEMENTED(op)
24 #else
25 # define IF_ACT_1_IMPLEMENTED(op) op(1)
26 #endif

```

```
27 #if !(defined RH_ACT_2) || (RH_ACT_2 != YES)
28 # undef RH_ACT_2
29 # define RH_ACT_2 NO
30 # define IF_ACT_2_IMPLEMENTED(op)
31 #else
32 # define IF_ACT_2_IMPLEMENTED(op) op(2)
33 #endif
34 #if !(defined RH_ACT_3) || (RH_ACT_3 != YES)
35 # undef RH_ACT_3
36 # define RH_ACT_3 NO
37 # define IF_ACT_3_IMPLEMENTED(op)
38 #else
39 # define IF_ACT_3_IMPLEMENTED(op) op(3)
40 #endif
41 #if !(defined RH_ACT_4) || (RH_ACT_4 != YES)
42 # undef RH_ACT_4
43 # define RH_ACT_4 NO
44 # define IF_ACT_4_IMPLEMENTED(op)
45 #else
46 # define IF_ACT_4_IMPLEMENTED(op) op(4)
47 #endif
48 #if !(defined RH_ACT_5) || (RH_ACT_5 != YES)
49 # undef RH_ACT_5
50 # define RH_ACT_5 NO
51 # define IF_ACT_5_IMPLEMENTED(op)
52 #else
53 # define IF_ACT_5_IMPLEMENTED(op) op(5)
54 #endif
55 #if !(defined RH_ACT_6) || (RH_ACT_6 != YES)
56 # undef RH_ACT_6
57 # define RH_ACT_6 NO
58 # define IF_ACT_6_IMPLEMENTED(op)
59 #else
60 # define IF_ACT_6_IMPLEMENTED(op) op(6)
61 #endif
62 #if !(defined RH_ACT_7) || (RH_ACT_7 != YES)
63 # undef RH_ACT_7
64 # define RH_ACT_7 NO
65 # define IF_ACT_7_IMPLEMENTED(op)
66 #else
67 # define IF_ACT_7_IMPLEMENTED(op) op(7)
68 #endif
69 #if !(defined RH_ACT_8) || (RH_ACT_8 != YES)
70 # undef RH_ACT_8
71 # define RH_ACT_8 NO
72 # define IF_ACT_8_IMPLEMENTED(op)
73 #else
74 # define IF_ACT_8_IMPLEMENTED(op) op(8)
75 #endif
76 #if !(defined RH_ACT_9) || (RH_ACT_9 != YES)
77 # undef RH_ACT_9
78 # define RH_ACT_9 NO
79 # define IF_ACT_9_IMPLEMENTED(op)
80 #else
81 # define IF_ACT_9_IMPLEMENTED(op) op(9)
82 #endif
83 #if !(defined RH_ACT_A) || (RH_ACT_A != YES)
84 # undef RH_ACT_A
85 # define RH_ACT_A NO
86 # define IF_ACT_A_IMPLEMENTED(op)
87 #else
88 # define IF_ACT_A_IMPLEMENTED(op) op(A)
89 #endif
90 #if !(defined RH_ACT_B) || (RH_ACT_B != YES)
91 # undef RH_ACT_B
92 # define RH_ACT_B NO
```

```

93  # define IF_ACT_B_IMPLEMENTED(op)
94  #else
95  # define IF_ACT_B_IMPLEMENTED(op) op(B)
96  #endif
97  #if !(defined RH_ACT_C) || (RH_ACT_C != YES)
98  # undef RH_ACT_C
99  # define RH_ACT_C NO
100 # define IF_ACT_C_IMPLEMENTED(op)
101 #else
102 # define IF_ACT_C_IMPLEMENTED(op) op(C)
103 #endif
104 #if !(defined RH_ACT_D) || (RH_ACT_D != YES)
105 # undef RH_ACT_D
106 # define RH_ACT_D NO
107 # define IF_ACT_D_IMPLEMENTED(op)
108 #else
109 # define IF_ACT_D_IMPLEMENTED(op) op(D)
110 #endif
111 #if !(defined RH_ACT_E) || (RH_ACT_E != YES)
112 # undef RH_ACT_E
113 # define RH_ACT_E NO
114 # define IF_ACT_E_IMPLEMENTED(op)
115 #else
116 # define IF_ACT_E_IMPLEMENTED(op) op(E)
117 #endif
118 #if !(defined RH_ACT_F) || (RH_ACT_F != YES)
119 # undef RH_ACT_F
120 # define RH_ACT_F NO
121 # define IF_ACT_F_IMPLEMENTED(op)
122 #else
123 # define IF_ACT_F_IMPLEMENTED(op) op(F)
124 #endif
125
126 #ifndef TPM_RH_ACT_0
127 # error Need numeric definition for TPM_RH_ACT_0
128 #endif
129
130 #ifndef TPM_RH_ACT_1
131 # define TPM_RH_ACT_1 (TPM_RH_ACT_0 + 1)
132 #endif
133 #ifndef TPM_RH_ACT_2
134 # define TPM_RH_ACT_2 (TPM_RH_ACT_0 + 2)
135 #endif
136 #ifndef TPM_RH_ACT_3
137 # define TPM_RH_ACT_3 (TPM_RH_ACT_0 + 3)
138 #endif
139 #ifndef TPM_RH_ACT_4
140 # define TPM_RH_ACT_4 (TPM_RH_ACT_0 + 4)
141 #endif
142 #ifndef TPM_RH_ACT_5
143 # define TPM_RH_ACT_5 (TPM_RH_ACT_0 + 5)
144 #endif
145 #ifndef TPM_RH_ACT_6
146 # define TPM_RH_ACT_6 (TPM_RH_ACT_0 + 6)
147 #endif
148 #ifndef TPM_RH_ACT_7
149 # define TPM_RH_ACT_7 (TPM_RH_ACT_0 + 7)
150 #endif
151 #ifndef TPM_RH_ACT_8
152 # define TPM_RH_ACT_8 (TPM_RH_ACT_0 + 8)
153 #endif
154 #ifndef TPM_RH_ACT_9
155 # define TPM_RH_ACT_9 (TPM_RH_ACT_0 + 9)
156 #endif
157 #ifndef TPM_RH_ACT_A
158 # define TPM_RH_ACT_A (TPM_RH_ACT_0 + 0xA)

```

```

159 #endif
160 #ifndef TPM_RH_ACT_B
161 # define TPM_RH_ACT_B (TPM_RH_ACT_0 + 0xB)
162 #endif
163 #ifndef TPM_RH_ACT_C
164 # define TPM_RH_ACT_C (TPM_RH_ACT_0 + 0xC)
165 #endif
166 #ifndef TPM_RH_ACT_D
167 # define TPM_RH_ACT_D (TPM_RH_ACT_0 + 0xD)
168 #endif
169 #ifndef TPM_RH_ACT_E
170 # define TPM_RH_ACT_E (TPM_RH_ACT_0 + 0xE)
171 #endif
172 #ifndef TPM_RH_ACT_F
173 # define TPM_RH_ACT_F (TPM_RH_ACT_0 + 0xF)
174 #endif
175
176 #define FOR_EACH_ACT(op) \
177     IF_ACT_0_IMPLEMENTED(op) \
178     IF_ACT_1_IMPLEMENTED(op) \
179     IF_ACT_2_IMPLEMENTED(op) \
180     IF_ACT_3_IMPLEMENTED(op) \
181     IF_ACT_4_IMPLEMENTED(op) \
182     IF_ACT_5_IMPLEMENTED(op) \
183     IF_ACT_6_IMPLEMENTED(op) \
184     IF_ACT_7_IMPLEMENTED(op) \
185     IF_ACT_8_IMPLEMENTED(op) \
186     IF_ACT_9_IMPLEMENTED(op) \
187     IF_ACT_A_IMPLEMENTED(op) \
188     IF_ACT_B_IMPLEMENTED(op) \
189     IF_ACT_C_IMPLEMENTED(op) \
190     IF_ACT_D_IMPLEMENTED(op) \
191     IF_ACT_E_IMPLEMENTED(op) \
192     IF_ACT_F_IMPLEMENTED(op)
193
194 // This is the mask for ACT that are implemented
195 // #define ACT_MASK(N) | (1 << 0x##N)
196 // #define ACT_IMPLEMENTED_MASK (0 FOR_EACH_ACT(ACT_MASK))
197
198 #define CASE_ACT_HANDLE(N) case TPM_RH_ACT_##N:
199 #define CASE_ACT_NUMBER(N) case 0x##N:
200
201 typedef struct ACT_STATE
202 {
203     UINT32    remaining;
204     TPM_ALG_ID hashAlg;
205     TPM2B_DIGEST authPolicy;
206 } ACT_STATE, *P_ACT_STATE;
207
208 #endif // _ACT_H_

```

6.239 /tpm/include/public/BaseTypes.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #ifndef _TPM_INCLUDE_PUBLIC_BASETYPES_H_
4 #define _TPM_INCLUDE_PUBLIC_BASETYPES_H_
5
6 // NULL definition
7 #ifndef NULL
8 # define NULL (0)
9 #endif // NULL
10
11 typedef uint8_t  UINT8;
12 typedef uint8_t  BYTE;

```

```

13 typedef int8_t    INT8;
14 typedef int      BOOL;
15 typedef uint16_t  UINT16;
16 typedef int16_t   INT16;
17 typedef uint32_t  UINT32;
18 typedef int32_t   INT32;
19 typedef uint64_t  UINT64;
20 typedef int64_t   INT64;
21
22 #endif // _TPM_INCLUDE_PUBLIC_BASETYPES_H_

```

6.240 /tpm/include/public/Capabilities.h

```

1 #ifndef _CAPABILITIES_H
2 #define _CAPABILITIES_H
3
4 #define MAX_CAP_DATA      (MAX_CAP_BUFFER - sizeof(TPM_CAP) - sizeof(UINT32))
5 #define MAX_CAP_ALGS      (MAX_CAP_DATA / sizeof(TPMS_ALG_PROPERTY))
6 #define MAX_CAP_HANDLES   (MAX_CAP_DATA / sizeof(TPM_HANDLE))
7 #define MAX_CAP_CC        (MAX_CAP_DATA / sizeof(TPM_CC))
8 #define MAX_TPM_PROPERTIES (MAX_CAP_DATA / sizeof(TPMS_TAGGED_PROPERTY))
9 #define MAX_PCR_PROPERTIES (MAX_CAP_DATA / sizeof(TPMS_TAGGED_PCR_SELECT))
10 #define MAX_ECC_CURVES    (MAX_CAP_DATA / sizeof(TPM_ECC_CURVE))
11 #define MAX_TAGGED_POLICIES (MAX_CAP_DATA / sizeof(TPMS_TAGGED_POLICY))
12 #define MAX_ACT_DATA      (MAX_CAP_DATA / sizeof(TPMS_ACT_DATA))
13 #define MAX_AC_CAPABILITIES (MAX_CAP_DATA / sizeof(TPMS_AC_OUTPUT))
14
15 #endif

```

6.241 /tpm/include/public/CompilerDependencies.h

```

1 // This file contains the build switches. This contains switches for multiple
2 // versions of the crypto-library so some may not apply to your environment.
3 //
4
5 #ifndef COMPILER_DEPENDENCIES_H_
6 #define COMPILER_DEPENDENCIES_H_
7
8 #if defined(__GNUC__)
9 # include "CompilerDependencies_gcc.h"
10 #elif defined(_MSC_VER)
11 # include "CompilerDependencies_msvc.h"
12 #else
13 # error unexpected
14 #endif
15
16 #include <stdint.h>
17
18 // Things that are not defined should be defined as NULL
19
20 #ifndef NORETURN
21 # define NORETURN
22 #endif
23 #ifndef LIB_EXPORT
24 # define LIB_EXPORT
25 #endif
26 #ifndef LIB_IMPORT
27 # define LIB_IMPORT
28 #endif
29 #ifndef REDUCE_WARNING_LEVEL
30 # define REDUCE_WARNING_LEVEL_(n)
31 #endif
32 #ifndef NORMAL_WARNING_LEVEL
33 # define NORMAL_WARNING_LEVEL_

```

```

34 #endif
35 #ifndef NOT_REFERENCED
36 # define NOT_REFERENCED(x) (x = x)
37 #endif
38
39 #ifdef _POSIX_
40 typedef int SOCKET;
41 #endif
42
43 #if !defined(TPM_STATIC_ASSERT) || !defined(COMPILER_CHECKS)
44 # error Expect definitions of COMPILER_CHECKS and TPM_STATIC_ASSERT
45 #elif COMPILER_CHECKS
46 // pre static_assert static_assert
47 # define MUST_BE(e) TPM_STATIC_ASSERT(e)
48
49 #else
50 // intentionally disabled, fine.
51 # define MUST_BE(e)
52 #endif
53
54 #endif // _COMPILER_DEPENDENCIES_H_

```

6.242 /tpm/include/public/CompilerDependencies_gcc.h

```

1 // This file contains compiler specific switches.
2 // These definitions are for the GCC compiler
3 //
4
5 #ifndef _COMPILER_DEPENDENCIES_GCC_H_
6 #define _COMPILER_DEPENDENCIES_GCC_H_
7
8 #if !defined(__GNUC__)
9 # error CompilerDependencies_gcc.h included for wrong compiler
10 #endif
11
12 // don't warn on unused local typedefs, they are used as a
13 // cross-compiler static_assert
14 #pragma GCC diagnostic push
15 #pragma GCC diagnostic ignored "-Wunused-local-typedefs"
16 #pragma GCC diagnostic pop
17
18 #undef _MSC_VER
19 #undef WIN32
20
21 #ifndef WINAPI
22 # define WINAPI
23 #endif
24 #ifndef __pragma
25 # define __pragma(x)
26 #endif
27 #define REVERSE_ENDIAN_16(_Number) __builtin_bswap16(_Number)
28 #define REVERSE_ENDIAN_32(_Number) __builtin_bswap32(_Number)
29 #define REVERSE_ENDIAN_64(_Number) __builtin_bswap64(_Number)
30
31 #define NORETURN __attribute__((noreturn))
32
33 #define TPM_INLINE inline __attribute__((always_inline))
34 #define TPM_STATIC_ASSERT(e) _Static_assert(e, "static assert")
35 #endif // _COMPILER_DEPENDENCIES_H_

```

6.243 /tpm/include/public/CompilerDependencies_msvc.h

```

1 // This file contains compiler specific switches.
2 // These definitions are for the Microsoft compiler

```



```

3  //
4
5  #ifndef _COMPILER_DEPENDENCIES_MSVC_H_
6  #define _COMPILER_DEPENDENCIES_MSVC_H_
7
8  #if !defined(_MSC_VER)
9  # error CompilerDependencies_msvc.h included for wrong compiler
10 #endif
11
12 // Endian conversion for aligned structures
13 #define REVERSE_ENDIAN_16( Number) _byteswap_ushort( Number)
14 #define REVERSE_ENDIAN_32( Number) _byteswap_ulong( Number)
15 #define REVERSE_ENDIAN_64( Number) _byteswap_uint64( Number)
16
17 // Avoid compiler warning for in line of stdio (or not)
18 // #define _NO_CRT_STDIO_INLINE
19
20 // This macro is used to handle LIB_EXPORT of function and variable names in lieu
21 // of a .def file. Visual Studio requires that functions be explicitly exported and
22 // imported.
23 #ifndef TPM_AS_DLL
24 # define LIB_EXPORT __declspec(dllexport) // VS compatible version
25 # define LIB_IMPORT __declspec(dllimport)
26 #else
27 // building static libraries
28 # define LIB_EXPORT
29 # define LIB_IMPORT
30 #endif
31
32 #define TPM_INLINE inline
33
34 // This is defined to indicate a function that does not return. Microsoft compilers
35 // do not support the _Noreturn function parameter.
36 #define NORETURN __declspec(noreturn)
37 #if _MSC_VER >= 1400 // SAL processing when needed
38 # include <sal.h>
39 #endif
40
41 // # ifdef _WIN64
42 // # define _INTPTR 2
43 // # else
44 // # define _INTPTR 1
45 // # endif
46
47 #define NOT_REFERENCED(x) (x)
48
49 // Lower the compiler error warning for system include
50 // files. They tend not to be that clean and there is no
51 // reason to sort through all the spurious errors that they
52 // generate when the normal error level is set to /Wall
53 #define REDUCE_WARNING_LEVEL(n) __pragma(warning(push, n))
54 // Restore the compiler warning level
55 #define NORMAL_WARNING_LEVEL __pragma(warning(pop))
56 #include <stdint.h>
57
58 #ifndef TPM_STATIC_ASSERT
59 # error TPM_STATIC_ASSERT already defined
60 #endif
61
62 // MSVC: failure results in error C2118: negative subscript error
63 #define TPM_STATIC_ASSERT(e) typedef char __C_ASSERT__[(e) ? 1 : -1]
64
65 #endif // _COMPILER_DEPENDENCIES_MSVC_H_

```

6.244 /tpm/include/public/endian_swap.h

```

1  #ifndef _SWAP_H
2  #define _SWAP_H
3
4  #if LITTLE_ENDIAN_TPM
5  # define TO_BIG_ENDIAN_UINT16(i)    REVERSE_ENDIAN_16(i)
6  # define FROM_BIG_ENDIAN_UINT16(i)  REVERSE_ENDIAN_16(i)
7  # define TO_BIG_ENDIAN_UINT32(i)    REVERSE_ENDIAN_32(i)
8  # define FROM_BIG_ENDIAN_UINT32(i)  REVERSE_ENDIAN_32(i)
9  # define TO_BIG_ENDIAN_UINT64(i)    REVERSE_ENDIAN_64(i)
10 # define FROM_BIG_ENDIAN_UINT64(i)  REVERSE_ENDIAN_64(i)
11 #else
12 # define TO_BIG_ENDIAN_UINT16(i)    (i)
13 # define FROM_BIG_ENDIAN_UINT16(i)  (i)
14 # define TO_BIG_ENDIAN_UINT32(i)    (i)
15 # define FROM_BIG_ENDIAN_UINT32(i)  (i)
16 # define TO_BIG_ENDIAN_UINT64(i)    (i)
17 # define FROM_BIG_ENDIAN_UINT64(i)  (i)
18 #endif
19
20 #if AUTO_ALIGN == NO
21
22 // The aggregation macros for machines that do not allow unaligned access or for
23 // little-endian machines.
24
25 // Aggregate bytes into a UINT
26
27 # define BYTE_ARRAY_TO_UINT8(b)      (uint8_t)((b)[0])
28 # define BYTE_ARRAY_TO_UINT16(b)     ByteArrayToUint16((BYTE*)(b))
29 # define BYTE_ARRAY_TO_UINT32(b)     ByteArrayToUint32((BYTE*)(b))
30 # define BYTE_ARRAY_TO_UINT64(b)     ByteArrayToUint64((BYTE*)(b))
31 # define UINT8_TO_BYTE_ARRAY(i, b)   ((b)[0] = (uint8_t)(i))
32 # define UINT16_TO_BYTE_ARRAY(i, b)  Uint16ToByteArray((i), (BYTE*)(b))
33 # define UINT32_TO_BYTE_ARRAY(i, b)  Uint32ToByteArray((i), (BYTE*)(b))
34 # define UINT64_TO_BYTE_ARRAY(i, b)  Uint64ToByteArray((i), (BYTE*)(b))
35
36 #else // AUTO_ALIGN
37
38 # if BIG_ENDIAN_TPM
39 // the big-endian macros for machines that allow unaligned memory access
40 // Aggregate a byte array into a UINT
41 # define BYTE_ARRAY_TO_UINT8(b)      *((uint8_t*)(b))
42 # define BYTE_ARRAY_TO_UINT16(b)     *((uint16_t*)(b))
43 # define BYTE_ARRAY_TO_UINT32(b)     *((uint32_t*)(b))
44 # define BYTE_ARRAY_TO_UINT64(b)     *((uint64_t*)(b))
45
46 // Disaggregate a UINT into a byte array
47
48 # define UINT8_TO_BYTE_ARRAY(i, b) \
49 { \
50     *((uint8_t*)(b)) = (i); \
51 }
52 # define UINT16_TO_BYTE_ARRAY(i, b) \
53 { \
54     *((uint16_t*)(b)) = (i); \
55 }
56 # define UINT32_TO_BYTE_ARRAY(i, b) \
57 { \
58     *((uint32_t*)(b)) = (i); \
59 }
60 # define UINT64_TO_BYTE_ARRAY(i, b) \
61 { \
62     *((uint64_t*)(b)) = (i); \
63 }
64 # else

```

```

65 // the little endian macros for machines that allow unaligned memory access
66 // the big-endian macros for machines that allow unaligned memory access
67 // Aggregate a byte array into a UINT
68 # define BYTE_ARRAY_TO_UINT8(b) *((uint8_t*)(b))
69 # define BYTE_ARRAY_TO_UINT16(b) REVERSE_ENDIAN_16(*((uint16_t*)(b)))
70 # define BYTE_ARRAY_TO_UINT32(b) REVERSE_ENDIAN_32(*((uint32_t*)(b)))
71 # define BYTE_ARRAY_TO_UINT64(b) REVERSE_ENDIAN_64(*((uint64_t*)(b)))
72
73 // Disaggregate a UINT into a byte array
74
75 # define UINT8_TO_BYTE_ARRAY(i, b) \
76 { \
77     *((uint8_t*)(b)) = (i); \
78 }
79 # define UINT16_TO_BYTE_ARRAY(i, b) \
80 { \
81     *((uint16_t*)(b)) = REVERSE_ENDIAN_16(i); \
82 }
83 # define UINT32_TO_BYTE_ARRAY(i, b) \
84 { \
85     *((uint32_t*)(b)) = REVERSE_ENDIAN_32(i); \
86 }
87 # define UINT64_TO_BYTE_ARRAY(i, b) \
88 { \
89     *((uint64_t*)(b)) = REVERSE_ENDIAN_64(i); \
90 }
91 # endif // BIG_ENDIAN_TPM
92
93 #endif // AUTO_ALIGN == NO
94
95 #endif // _SWAP_H

```

6.245 /tpm/include/public/GpMacros.h

```

1 /** Introduction
2 // This file is a collection of miscellaneous macros.
3
4 #ifndef GP_MACROS_H
5 #define GP_MACROS_H
6
7 #ifndef NULL
8 # define NULL 0
9 #endif
10
11 #include "endian_swap.h"
12 #include <TpmConfiguration/VendorInfo.h>
13
14 /** For Self-test
15 // These macros are used in CryptUtil to invoke the incremental self test.
16 #if SELF_TEST
17 # define TEST(alg) \
18     do \
19     { \
20         if(TEST_BIT(alg, g_toTest)) \
21             CryptTestAlgorithm(alg, NULL); \
22     } while(0)
23 #else
24 # define TEST(alg)
25 #endif // SELF_TEST
26
27 /** For Failures
28 #if defined _POSIX_
29 # define FUNCTION_NAME 0
30 #else
31 # define FUNCTION_NAME __FUNCTION__

```

```

32 #endif
33
34 #if defined(FAIL_TRACE) && FAIL_TRACE != 0
35 # define CODELOCATOR() FUNCTION_NAME, __LINE__
36 #else // !FAIL_TRACE
37 // if provided, use the definition of CODELOCATOR from TpmConfiguration so
38 // implementor can customize this.
39 # ifndef CODELOCATOR
40 # define CODELOCATOR() 0
41 # endif
42 #endif // FAIL_TRACE
43
44 // SETFAILED calls TpmFail. It may or may not return based on the NO_LONGJMP flag.
45 // CODELOCATOR is a macro that expands to either one 64-bit value that encodes the
46 // location, or two parameters: Function Name and Line Number.
47 #define SETFAILED(errorCode) (TpmFail(CODELOCATOR(), errorCode))
48
49 // If implementation is using longjmp, then calls to TpmFail() will never
50 // return. However, without longjmp facility, TpmFail will return while most of
51 // the code currently expects FAIL() calls to immediately abort the current
52 // command. If they don't, some commands return success instead of failure. The
53 // family of macros below are provided to allow the code to be modified to
54 // correctly propagate errors correctly, based on the context.
55 //
56 // * Some functions, particularly the ECC crypto have state cleanup at the end
57 // of the function and need to use the goto Exit pattern.
58 // * Other functions return TPM_RC values, which should return TPM_RC_FAILURE
59 // * Still other functions return an isOK boolean and need to return FALSE.
60 //
61 // if longjmp is available, all these macros just call SETFAILED and immediately
62 // abort. Note any of these approaches could leak memory if the crypto adapter
63 // libraries are using dynamic memory.
64 //
65 // FAIL vs. FAIL_NORET
66 // =====
67 // Be cautious with these macros. FAIL_NORET is intended as an affirmation
68 // that the upstream code calling the function using this macro has been
69 // investigated to confirm that upstream functions correctly handle this
70 // function putting the TPM into failure mode without returning an error.
71 //
72 // The TPM library was originally written with a lot of error checking omitted,
73 // which means code occurring after a FAIL macro may not expect to be called
74 // when the TPM is in failure mode. When NO_LONGJMP is false (the system has a
75 // longjmp API), then none of that code is executed because the sample platform
76 // sets up longjmp before calling ExecuteCommand. However, in the NO_LONGJMP
77 // case, code following a FAIL or FAIL_NORET macro will get run. The
78 // conservative assumption is that code is untested and may be unsafe in such a
79 // situation. FAIL_NORET can replace FAIL when the code has been reviewed to
80 // ensure the post-FAIL code is safe. Of course, this is a point-in-time
81 // assertion that is only true when the FAIL_NORET macro is first inserted;
82 // hence it is better to use one of the early-exit macros to immediately return.
83 // However, the necessary return-code plumbing may be large and FAIL/FAIL_NORET
84 // are provided to support gradual improvement over time.
85
86 #ifndef NO_LONGJMP
87 // has longjmp
88 // necessary to reference Exit, even though the code is no-return
89 # define TPM_FAIL_RETURN NORETURN void
90
91 // see discussion above about FAIL/FAIL_NORET
92 # define FAIL(failCode) SETFAILED(failCode)
93 # define FAIL_NORET(failCode) SETFAILED(failCode)
94 # define FAIL_IMMEDIATE(failCode, retval) SETFAILED(failCode)
95 # define FAIL_BOOL(failCode) SETFAILED(failCode)
96 # define FAIL_RC(failCode) SETFAILED(failCode)
97 # define FAIL_VOID(failCode) SETFAILED(failCode)

```

```

98  # define FAIL_NULL(failCode)          SETFAILED(failCode)
99  # define FAIL_EXIT(failCode, returnVar, returnCode) \
100      do                                \
101      {                                  \
102          SETFAILED(failCode);          \
103          goto Exit;                    \
104      } while(0)
105
106  #else // NO_LONGJMP
107  // no longjmp service is available
108  # define TPM_FAIL_RETURN      void
109
110  // This macro is provided for existing code and should not be used in new code.
111  // see discussion above.
112  # define FAIL(failCode)        FAIL_NORET(failCode)
113
114  // Be cautious with this macro, see discussion above.
115  # define FAIL_NORET(failCode) SETFAILED(failCode)
116
117  // fail and immediately return void
118  # define FAIL_VOID(failCode) \
119      do                        \
120      {                        \
121          SETFAILED(failCode); \
122          return;              \
123      } while(0)
124
125  // fail and immediately return a value
126  # define FAIL_IMMEDIATE(failCode, retval) \
127      do                                    \
128      {                                    \
129          SETFAILED(failCode);             \
130          return retval;                   \
131      } while(0)
132
133  // fail and return FALSE
134  # define FAIL_BOOL(failCode) FAIL_IMMEDIATE(failCode, FALSE)
135
136  // fail and return TPM_RC_FAILURE
137  # define FAIL_RC(failCode)   FAIL_IMMEDIATE(failCode, TPM_RC_FAILURE)
138
139  // fail and return NULL
140  # define FAIL_NULL(failCode) FAIL_IMMEDIATE(failCode, NULL)
141
142  // fail and return using the goto exit pattern
143  # define FAIL_EXIT(failCode, returnVar, returnCode) \
144      do                                              \
145      {                                              \
146          SETFAILED(failCode);                      \
147          returnVar = returnCode;                   \
148          goto Exit;                                \
149      } while(0)
150
151  #endif
152
153  // This macro tests that a condition is TRUE and puts the TPM into failure mode
154  // if it is not. If longjmp is being used, then the macro makes a call from
155  // which there is no return. Otherwise, the function will return the given
156  // return code.
157  #define VERIFY(condition, failCode, returnCode) \
158      do                                          \
159      {                                          \
160          if(!(condition))                     \
161          {                                     \
162              FAIL_IMMEDIATE(failCode, returnCode); \
163          }                                     \

```

```

164     } while(0)
165
166 // this function also verifies a condition and enters failure mode, but sets a
167 // return value and jumps to Exit on failure - allowing for cleanup.
168 #define VERIFY_OR_EXIT(condition, failCode, returnVar, returnCode) \
169     do \
170     { \
171         if(!(condition)) \
172         { \
173             FAIL_EXIT(failCode, returnVar, returnCode); \
174         } \
175     } while(0)
176
177 // verify the given TPM_RC is success and we are not in
178 // failure mode. Otherwise, return immediately with TPM_RC_FAILURE.
179 // note that failure mode is checked first so that an existing FATAL_* error code
180 // is not overwritten with the default from this macro.
181 #define VERIFY_RC(rc) \
182     do \
183     { \
184         if(g_inFailureMode) \
185         { \
186             return TPM_RC_FAILURE; \
187         } \
188         if(rc != TPM_RC_SUCCESS) \
189         { \
190             FAIL_IMMEDIATE(FATAL_ERROR_ASSERT, TPM_RC_FAILURE); \
191         } \
192     } while(0)
193
194 // verify the TPM is not in failure mode or return failure
195 #define VERIFY_NOT_FAILED() \
196     do \
197     { \
198         if(g_inFailureMode) \
199         { \
200             return TPM_RC_FAILURE; \
201         } \
202     } while(0)
203
204 // Enter failure mode if the given TPM_RC is not success, return void.
205 #define VERIFY_RC_VOID(rc) \
206     do \
207     { \
208         if(g_inFailureMode) \
209         { \
210             return; \
211         } \
212         if(rc != TPM_RC_SUCCESS) \
213         { \
214             FAIL_VOID(FATAL_ERROR_ASSERT); \
215         } \
216     } while(0)
217
218 // These VERIFY_CRYPTO macros all set failure mode to FATAL_ERROR_CRYPTO
219 // and immediately return. The general way to parse the names is:
220 // VERIFY_CRYPTO_[conditionType]_[OR_EXIT]_[retValType]
221 // if conditionType is omitted, it is taken as BOOL.
222 // Without OR_EXIT, implies an immediate return. Thus VERIFY_CRYPTO_BOOL:
223 // 1. check fn against TRUE
224 // 2. if false, set failure mode to FATAL_ERROR_CRYPTO
225 // 3. immediately return FALSE.
226 // and, VERIFY_CRYPTO_OR_EXIT_RC translates to:
227 // 1. Check a BOOL
228 // 2. If false, set failure mode with FATAL_ERROR_CRYPTO,
229 // 3. assume retVal is type TPM_RC, set it to TPM_RC_FAILURE

```



```

230 // 4. Goto Exit
231 // while VERIFY_CRYPTO_RC_OR_EXIT translates to:
232 // 1. Check fn result against TPM_RC_SUCCESS
233 // 2. if not equal, set failure mode to FATAL_ERROR_CRYPTO
234 // 3. assume retVal is type TPM_RC, set it to TPM_RC_FAILURE
235 // 4. Goto Exit.
236 #define VERIFY_CRYPTO(fn) VERIFY((fn), FATAL_ERROR_CRYPTO, TPM_RC_FAILURE)
237
238 #define VERIFY_CRYPTO_BOOL(fn) VERIFY((fn), FATAL_ERROR_CRYPTO, FALSE)
239
240 #define VERIFY_CRYPTO_OR_NULL(fn) VERIFY((fn), FATAL_ERROR_CRYPTO, NULL)
241
242 // these VERIFY_CRYPTO macros all set a result value and goto Exit
243 #define VERIFY_CRYPTO_OR_EXIT(fn, returnVar, returnCode) \
244     VERIFY_OR_EXIT(fn, FATAL_ERROR_CRYPTO, returnVar, returnCode);
245
246 // these VERIFY_CRYPTO_OR_EXIT functions assume the return value variable is
247 // named retVal
248 #define VERIFY_CRYPTO_OR_EXIT_RC(fn) \
249     VERIFY_CRYPTO_OR_EXIT_GENERIC(fn, retVal, TPM_RC_FAILURE)
250
251 #define VERIFY_CRYPTO_OR_EXIT_FALSE(fn) \
252     VERIFY_CRYPTO_OR_EXIT_GENERIC(fn, retVal, FALSE)
253
254 #define VERIFY_CRYPTO_RC_OR_EXIT(fn) \
255     do \
256     { \
257         TPM_RC rc = fn; \
258         if(rc != TPM_RC_SUCCESS) \
259         { \
260             FAIL_EXIT(FATAL_ERROR_CRYPTO, retVal, rc); \
261         } \
262     } while(0)
263
264 #if defined EMPTY_ASSERT) && (EMPTY_ASSERT != NO)
265 # define pAssert(a) ((void)0)
266 #else
267 # define pAssert(a) \
268     do \
269     { \
270         if(!(a)) \
271             FAIL(FATAL_ERROR_PARAMETER); \
272     } while(0)
273
274 # define pAssert_ZERO(a) \
275     do \
276     { \
277         if(!(a)) \
278             FAIL_IMMEDIATE(FATAL_ERROR_ASSERT, 0); \
279     } while(0);
280
281 # define pAssert_RC(a) \
282     do \
283     { \
284         if(!(a)) \
285             FAIL_RC(FATAL_ERROR_ASSERT); \
286     } while(0);
287
288 # define pAssert_BOOL(a) \
289     do \
290     { \
291         if(!(a)) \
292             FAIL_BOOL(FATAL_ERROR_ASSERT); \
293     } while(0);
294
295 # define pAssert_NULL(a) \

```

```

296         do                                     \
297         {                                     \
298             if(!(a))                         \
299                 FAIL_NULL(FATAL_ERROR_ASSERT); \
300         } while(0);
301
302 // using FAIL_NORET isn't optimum but is available in limited cases that
303 // result in wrong calculated values, and can be checked later
304 // but should have no vulnerability implications.
305 # define pAssert_NORET(a)                       \
306     {                                           \
307         if(!(a))                               \
308             FAIL_NORET(FATAL_ERROR_ASSERT); \
309     }
310
311 // this macro is used where a calling code has been verified to function correctly
312 // when the failing assert immediately returns without an error code.
313 // this can be because either the caller checks the fatal error flag, or
314 // the state is safe and a higher-level check will catch it.
315 # define pAssert_VOID_OK(a)                     \
316     {                                           \
317         if(!(a))                               \
318             FAIL_VOID(FATAL_ERROR_ASSERT); \
319     }
320
321 #endif
322
323 // These macros are commonly used in the "Crypt" code as a way to keep listings from
324 // getting too long. This is not to save paper but to allow one to see more
325 // useful stuff on the screen at any given time. Neither macro sets failure mode.
326 #define ERROR_EXIT(returnCode) \
327     do                          \
328     {                          \
329         retVal = returnCode; \
330         goto Exit;          \
331     } while(0)
332
333 // braces are necessary for this usage:
334 // if (y)
335 //     GOTO_ERROR_UNLESS(x)
336 // else ...
337 // without braces the else would attach to the GOTO macro instead of the
338 // outer if statement; given the amount of TPM code that doesn't use braces on
339 // if statements, this is a live risk.
340 #define GOTO_ERROR_UNLESS(_X) \
341     do                        \
342     {                        \
343         if(!(_X))           \
344             goto Error; \
345     } while(0)
346
347 #include "public/MinMax.h"
348
349 #ifndef IsOdd
350 # define IsOdd(a) ((a)&1) != 0
351 #endif
352
353 #ifndef BITS_TO_BYTES
354 # define BITS_TO_BYTES(bits) (((bits) + 7) >> 3)
355 #endif
356
357 // These are defined for use when the size of the vector being checked is known
358 // at compile time.
359 #define TEST_BIT(bit, vector) TestBit((bit), (BYTE*)&(vector), sizeof(vector))
360 #define SET_BIT(bit, vector) SetBit((bit), (BYTE*)&(vector), sizeof(vector))
361 #define CLEAR_BIT(bit, vector) ClearBit((bit), (BYTE*)&(vector), sizeof(vector))

```

```

362
363 // The following definitions are used if they have not already been defined. The
364 // defaults for these settings are compatible with ISO/IEC 9899:2011 (E)
365 #ifndef LIB_EXPORT
366 # define LIB_EXPORT
367 # define LIB_IMPORT
368 #endif
369 #ifndef NORETURN
370 # define NORETURN _Noreturn
371 #endif
372 #ifndef NOT_REFERENCED
373 # define NOT_REFERENCED(x = x) ((void)(x))
374 #endif
375
376 #define STD_RESPONSE_HEADER (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC))
377
378 // This bit is used to indicate that an authorization ticket expires on TPM Reset
379 // and TPM Restart. It is added to the timeout value returned by TPM2_PolicySigned()
380 // and TPM2_PolicySecret() and used by TPM2_PolicyTicket(). The timeout value is
381 // relative to Time (g_time). Time is reset whenever the TPM loses power and cannot
382 // be moved forward by the user (as can Clock). 'g_time' is a 64-bit value expressing
383 // time in ms. Stealing the MSb for a flag means that the TPM needs to be reset
384 // at least once every 292,471,208 years rather than once every 584,942,417 years.
385 #define EXPIRATION_BIT ((UINT64)1 << 63)
386
387 // Check for consistency of the bit ordering of bit fields
388 #if BIG_ENDIAN_TPM && MOST_SIGNIFICANT_BIT_0 && USE_BIT_FIELD_STRUCTURES
389 # error "Settings not consistent"
390 #endif
391
392 // These macros are used to handle the variation in handling of bit fields. If
393 #if USE_BIT_FIELD_STRUCTURES // The default, old version, with bit fields
394 # define IS_ATTRIBUTE(a, type, b) ((a.b) != 0)
395 # define SET_ATTRIBUTE(a, type, b) (a.b = SET)
396 # define CLEAR_ATTRIBUTE(a, type, b) (a.b = CLEAR)
397 # define GET_ATTRIBUTE(a, type, b) (a.b)
398 # define TPMA_ZERO_INITIALIZER() \
399 { \
400     0 \
401 }
402 #else
403 # define IS_ATTRIBUTE(a, type, b) ((a & type##_##b) != 0)
404 # define SET_ATTRIBUTE(a, type, b) (a |= type##_##b)
405 # define CLEAR_ATTRIBUTE(a, type, b) (a &= ~type##_##b)
406 # define GET_ATTRIBUTE(a, type, b) (type)((a & type##_##b) >> type##_##b##_SHIFT)
407 # define TPMA_ZERO_INITIALIZER() (0)
408 #endif
409
410 // These macros determine if the values in this file are referenced or instantiated.
411 // Global.c defines GLOBAL_C so all the values in this file will be instantiated in
412 // Global.obj. For all other files that include this file, the values will simply
413 // be external references. For constants, there can be an initializer.
414 #ifndef EXTERN
415 # ifdef GLOBAL_C
416 # define EXTERN
417 # else
418 # define EXTERN extern
419 # endif
420 #endif // EXTERN
421
422 #ifndef GLOBAL_C
423 # define INITIALIZER(_value_) = _value_
424 #else
425 # define INITIALIZER(_value_)
426 #endif
427

```

```

428 // This macro will create an OID. All OIDs are in DER form with a first octet of
429 // 0x06 indicating an OID followed by an octet indicating the number of octets in the
430 // rest of the OID. This allows a user of this OID to know how much/little to copy.
431 #define MAKE_OID(NAME) EXTERN const BYTE OID##NAME[] INITIALIZER({OID##NAME##_VALUE})
432
433 // This definition is moved from TpmProfile.h because it is not actually vendor-
434 // specific. It has to be the same size as the 'sequence' parameter of a TPMS_CONTEXT
435 // and that is a UINT64. So, this is an invariant value
436 #define CONTEXT_COUNTER UINT64
437
438 #include "public/TpmCalculatedAttributes.h"
439
440 #endif // GP_MACROS_H

```

6.246 /tpm/include/public/MinMax.h

```

1  #ifndef _MIN_MAX_H_
2  #define _MIN_MAX_H_
3
4  #ifndef MAX
5  # define MAX(a, b) ((a) > (b) ? (a) : (b))
6  #endif
7  #ifndef MIN
8  # define MIN(a, b) ((a) < (b) ? (a) : (b))
9  #endif
10
11 #ifndef SIZEOF_MEMBER
12 # define SIZEOF_MEMBER(type, member) sizeof(((type*)0)->member)
13 #endif
14
15 #endif // _MIN_MAX_H_

```

6.247 /tpm/include/public/TpmAlgorithmDefines.h

```

1  // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3  #ifndef TPM_INCLUDE_PRIVATE_TPMALGORITHMDEFINES_H_
4  #define TPM_INCLUDE_PRIVATE_TPMALGORITHMDEFINES_H_
5
6  #include <TpmConfiguration/TpmProfile.h>
7  #include "public/MinMax.h"
8  #include "public/TPMB.h"
9
10 // Table "Defines for NIST_P192 ECC Values" (TCG Algorithm Registry)
11 #define NIST_P192_ID TPM_ECC_NIST_P192
12 #define NIST_P192_KEY_SIZE 192
13
14 // Table "Defines for NIST_P224 ECC Values" (TCG Algorithm Registry)
15 #define NIST_P224_ID TPM_ECC_NIST_P224
16 #define NIST_P224_KEY_SIZE 224
17
18 // Table "Defines for NIST_P256 ECC Values" (TCG Algorithm Registry)
19 #define NIST_P256_ID TPM_ECC_NIST_P256
20 #define NIST_P256_KEY_SIZE 256
21
22 // Table "Defines for NIST_P384 ECC Values" (TCG Algorithm Registry)
23 #define NIST_P384_ID TPM_ECC_NIST_P384
24 #define NIST_P384_KEY_SIZE 384
25
26 // Table "Defines for NIST_P521 ECC Values" (TCG Algorithm Registry)
27 #define NIST_P521_ID TPM_ECC_NIST_P521
28 #define NIST_P521_KEY_SIZE 521
29
30 // Table "Defines for BN_P256 ECC Values" (TCG Algorithm Registry)

```

```

31 #define BN_P256_ID          TPM_ECC_BN_P256
32 #define BN_P256_KEY_SIZE 256
33
34 // Table "Defines for BN_P638 ECC Values" (TCG Algorithm Registry)
35 #define BN_P638_ID          TPM_ECC_BN_P638
36 #define BN_P638_KEY_SIZE 638
37
38 // Table "Defines for SM2_P256 ECC Values" (TCG Algorithm Registry)
39 #define SM2_P256_ID          TPM_ECC_SM2_P256
40 #define SM2_P256_KEY_SIZE 256
41
42 // Table "Defines for BP_P256_R1 ECC Values" (TCG Algorithm Registry)
43 #define BP_P256_R1_ID          TPM_ECC_BP_P256_R1
44 #define BP_P256_R1_KEY_SIZE 256
45
46 // Table "Defines for BP_P384_R1 ECC Values" (TCG Algorithm Registry)
47 #define BP_P384_R1_ID          TPM_ECC_BP_P384_R1
48 #define BP_P384_R1_KEY_SIZE 384
49
50 // Table "Defines for BP_P512_R1 ECC Values" (TCG Algorithm Registry)
51 #define BP_P512_R1_ID          TPM_ECC_BP_P512_R1
52 #define BP_P512_R1_KEY_SIZE 512
53
54 // Table "Defines for CURVE_25519 ECC Values" (TCG Algorithm Registry)
55 #define CURVE_25519_ID          TPM_ECC_CURVE_25519
56 #define CURVE_25519_KEY_SIZE 256
57
58 // Table "Defines for CURVE_448 ECC Values" (TCG Algorithm Registry)
59 #define CURVE_448_ID          TPM_ECC_CURVE_448
60 #define CURVE_448_KEY_SIZE 448
61
62 // Table "Defines for SHA1 Hash Values" (TCG Algorithm Registry)
63 #define SHA1_DIGEST_SIZE 20
64 #define SHA1_BLOCK_SIZE 64
65
66 // Table "Defines for SHA256 Hash Values" (TCG Algorithm Registry)
67 #define SHA256_DIGEST_SIZE 32
68 #define SHA256_BLOCK_SIZE 64
69
70 // Table "Defines for SHA384 Hash Values" (TCG Algorithm Registry)
71 #define SHA384_DIGEST_SIZE 48
72 #define SHA384_BLOCK_SIZE 128
73
74 // Table "Defines for SHA512 Hash Values" (TCG Algorithm Registry)
75 #define SHA512_DIGEST_SIZE 64
76 #define SHA512_BLOCK_SIZE 128
77
78 // Table "Defines for SM3_256 Hash Values" (TCG Algorithm Registry)
79 #define SM3_256_DIGEST_SIZE 32
80 #define SM3_256_BLOCK_SIZE 64
81
82 // Table "Defines for SHA3_256 Hash Values" (TCG Algorithm Registry)
83 #define SHA3_256_DIGEST_SIZE 32
84 #define SHA3_256_BLOCK_SIZE 136
85
86 // Table "Defines for SHA3_384 Hash Values" (TCG Algorithm Registry)
87 #define SHA3_384_DIGEST_SIZE 48
88 #define SHA3_384_BLOCK_SIZE 104
89
90 // Table "Defines for SHA3_512 Hash Values" (TCG Algorithm Registry)
91 #define SHA3_512_DIGEST_SIZE 64
92 #define SHA3_512_BLOCK_SIZE 72
93
94 // Table "Defines for RSA Asymmetric Cipher Algorithm Constants" (TCG Algorithm
Registry)
95 #define RSA_KEY_SIZES_BITS

```



```

96     (RSA_1024 * 1024), (RSA_2048 * 2048), (RSA_3072 * 3072), (RSA_4096 * 4096), \
97     (RSA_16384 * 16384)
98 #define RSA_MAX_KEY_SIZE_BITS \
99     MAX((RSA_16384 * 16384), \
100        MAX((RSA_4096 * 4096), \
101            MAX((RSA_3072 * 3072), MAX((RSA_2048 * 2048), (RSA_1024 * 1024)))))
102 #define MAX_RSA_KEY_BITS RSA_MAX_KEY_SIZE_BITS
103 #define MAX_RSA_KEY_BYTES BITS_TO_BYTES(MAX_RSA_KEY_BITS)
104
105 // Table "Defines for AES Symmetric Cipher Algorithm Constants" (TCG Algorithm
106 Registry)
107 #define AES_KEY_SIZES_BITS (AES_128 * 128), (AES_192 * 192), (AES_256 * 256)
108 #define AES_MAX_KEY_SIZE_BITS \
109     MAX((AES_256 * 256), MAX((AES_192 * 192), (AES_128 * 128)))
110 #define MAX_AES_KEY_BITS AES_MAX_KEY_SIZE_BITS
111 #define MAX_AES_KEY_BYTES BITS_TO_BYTES(MAX_AES_KEY_BITS)
112 #define AES_BLOCK_SIZES (AES_128 * 128 / 8), (AES_192 * 128 / 8), (AES_256 * 128 /
113 8)
114 #define MAX_AES_BLOCK_SIZE_BYTES \
115     MAX((AES_256 * 128 / 8), MAX((AES_192 * 128 / 8), (AES_128 * 128 / 8)))
116 #define AES_MAX_BLOCK_SIZE MAX_AES_BLOCK_SIZE_BYTES
117
118 // Table "Defines for SM4 Symmetric Cipher Algorithm Constants" (TCG Algorithm
119 Registry)
120 #define SM4_KEY_SIZES_BITS (SM4_128 * 128)
121 #define SM4_MAX_KEY_SIZE_BITS (SM4_128 * 128)
122 #define MAX_SM4_KEY_BITS SM4_MAX_KEY_SIZE_BITS
123 #define MAX_SM4_KEY_BYTES BITS_TO_BYTES(MAX_SM4_KEY_BITS)
124 #define SM4_BLOCK_SIZES (SM4_128 * 128 / 8)
125 #define MAX_SM4_BLOCK_SIZE_BYTES (SM4_128 * 128 / 8)
126 #define SM4_MAX_BLOCK_SIZE MAX_SM4_BLOCK_SIZE_BYTES
127
128 // Table "Defines for CAMELLIA Symmetric Cipher Algorithm Constants" (TCG Algorithm
129 Registry)
130 #define CAMELLIA_KEY_SIZES_BITS \
131     (CAMELLIA_128 * 128), (CAMELLIA_192 * 192), (CAMELLIA_256 * 256)
132 #define CAMELLIA_MAX_KEY_SIZE_BITS \
133     MAX((CAMELLIA_256 * 256), MAX((CAMELLIA_192 * 192), (CAMELLIA_128 * 128)))
134 #define MAX_CAMELLIA_KEY_BITS CAMELLIA_MAX_KEY_SIZE_BITS
135 #define MAX_CAMELLIA_KEY_BYTES BITS_TO_BYTES(MAX_CAMELLIA_KEY_BITS)
136 #define CAMELLIA_BLOCK_SIZES \
137     (CAMELLIA_128 * 128 / 8), (CAMELLIA_192 * 128 / 8), (CAMELLIA_256 * 128 / 8)
138 #define MAX_CAMELLIA_BLOCK_SIZE_BYTES \
139     MAX((CAMELLIA_256 * 128 / 8), \
140         MAX((CAMELLIA_192 * 128 / 8), (CAMELLIA_128 * 128 / 8)))
141 #define CAMELLIA_MAX_BLOCK_SIZE MAX_CAMELLIA_BLOCK_SIZE_BYTES
142
143 // Derived ECC Value
144 #define ECC_CURVES
145 {
146     TPM_ECC_NIST_P192, TPM_ECC_NIST_P224, TPM_ECC_NIST_P256, TPM_ECC_NIST_P384, \
147     TPM_ECC_NIST_P521, TPM_ECC_BN_P256, TPM_ECC_BN_P638, TPM_ECC_SM2_P256, \
148     TPM_ECC_BP_P256_R1, TPM_ECC_BP_P384_R1, TPM_ECC_BP_P512_R1, \
149     TPM_ECC_CURVE_25519, TPM_ECC_CURVE_448
150 }
151 #define ECC_CURVE_COUNT
152 (ECC_NIST_P192 + ECC_NIST_P224 + ECC_NIST_P256 + ECC_NIST_P384 + ECC_NIST_P521 \
153 + ECC_BN_P256 + ECC_BN_P638 + ECC_SM2_P256 + ECC_BP_P256_R1 + ECC_BP_P384_R1 \
154 + ECC_BP_P512_R1 + ECC_CURVE_25519 + ECC_CURVE_448)
155 #define MAX_ECC_KEY_BITS
156 MAX((CURVE_448_KEY_SIZE * ECC_CURVE_448), \
157     MAX((CURVE_25519_KEY_SIZE * ECC_CURVE_25519), \
158         MAX((BP_P512_R1_KEY_SIZE * ECC_BP_P512_R1), \
159             MAX((BP_P384_R1_KEY_SIZE * ECC_BP_P384_R1), \
160                 MAX((BP_P256_R1_KEY_SIZE * ECC_BP_P256_R1), \
161                     MAX((SM2_P256_KEY_SIZE * ECC_SM2_P256), \

```



```

158             MAX((BN_P638_KEY_SIZE * ECC_BN_P638), \
159             MAX((BN_P256_KEY_SIZE * ECC_BN_P256), \
160             MAX((NIST_P521_KEY_SIZE * ECC_NIST_P521), \
161             MAX((NIST_P384_KEY_SIZE * ECC_NIST_P384), \
162             MAX((NIST_P256_KEY_SIZE \
163             * ECC_NIST_P256), \
164             MAX((NIST_P224_KEY_SIZE \
165             * ECC_NIST_P224), \
166             (NIST_P192_KEY_SIZE \
167             * ECC_NIST_P192)))))))))
168 #define MAX_ECC_KEY_BYTES ((MAX_ECC_KEY_BITS + 7) / 8)
169 // Derived Hash Values
170 #define HASH_COUNT \
171     (ALG_SHA1 + ALG_SHA256 + ALG_SHA384 + ALG_SHA512 + ALG_SM3_256 + ALG_SHA3_256 \
172     + ALG_SHA3_384 + ALG_SHA3_512)
173 #define MAX_HASH_BLOCK_SIZE \
174     MAX((ALG_SHA3_512 * SHA3_512_BLOCK_SIZE), \
175     MAX((ALG_SHA3_384 * SHA3_384_BLOCK_SIZE), \
176     MAX((ALG_SHA3_256 * SHA3_256_BLOCK_SIZE), \
177     MAX((ALG_SM3_256 * SM3_256_BLOCK_SIZE), \
178     MAX((ALG_SHA512 * SHA512_BLOCK_SIZE), \
179     MAX((ALG_SHA384 * SHA384_BLOCK_SIZE), \
180     MAX((ALG_SHA256 * SHA256_BLOCK_SIZE), \
181     (ALG_SHA1 * SHA1_BLOCK_SIZE))))))
182 #define MAX_HASH_DIGEST_SIZE \
183     MAX((ALG_SHA3_512 * SHA3_512_DIGEST_SIZE), \
184     MAX((ALG_SHA3_384 * SHA3_384_DIGEST_SIZE), \
185     MAX((ALG_SHA3_256 * SHA3_256_DIGEST_SIZE), \
186     MAX((ALG_SM3_256 * SM3_256_DIGEST_SIZE), \
187     MAX((ALG_SHA512 * SHA512_DIGEST_SIZE), \
188     MAX((ALG_SHA384 * SHA384_DIGEST_SIZE), \
189     MAX((ALG_SHA256 * SHA256_DIGEST_SIZE), \
190     (ALG_SHA1 * SHA1_DIGEST_SIZE))))))
191 #define MAX_DIGEST_SIZE MAX_HASH_DIGEST_SIZE
192
193 #if MAX_HASH_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
194 # error "Hash data not valid"
195 #endif
196
197 // Define the 2B structure that would hold any hash block
198 TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
199
200 // Following typedef is for some old code
201 typedef TPM2B_MAX_HASH_BLOCK TPM2B_HASH_BLOCK;
202 // Derived Symmetric Values
203 #define SYM_COUNT ALG_AES + ALG_SM4 + ALG_CAMELLIA
204 #define MAX_SYM_BLOCK_SIZE \
205     MAX(CAMELLIA_MAX_BLOCK_SIZE, MAX(SM4_MAX_BLOCK_SIZE, AES_MAX_BLOCK_SIZE))
206 #define MAX_SYM_KEY_BITS \
207     MAX(CAMELLIA_MAX_KEY_SIZE_BITS, MAX(SM4_MAX_KEY_SIZE_BITS, AES_MAX_KEY_SIZE_BITS))
208 #define MAX_SYM_KEY_BYTES ((MAX_SYM_KEY_BITS + 7) / 8)
209
210 #endif // _TPM_INCLUDE_PRIVATE_TPMALGORITHMDEFINES_H

```

6.248 /tpm/include/public/TPMB.h

```

1 //
2 // This file contains extra TPM2B structures
3 //
4
5 #ifndef TPMB_H
6 #define TPMB_H
7
8 /*** Size Types
9 // These types are used to differentiate the two different size values used.

```

```

10 //
11 // NUMBYTES is used when a size is a number of bytes (usually a TPM2B)
12 typedef UINT16 NUMBYTES;
13
14 // TPM2B Types
15 typedef struct
16 {
17     NUMBYTES size;
18     BYTE     buffer[1];
19 } TPM2B, *P2B;
20 typedef const TPM2B* PC2B;
21
22 // This macro helps avoid having to type in the structure in order to create
23 // a new TPM2B type that is used in a function.
24 #define TPM2B_TYPE(name, bytes) \
25     typedef union \
26     { \
27         struct \
28         { \
29             NUMBYTES size; \
30             BYTE     buffer[(bytes)]; \
31         } t; \
32         TPM2B b; \
33     } TPM2B_##name
34
35 // This macro defines a TPM2B with a constant character value. This macro
36 // sets the size of the string to the size minus the terminating zero byte.
37 // This lets the user of the label add their terminating 0. This method
38 // is chosen so that existing code that provides a label will continue
39 // to work correctly.
40
41 // Macro to instance and initialize a TPM2B value
42 #define TPM2B_INIT(TYPE, name) TPM2B_##TYPE name = {sizeof(name.t.buffer), {0}}
43
44 #define TPM2B_BYTE_VALUE(bytes) TPM2B_TYPE(bytes##_BYTE_VALUE, bytes)
45
46 #endif

```

6.249 /tpm/include/public/TpmCalculatedAttributes.h

```

1  #ifndef TPM_CALCULATED_ATTRIBUTES_H
2  #define _TPM_CALCULATED_ATTRIBUTES_H_
3
4  #include "public/TpmAlgorithmDefines.h"
5  #include "public/GpMacros.h"
6
7  #define JOIN(x, y)      x##y
8  #define JOIN3(x, y, z)  x##y##z
9  #define CONCAT(x, y)    JOIN(x, y)
10 #define CONCAT3(x, y, z) JOIN3(x, y, z)
11
12 /** Derived from Vendor-specific values
13  ** Values derived from vendor specific settings in TpmProfile.h
14  */
15 #define PCR_SELECT_MIN ((PLATFORM_PCR + 7) / 8)
16 #define PCR_SELECT_MAX ((IMPLEMENTATION_PCR + 7) / 8)
17 #define MAX_ORDERLY_COUNT ((1 << ORDERLY_BITS) - 1)
18 #define RSA_MAX_PRIME (MAX_RSA_KEY_BYTES / 2)
19 #define RSA_PRIVATE_SIZE (RSA_MAX_PRIME * 5)
20
21 // If CONTEXT_INTEGRITY_HASH_ALG is defined, then the vendor is using the old style
22 // table. Otherwise, pick the "strongest" implemented hash algorithm as the context
23 // hash.
24 #ifndef CONTEXT_HASH_ALGORITHM
25 # if defined ALG_SHA3_512 && ALG_SHA3_512 == YES
26 #   define CONTEXT_HASH_ALGORITHM SHA3_512

```

```

26 # elif defined ALG_SHA512 && ALG_SHA512 == YES
27 #   define CONTEXT_HASH_ALGORITHM SHA512
28 # elif defined ALG_SHA3_384 && ALG_SHA3_384 == YES
29 #   define CONTEXT_HASH_ALGORITHM SHA3_384
30 # elif defined ALG_SHA384 && ALG_SHA384 == YES
31 #   define CONTEXT_HASH_ALGORITHM SHA384
32 # elif defined ALG_SHA3_256 && ALG_SHA3_256 == YES
33 #   define CONTEXT_HASH_ALGORITHM SHA3_256
34 # elif defined ALG_SHA256 && ALG_SHA256 == YES
35 #   define CONTEXT_HASH_ALGORITHM SHA256
36 # elif defined ALG_SM3_256 && ALG_SM3_256 == YES
37 #   define CONTEXT_HASH_ALGORITHM SM3_256
38 # elif defined ALG_SHA1 && ALG_SHA1 == YES
39 #   define CONTEXT_HASH_ALGORITHM SHA1
40 # endif
41 # define CONTEXT_INTEGRITY_HASH_ALG CONCAT(TPM_ALG_, CONTEXT_HASH_ALGORITHM)
42 #endif
43
44 #ifndef CONTEXT_INTEGRITY_HASH_SIZE
45 # define CONTEXT_INTEGRITY_HASH_SIZE CONCAT(CONTEXT_HASH_ALGORITHM, _DIGEST_SIZE)
46 #endif
47 #if ALG_RSA
48 # define RSA_SECURITY_STRENGTH \
49     (MAX_RSA_KEY_BITS >= 15360 \
50     ? 256 \
51     : (MAX_RSA_KEY_BITS >= 7680 \
52     ? 192 \
53     : (MAX_RSA_KEY_BITS >= 3072 \
54     ? 128 \
55     : (MAX_RSA_KEY_BITS >= 2048 \
56     ? 112 \
57     : (MAX_RSA_KEY_BITS >= 1024 ? 80 : 0))))))
58 #else
59 # define RSA_SECURITY_STRENGTH 0
60 #endif // ALG_RSA
61
62 #if ALG_ECC
63 # define ECC_SECURITY_STRENGTH \
64     (MAX_ECC_KEY_BITS >= 521 \
65     ? 256 \
66     : (MAX_ECC_KEY_BITS >= 384 ? 192 : (MAX_ECC_KEY_BITS >= 256 ? 128 : 0)))
67 #else
68 # define ECC_SECURITY_STRENGTH 0
69 #endif // ALG_ECC
70
71 #define MAX_ASYM_SECURITY_STRENGTH MAX(RSA_SECURITY_STRENGTH, ECC_SECURITY_STRENGTH)
72
73 #define MAX_HASH_SECURITY_STRENGTH ((CONTEXT_INTEGRITY_HASH_SIZE * 8) / 2)
74
75 // Unless some algorithm is broken...
76 #define MAX_SYM_SECURITY_STRENGTH MAX_SYM_KEY_BITS
77
78 #define MAX_SECURITY_STRENGTH_BITS \
79     MAX(MAX_ASYM_SECURITY_STRENGTH, \
80     MAX(MAX_SYM_SECURITY_STRENGTH, MAX_HASH_SECURITY_STRENGTH))
81
82 // This is the size that was used before the 1.38 errata requiring that P1.14.4 be
83 // followed
84 #define PROOF_SIZE CONTEXT_INTEGRITY_HASH_SIZE
85
86 // As required by P1.14.4
87 #define COMPLIANT_PROOF_SIZE \
88     (MAX(CONTEXT_INTEGRITY_HASH_SIZE, (2 * MAX_SYM_KEY_BYTES)))
89
90 // As required by P1.14.3.1
91 #define COMPLIANT_PRIMARY_SEED_SIZE BITS_TO_BYTES(MAX_SECURITY_STRENGTH_BITS * 2)

```

```

92
93 // This is the pre-errata version
94 #ifndef PRIMARY_SEED_SIZE
95 # define PRIMARY_SEED_SIZE PROOF_SIZE
96 #endif
97
98 #if USE_SPEC_COMPLIANT_PROOFS
99 # undef PROOF_SIZE
100 # define PROOF_SIZE COMPLIANT_PROOF_SIZE
101 # undef PRIMARY_SEED_SIZE
102 # define PRIMARY_SEED_SIZE COMPLIANT_PRIMARY_SEED_SIZE
103 #endif // USE_SPEC_COMPLIANT_PROOFS
104
105 #if !SKIP_PROOF_ERRORS
106 # if PROOF_SIZE < COMPLIANT_PROOF_SIZE
107 #   error "PROOF_SIZE is not compliant with TPM specification"
108 # endif
109 # if PRIMARY_SEED_SIZE < COMPLIANT_PRIMARY_SEED_SIZE
110 #   error Non-compliant PRIMARY_SEED_SIZE
111 # endif
112 #endif // !SKIP_PROOF_ERRORS
113
114 // If CONTEXT_ENCRYPT_ALG is defined, then the vendor is using the old style table
115 #if defined CONTEXT_ENCRYPT_ALG
116 # undef CONTEXT_ENCRYPT_ALGORITHM
117 # if CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
118 #   define CONTEXT_ENCRYPT_ALGORITHM AES
119 # elif CONTEXT_ENCRYPT_ALG == ALG_SM4_VALUE
120 #   define CONTEXT_ENCRYPT_ALGORITHM SM4
121 # elif CONTEXT_ENCRYPT_ALG == ALG_CAMELLIA_VALUE
122 #   define CONTEXT_ENCRYPT_ALGORITHM CAMELLIA
123 # else
124 #   error Unknown value for CONTEXT_ENCRYPT_ALG
125 # endif // CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
126 #else
127 # define CONTEXT_ENCRYPT_ALG CONCAT3(ALG_, CONTEXT_ENCRYPT_ALGORITHM, _VALUE)
128 #endif // CONTEXT_ENCRYPT_ALG
129 #define CONTEXT_ENCRYPT_KEY_BITS  CONCAT(CONTEXT_ENCRYPT_ALGORITHM,
130 _MAX_KEY_SIZE_BITS)
131 #define CONTEXT_ENCRYPT_KEY_BYTES ((CONTEXT_ENCRYPT_KEY_BITS + 7) / 8)
132
133 // This is updated to follow the requirement of P2 that the label not be larger
134 // than 32 bytes.
135 #ifndef LABEL_MAX_BUFFER
136 # define LABEL_MAX_BUFFER MIN(32, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE))
137 #endif
138
139 // This bit is used to indicate that an authorization ticket expires on TPM Reset
140 // and TPM Restart. It is added to the timeout value returned by TPM2_PolicySigned()
141 // and TPM2_PolicySecret() and used by TPM2_PolicyTicket(). The timeout value is
142 // relative to Time (g_time). Time is reset whenever the TPM loses power and cannot
143 // be moved forward by the user (as can Clock). 'g_time' is a 64-bit value expressing
144 // time in ms. Stealing the MSb for a flag means that the TPM needs to be reset
145 // at least once every 292,471,208 years rather than once every 584,942,417 years.
146 #define EXPIRATION_BIT ((UINT64)1 << 63)
147
148 // This definition is moved from TpmProfile.h because it is not actually vendor-
149 // specific. It has to be the same size as the 'sequence' parameter of a TPMS_CONTEXT
150 // and that is a UINT64. So, this is an invariant value
151 #define CONTEXT_COUNTER UINT64
152 #endif // _TPM_CALCULATED_ATTRIBUTES_H_

```

6.250 /tpm/include/public/TpmTypes.h

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT

```

```

2
3 #ifndef _TPM_INCLUDE_PRIVATE_TPMTYPES_H_
4 #define _TPM_INCLUDE_PRIVATE_TPMTYPES_H_
5
6 #ifndef MAX_CAP_BUFFER
7 # error MAX_CAP_BUFFER must be defined before this file so it can calculate maximum
  capability sizes
8 #endif
9 #include "public/Capabilities.h"
10 #include "public/TpmAlgorithmDefines.h"
11 #include "public/TpmCalculatedAttributes.h"
12 #include "public/GpMacros.h"
13
14 // Table "Definition of Types for Documentation Clarity" (Part 2: Structures)
15 typedef UINT32 TPM_ALGORITHM_ID;
16 #define TYPE_OF_TPM_ALGORITHM_ID UINT32
17 typedef UINT32 TPM_MODIFIER_INDICATOR;
18 #define TYPE_OF_TPM_MODIFIER_INDICATOR UINT32
19 typedef UINT32 TPM_AUTHORIZATION_SIZE;
20 #define TYPE_OF_TPM_AUTHORIZATION_SIZE UINT32
21 typedef UINT32 TPM_PARAMETER_SIZE;
22 #define TYPE_OF_TPM_PARAMETER_SIZE UINT32
23 typedef UINT16 TPM_KEY_SIZE;
24 #define TYPE_OF_TPM_KEY_SIZE UINT16
25 typedef UINT16 TPM_KEY_BITS;
26 #define TYPE_OF_TPM_KEY_BITS UINT16
27
28 // Table "Definition of TPM_CONSTANTS32 Constants" (Part 2: Structures)
29 typedef UINT32 TPM_CONSTANTS32;
30 #define TYPE_OF_TPM_CONSTANTS32 UINT32
31 #define TPM_GENERATED_VALUE (TPM_CONSTANTS32) (0xFF544347)
32 #define TPM_MAX_DERIVATION_BITS (TPM_CONSTANTS32) (8192)
33
34 // Table "Definition of TPM_ALG_ID Constants" (Part 2: Structures)
35 typedef UINT16 TPM_ALG_ID;
36 #define TYPE_OF_TPM_ALG_ID
37 #define ALG_ERROR_VALUE          UINT16
38 #define ALG_ERROR_VALUE          0x0000
39 #define TPM_ALG_ERROR            (TPM_ALG_ID) (ALG_ERROR_VALUE)
40 #define ALG_RSA_VALUE           0x0001
41 #define TPM_ALG_RSA              (TPM_ALG_ID) (ALG_RSA_VALUE)
42 #define ALG_TDES_VALUE          0x0003
43 #define TPM_ALG_TDES            (TPM_ALG_ID) (ALG_TDES_VALUE)
44 #define ALG_SHA1_VALUE          0x0004
45 #define TPM_ALG_SHA1            (TPM_ALG_ID) (ALG_SHA1_VALUE)
46 #define ALG_HMAC_VALUE          0x0005
47 #define TPM_ALG_HMAC            (TPM_ALG_ID) (ALG_HMAC_VALUE)
48 #define ALG_AES_VALUE           0x0006
49 #define TPM_ALG_AES             (TPM_ALG_ID) (ALG_AES_VALUE)
50 #define ALG_MGF1_VALUE          0x0007
51 #define TPM_ALG_MGF1            (TPM_ALG_ID) (ALG_MGF1_VALUE)
52 #define ALG_KEYEDHASH_VALUE     0x0008
53 #define TPM_ALG_KEYEDHASH       (TPM_ALG_ID) (ALG_KEYEDHASH_VALUE)
54 #define ALG_XOR_VALUE           0x000A
55 #define TPM_ALG_XOR             (TPM_ALG_ID) (ALG_XOR_VALUE)
56 #define ALG_SHA256_VALUE        0x000B
57 #define TPM_ALG_SHA256          (TPM_ALG_ID) (ALG_SHA256_VALUE)
58 #define ALG_SHA384_VALUE        0x000C
59 #define TPM_ALG_SHA384          (TPM_ALG_ID) (ALG_SHA384_VALUE)
60 #define ALG_SHA512_VALUE        0x000D
61 #define TPM_ALG_SHA512          (TPM_ALG_ID) (ALG_SHA512_VALUE)
62 #define ALG_SHA256_192_VALUE    0x000E
63 #define TPM_ALG_SHA256_192      (TPM_ALG_ID) (ALG_SHA256_192_VALUE)
64 #define ALG_NULL_VALUE          0x0010
65 #define TPM_ALG_NULL            (TPM_ALG_ID) (ALG_NULL_VALUE)
66 #define ALG_SM3_256_VALUE       0x0012
67 #define TPM_ALG_SM3_256         (TPM_ALG_ID) (ALG_SM3_256_VALUE)

```



```
67 #define ALG_SM4_VALUE 0x0013
68 #define TPM_ALG_SM4 (TPM_ALG_ID) (ALG_SM4_VALUE)
69 #define ALG_RSASSA_VALUE 0x0014
70 #define TPM_ALG_RSASSA (TPM_ALG_ID) (ALG_RSASSA_VALUE)
71 #define ALG_RSAES_VALUE 0x0015
72 #define TPM_ALG_RSAES (TPM_ALG_ID) (ALG_RSAES_VALUE)
73 #define ALG_RSAPSS_VALUE 0x0016
74 #define TPM_ALG_RSAPSS (TPM_ALG_ID) (ALG_RSAPSS_VALUE)
75 #define ALG_OAEP_VALUE 0x0017
76 #define TPM_ALG_OAEP (TPM_ALG_ID) (ALG_OAEP_VALUE)
77 #define ALG_ECDSA_VALUE 0x0018
78 #define TPM_ALG_ECDSA (TPM_ALG_ID) (ALG_ECDSA_VALUE)
79 #define ALG_ECDH_VALUE 0x0019
80 #define TPM_ALG_ECDH (TPM_ALG_ID) (ALG_ECDH_VALUE)
81 #define ALG_ECDAA_VALUE 0x001A
82 #define TPM_ALG_ECDAA (TPM_ALG_ID) (ALG_ECDAA_VALUE)
83 #define ALG_SM2_VALUE 0x001B
84 #define TPM_ALG_SM2 (TPM_ALG_ID) (ALG_SM2_VALUE)
85 #define ALG_ECSCHNORR_VALUE 0x001C
86 #define TPM_ALG_ECSCHNORR (TPM_ALG_ID) (ALG_ECSCHNORR_VALUE)
87 #define ALG_ECMQV_VALUE 0x001D
88 #define TPM_ALG_ECMQV (TPM_ALG_ID) (ALG_ECMQV_VALUE)
89 #define ALG_KDF1_SP800_56A_VALUE 0x0020
90 #define TPM_ALG_KDF1_SP800_56A (TPM_ALG_ID) (ALG_KDF1_SP800_56A_VALUE)
91 #define ALG_KDF2_VALUE 0x0021
92 #define TPM_ALG_KDF2 (TPM_ALG_ID) (ALG_KDF2_VALUE)
93 #define ALG_KDF1_SP800_108_VALUE 0x0022
94 #define TPM_ALG_KDF1_SP800_108 (TPM_ALG_ID) (ALG_KDF1_SP800_108_VALUE)
95 #define ALG_ECC_VALUE 0x0023
96 #define TPM_ALG_ECC (TPM_ALG_ID) (ALG_ECC_VALUE)
97 #define ALG_SYMCIPHER_VALUE 0x0025
98 #define TPM_ALG_SYMCIPHER (TPM_ALG_ID) (ALG_SYMCIPHER_VALUE)
99 #define ALG_CAMELLIA_VALUE 0x0026
100 #define TPM_ALG_CAMELLIA (TPM_ALG_ID) (ALG_CAMELLIA_VALUE)
101 #define ALG_SHA3_256_VALUE 0x0027
102 #define TPM_ALG_SHA3_256 (TPM_ALG_ID) (ALG_SHA3_256_VALUE)
103 #define ALG_SHA3_384_VALUE 0x0028
104 #define TPM_ALG_SHA3_384 (TPM_ALG_ID) (ALG_SHA3_384_VALUE)
105 #define ALG_SHA3_512_VALUE 0x0029
106 #define TPM_ALG_SHA3_512 (TPM_ALG_ID) (ALG_SHA3_512_VALUE)
107 #define ALG_SHAKE128_VALUE 0x002A
108 #define TPM_ALG_SHAKE128 (TPM_ALG_ID) (ALG_SHAKE128_VALUE)
109 #define ALG_SHAKE256_VALUE 0x002B
110 #define TPM_ALG_SHAKE256 (TPM_ALG_ID) (ALG_SHAKE256_VALUE)
111 #define ALG_SHAKE256_192_VALUE 0x002C
112 #define TPM_ALG_SHAKE256_192 (TPM_ALG_ID) (ALG_SHAKE256_192_VALUE)
113 #define ALG_SHAKE256_256_VALUE 0x002D
114 #define TPM_ALG_SHAKE256_256 (TPM_ALG_ID) (ALG_SHAKE256_256_VALUE)
115 #define ALG_SHAKE256_512_VALUE 0x002E
116 #define TPM_ALG_SHAKE256_512 (TPM_ALG_ID) (ALG_SHAKE256_512_VALUE)
117 #define ALG_CMAC_VALUE 0x003F
118 #define TPM_ALG_CMAC (TPM_ALG_ID) (ALG_CMAC_VALUE)
119 #define ALG_CTR_VALUE 0x0040
120 #define TPM_ALG_CTR (TPM_ALG_ID) (ALG_CTR_VALUE)
121 #define ALG_OFB_VALUE 0x0041
122 #define TPM_ALG_OFB (TPM_ALG_ID) (ALG_OFB_VALUE)
123 #define ALG_CBC_VALUE 0x0042
124 #define TPM_ALG_CBC (TPM_ALG_ID) (ALG_CBC_VALUE)
125 #define ALG_CFB_VALUE 0x0043
126 #define TPM_ALG_CFB (TPM_ALG_ID) (ALG_CFB_VALUE)
127 #define ALG_ECB_VALUE 0x0044
128 #define TPM_ALG_ECB (TPM_ALG_ID) (ALG_ECB_VALUE)
129 #define ALG_CCM_VALUE 0x0050
130 #define TPM_ALG_CCM (TPM_ALG_ID) (ALG_CCM_VALUE)
131 #define ALG_GCM_VALUE 0x0051
132 #define TPM_ALG_GCM (TPM_ALG_ID) (ALG_GCM_VALUE)
```



```

133 #define ALG_KW_VALUE 0x0052
134 #define TPM_ALG_KW (TPM_ALG_ID) (ALG_KW_VALUE)
135 #define ALG_KWP_VALUE 0x0053
136 #define TPM_ALG_KWP (TPM_ALG_ID) (ALG_KWP_VALUE)
137 #define ALG_EAX_VALUE 0x0054
138 #define TPM_ALG_EAX (TPM_ALG_ID) (ALG_EAX_VALUE)
139 #define ALG_EDDSA_VALUE 0x0060
140 #define TPM_ALG_EDDSA (TPM_ALG_ID) (ALG_EDDSA_VALUE)
141 #define ALG_EDDSA_PH_VALUE 0x0061
142 #define TPM_ALG_EDDSA_PH (TPM_ALG_ID) (ALG_EDDSA_PH_VALUE)
143 #define ALG_LMS_VALUE 0x0070
144 #define TPM_ALG_LMS (TPM_ALG_ID) (ALG_LMS_VALUE)
145 #define ALG_XMSS_VALUE 0x0071
146 #define TPM_ALG_XMSS (TPM_ALG_ID) (ALG_XMSS_VALUE)
147 // Values derived from Table "Definition of TPM_ALG_ID Constants" (Part 2:
    Structures)
148 #define ALG_FIRST_VALUE 0x0001
149 #define TPM_ALG_FIRST (TPM_ALG_ID) (ALG_FIRST_VALUE)
150 #define ALG_LAST_VALUE 0x0071
151 #define TPM_ALG_LAST (TPM_ALG_ID) (ALG_LAST_VALUE)
152
153 // Table "Definition of TPM_ECC_CURVE Constants" (Part 2: Structures)
154 typedef UINT16 TPM_ECC_CURVE;
155
156 #define TYPE_OF_TPM_ECC_CURVE UINT16
157 #define TPM_ECC_NONE (TPM_ECC_CURVE) (0x0000)
158 #define TPM_ECC_NIST_P192 (TPM_ECC_CURVE) (0x0001)
159 #define TPM_ECC_NIST_P224 (TPM_ECC_CURVE) (0x0002)
160 #define TPM_ECC_NIST_P256 (TPM_ECC_CURVE) (0x0003)
161 #define TPM_ECC_NIST_P384 (TPM_ECC_CURVE) (0x0004)
162 #define TPM_ECC_NIST_P521 (TPM_ECC_CURVE) (0x0005)
163 #define TPM_ECC_BN_P256 (TPM_ECC_CURVE) (0x0010)
164 #define TPM_ECC_BN_P638 (TPM_ECC_CURVE) (0x0011)
165 #define TPM_ECC_SM2_P256 (TPM_ECC_CURVE) (0x0020)
166 #define TPM_ECC_BP_P256_R1 (TPM_ECC_CURVE) (0x0030)
167 #define TPM_ECC_BP_P384_R1 (TPM_ECC_CURVE) (0x0031)
168 #define TPM_ECC_BP_P512_R1 (TPM_ECC_CURVE) (0x0032)
169 #define TPM_ECC_CURVE_25519 (TPM_ECC_CURVE) (0x0040)
170 #define TPM_ECC_CURVE_448 (TPM_ECC_CURVE) (0x0041)
171
172 // Table "Definition of TPM_CC Constants" (Part 2: Structures)
173 typedef UINT32 TPM_CC;
174
175 #define TYPE_OF_TPM_CC UINT32
176 #define TPM_CC_FIRST (TPM_CC) (0x0000011F)
177 #define TPM_CC_NV_UndefineSpaceSpecial (TPM_CC) (0x0000011F)
178 #define TPM_CC_EvictControl (TPM_CC) (0x00000120)
179 #define TPM_CC_HierarchyControl (TPM_CC) (0x00000121)
180 #define TPM_CC_NV_UndefineSpace (TPM_CC) (0x00000122)
181 #define TPM_CC_ChangeEPS (TPM_CC) (0x00000124)
182 #define TPM_CC_ChangePPS (TPM_CC) (0x00000125)
183 #define TPM_CC_Clear (TPM_CC) (0x00000126)
184 #define TPM_CC_ClearControl (TPM_CC) (0x00000127)
185 #define TPM_CC_ClockSet (TPM_CC) (0x00000128)
186 #define TPM_CC_HierarchyChangeAuth (TPM_CC) (0x00000129)
187 #define TPM_CC_NV_DefineSpace (TPM_CC) (0x0000012A)
188 #define TPM_CC_PCR_Allocate (TPM_CC) (0x0000012B)
189 #define TPM_CC_PCR_SetAuthPolicy (TPM_CC) (0x0000012C)
190 #define TPM_CC_PP_Commands (TPM_CC) (0x0000012D)
191 #define TPM_CC_SetPrimaryPolicy (TPM_CC) (0x0000012E)
192 #define TPM_CC_FieldUpgradeStart (TPM_CC) (0x0000012F)
193 #define TPM_CC_ClockRateAdjust (TPM_CC) (0x00000130)
194 #define TPM_CC_CreatePrimary (TPM_CC) (0x00000131)
195 #define TPM_CC_NV_GlobalWriteLock (TPM_CC) (0x00000132)
196 #define TPM_CC_GetCommandAuditDigest (TPM_CC) (0x00000133)
197 #define TPM_CC_NV_Increment (TPM_CC) (0x00000134)

```

```
198 #define TPM_CC_NV_SetBits (TPM_CC) (0x00000135)
199 #define TPM_CC_NV_Extend (TPM_CC) (0x00000136)
200 #define TPM_CC_NV_Write (TPM_CC) (0x00000137)
201 #define TPM_CC_NV_WriteLock (TPM_CC) (0x00000138)
202 #define TPM_CC_DictionaryAttackLockReset (TPM_CC) (0x00000139)
203 #define TPM_CC_DictionaryAttackParameters (TPM_CC) (0x0000013A)
204 #define TPM_CC_NV_ChangeAuth (TPM_CC) (0x0000013B)
205 #define TPM_CC_PCR_Event (TPM_CC) (0x0000013C)
206 #define TPM_CC_PCR_Reset (TPM_CC) (0x0000013D)
207 #define TPM_CC_SequenceComplete (TPM_CC) (0x0000013E)
208 #define TPM_CC_SetAlgorithmSet (TPM_CC) (0x0000013F)
209 #define TPM_CC_SetCommandCodeAuditStatus (TPM_CC) (0x00000140)
210 #define TPM_CC_FieldUpgradeData (TPM_CC) (0x00000141)
211 #define TPM_CC_IncrementalSelfTest (TPM_CC) (0x00000142)
212 #define TPM_CC_SelfTest (TPM_CC) (0x00000143)
213 #define TPM_CC_Startup (TPM_CC) (0x00000144)
214 #define TPM_CC_Shutdown (TPM_CC) (0x00000145)
215 #define TPM_CC_StirRandom (TPM_CC) (0x00000146)
216 #define TPM_CC_ActivateCredential (TPM_CC) (0x00000147)
217 #define TPM_CC_Certify (TPM_CC) (0x00000148)
218 #define TPM_CC_PolicyNV (TPM_CC) (0x00000149)
219 #define TPM_CC_CertifyCreation (TPM_CC) (0x0000014A)
220 #define TPM_CC_Duplicate (TPM_CC) (0x0000014B)
221 #define TPM_CC_GetTime (TPM_CC) (0x0000014C)
222 #define TPM_CC_GetSessionAuditDigest (TPM_CC) (0x0000014D)
223 #define TPM_CC_NV_Read (TPM_CC) (0x0000014E)
224 #define TPM_CC_NV_ReadLock (TPM_CC) (0x0000014F)
225 #define TPM_CC_ObjectChangeAuth (TPM_CC) (0x00000150)
226 #define TPM_CC_PolicySecret (TPM_CC) (0x00000151)
227 #define TPM_CC_Rewrap (TPM_CC) (0x00000152)
228 #define TPM_CC_Create (TPM_CC) (0x00000153)
229 #define TPM_CC_ECDH_ZGen (TPM_CC) (0x00000154)
230 #define TPM_CC_HMAC (TPM_CC) (0x00000155)
231 #define TPM_CC_MAC (TPM_CC) (0x00000155)
232 #define TPM_CC_Import (TPM_CC) (0x00000156)
233 #define TPM_CC_Load (TPM_CC) (0x00000157)
234 #define TPM_CC_Quote (TPM_CC) (0x00000158)
235 #define TPM_CC_RSA_Decrypt (TPM_CC) (0x00000159)
236 #define TPM_CC_HMAC_Start (TPM_CC) (0x0000015B)
237 #define TPM_CC_MAC_Start (TPM_CC) (0x0000015B)
238 #define TPM_CC_SequenceUpdate (TPM_CC) (0x0000015C)
239 #define TPM_CC_Sign (TPM_CC) (0x0000015D)
240 #define TPM_CC_Unseal (TPM_CC) (0x0000015E)
241 #define TPM_CC_PolicySigned (TPM_CC) (0x00000160)
242 #define TPM_CC_ContextLoad (TPM_CC) (0x00000161)
243 #define TPM_CC_ContextSave (TPM_CC) (0x00000162)
244 #define TPM_CC_ECDH_KeyGen (TPM_CC) (0x00000163)
245 #define TPM_CC_EncryptDecrypt (TPM_CC) (0x00000164)
246 #define TPM_CC_FlushContext (TPM_CC) (0x00000165)
247 #define TPM_CC_LoadExternal (TPM_CC) (0x00000167)
248 #define TPM_CC_MakeCredential (TPM_CC) (0x00000168)
249 #define TPM_CC_NV_ReadPublic (TPM_CC) (0x00000169)
250 #define TPM_CC_PolicyAuthorize (TPM_CC) (0x0000016A)
251 #define TPM_CC_PolicyAuthValue (TPM_CC) (0x0000016B)
252 #define TPM_CC_PolicyCommandCode (TPM_CC) (0x0000016C)
253 #define TPM_CC_PolicyCounterTimer (TPM_CC) (0x0000016D)
254 #define TPM_CC_PolicyCpHash (TPM_CC) (0x0000016E)
255 #define TPM_CC_PolicyLocality (TPM_CC) (0x0000016F)
256 #define TPM_CC_PolicyNameHash (TPM_CC) (0x00000170)
257 #define TPM_CC_PolicyOR (TPM_CC) (0x00000171)
258 #define TPM_CC_PolicyTicket (TPM_CC) (0x00000172)
259 #define TPM_CC_ReadPublic (TPM_CC) (0x00000173)
260 #define TPM_CC_RSA_Encrypt (TPM_CC) (0x00000174)
261 #define TPM_CC_StartAuthSession (TPM_CC) (0x00000176)
262 #define TPM_CC_VerifySignature (TPM_CC) (0x00000177)
263 #define TPM_CC_ECC_Parameters (TPM_CC) (0x00000178)
```

```

264 #define TPM_CC_FirmwareRead (TPM_CC) (0x00000179)
265 #define TPM_CC_GetCapability (TPM_CC) (0x0000017A)
266 #define TPM_CC_GetRandom (TPM_CC) (0x0000017B)
267 #define TPM_CC_GetTestResult (TPM_CC) (0x0000017C)
268 #define TPM_CC_Hash (TPM_CC) (0x0000017D)
269 #define TPM_CC_PCR_Read (TPM_CC) (0x0000017E)
270 #define TPM_CC_PolicyPCR (TPM_CC) (0x0000017F)
271 #define TPM_CC_PolicyRestart (TPM_CC) (0x00000180)
272 #define TPM_CC_ReadClock (TPM_CC) (0x00000181)
273 #define TPM_CC_PCR_Extend (TPM_CC) (0x00000182)
274 #define TPM_CC_PCR_SetAuthValue (TPM_CC) (0x00000183)
275 #define TPM_CC_NV_Certify (TPM_CC) (0x00000184)
276 #define TPM_CC_EventSequenceComplete (TPM_CC) (0x00000185)
277 #define TPM_CC_HashSequenceStart (TPM_CC) (0x00000186)
278 #define TPM_CC_PolicyPhysicalPresence (TPM_CC) (0x00000187)
279 #define TPM_CC_PolicyDuplicationSelect (TPM_CC) (0x00000188)
280 #define TPM_CC_PolicyGetDigest (TPM_CC) (0x00000189)
281 #define TPM_CC_TestParms (TPM_CC) (0x0000018A)
282 #define TPM_CC_Commit (TPM_CC) (0x0000018B)
283 #define TPM_CC_PolicyPassword (TPM_CC) (0x0000018C)
284 #define TPM_CC_ZGen_2Phase (TPM_CC) (0x0000018D)
285 #define TPM_CC_EC_Ephemeral (TPM_CC) (0x0000018E)
286 #define TPM_CC_PolicyNvWritten (TPM_CC) (0x0000018F)
287 #define TPM_CC_PolicyTemplate (TPM_CC) (0x00000190)
288 #define TPM_CC_CreateLoaded (TPM_CC) (0x00000191)
289 #define TPM_CC_PolicyAuthorizeNV (TPM_CC) (0x00000192)
290 #define TPM_CC_EncryptDecrypt2 (TPM_CC) (0x00000193)
291 #define TPM_CC_AC_GetCapability (TPM_CC) (0x00000194)
292 #define TPM_CC_AC_Send (TPM_CC) (0x00000195)
293 #define TPM_CC_Policy_AC_SendSelect (TPM_CC) (0x00000196)
294 #define TPM_CC_CertifyX509 (TPM_CC) (0x00000197)
295 #define TPM_CC_ACT_SetTimeout (TPM_CC) (0x00000198)
296 #define TPM_CC_ECC_Encrypt (TPM_CC) (0x00000199)
297 #define TPM_CC_ECC_Decrypt (TPM_CC) (0x0000019A)
298 #define TPM_CC_PolicyCapability (TPM_CC) (0x0000019B)
299 #define TPM_CC_PolicyParameters (TPM_CC) (0x0000019C)
300 #define TPM_CC_NV_DefineSpace2 (TPM_CC) (0x0000019D)
301 #define TPM_CC_NV_ReadPublic2 (TPM_CC) (0x0000019E)
302 #define TPM_CC_SetCapability (TPM_CC) (0x0000019F)
303 #define TPM_CC_LAST (TPM_CC) (0x0000019F)
304 #define CC_VEND (TPM_CC) (0x20000000)
305 #define TPM_CC_Vendor_TCG_Test (TPM_CC) (0x20000000)
306
307 // This large macro is needed to determine the maximum commandIndex. This value
308 // is needed in order to size typedef'd structures. As a consequence, the
309 // computation cannot be deferred until the command array is instantiated and
310 // so that the number of entries can be determined by
311 // sizeof(array)/sizeof(entry).
312 //
313 // Size the array of library commands based on whether or not the array is
314 // packed (only defined commands) or dense
315 // (having entries for unimplemented commands). This overly large macro
316 // computes the size of the array and sets some global constants
317 #if COMPRESSED_LISTS
318 # define ADD_FILL 0
319 #else
320 # define ADD_FILL 1
321 #endif
322 #define LIBRARY_COMMAND_ARRAY_SIZE \
323 (0 + (ADD_FILL || CC_NV_UndefineSpaceSpecial) /* 0x0000011F */ \
324 + (ADD_FILL || CC_EvictControl) /* 0x00000120 */ \
325 + (ADD_FILL || CC_HierarchyControl) /* 0x00000121 */ \
326 + (ADD_FILL || CC_NV_UndefineSpace) /* 0x00000122 */ \
327 + ADD_FILL /* 0x00000123 */ \
328 + (ADD_FILL || CC_ChangeEPS) /* 0x00000124 */ \
329 + (ADD_FILL || CC_ChangePPS) /* 0x00000125 */ \

```

```

330 + (ADD_FILL || CC_Clear) /* 0x00000126 */ \
331 + (ADD_FILL || CC_ClearControl) /* 0x00000127 */ \
332 + (ADD_FILL || CC_ClockSet) /* 0x00000128 */ \
333 + (ADD_FILL || CC_HierarchyChangeAuth) /* 0x00000129 */ \
334 + (ADD_FILL || CC_NV_DefineSpace) /* 0x0000012A */ \
335 + (ADD_FILL || CC_PCR_Allocate) /* 0x0000012B */ \
336 + (ADD_FILL || CC_PCR_SetAuthPolicy) /* 0x0000012C */ \
337 + (ADD_FILL || CC_PP_Commands) /* 0x0000012D */ \
338 + (ADD_FILL || CC_SetPrimaryPolicy) /* 0x0000012E */ \
339 + (ADD_FILL || CC_FieldUpgradeStart) /* 0x0000012F */ \
340 + (ADD_FILL || CC_ClockRateAdjust) /* 0x00000130 */ \
341 + (ADD_FILL || CC_CreatePrimary) /* 0x00000131 */ \
342 + (ADD_FILL || CC_NV_GlobalWriteLock) /* 0x00000132 */ \
343 + (ADD_FILL || CC_GetCommandAuditDigest) /* 0x00000133 */ \
344 + (ADD_FILL || CC_NV_Increment) /* 0x00000134 */ \
345 + (ADD_FILL || CC_NV_SetBits) /* 0x00000135 */ \
346 + (ADD_FILL || CC_NV_Extend) /* 0x00000136 */ \
347 + (ADD_FILL || CC_NV_Write) /* 0x00000137 */ \
348 + (ADD_FILL || CC_NV_WriteLock) /* 0x00000138 */ \
349 + (ADD_FILL || CC_DictionaryAttackLockReset) /* 0x00000139 */ \
350 + (ADD_FILL || CC_DictionaryAttackParameters) /* 0x0000013A */ \
351 + (ADD_FILL || CC_NV_ChangeAuth) /* 0x0000013B */ \
352 + (ADD_FILL || CC_PCR_Event) /* 0x0000013C */ \
353 + (ADD_FILL || CC_PCR_Reset) /* 0x0000013D */ \
354 + (ADD_FILL || CC_SequenceComplete) /* 0x0000013E */ \
355 + (ADD_FILL || CC_SetAlgorithmSet) /* 0x0000013F */ \
356 + (ADD_FILL || CC_SetCommandCodeAuditStatus) /* 0x00000140 */ \
357 + (ADD_FILL || CC_FieldUpgradeData) /* 0x00000141 */ \
358 + (ADD_FILL || CC_IncrementalSelfTest) /* 0x00000142 */ \
359 + (ADD_FILL || CC_SelfTest) /* 0x00000143 */ \
360 + (ADD_FILL || CC_Startup) /* 0x00000144 */ \
361 + (ADD_FILL || CC_Shutdown) /* 0x00000145 */ \
362 + (ADD_FILL || CC_StirRandom) /* 0x00000146 */ \
363 + (ADD_FILL || CC_ActivateCredential) /* 0x00000147 */ \
364 + (ADD_FILL || CC_Certify) /* 0x00000148 */ \
365 + (ADD_FILL || CC_PolicyNV) /* 0x00000149 */ \
366 + (ADD_FILL || CC_CertifyCreation) /* 0x0000014A */ \
367 + (ADD_FILL || CC_Duplicate) /* 0x0000014B */ \
368 + (ADD_FILL || CC_GetTime) /* 0x0000014C */ \
369 + (ADD_FILL || CC_GetSessionAuditDigest) /* 0x0000014D */ \
370 + (ADD_FILL || CC_NV_Read) /* 0x0000014E */ \
371 + (ADD_FILL || CC_NV_ReadLock) /* 0x0000014F */ \
372 + (ADD_FILL || CC_ObjectChangeAuth) /* 0x00000150 */ \
373 + (ADD_FILL || CC_PolicySecret) /* 0x00000151 */ \
374 + (ADD_FILL || CC_Rewrap) /* 0x00000152 */ \
375 + (ADD_FILL || CC_Create) /* 0x00000153 */ \
376 + (ADD_FILL || CC_ECDH_ZGen) /* 0x00000154 */ \
377 + (ADD_FILL || (CC_HMAC || CC_MAC)) /* 0x00000155 */ \
378 + (ADD_FILL || CC_Import) /* 0x00000156 */ \
379 + (ADD_FILL || CC_Load) /* 0x00000157 */ \
380 + (ADD_FILL || CC_Quote) /* 0x00000158 */ \
381 + (ADD_FILL || CC_RSA_Decrypt) /* 0x00000159 */ \
382 + ADD_FILL /* 0x0000015A */ \
383 + (ADD_FILL || (CC_HMAC_Start || CC_MAC_Start)) /* 0x0000015B */ \
384 + (ADD_FILL || CC_SequenceUpdate) /* 0x0000015C */ \
385 + (ADD_FILL || CC_Sign) /* 0x0000015D */ \
386 + (ADD_FILL || CC_Unseal) /* 0x0000015E */ \
387 + ADD_FILL /* 0x0000015F */ \
388 + (ADD_FILL || CC_PolicySigned) /* 0x00000160 */ \
389 + (ADD_FILL || CC_ContextLoad) /* 0x00000161 */ \
390 + (ADD_FILL || CC_ContextSave) /* 0x00000162 */ \
391 + (ADD_FILL || CC_ECDH_KeyGen) /* 0x00000163 */ \
392 + (ADD_FILL || CC_EncryptDecrypt) /* 0x00000164 */ \
393 + (ADD_FILL || CC_FlushContext) /* 0x00000165 */ \
394 + ADD_FILL /* 0x00000166 */ \
395 + (ADD_FILL || CC_LoadExternal) /* 0x00000167 */ \

```



```

396 + (ADD_FILL || CC_MakeCredential) /* 0x00000168 */ \
397 + (ADD_FILL || CC_NV_ReadPublic) /* 0x00000169 */ \
398 + (ADD_FILL || CC_PolicyAuthorize) /* 0x0000016A */ \
399 + (ADD_FILL || CC_PolicyAuthValue) /* 0x0000016B */ \
400 + (ADD_FILL || CC_PolicyCommandCode) /* 0x0000016C */ \
401 + (ADD_FILL || CC_PolicyCounterTimer) /* 0x0000016D */ \
402 + (ADD_FILL || CC_PolicyCpHash) /* 0x0000016E */ \
403 + (ADD_FILL || CC_PolicyLocality) /* 0x0000016F */ \
404 + (ADD_FILL || CC_PolicyNameHash) /* 0x00000170 */ \
405 + (ADD_FILL || CC_PolicyOR) /* 0x00000171 */ \
406 + (ADD_FILL || CC_PolicyTicket) /* 0x00000172 */ \
407 + (ADD_FILL || CC_ReadPublic) /* 0x00000173 */ \
408 + (ADD_FILL || CC_RSA_Encrypt) /* 0x00000174 */ \
409 + ADD_FILL /* 0x00000175 */ \
410 + (ADD_FILL || CC_StartAuthSession) /* 0x00000176 */ \
411 + (ADD_FILL || CC_VerifySignature) /* 0x00000177 */ \
412 + (ADD_FILL || CC_ECC_Parameters) /* 0x00000178 */ \
413 + (ADD_FILL || CC_FirmwareRead) /* 0x00000179 */ \
414 + (ADD_FILL || CC_GetCapability) /* 0x0000017A */ \
415 + (ADD_FILL || CC_GetRandom) /* 0x0000017B */ \
416 + (ADD_FILL || CC_GetTestResult) /* 0x0000017C */ \
417 + (ADD_FILL || CC_Hash) /* 0x0000017D */ \
418 + (ADD_FILL || CC_PCR_Read) /* 0x0000017E */ \
419 + (ADD_FILL || CC_PolicyPCR) /* 0x0000017F */ \
420 + (ADD_FILL || CC_PolicyRestart) /* 0x00000180 */ \
421 + (ADD_FILL || CC_ReadClock) /* 0x00000181 */ \
422 + (ADD_FILL || CC_PCR_Extend) /* 0x00000182 */ \
423 + (ADD_FILL || CC_PCR_SetAuthValue) /* 0x00000183 */ \
424 + (ADD_FILL || CC_NV_Certify) /* 0x00000184 */ \
425 + (ADD_FILL || CC_EventSequenceComplete) /* 0x00000185 */ \
426 + (ADD_FILL || CC_HashSequenceStart) /* 0x00000186 */ \
427 + (ADD_FILL || CC_PolicyPhysicalPresence) /* 0x00000187 */ \
428 + (ADD_FILL || CC_PolicyDuplicationSelect) /* 0x00000188 */ \
429 + (ADD_FILL || CC_PolicyGetDigest) /* 0x00000189 */ \
430 + (ADD_FILL || CC_TestParms) /* 0x0000018A */ \
431 + (ADD_FILL || CC_Commit) /* 0x0000018B */ \
432 + (ADD_FILL || CC_PolicyPassword) /* 0x0000018C */ \
433 + (ADD_FILL || CC_ZGen_2Phase) /* 0x0000018D */ \
434 + (ADD_FILL || CC_EC_Ephemeral) /* 0x0000018E */ \
435 + (ADD_FILL || CC_PolicyNvWritten) /* 0x0000018F */ \
436 + (ADD_FILL || CC_PolicyTemplate) /* 0x00000190 */ \
437 + (ADD_FILL || CC_CreateLoaded) /* 0x00000191 */ \
438 + (ADD_FILL || CC_PolicyAuthorizeNV) /* 0x00000192 */ \
439 + (ADD_FILL || CC_EncryptDecrypt2) /* 0x00000193 */ \
440 + (ADD_FILL || CC_AC_GetCapability) /* 0x00000194 */ \
441 + (ADD_FILL || CC_AC_Send) /* 0x00000195 */ \
442 + (ADD_FILL || CC_Policy_AC_SendSelect) /* 0x00000196 */ \
443 + (ADD_FILL || CC_CertifyX509) /* 0x00000197 */ \
444 + (ADD_FILL || CC_ACT_SetTimeout) /* 0x00000198 */ \
445 + (ADD_FILL || CC_ECC_Encrypt) /* 0x00000199 */ \
446 + (ADD_FILL || CC_ECC_Decrypt) /* 0x0000019A */ \
447 + (ADD_FILL || CC_PolicyCapability) /* 0x0000019B */ \
448 + (ADD_FILL || CC_PolicyParameters) /* 0x0000019C */ \
449 + (ADD_FILL || CC_NV_DefineSpace2) /* 0x0000019D */ \
450 + (ADD_FILL || CC_NV_ReadPublic2) /* 0x0000019E */ \
451 + (ADD_FILL || CC_SetCapability) /* 0x0000019F */ \
452 )
453 #define VENDOR_COMMAND_ARRAY_SIZE (CC_Vendor_TCG_Test)
454 #define COMMAND_COUNT (LIBRARY_COMMAND_ARRAY_SIZE +
VENDOR_COMMAND_ARRAY_SIZE)
455
456 // Table "Definition of TPM_RC Constants" (Part 2: Structures)
457 typedef UINT32 TPM_RC;
458 #define TYPE_OF_TPM_RC UINT32
459 #define TPM_RC_SUCCESS (TPM_RC) (0x000)
460 #define TPM_RC_BAD_TAG (TPM_RC) (0x01E)

```

```

461 #define RC_VER1 (TPM_RC) (0x100)
462 #define TPM_RC_INITIALIZE (TPM_RC) (RC_VER1 + 0x000)
463 #define TPM_RC_FAILURE (TPM_RC) (RC_VER1 + 0x001)
464 #define TPM_RC_SEQUENCE (TPM_RC) (RC_VER1 + 0x003)
465 #define TPM_RC_PRIVATE (TPM_RC) (RC_VER1 + 0x00B)
466 #define TPM_RC_HMAC (TPM_RC) (RC_VER1 + 0x019)
467 #define TPM_RC_DISABLED (TPM_RC) (RC_VER1 + 0x020)
468 #define TPM_RC_EXCLUSIVE (TPM_RC) (RC_VER1 + 0x021)
469 #define TPM_RC_AUTH_TYPE (TPM_RC) (RC_VER1 + 0x024)
470 #define TPM_RC_AUTH_MISSING (TPM_RC) (RC_VER1 + 0x025)
471 #define TPM_RC_POLICY (TPM_RC) (RC_VER1 + 0x026)
472 #define TPM_RC_PCR (TPM_RC) (RC_VER1 + 0x027)
473 #define TPM_RC_PCR_CHANGED (TPM_RC) (RC_VER1 + 0x028)
474 #define TPM_RC_UPGRADE (TPM_RC) (RC_VER1 + 0x02D)
475 #define TPM_RC_TOO_MANY_CONTEXTS (TPM_RC) (RC_VER1 + 0x02E)
476 #define TPM_RC_AUTH_UNAVAILABLE (TPM_RC) (RC_VER1 + 0x02F)
477 #define TPM_RC_REBOOT (TPM_RC) (RC_VER1 + 0x030)
478 #define TPM_RC_UNBALANCED (TPM_RC) (RC_VER1 + 0x031)
479 #define TPM_RC_COMMAND_SIZE (TPM_RC) (RC_VER1 + 0x042)
480 #define TPM_RC_COMMAND_CODE (TPM_RC) (RC_VER1 + 0x043)
481 #define TPM_RC_AUTHSIZE (TPM_RC) (RC_VER1 + 0x044)
482 #define TPM_RC_AUTH_CONTEXT (TPM_RC) (RC_VER1 + 0x045)
483 #define TPM_RC_NV_RANGE (TPM_RC) (RC_VER1 + 0x046)
484 #define TPM_RC_NV_SIZE (TPM_RC) (RC_VER1 + 0x047)
485 #define TPM_RC_NV_LOCKED (TPM_RC) (RC_VER1 + 0x048)
486 #define TPM_RC_NV_AUTHORIZATION (TPM_RC) (RC_VER1 + 0x049)
487 #define TPM_RC_NV_UNINITIALIZED (TPM_RC) (RC_VER1 + 0x04A)
488 #define TPM_RC_NV_SPACE (TPM_RC) (RC_VER1 + 0x04B)
489 #define TPM_RC_NV_DEFINED (TPM_RC) (RC_VER1 + 0x04C)
490 #define TPM_RC_BAD_CONTEXT (TPM_RC) (RC_VER1 + 0x050)
491 #define TPM_RC_CPHASH (TPM_RC) (RC_VER1 + 0x051)
492 #define TPM_RC_PARENT (TPM_RC) (RC_VER1 + 0x052)
493 #define TPM_RC_NEEDS_TEST (TPM_RC) (RC_VER1 + 0x053)
494 #define TPM_RC_NO_RESULT (TPM_RC) (RC_VER1 + 0x054)
495 #define TPM_RC_SENSITIVE (TPM_RC) (RC_VER1 + 0x055)
496 #define RC_MAX_FMO (TPM_RC) (RC_VER1 + 0x07F)
497 #define RC_FMT1 (TPM_RC) (0x080)
498 #define TPM_RC_ASYMMETRIC (TPM_RC) (RC_FMT1 + 0x001)
499 #define TPM_RCS_ASYMMETRIC (TPM_RC) (RC_FMT1 + 0x001)
500 #define TPM_RC_ATTRIBUTES (TPM_RC) (RC_FMT1 + 0x002)
501 #define TPM_RCS_ATTRIBUTES (TPM_RC) (RC_FMT1 + 0x002)
502 #define TPM_RC_HASH (TPM_RC) (RC_FMT1 + 0x003)
503 #define TPM_RCS_HASH (TPM_RC) (RC_FMT1 + 0x003)
504 #define TPM_RC_VALUE (TPM_RC) (RC_FMT1 + 0x004)
505 #define TPM_RCS_VALUE (TPM_RC) (RC_FMT1 + 0x004)
506 #define TPM_RC_HIERARCHY (TPM_RC) (RC_FMT1 + 0x005)
507 #define TPM_RCS_HIERARCHY (TPM_RC) (RC_FMT1 + 0x005)
508 #define TPM_RC_KEY_SIZE (TPM_RC) (RC_FMT1 + 0x007)
509 #define TPM_RCS_KEY_SIZE (TPM_RC) (RC_FMT1 + 0x007)
510 #define TPM_RC_MGF (TPM_RC) (RC_FMT1 + 0x008)
511 #define TPM_RCS_MGF (TPM_RC) (RC_FMT1 + 0x008)
512 #define TPM_RC_MODE (TPM_RC) (RC_FMT1 + 0x009)
513 #define TPM_RCS_MODE (TPM_RC) (RC_FMT1 + 0x009)
514 #define TPM_RC_TYPE (TPM_RC) (RC_FMT1 + 0x00A)
515 #define TPM_RCS_TYPE (TPM_RC) (RC_FMT1 + 0x00A)
516 #define TPM_RC_HANDLE (TPM_RC) (RC_FMT1 + 0x00B)
517 #define TPM_RCS_HANDLE (TPM_RC) (RC_FMT1 + 0x00B)
518 #define TPM_RC_KDF (TPM_RC) (RC_FMT1 + 0x00C)
519 #define TPM_RCS_KDF (TPM_RC) (RC_FMT1 + 0x00C)
520 #define TPM_RC_RANGE (TPM_RC) (RC_FMT1 + 0x00D)
521 #define TPM_RCS_RANGE (TPM_RC) (RC_FMT1 + 0x00D)
522 #define TPM_RC_AUTH_FAIL (TPM_RC) (RC_FMT1 + 0x00E)
523 #define TPM_RCS_AUTH_FAIL (TPM_RC) (RC_FMT1 + 0x00E)
524 #define TPM_RC_NONCE (TPM_RC) (RC_FMT1 + 0x00F)
525 #define TPM_RCS_NONCE (TPM_RC) (RC_FMT1 + 0x00F)
526 #define TPM_RC_PP (TPM_RC) (RC_FMT1 + 0x010)

```



```
527 #define TPM_RCS_PP (TPM_RC) (RC_FMT1 + 0x010)
528 #define TPM_RC_SCHEME (TPM_RC) (RC_FMT1 + 0x012)
529 #define TPM_RCS_SCHEME (TPM_RC) (RC_FMT1 + 0x012)
530 #define TPM_RC_SIZE (TPM_RC) (RC_FMT1 + 0x015)
531 #define TPM_RCS_SIZE (TPM_RC) (RC_FMT1 + 0x015)
532 #define TPM_RC_SYMMETRIC (TPM_RC) (RC_FMT1 + 0x016)
533 #define TPM_RCS_SYMMETRIC (TPM_RC) (RC_FMT1 + 0x016)
534 #define TPM_RC_TAG (TPM_RC) (RC_FMT1 + 0x017)
535 #define TPM_RCS_TAG (TPM_RC) (RC_FMT1 + 0x017)
536 #define TPM_RC_SELECTOR (TPM_RC) (RC_FMT1 + 0x018)
537 #define TPM_RCS_SELECTOR (TPM_RC) (RC_FMT1 + 0x018)
538 #define TPM_RC_INSUFFICIENT (TPM_RC) (RC_FMT1 + 0x01A)
539 #define TPM_RCS_INSUFFICIENT (TPM_RC) (RC_FMT1 + 0x01A)
540 #define TPM_RC_SIGNATURE (TPM_RC) (RC_FMT1 + 0x01B)
541 #define TPM_RCS_SIGNATURE (TPM_RC) (RC_FMT1 + 0x01B)
542 #define TPM_RC_KEY (TPM_RC) (RC_FMT1 + 0x01C)
543 #define TPM_RCS_KEY (TPM_RC) (RC_FMT1 + 0x01C)
544 #define TPM_RC_POLICY_FAIL (TPM_RC) (RC_FMT1 + 0x01D)
545 #define TPM_RCS_POLICY_FAIL (TPM_RC) (RC_FMT1 + 0x01D)
546 #define TPM_RC_INTEGRITY (TPM_RC) (RC_FMT1 + 0x01F)
547 #define TPM_RCS_INTEGRITY (TPM_RC) (RC_FMT1 + 0x01F)
548 #define TPM_RC_TICKET (TPM_RC) (RC_FMT1 + 0x020)
549 #define TPM_RCS_TICKET (TPM_RC) (RC_FMT1 + 0x020)
550 #define TPM_RC_RESERVED_BITS (TPM_RC) (RC_FMT1 + 0x021)
551 #define TPM_RCS_RESERVED_BITS (TPM_RC) (RC_FMT1 + 0x021)
552 #define TPM_RC_BAD_AUTH (TPM_RC) (RC_FMT1 + 0x022)
553 #define TPM_RCS_BAD_AUTH (TPM_RC) (RC_FMT1 + 0x022)
554 #define TPM_RC_EXPIRED (TPM_RC) (RC_FMT1 + 0x023)
555 #define TPM_RCS_EXPIRED (TPM_RC) (RC_FMT1 + 0x023)
556 #define TPM_RC_POLICY_CC (TPM_RC) (RC_FMT1 + 0x024)
557 #define TPM_RCS_POLICY_CC (TPM_RC) (RC_FMT1 + 0x024)
558 #define TPM_RC_BINDING (TPM_RC) (RC_FMT1 + 0x025)
559 #define TPM_RCS_BINDING (TPM_RC) (RC_FMT1 + 0x025)
560 #define TPM_RC_CURVE (TPM_RC) (RC_FMT1 + 0x026)
561 #define TPM_RCS_CURVE (TPM_RC) (RC_FMT1 + 0x026)
562 #define TPM_RC_ECC_POINT (TPM_RC) (RC_FMT1 + 0x027)
563 #define TPM_RCS_ECC_POINT (TPM_RC) (RC_FMT1 + 0x027)
564 #define TPM_RC_FW_LIMITED (TPM_RC) (RC_FMT1 + 0x028)
565 #define TPM_RC_SVN_LIMITED (TPM_RC) (RC_FMT1 + 0x029)
566 #define RC_WARN (TPM_RC) (0x900)
567 #define TPM_RC_CONTEXT_GAP (TPM_RC) (RC_WARN + 0x001)
568 #define TPM_RC_OBJECT_MEMORY (TPM_RC) (RC_WARN + 0x002)
569 #define TPM_RC_SESSION_MEMORY (TPM_RC) (RC_WARN + 0x003)
570 #define TPM_RC_MEMORY (TPM_RC) (RC_WARN + 0x004)
571 #define TPM_RC_SESSION_HANDLES (TPM_RC) (RC_WARN + 0x005)
572 #define TPM_RC_OBJECT_HANDLES (TPM_RC) (RC_WARN + 0x006)
573 #define TPM_RC_LOCALITY (TPM_RC) (RC_WARN + 0x007)
574 #define TPM_RC_YIELDED (TPM_RC) (RC_WARN + 0x008)
575 #define TPM_RC_CANCELED (TPM_RC) (RC_WARN + 0x009)
576 #define TPM_RC_TESTING (TPM_RC) (RC_WARN + 0x00A)
577 #define TPM_RC_REFERENCE_H0 (TPM_RC) (RC_WARN + 0x010)
578 #define TPM_RC_REFERENCE_H1 (TPM_RC) (RC_WARN + 0x011)
579 #define TPM_RC_REFERENCE_H2 (TPM_RC) (RC_WARN + 0x012)
580 #define TPM_RC_REFERENCE_H3 (TPM_RC) (RC_WARN + 0x013)
581 #define TPM_RC_REFERENCE_H4 (TPM_RC) (RC_WARN + 0x014)
582 #define TPM_RC_REFERENCE_H5 (TPM_RC) (RC_WARN + 0x015)
583 #define TPM_RC_REFERENCE_H6 (TPM_RC) (RC_WARN + 0x016)
584 #define TPM_RC_REFERENCE_S0 (TPM_RC) (RC_WARN + 0x018)
585 #define TPM_RC_REFERENCE_S1 (TPM_RC) (RC_WARN + 0x019)
586 #define TPM_RC_REFERENCE_S2 (TPM_RC) (RC_WARN + 0x01A)
587 #define TPM_RC_REFERENCE_S3 (TPM_RC) (RC_WARN + 0x01B)
588 #define TPM_RC_REFERENCE_S4 (TPM_RC) (RC_WARN + 0x01C)
589 #define TPM_RC_REFERENCE_S5 (TPM_RC) (RC_WARN + 0x01D)
590 #define TPM_RC_REFERENCE_S6 (TPM_RC) (RC_WARN + 0x01E)
591 #define TPM_RC_NV_RATE (TPM_RC) (RC_WARN + 0x020)
592 #define TPM_RC_LOCKOUT (TPM_RC) (RC_WARN + 0x021)
```

```

593 #define TPM_RC_RETRY (TPM_RC) (RC_WARN + 0x022)
594 #define TPM_RC_NV_UNAVAILABLE (TPM_RC) (RC_WARN + 0x023)
595 #define TPM_RC_NOT_USED (TPM_RC) (RC_WARN + 0x7F)
596 #define TPM_RC_H (TPM_RC) (0x000)
597 #define TPM_RC_P (TPM_RC) (0x040)
598 #define TPM_RC_S (TPM_RC) (0x800)
599 #define TPM_RC_1 (TPM_RC) (0x100)
600 #define TPM_RC_2 (TPM_RC) (0x200)
601 #define TPM_RC_3 (TPM_RC) (0x300)
602 #define TPM_RC_4 (TPM_RC) (0x400)
603 #define TPM_RC_5 (TPM_RC) (0x500)
604 #define TPM_RC_6 (TPM_RC) (0x600)
605 #define TPM_RC_7 (TPM_RC) (0x700)
606 #define TPM_RC_8 (TPM_RC) (0x800)
607 #define TPM_RC_9 (TPM_RC) (0x900)
608 #define TPM_RC_A (TPM_RC) (0xA00)
609 #define TPM_RC_B (TPM_RC) (0xB00)
610 #define TPM_RC_C (TPM_RC) (0xC00)
611 #define TPM_RC_D (TPM_RC) (0xD00)
612 #define TPM_RC_E (TPM_RC) (0xE00)
613 #define TPM_RC_F (TPM_RC) (0xF00)
614 #define TPM_RC_N_MASK (TPM_RC) (0xF00)
615
616 // Table "Definition of TPM_CLOCK_ADJUST Constants" (Part 2: Structures)
617 typedef INT8 TPM_CLOCK_ADJUST;
618 #define TYPE_OF_TPM_CLOCK_ADJUST INT8
619 #define TPM_CLOCK_COARSE_SLOWER (TPM_CLOCK_ADJUST) (-3)
620 #define TPM_CLOCK_MEDIUM_SLOWER (TPM_CLOCK_ADJUST) (-2)
621 #define TPM_CLOCK_FINE_SLOWER (TPM_CLOCK_ADJUST) (-1)
622 #define TPM_CLOCK_NO_CHANGE (TPM_CLOCK_ADJUST) (0)
623 #define TPM_CLOCK_FINE_FASTER (TPM_CLOCK_ADJUST) (1)
624 #define TPM_CLOCK_MEDIUM_FASTER (TPM_CLOCK_ADJUST) (2)
625 #define TPM_CLOCK_COARSE_FASTER (TPM_CLOCK_ADJUST) (3)
626
627 // Table "Definition of TPM_EO Constants" (Part 2: Structures)
628 typedef UINT16 TPM_EO;
629 #define TYPE_OF_TPM_EO UINT16
630 #define TPM_EO_EQ (TPM_EO) (0x0000)
631 #define TPM_EO_NEQ (TPM_EO) (0x0001)
632 #define TPM_EO_SIGNED_GT (TPM_EO) (0x0002)
633 #define TPM_EO_UNSIGNED_GT (TPM_EO) (0x0003)
634 #define TPM_EO_SIGNED_LT (TPM_EO) (0x0004)
635 #define TPM_EO_UNSIGNED_LT (TPM_EO) (0x0005)
636 #define TPM_EO_SIGNED_GE (TPM_EO) (0x0006)
637 #define TPM_EO_UNSIGNED_GE (TPM_EO) (0x0007)
638 #define TPM_EO_SIGNED_LE (TPM_EO) (0x0008)
639 #define TPM_EO_UNSIGNED_LE (TPM_EO) (0x0009)
640 #define TPM_EO_BITSET (TPM_EO) (0x000A)
641 #define TPM_EO_BITCLEAR (TPM_EO) (0x000B)
642
643 // Table "Definition of TPM_ST Constants" (Part 2: Structures)
644 typedef UINT16 TPM_ST;
645 #define TYPE_OF_TPM_ST UINT16
646 #define TPM_ST_RSP_COMMAND (TPM_ST) (0x00C4)
647 #define TPM_ST_NULL (TPM_ST) (0x8000)
648 #define TPM_ST_NO_SESSIONS (TPM_ST) (0x8001)
649 #define TPM_ST_SESSIONS (TPM_ST) (0x8002)
650 #define TPM_ST_ATTEST_NV (TPM_ST) (0x8014)
651 #define TPM_ST_ATTEST_COMMAND_AUDIT (TPM_ST) (0x8015)
652 #define TPM_ST_ATTEST_SESSION_AUDIT (TPM_ST) (0x8016)
653 #define TPM_ST_ATTEST_CERTIFY (TPM_ST) (0x8017)
654 #define TPM_ST_ATTEST_QUOTE (TPM_ST) (0x8018)
655 #define TPM_ST_ATTEST_TIME (TPM_ST) (0x8019)
656 #define TPM_ST_ATTEST_CREATION (TPM_ST) (0x801A)
657 #define TPM_ST_ATTEST_NV_DIGEST (TPM_ST) (0x801C)
658 #define TPM_ST_CREATION (TPM_ST) (0x8021)

```

```

659 #define TPM_ST VERIFIED (TPM_ST) (0x8022)
660 #define TPM_ST AUTH_SECRET (TPM_ST) (0x8023)
661 #define TPM_ST HASHCHECK (TPM_ST) (0x8024)
662 #define TPM_ST AUTH_SIGNED (TPM_ST) (0x8025)
663 #define TPM_ST FU_MANIFEST (TPM_ST) (0x8029)
664
665 // Table "Definition of TPM_SU Constants" (Part 2: Structures)
666 typedef UINT16 TPM_SU;
667 #define TYPE_OF_TPM_SU UINT16
668 #define TPM_SU_CLEAR (TPM_SU) (0x0000)
669 #define TPM_SU_STATE (TPM_SU) (0x0001)
670
671 // Table "Definition of TPM_SE Constants" (Part 2: Structures)
672 typedef UINT8 TPM_SE;
673 #define TYPE_OF_TPM_SE UINT8
674 #define TPM_SE_HMAC (TPM_SE) (0x00)
675 #define TPM_SE_POLICY (TPM_SE) (0x01)
676 #define TPM_SE_TRIAL (TPM_SE) (0x03)
677
678 // Table "Definition of TPM_CAP Constants" (Part 2: Structures)
679 typedef UINT32 TPM_CAP;
680 #define TYPE_OF_TPM_CAP UINT32
681 #define TPM_CAP_FIRST (TPM_CAP) (0x00000000)
682 #define TPM_CAP_ALGS (TPM_CAP) (0x00000000)
683 #define TPM_CAP_HANDLES (TPM_CAP) (0x00000001)
684 #define TPM_CAP_COMMANDS (TPM_CAP) (0x00000002)
685 #define TPM_CAP_PP_COMMANDS (TPM_CAP) (0x00000003)
686 #define TPM_CAP_AUDIT_COMMANDS (TPM_CAP) (0x00000004)
687 #define TPM_CAP_PCERS (TPM_CAP) (0x00000005)
688 #define TPM_CAP_TPM_PROPERTIES (TPM_CAP) (0x00000006)
689 #define TPM_CAP_PCR_PROPERTIES (TPM_CAP) (0x00000007)
690 #define TPM_CAP_ECC_CURVES (TPM_CAP) (0x00000008)
691 #define TPM_CAP_AUTH_POLICIES (TPM_CAP) (0x00000009)
692 #define TPM_CAP_ACT (TPM_CAP) (0x0000000A)
693 #define TPM_CAP_LAST (TPM_CAP) (0x0000000A)
694 #define TPM_CAP_VENDOR_PROPERTY (TPM_CAP) (0x00000100)
695
696 // Table "Definition of TPM_PT Constants" (Part 2: Structures)
697 typedef UINT32 TPM_PT;
698 #define TYPE_OF_TPM_PT UINT32
699 #define TPM_PT_NONE (TPM_PT) (0x00000000)
700 #define PT_GROUP (TPM_PT) (0x00000100)
701 #define PT_FIXED (TPM_PT) (PT_GROUP * 1)
702 #define TPM_PT_FAMILY_INDICATOR (TPM_PT) (PT_FIXED + 0)
703 #define TPM_PT_LEVEL (TPM_PT) (PT_FIXED + 1)
704 #define TPM_PT_REVISION (TPM_PT) (PT_FIXED + 2)
705 #define TPM_PT_DAY_OF_YEAR (TPM_PT) (PT_FIXED + 3)
706 #define TPM_PT_YEAR (TPM_PT) (PT_FIXED + 4)
707 #define TPM_PT_MANUFACTURER (TPM_PT) (PT_FIXED + 5)
708 #define TPM_PT_VENDOR_STRING_1 (TPM_PT) (PT_FIXED + 6)
709 #define TPM_PT_VENDOR_STRING_2 (TPM_PT) (PT_FIXED + 7)
710 #define TPM_PT_VENDOR_STRING_3 (TPM_PT) (PT_FIXED + 8)
711 #define TPM_PT_VENDOR_STRING_4 (TPM_PT) (PT_FIXED + 9)
712 #define TPM_PT_VENDOR_TPM_TYPE (TPM_PT) (PT_FIXED + 10)
713 #define TPM_PT_FIRMWARE_VERSION_1 (TPM_PT) (PT_FIXED + 11)
714 #define TPM_PT_FIRMWARE_VERSION_2 (TPM_PT) (PT_FIXED + 12)
715 #define TPM_PT_INPUT_BUFFER (TPM_PT) (PT_FIXED + 13)
716 #define TPM_PT_HR_TRANSIENT_MIN (TPM_PT) (PT_FIXED + 14)
717 #define TPM_PT_HR_PERSISTENT_MIN (TPM_PT) (PT_FIXED + 15)
718 #define TPM_PT_HR_LOADED_MIN (TPM_PT) (PT_FIXED + 16)
719 #define TPM_PT_ACTIVE_SESSIONS_MAX (TPM_PT) (PT_FIXED + 17)
720 #define TPM_PT_PCR_COUNT (TPM_PT) (PT_FIXED + 18)
721 #define TPM_PT_PCR_SELECT_MIN (TPM_PT) (PT_FIXED + 19)
722 #define TPM_PT_CONTEXT_GAP_MAX (TPM_PT) (PT_FIXED + 20)
723 #define TPM_PT_NV_COUNTERS_MAX (TPM_PT) (PT_FIXED + 22)
724 #define TPM_PT_NV_INDEX_MAX (TPM_PT) (PT_FIXED + 23)

```

```

725 #define TPM_PT_MEMORY (TPM_PT) (PT_FIXED + 24)
726 #define TPM_PT_CLOCK_UPDATE (TPM_PT) (PT_FIXED + 25)
727 #define TPM_PT_CONTEXT_HASH (TPM_PT) (PT_FIXED + 26)
728 #define TPM_PT_CONTEXT_SYM (TPM_PT) (PT_FIXED + 27)
729 #define TPM_PT_CONTEXT_SYM_SIZE (TPM_PT) (PT_FIXED + 28)
730 #define TPM_PT_ORDERLY_COUNT (TPM_PT) (PT_FIXED + 29)
731 #define TPM_PT_MAX_COMMAND_SIZE (TPM_PT) (PT_FIXED + 30)
732 #define TPM_PT_MAX_RESPONSE_SIZE (TPM_PT) (PT_FIXED + 31)
733 #define TPM_PT_MAX_DIGEST (TPM_PT) (PT_FIXED + 32)
734 #define TPM_PT_MAX_OBJECT_CONTEXT (TPM_PT) (PT_FIXED + 33)
735 #define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED + 34)
736 #define TPM_PT_PS_FAMILY_INDICATOR (TPM_PT) (PT_FIXED + 35)
737 #define TPM_PT_PS_LEVEL (TPM_PT) (PT_FIXED + 36)
738 #define TPM_PT_PS_REVISION (TPM_PT) (PT_FIXED + 37)
739 #define TPM_PT_PS_DAY_OF_YEAR (TPM_PT) (PT_FIXED + 38)
740 #define TPM_PT_PS_YEAR (TPM_PT) (PT_FIXED + 39)
741 #define TPM_PT_SPLIT_MAX (TPM_PT) (PT_FIXED + 40)
742 #define TPM_PT_TOTAL_COMMANDS (TPM_PT) (PT_FIXED + 41)
743 #define TPM_PT_LIBRARY_COMMANDS (TPM_PT) (PT_FIXED + 42)
744 #define TPM_PT_VENDOR_COMMANDS (TPM_PT) (PT_FIXED + 43)
745 #define TPM_PT_NV_BUFFER_MAX (TPM_PT) (PT_FIXED + 44)
746 #define TPM_PT_MODES (TPM_PT) (PT_FIXED + 45)
747 #define TPM_PT_MAX_CAP_BUFFER (TPM_PT) (PT_FIXED + 46)
748 #define TPM_PT_FIRMWARE_SVN (TPM_PT) (PT_FIXED + 47)
749 #define TPM_PT_FIRMWARE_MAX_SVN (TPM_PT) (PT_FIXED + 48)
750 #define PT_VAR (TPM_PT) (PT_GROUP * 2)
751 #define TPM_PT_PERMANENT (TPM_PT) (PT_VAR + 0)
752 #define TPM_PT_STARTUP_CLEAR (TPM_PT) (PT_VAR + 1)
753 #define TPM_PT_HR_NV_INDEX (TPM_PT) (PT_VAR + 2)
754 #define TPM_PT_HR_LOADED (TPM_PT) (PT_VAR + 3)
755 #define TPM_PT_HR_LOADED_AVAIL (TPM_PT) (PT_VAR + 4)
756 #define TPM_PT_HR_ACTIVE (TPM_PT) (PT_VAR + 5)
757 #define TPM_PT_HR_ACTIVE_AVAIL (TPM_PT) (PT_VAR + 6)
758 #define TPM_PT_HR_TRANSIENT_AVAIL (TPM_PT) (PT_VAR + 7)
759 #define TPM_PT_HR_PERSISTENT (TPM_PT) (PT_VAR + 8)
760 #define TPM_PT_HR_PERSISTENT_AVAIL (TPM_PT) (PT_VAR + 9)
761 #define TPM_PT_NV_COUNTERS (TPM_PT) (PT_VAR + 10)
762 #define TPM_PT_NV_COUNTERS_AVAIL (TPM_PT) (PT_VAR + 11)
763 #define TPM_PT_ALGORITHM_SET (TPM_PT) (PT_VAR + 12)
764 #define TPM_PT_LOADED_CURVES (TPM_PT) (PT_VAR + 13)
765 #define TPM_PT_LOCKOUT_COUNTER (TPM_PT) (PT_VAR + 14)
766 #define TPM_PT_MAX_AUTH_FAIL (TPM_PT) (PT_VAR + 15)
767 #define TPM_PT_LOCKOUT_INTERVAL (TPM_PT) (PT_VAR + 16)
768 #define TPM_PT_LOCKOUT_RECOVERY (TPM_PT) (PT_VAR + 17)
769 #define TPM_PT_NV_WRITE_RECOVERY (TPM_PT) (PT_VAR + 18)
770 #define TPM_PT_AUDIT_COUNTER_0 (TPM_PT) (PT_VAR + 19)
771 #define TPM_PT_AUDIT_COUNTER_1 (TPM_PT) (PT_VAR + 20)
772
773 // Table "Definition of TPM_PT_PCR Constants" (Part 2: Structures)
774 typedef UINT32 TPM_PT_PCR;
775 #define TYPE_OF_TPM_PT_PCR UINT32
776 #define TPM_PT_PCR_FIRST (TPM_PT_PCR) (0x00000000)
777 #define TPM_PT_PCR_SAVE (TPM_PT_PCR) (0x00000000)
778 #define TPM_PT_PCR_EXTEND_L0 (TPM_PT_PCR) (0x00000001)
779 #define TPM_PT_PCR_RESET_L0 (TPM_PT_PCR) (0x00000002)
780 #define TPM_PT_PCR_EXTEND_L1 (TPM_PT_PCR) (0x00000003)
781 #define TPM_PT_PCR_RESET_L1 (TPM_PT_PCR) (0x00000004)
782 #define TPM_PT_PCR_EXTEND_L2 (TPM_PT_PCR) (0x00000005)
783 #define TPM_PT_PCR_RESET_L2 (TPM_PT_PCR) (0x00000006)
784 #define TPM_PT_PCR_EXTEND_L3 (TPM_PT_PCR) (0x00000007)
785 #define TPM_PT_PCR_RESET_L3 (TPM_PT_PCR) (0x00000008)
786 #define TPM_PT_PCR_EXTEND_L4 (TPM_PT_PCR) (0x00000009)
787 #define TPM_PT_PCR_RESET_L4 (TPM_PT_PCR) (0x0000000A)
788 #define TPM_PT_PCR_NO_INCREMENT (TPM_PT_PCR) (0x00000011)
789 #define TPM_PT_PCR_DRTM_RESET (TPM_PT_PCR) (0x00000012)
790 #define TPM_PT_PCR_POLICY (TPM_PT_PCR) (0x00000013)

```



```

791 #define TPM_PT_PCR_AUTH          (TPM_PT_PCR) (0x00000014)
792 #define TPM_PT_PCR_LAST          (TPM_PT_PCR) (0x00000014)
793
794 // Table "Definition of TPM_PS Constants" (Part 2: Structures)
795 typedef UINT32 TPM_PS;
796 #define TYPE_OF_TPM_PS          UINT32
797 #define TPM_PS_MAIN              (TPM_PS) (0x00000000)
798 #define TPM_PS_PC                (TPM_PS) (0x00000001)
799 #define TPM_PS_PDA               (TPM_PS) (0x00000002)
800 #define TPM_PS_CELL_PHONE       (TPM_PS) (0x00000003)
801 #define TPM_PS_SERVER           (TPM_PS) (0x00000004)
802 #define TPM_PS_PERIPHERAL       (TPM_PS) (0x00000005)
803 #define TPM_PS_TSS               (TPM_PS) (0x00000006)
804 #define TPM_PS_STORAGE          (TPM_PS) (0x00000007)
805 #define TPM_PS_AUTHENTICATION   (TPM_PS) (0x00000008)
806 #define TPM_PS_EMBEDDED         (TPM_PS) (0x00000009)
807 #define TPM_PS_HARDCOPY         (TPM_PS) (0x0000000A)
808 #define TPM_PS_INFRASTRUCTURE   (TPM_PS) (0x0000000B)
809 #define TPM_PS_VIRTUALIZATION   (TPM_PS) (0x0000000C)
810 #define TPM_PS_TNC              (TPM_PS) (0x0000000D)
811 #define TPM_PS_MULTI_TENANT     (TPM_PS) (0x0000000E)
812 #define TPM_PS_TC               (TPM_PS) (0x0000000F)
813
814 // Table "Definition of Types for Handles" (Part 2: Structures)
815 typedef UINT32 TPM_HANDLE;
816 #define TYPE_OF_TPM_HANDLE      UINT32
817
818 // Table "Definition of TPM_HT Constants" (Part 2: Structures)
819 typedef UINT8 TPM_HT;
820 #define TYPE_OF_TPM_HT          UINT8
821 #define TPM_HT_PCR              (TPM_HT) (0x00)
822 #define TPM_HT_NV_INDEX         (TPM_HT) (0x01)
823 #define TPM_HT_HMAC_SESSION     (TPM_HT) (0x02)
824 #define TPM_HT_LOADED_SESSION   (TPM_HT) (0x02)
825 #define TPM_HT_POLICY_SESSION   (TPM_HT) (0x03)
826 #define TPM_HT_SAVED_SESSION    (TPM_HT) (0x03)
827 #define TPM_HT_EXTERNAL_NV      (TPM_HT) (0x11)
828 #define TPM_HT_PERMANENT_NV     (TPM_HT) (0x12)
829 #define TPM_HT_PERMANENT        (TPM_HT) (0x40)
830 #define TPM_HT_TRANSIENT        (TPM_HT) (0x80)
831 #define TPM_HT_PERSISTENT       (TPM_HT) (0x81)
832 #define TPM_HT_AC               (TPM_HT) (0x90)
833
834 // Table "Definition of TPM_RH Constants" (Part 2: Structures)
835 typedef TPM_HANDLE TPM_RH;
836 #define TYPE_OF_TPM_RH          TPM_HANDLE
837 #define TPM_RH_FIRST            (TPM_RH) (0x40000000)
838 #define TPM_RH_SRK              (TPM_RH) (0x40000000)
839 #define TPM_RH_OWNER            (TPM_RH) (0x40000001)
840 #define TPM_RH_REVOKE           (TPM_RH) (0x40000002)
841 #define TPM_RH_TRANSPORT        (TPM_RH) (0x40000003)
842 #define TPM_RH_OPERATOR         (TPM_RH) (0x40000004)
843 #define TPM_RH_ADMIN            (TPM_RH) (0x40000005)
844 #define TPM_RH_EK               (TPM_RH) (0x40000006)
845 #define TPM_RH_NULL             (TPM_RH) (0x40000007)
846 #define TPM_RH_UNASSIGNED       (TPM_RH) (0x40000008)
847 #define TPM_RS_PW               (TPM_RH) (0x40000009)
848 #define TPM_RH_LOCKOUT          (TPM_RH) (0x4000000A)
849 #define TPM_RH_ENDORSEMENT      (TPM_RH) (0x4000000B)
850 #define TPM_RH_PLATFORM         (TPM_RH) (0x4000000C)
851 #define TPM_RH_PLATFORM_NV      (TPM_RH) (0x4000000D)
852 #define TPM_RH_AUTH_00          (TPM_RH) (0x40000010)
853 #define TPM_RH_AUTH_FF          (TPM_RH) (0x4000010F)
854 #define TPM_RH_ACT_0            (TPM_RH) (0x40000110)
855 #define TPM_RH_ACT_F            (TPM_RH) (0x4000011F)
856 #define TPM_RH_FW_OWNER         (TPM_RH) (0x40000140)

```

```

857 #define TPM_RH_FW_ENDORSEMENT (TPM_RH) (0x40000141)
858 #define TPM_RH_FW_PLATFORM (TPM_RH) (0x40000142)
859 #define TPM_RH_FW_NULL (TPM_RH) (0x40000143)
860 #define TPM_RH_SVN_OWNER_BASE (TPM_RH) (0x40010000)
861 #define TPM_RH_SVN_ENDORSEMENT_BASE (TPM_RH) (0x40020000)
862 #define TPM_RH_SVN_PLATFORM_BASE (TPM_RH) (0x40030000)
863 #define TPM_RH_SVN_NULL_BASE (TPM_RH) (0x40040000)
864 #define TPM_RH_LAST (TPM_RH) (0x4004FFFF)
865 // Note: 0x40010001-0x4001FFFF, 0x40020001-0x4002FFFF,
866 // 0x40030001-0x4003FFFF, and 0x40040001-0x4004FFFF are
867 // valid reserved handles, but are not returned from
868 // TPM2_GetCapability().
869
870 // Table "Definition of TPM_HC Constants" (Part 2: Structures)
871 typedef TPM_HANDLE TPM_HC;
872 #define TYPE_OF_TPM_HC TPM_HANDLE
873 #define HR_HANDLE_MASK (TPM_HC) (0x00FFFFFF)
874 #define HR_RANGE_MASK (TPM_HC) (0xFF000000)
875 #define HR_SHIFT (TPM_HC) (24)
876 #define HR_PCR (TPM_HC) ((TPM_HT_PCR << HR_SHIFT))
877 #define HR_HMAC_SESSION (TPM_HC) ((TPM_HT_HMAC_SESSION << HR_SHIFT))
878 #define HR_POLICY_SESSION (TPM_HC) ((TPM_HT_POLICY_SESSION << HR_SHIFT))
879 #define HR_TRANSIENT (TPM_HC) ((TPM_HT_TRANSIENT << HR_SHIFT))
880 #define HR_PERSISTENT (TPM_HC) ((TPM_HT_PERSISTENT << HR_SHIFT))
881 #define HR_NV_INDEX (TPM_HC) ((TPM_HT_NV_INDEX << HR_SHIFT))
882 #define HR_EXTERNAL_NV (TPM_HC) ((TPM_HT_EXTERNAL_NV << HR_SHIFT))
883 #define HR_PERMANENT_NV (TPM_HC) ((TPM_HT_PERMANENT_NV << HR_SHIFT))
884 #define HR_PERMANENT (TPM_HC) ((TPM_HT_PERMANENT << HR_SHIFT))
885 #define PCR_FIRST (TPM_HC) ((HR_PCR + 0))
886 #define PCR_LAST (TPM_HC) ((PCR_FIRST + IMPLEMENTATION_PCR - 1))
887 #define HMAC_SESSION_FIRST (TPM_HC) ((HR_HMAC_SESSION + 0))
888 #define HMAC_SESSION_LAST (TPM_HC) ((HMAC_SESSION_FIRST + MAX_ACTIVE_SESSIONS - 1))
889 #define LOADED_SESSION_FIRST (TPM_HC) (HMAC_SESSION_FIRST)
890 #define LOADED_SESSION_LAST (TPM_HC) (HMAC_SESSION_LAST)
891 #define POLICY_SESSION_FIRST (TPM_HC) ((HR_POLICY_SESSION + 0))
892 #define POLICY_SESSION_LAST (TPM_HC) ((POLICY_SESSION_FIRST + MAX_ACTIVE_SESSIONS -
1))
893 #define TRANSIENT_FIRST (TPM_HC) ((HR_TRANSIENT + 0))
894 #define ACTIVE_SESSION_FIRST (TPM_HC) (POLICY_SESSION_FIRST)
895 #define ACTIVE_SESSION_LAST (TPM_HC) (POLICY_SESSION_LAST)
896 #define TRANSIENT_LAST (TPM_HC) ((TRANSIENT_FIRST + MAX_LOADED_OBJECTS - 1))
897 #define PERSISTENT_FIRST (TPM_HC) ((HR_PERSISTENT + 0))
898 #define PERSISTENT_LAST (TPM_HC) ((PERSISTENT_FIRST + 0x00FFFFFF))
899 #define SVN_OWNER_FIRST (TPM_HC) ((TPM_RH_SVN_OWNER_BASE + 0x0000))
900 #define SVN_OWNER_LAST (TPM_HC) ((TPM_RH_SVN_OWNER_BASE + 0xFFFF))
901 #define SVN_ENDORSEMENT_FIRST (TPM_HC) ((TPM_RH_SVN_ENDORSEMENT_BASE + 0x0000))
902 #define SVN_ENDORSEMENT_LAST (TPM_HC) ((TPM_RH_SVN_ENDORSEMENT_BASE + 0xFFFF))
903 #define SVN_PLATFORM_FIRST (TPM_HC) ((TPM_RH_SVN_PLATFORM_BASE + 0x0000))
904 #define SVN_PLATFORM_LAST (TPM_HC) ((TPM_RH_SVN_PLATFORM_BASE + 0xFFFF))
905 #define SVN_NULL_FIRST (TPM_HC) ((TPM_RH_SVN_NULL_BASE + 0x0000))
906 #define SVN_NULL_LAST (TPM_HC) ((TPM_RH_SVN_NULL_BASE + 0xFFFF))
907 #define PLATFORM_PERSISTENT (TPM_HC) ((PERSISTENT_FIRST + 0x00800000))
908 #define NV_INDEX_FIRST (TPM_HC) ((HR_NV_INDEX + 0))
909 #define NV_INDEX_LAST (TPM_HC) ((NV_INDEX_FIRST + 0x00FFFFFF))
910 #define EXTERNAL_NV_FIRST (TPM_HC) ((HR_EXTERNAL_NV + 0))
911 #define EXTERNAL_NV_LAST (TPM_HC) ((EXTERNAL_NV_FIRST + 0x00FFFFFF))
912 #define PERMANENT_NV_FIRST (TPM_HC) ((HR_PERMANENT_NV + 0))
913 #define PERMANENT_NV_LAST (TPM_HC) ((PERMANENT_NV_FIRST + 0x00FFFFFF))
914 #define PERMANENT_FIRST (TPM_HC) (TPM_RH_FIRST)
915 #define PERMANENT_LAST (TPM_HC) (TPM_RH_LAST)
916 #define HR_NV_AC (TPM_HC) (((TPM_HT_NV_INDEX << HR_SHIFT) + 0xD00000))
917 #define NV_AC_FIRST (TPM_HC) ((HR_NV_AC + 0))
918 #define NV_AC_LAST (TPM_HC) ((HR_NV_AC + 0x0000FFFF))
919 #define HR_AC (TPM_HC) ((TPM_HT_AC << HR_SHIFT))
920 #define AC_FIRST (TPM_HC) ((HR_AC + 0))
921 #define AC_LAST (TPM_HC) ((HR_AC + 0x0000FFFF))

```



```

922
923 // Table "Definition of TPMA_ALGORITHM Bits" (Part 2: Structures)
924 #define TYPE_OF_TPMA_ALGORITHM      UINT32
925 #define TPMA_ALGORITHM_TO_UINT32(a) (*(UINT32*)&(a))
926 #define UINT32_TO_TPMA_ALGORITHM(a) (*(TPMA_ALGORITHM*)&(a))
927 #define TPMA_ALGORITHM_TO_BYTE_ARRAY(i, a) \
928     UINT32_TO_BYTE_ARRAY((TPMA_ALGORITHM_TO_UINT32(i)), (a))
929 #define BYTE_ARRAY_TO_TPMA_ALGORITHM(i, a) \
930     { \
931         UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
932         i = UINT32_TO_TPMA_ALGORITHM(x); \
933     }
934 #if USE_BIT_FIELD_STRUCTURES
935 typedef struct
936 {
937     unsigned asymmetric      : 1;
938     unsigned symmetric      : 1;
939     unsigned hash            : 1;
940     unsigned object         : 1;
941     unsigned Reserved_bits_at_4 : 4;
942     unsigned signing        : 1;
943     unsigned encrypting     : 1;
944     unsigned method         : 1;
945     unsigned Reserved_bits_at_11 : 21;
946 } TPMA_ALGORITHM;
947
948 // Initializer for the bit-field structure
949 # define TPMA_ALGORITHM_INITIALIZER(asymmetric, \
950     symmetric, \
951     hash, \
952     object, \
953     bits_at_4, \
954     signing, \
955     encrypting, \
956     method, \
957     bits_at_11) \
958 { \
959     asymmetric, symmetric, hash, object, bits_at_4, signing, encrypting, \
960     method, bits_at_11 \
961 }
962 #else // USE_BIT_FIELD_STRUCTURES
963
964 // This implements Table "Definition of TPMA_ALGORITHM Bits" (Part 2: Structures)
965 using bit masking
966 typedef UINT32 TPMA_ALGORITHM;
967 # define TPMA_ALGORITHM_asymmetric (TPMA_ALGORITHM) (1 << 0)
968 # define TPMA_ALGORITHM_symmetric (TPMA_ALGORITHM) (1 << 1)
969 # define TPMA_ALGORITHM_hash (TPMA_ALGORITHM) (1 << 2)
970 # define TPMA_ALGORITHM_object (TPMA_ALGORITHM) (1 << 3)
971 # define TPMA_ALGORITHM_signing (TPMA_ALGORITHM) (1 << 8)
972 # define TPMA_ALGORITHM_encrypting (TPMA_ALGORITHM) (1 << 9)
973 # define TPMA_ALGORITHM_method (TPMA_ALGORITHM) (1 << 10)
974
975 // This is the initializer for a TPMA_ALGORITHM bit array.
976 # define TPMA_ALGORITHM_INITIALIZER(asymmetric, \
977     symmetric, \
978     hash, \
979     object, \
980     bits_at_4, \
981     signing, \
982     encrypting, \
983     method, \
984     bits_at_11) \
985     (TPMA_ALGORITHM) ((asymmetric << 0) + (symmetric << 1) + (hash << 2) \
986     + (object << 3) + (signing << 8) + (encrypting << 9) \
987     + (method << 10))

```

```

987
988 #endif // USE_BIT_FIELD_STRUCTURES
989
990 // Table "Definition of TPMA_OBJECT Bits" (Part 2: Structures)
991 #define TYPE_OF_TPMA_OBJECT      UINT32
992 #define TPMA_OBJECT_TO_UINT32(a) (*(UINT32*)&(a))
993 #define UINT32_TO_TPMA_OBJECT(a) (*(TPMA_OBJECT*)&(a))
994 #define TPMA_OBJECT_TO_BYTE_ARRAY(i, a) \
995     UINT32_TO_BYTE_ARRAY((TPMA_OBJECT_TO_UINT32(i)), (a))
996 #define BYTE_ARRAY_TO_TPMA_OBJECT(i, a) \
997     { \
998         UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
999         i = UINT32_TO_TPMA_OBJECT(x); \
1000     }
1001 #if USE_BIT_FIELD_STRUCTURES
1002 typedef struct
1003 {
1004     unsigned Reserved_bit_at_0      : 1;
1005     unsigned fixedTPM               : 1;
1006     unsigned stClear                : 1;
1007     unsigned fixedFirmware          : 1;
1008     unsigned fixedParent            : 1;
1009     unsigned sensitiveDataOrigin    : 1;
1010     unsigned userWithAuth           : 1;
1011     unsigned adminWithPolicy        : 1;
1012     unsigned firmwareLimited        : 1;
1013     unsigned svnLimited             : 1;
1014     unsigned noDA                   : 1;
1015     unsigned encryptedDuplication   : 1;
1016     unsigned Reserved_bits_at_12    : 4;
1017     unsigned restricted              : 1;
1018     unsigned decrypt                : 1;
1019     unsigned sign                   : 1;
1020     unsigned x509sign               : 1;
1021     unsigned Reserved_bits_at_20    : 12;
1022 } TPMA_OBJECT;
1023
1024 #else // USE_BIT_FIELD_STRUCTURES
1025
1026 // This implements Table "Definition of TPMA_OBJECT Bits" (Part 2: Structures) using
1027 // bit masking
1028 typedef UINT32 TPMA_OBJECT;
1029 # define TPMA_OBJECT_fixedTPM                (TPMA_OBJECT) (1 << 1)
1030 # define TPMA_OBJECT_stClear                  (TPMA_OBJECT) (1 << 2)
1031 # define TPMA_OBJECT_fixedFirmware            (TPMA_OBJECT) (1 << 3)
1032 # define TPMA_OBJECT_fixedParent              (TPMA_OBJECT) (1 << 4)
1033 # define TPMA_OBJECT_sensitiveDataOrigin      (TPMA_OBJECT) (1 << 5)
1034 # define TPMA_OBJECT_userWithAuth            (TPMA_OBJECT) (1 << 6)
1035 # define TPMA_OBJECT_adminWithPolicy          (TPMA_OBJECT) (1 << 7)
1036 # define TPMA_OBJECT_firmwareLimited          (TPMA_OBJECT) (1 << 8)
1037 # define TPMA_OBJECT_svnLimited               (TPMA_OBJECT) (1 << 9)
1038 # define TPMA_OBJECT_noDA                     (TPMA_OBJECT) (1 << 10)
1039 # define TPMA_OBJECT_encryptedDuplication     (TPMA_OBJECT) (1 << 11)
1040 # define TPMA_OBJECT_restricted               (TPMA_OBJECT) (1 << 16)
1041 # define TPMA_OBJECT_decrypt                  (TPMA_OBJECT) (1 << 17)
1042 # define TPMA_OBJECT_sign                     (TPMA_OBJECT) (1 << 18)
1043 # define TPMA_OBJECT_x509sign                 (TPMA_OBJECT) (1 << 19)
1044 #endif // USE_BIT_FIELD_STRUCTURES
1045
1046 // Table "Definition of TPMA_SESSION Bits" (Part 2: Structures)
1047 #define TYPE_OF_TPMA_SESSION      UINT8
1048 #define TPMA_SESSION_TO_UINT8(a) (*(UINT8*)&(a))
1049 #define UINT8_TO_TPMA_SESSION(a) (*(TPMA_SESSION*)&(a))
1050 #define TPMA_SESSION_TO_BYTE_ARRAY(i, a) \
1051     UINT8_TO_BYTE_ARRAY((TPMA_SESSION_TO_UINT8(i)), (a))

```

```

1052 #define BYTE_ARRAY_TO_TPMA_SESSION(i, a) \
1053 { \
1054     UINT8 x = BYTE_ARRAY_TO_UINT8(a); \
1055     i      = UINT8_TO_TPMA_SESSION(x); \
1056 }
1057 #if USE_BIT_FIELD_STRUCTURES
1058 typedef struct
1059 {
1060     unsigned continueSession      : 1;
1061     unsigned auditExclusive       : 1;
1062     unsigned auditReset          : 1;
1063     unsigned Reserved_bits_at_3  : 2;
1064     unsigned decrypt              : 1;
1065     unsigned encrypt              : 1;
1066     unsigned audit                : 1;
1067 } TPMA_SESSION;
1068
1069 // Initializer for the bit-field structure
1070 # define TPMA_SESSION_INITIALIZER(continuesession, \
1071     auditexclusive, \
1072     auditreset, \
1073     bits_at_3, \
1074     decrypt, \
1075     encrypt, \
1076     audit) \
1077 { \
1078     continuesession, auditexclusive, auditreset, bits_at_3, decrypt, encrypt, \
1079     audit \
1080 }
1081 #else // USE_BIT_FIELD_STRUCTURES
1082
1083 // This implements Table "Definition of TPMA_SESSION Bits" (Part 2: Structures) using
1084 // bit masking
1085 typedef UINT8 TPMA_SESSION;
1086 # define TPMA_SESSION_continueSession (TPMA_SESSION) (1 << 0)
1087 # define TPMA_SESSION_auditExclusive (TPMA_SESSION) (1 << 1)
1088 # define TPMA_SESSION_auditReset (TPMA_SESSION) (1 << 2)
1089 # define TPMA_SESSION_decrypt (TPMA_SESSION) (1 << 5)
1090 # define TPMA_SESSION_encrypt (TPMA_SESSION) (1 << 6)
1091 # define TPMA_SESSION_audit (TPMA_SESSION) (1 << 7)
1092
1093 // This is the initializer for a TPMA_SESSION bit array.
1094 # define TPMA_SESSION_INITIALIZER(continuesession, \
1095     auditexclusive, \
1096     auditreset, \
1097     bits_at_3, \
1098     decrypt, \
1099     encrypt, \
1100     audit) \
1101     (TPMA_SESSION) ((continuesession << 0) + (auditexclusive << 1) \
1102     + (auditreset << 2) + (decrypt << 5) + (encrypt << 6) \
1103     + (audit << 7))
1104 #endif // USE_BIT_FIELD_STRUCTURES
1105
1106 // Table "Definition of TPMA_LOCALITY Bits" (Part 2: Structures)
1107 #define TYPE_OF_TPMA_LOCALITY      UINT8
1108 #define TPMA_LOCALITY_TO_UINT8(a) (*(UINT8*)&(a))
1109 #define UINT8_TO_TPMA_LOCALITY(a) (*(TPMA_LOCALITY*)&(a))
1110 #define TPMA_LOCALITY_TO_BYTE_ARRAY(i, a) \
1111     UINT8_TO_BYTE_ARRAY(TPMA_LOCALITY_TO_UINT8(i), (a))
1112 #define BYTE_ARRAY_TO_TPMA_LOCALITY(i, a) \
1113 { \
1114     UINT8 x = BYTE_ARRAY_TO_UINT8(a); \
1115     i      = UINT8_TO_TPMA_LOCALITY(x); \
1116 }

```

```

1117 #if USE_BIT_FIELD_STRUCTURES
1118 typedef struct
1119 {
1120     unsigned TPM_LOC_ZERO : 1;
1121     unsigned TPM_LOC_ONE : 1;
1122     unsigned TPM_LOC_TWO : 1;
1123     unsigned TPM_LOC_THREE : 1;
1124     unsigned TPM_LOC_FOUR : 1;
1125     unsigned Extended : 3;
1126 } TPMA_LOCALITY;
1127
1128 // Initializer for the bit-field structure
1129 # define TPMA_LOCALITY_INITIALIZER( \
1130     tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three, tpm_loc_four, extended) \
1131     { \
1132         tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three, tpm_loc_four, \
1133         extended \
1134     }
1135 #else // USE_BIT_FIELD_STRUCTURES
1136
1137 // This implements Table "Definition of TPMA_LOCALITY Bits" (Part 2: Structures) using
1138 // bit masking
1139 typedef UINT8 TPMA_LOCALITY;
1140 # define TPMA_LOCALITY_TPM_LOC_ZERO (TPMA_LOCALITY) (1 << 0)
1141 # define TPMA_LOCALITY_TPM_LOC_ONE (TPMA_LOCALITY) (1 << 1)
1142 # define TPMA_LOCALITY_TPM_LOC_TWO (TPMA_LOCALITY) (1 << 2)
1143 # define TPMA_LOCALITY_TPM_LOC_THREE (TPMA_LOCALITY) (1 << 3)
1144 # define TPMA_LOCALITY_TPM_LOC_FOUR (TPMA_LOCALITY) (1 << 4)
1145 # define TPMA_LOCALITY_Extended (TPMA_LOCALITY) (7 << 5)
1146 # define TPMA_LOCALITY_Extended_SHIFT 5
1147
1148 // This is the initializer for a TPMA_LOCALITY bit array.
1149 # define TPMA_LOCALITY_INITIALIZER( \
1150     tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three, tpm_loc_four, extended) \
1151     (TPMA_LOCALITY) ((tpm_loc_zero << 0) + (tpm_loc_one << 1) + (tpm_loc_two << 2) \
1152     + (tpm_loc_three << 3) + (tpm_loc_four << 4) \
1153     + (extended << 5))
1154 #endif // USE_BIT_FIELD_STRUCTURES
1155
1156 // Table "Definition of TPMA_PERMANENT Bits" (Part 2: Structures)
1157 #define TYPE_OF_TPMA_PERMANENT UINT32
1158 #define TPMA_PERMANENT_TO_UINT32(a) (*(UINT32*)&(a))
1159 #define UINT32_TO_TPMA_PERMANENT(a) (*(TPMA_PERMANENT*)&(a))
1160 #define TPMA_PERMANENT_TO_BYTE_ARRAY(i, a) \
1161     UINT32_TO_BYTE_ARRAY(TPMA_PERMANENT_TO_UINT32(i), (a))
1162 #define BYTE_ARRAY_TO_TPMA_PERMANENT(i, a) \
1163     { \
1164         UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
1165         i = UINT32_TO_TPMA_PERMANENT(x); \
1166     }
1167 #if USE_BIT_FIELD_STRUCTURES
1168 typedef struct
1169 {
1170     unsigned ownerAuthSet : 1;
1171     unsigned endorsementAuthSet : 1;
1172     unsigned lockoutAuthSet : 1;
1173     unsigned Reserved_bits_at_3 : 5;
1174     unsigned disableClear : 1;
1175     unsigned inLockout : 1;
1176     unsigned tpmGeneratedEPS : 1;
1177     unsigned Reserved_bits_at_11 : 21;
1178 } TPMA_PERMANENT;
1179
1180 // Initializer for the bit-field structure
1181 # define TPMA_PERMANENT_INITIALIZER(ownerauthset, \

```

```

1182         endorsementauthset, \
1183         lockoutauthset, \
1184         bits_at_3, \
1185         disableclear, \
1186         inlockout, \
1187         tpmgeneratedeps, \
1188         bits_at_11) \
1189     { \
1190         ownerauthset, endorsementauthset, lockoutauthset, bits_at_3, disableclear, \
1191         inlockout, tpmgeneratedeps, bits_at_11 \
1192     } \
1193 #else // USE_BIT_FIELD_STRUCTURES
1194
1195 // This implements Table "Definition of TPMA_PERMANENT Bits" (Part 2: Structures)
1196 using bit masking
1197 typedef UINT32 TPMA_PERMANENT;
1198 # define TPMA_PERMANENT_ownerAuthSet (TPMA_PERMANENT) (1 << 0)
1199 # define TPMA_PERMANENT_endorsementAuthSet (TPMA_PERMANENT) (1 << 1)
1200 # define TPMA_PERMANENT_lockoutAuthSet (TPMA_PERMANENT) (1 << 2)
1201 # define TPMA_PERMANENT_disableClear (TPMA_PERMANENT) (1 << 8)
1202 # define TPMA_PERMANENT_inLockout (TPMA_PERMANENT) (1 << 9)
1203 # define TPMA_PERMANENT_tpmGeneratedEPS (TPMA_PERMANENT) (1 << 10)
1204
1205 // This is the initializer for a TPMA_PERMANENT bit array.
1206 # define TPMA_PERMANENT_INITIALIZER(ownerauthset, \
1207                                     endorsementauthset, \
1208                                     lockoutauthset, \
1209                                     bits_at_3, \
1210                                     disableclear, \
1211                                     inlockout, \
1212                                     tpmgeneratedeps, \
1213                                     bits_at_11) \
1214     (TPMA_PERMANENT) ((ownerauthset << 0) + (endorsementauthset << 1) \
1215                       + (lockoutauthset << 2) + (disableclear << 8) \
1216                       + (inlockout << 9) + (tpmgeneratedeps << 10))
1217 #endif // USE_BIT_FIELD_STRUCTURES
1218
1219 // Table "Definition of TPMA_STARTUP_CLEAR Bits" (Part 2: Structures)
1220 #define TYPE_OF_TPMA_STARTUP_CLEAR UINT32
1221 #define TPMA_STARTUP_CLEAR_TO_UINT32(a) (*(UINT32*)&(a))
1222 #define UINT32_TO_TPMA_STARTUP_CLEAR(a) (*(TPMA_STARTUP_CLEAR*)&(a))
1223 #define TPMA_STARTUP_CLEAR_TO_BYTE_ARRAY(i, a) \
1224     UINT32_TO_BYTE_ARRAY((TPMA_STARTUP_CLEAR_TO_UINT32(i)), (a))
1225 #define BYTE_ARRAY_TO_TPMA_STARTUP_CLEAR(i, a) \
1226     { \
1227         UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
1228         i = TPMA_STARTUP_CLEAR_TO_UINT32(x); \
1229     }
1230 #if USE_BIT_FIELD_STRUCTURES
1231 typedef struct
1232 {
1233     unsigned phEnable : 1;
1234     unsigned shEnable : 1;
1235     unsigned ehEnable : 1;
1236     unsigned phEnableNV : 1;
1237     unsigned Reserved_bits_at_4 : 27;
1238     unsigned orderly : 1;
1239 } TPMA_STARTUP_CLEAR;
1240
1241 // Initializer for the bit-field structure
1242 # define TPMA_STARTUP_CLEAR_INITIALIZER( \
1243     phenable, shenable, ehenable, phenablenv, bits_at_4, orderly) \
1244     { \
1245         phenable, shenable, ehenable, phenablenv, bits_at_4, orderly \
1246     }

```



```

1247 #else // USE_BIT_FIELD_STRUCTURES
1248
1249 // This implements Table "Definition of TPMA_STARTUP_CLEAR Bits" (Part 2: Structures)
1250 using bit masking
1251 typedef UINT32 TPMA_STARTUP_CLEAR;
1252 # define TPMA_STARTUP_CLEAR_phEnable (TPMA_STARTUP_CLEAR) (1 << 0)
1253 # define TPMA_STARTUP_CLEAR_shEnable (TPMA_STARTUP_CLEAR) (1 << 1)
1254 # define TPMA_STARTUP_CLEAR_ehEnable (TPMA_STARTUP_CLEAR) (1 << 2)
1255 # define TPMA_STARTUP_CLEAR_phEnableNV (TPMA_STARTUP_CLEAR) (1 << 3)
1256 # define TPMA_STARTUP_CLEAR_orderly (TPMA_STARTUP_CLEAR) (1 << 31)
1257
1258 // This is the initializer for a TPMA_STARTUP_CLEAR bit array.
1259 # define TPMA_STARTUP_CLEAR_INITIALIZER( \
1260     phenable, shenable, ehenable, phenablenv, bits_at_4, orderly) \
1261     (TPMA_STARTUP_CLEAR) ((phenable << 0) + (shenable << 1) + (ehenable << 2) \
1262     + (phenablenv << 3) + (orderly << 31))
1263 #endif // USE_BIT_FIELD_STRUCTURES
1264
1265 // Table "Definition of TPMA_MEMORY Bits" (Part 2: Structures)
1266 #define TYPE_OF TPMA_MEMORY UINT32
1267 #define TPMA_MEMORY_TO_UINT32(a) (*(UINT32*)&(a))
1268 #define UINT32_TO_TPMA_MEMORY(a) (*(TPMA_MEMORY*)&(a))
1269 #define TPMA_MEMORY_TO_BYTE_ARRAY(i, a) \
1270     UINT32_TO_BYTE_ARRAY(TPMA_MEMORY_TO_UINT32(i), (a))
1271 #define BYTE_ARRAY_TO_TPMA_MEMORY(i, a) \
1272     { \
1273         UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
1274         i = UINT32_TO_TPMA_MEMORY(x); \
1275     }
1276 #if USE_BIT_FIELD_STRUCTURES
1277 typedef struct
1278 {
1279     unsigned sharedRAM : 1;
1280     unsigned sharedNV : 1;
1281     unsigned objectCopiedToRam : 1;
1282     unsigned Reserved_bits_at_3 : 29;
1283 } TPMA_MEMORY;
1284
1285 // Initializer for the bit-field structure
1286 # define TPMA_MEMORY_INITIALIZER(sharedram, sharednv, objectcopiedtoram, bits_at_3) \
1287     { \
1288         sharedram, sharednv, objectcopiedtoram, bits_at_3 \
1289     }
1290 #else // USE_BIT_FIELD_STRUCTURES
1291
1292 // This implements Table "Definition of TPMA_MEMORY Bits" (Part 2: Structures) using
1293 bit masking
1294 typedef UINT32 TPMA_MEMORY;
1295 # define TPMA_MEMORY_sharedRAM (TPMA_MEMORY) (1 << 0)
1296 # define TPMA_MEMORY_sharedNV (TPMA_MEMORY) (1 << 1)
1297 # define TPMA_MEMORY_objectCopiedToRam (TPMA_MEMORY) (1 << 2)
1298
1299 // This is the initializer for a TPMA_MEMORY bit array.
1300 # define TPMA_MEMORY_INITIALIZER(sharedram, sharednv, objectcopiedtoram, bits_at_3) \
1301     (TPMA_MEMORY) ((sharedram << 0) + (sharednv << 1) + (objectcopiedtoram << 2))
1302 #endif // USE_BIT_FIELD_STRUCTURES
1303
1304 // Table "Definition of TPMA_CC Bits" (Part 2: Structures)
1305 #define TYPE_OF TPMA_CC UINT32
1306 #define TPMA_CC_TO_UINT32(a) (*(UINT32*)&(a))
1307 #define UINT32_TO_TPMA_CC(a) (*(TPMA_CC*)&(a))
1308 #define TPMA_CC_TO_BYTE_ARRAY(i, a) UINT32_TO_BYTE_ARRAY(TPMA_CC_TO_UINT32(i), (a))
1309 #define BYTE_ARRAY_TO_TPMA_CC(i, a) \
1310     { \

```



```

1311     UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
1312     i       = UINT32_TO_TPMA_CC(x); \
1313 }
1314 #if USE_BIT_FIELD_STRUCTURES
1315 typedef struct
1316 {
1317     unsigned commandIndex      : 16;
1318     unsigned Reserved_bits_at_16 : 6;
1319     unsigned nv                : 1;
1320     unsigned extensive         : 1;
1321     unsigned flushed           : 1;
1322     unsigned cHandles          : 3;
1323     unsigned rHandle           : 1;
1324     unsigned V                 : 1;
1325     unsigned Reserved_bits_at_30 : 2;
1326 } TPMA_CC;
1327
1328 // Initializer for the bit-field structure
1329 # define TPMA_CC_INITIALIZER(commandindex, \
1330                             bits_at_16, \
1331                             nv, \
1332                             extensive, \
1333                             flushed, \
1334                             chandles, \
1335                             rhandle, \
1336                             v, \
1337                             bits_at_30) \
1338 { \
1339     commandindex, bits_at_16, nv, extensive, flushed, chandles, rhandle, v, \
1340     bits_at_30 \
1341 }
1342 #else // USE_BIT_FIELD_STRUCTURES
1343
1344 // This implements Table "Definition of TPMA_CC Bits" (Part 2: Structures) using bit
1345 // masking
1346 typedef TPM_CC TPMA_CC;
1347 # define TPMA_CC_commandIndex      (TPMA_CC) (0xFFFF << 0)
1348 # define TPMA_CC_commandIndex_SHIFT 0
1349 # define TPMA_CC_nv                (TPMA_CC) (1 << 22)
1350 # define TPMA_CC_extensive         (TPMA_CC) (1 << 23)
1351 # define TPMA_CC_flushed           (TPMA_CC) (1 << 24)
1352 # define TPMA_CC_cHandles          (TPMA_CC) (7 << 25)
1353 # define TPMA_CC_cHandles_SHIFT    25
1354 # define TPMA_CC_rHandle           (TPMA_CC) (1 << 28)
1355 # define TPMA_CC_V                 (TPMA_CC) (1 << 29)
1356
1357 // This is the initializer for a TPMA_CC bit array.
1358 # define TPMA_CC_INITIALIZER(commandindex, \
1359                             bits_at_16, \
1360                             nv, \
1361                             extensive, \
1362                             flushed, \
1363                             chandles, \
1364                             rhandle, \
1365                             v, \
1366                             bits_at_30) \
1367     (TPMA_CC) ((commandindex << 0) + (nv << 22) + (extensive << 23) \
1368               + (flushed << 24) + (chandles << 25) + (rhandle << 28) + (v << 29))
1369 #endif // USE_BIT_FIELD_STRUCTURES
1370
1371 // Table "Definition of TPMA_MODES Bits" (Part 2: Structures)
1372 #define TYPE_OF_TPMA_MODES      UINT32
1373 #define TPMA_MODES_TO_UINT32(a) (*(UINT32*)&(a))
1374 #define UINT32_TO_TPMA_MODES(a) (*(TPMA_MODES*)&(a))
1375 #define TPMA_MODES_TO_BYTE_ARRAY(i, a) \

```

```

1376     UINT32_TO_BYTE_ARRAY((TPMA_MODES_TO_UINT32(i)), (a))
1377 #define BYTE_ARRAY_TO_TPMA_MODES(i, a) \
1378 { \
1379     UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
1380     i = UINT32_TO_TPMA_MODES(x); \
1381 }
1382 #if USE_BIT_FIELD_STRUCTURES
1383 typedef struct
1384 {
1385     unsigned FIPS_140_2 : 1;
1386     unsigned FIPS_140_3 : 1;
1387     unsigned FIPS_140_3_INDICATOR : 2;
1388     unsigned Reserved_bits_at_4 : 28;
1389 } TPMA_MODES;
1390
1391 // Initializer for the bit-field structure
1392 # define TPMA_MODES_INITIALIZER( \
1393     fips_140_2, fips_140_3, fips_140_3_indicator, bits_at_4) \
1394 { \
1395     fips_140_2, fips_140_3, fips_140_3_indicator, bits_at_4 \
1396 }
1397 #else // USE_BIT_FIELD_STRUCTURES
1398
1399 // This implements Table "Definition of TPMA_MODES Bits" (Part 2: Structures) using
1400 bit masking
1401 typedef UINT32 TPMA_MODES;
1402 # define TPMA_MODES_FIPS_140_2 (TPMA_MODES) (1 << 0)
1403 # define TPMA_MODES_FIPS_140_3 (TPMA_MODES) (1 << 1)
1404 # define TPMA_MODES_FIPS_140_3_INDICATOR (TPMA_MODES) (3 << 2)
1405 # define TPMA_MODES_FIPS_140_3_INDICATOR_SHIFT 2
1406
1407 // This is the initializer for a TPMA_MODES bit array.
1408 # define TPMA_MODES_INITIALIZER( \
1409     fips_140_2, fips_140_3, fips_140_3_indicator, bits_at_4) \
1410 (TPMA_MODES) ( \
1411     (fips_140_2 << 0) + (fips_140_3 << 1) + (fips_140_3_indicator << 2))
1412 #endif // USE_BIT_FIELD_STRUCTURES
1413
1414 // Table "Definition of TPMA_X509_KEY_USAGE Bits" (Part 2: Structures)
1415 #define TYPE_OF_TPMA_X509_KEY_USAGE UINT32
1416 #define TPMA_X509_KEY_USAGE_TO_UINT32(a) (*(UINT32*)&(a))
1417 #define UINT32_TO_TPMA_X509_KEY_USAGE(a) (*(TPMA_X509_KEY_USAGE*)&(a))
1418 #define TPMA_X509_KEY_USAGE_TO_BYTE_ARRAY(i, a) \
1419     UINT32_TO_BYTE_ARRAY((TPMA_X509_KEY_USAGE_TO_UINT32(i)), (a))
1420 #define BYTE_ARRAY_TO_TPMA_X509_KEY_USAGE(i, a) \
1421 { \
1422     UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
1423     i = UINT32_TO_TPMA_X509_KEY_USAGE(x); \
1424 }
1425 #define TPMA_X509_KEY_USAGE_ALLOWED_BITS (0xff800000)
1426 #if USE_BIT_FIELD_STRUCTURES
1427 typedef struct
1428 {
1429     unsigned Reserved_bits_at_0 : 23;
1430     unsigned decipherOnly : 1;
1431     unsigned encipherOnly : 1;
1432     unsigned cRLSign : 1;
1433     unsigned keyCertSign : 1;
1434     unsigned keyAgreement : 1;
1435     unsigned dataEncipherment : 1;
1436     unsigned keyEncipherment : 1;
1437     unsigned nonrepudiation : 1;
1438     unsigned digitalSignature : 1;
1439 } TPMA_X509_KEY_USAGE;
1440

```

```

1441 // Initializer for the bit-field structure
1442 # define TPMA_X509_KEY_USAGE_INITIALIZER(bits_at_0, \
1443                                         decipheronly, \
1444                                         encipheronly, \
1445                                         crlsign, \
1446                                         keycertsign, \
1447                                         keyagreement, \
1448                                         dataencipherment, \
1449                                         keyencipherment, \
1450                                         nonrepudiation, \
1451                                         digitalsignature) \
1452 { \
1453     bits_at_0, decipheronly, encipheronly, crlsign, keycertsign, keyagreement, \
1454     dataencipherment, keyencipherment, nonrepudiation, digitalsignature \
1455 }
1456 #else // USE_BIT_FIELD_STRUCTURES
1457
1458 // This implements Table "Definition of TPMA_X509_KEY_USAGE Bits" (Part 2: Structures)
1459 using bit masking
1460 typedef UINT32 TPMA_X509_KEY_USAGE;
1461 # define TPMA_X509_KEY_USAGE_decipherOnly (TPMA_X509_KEY_USAGE) (1 << 23)
1462 # define TPMA_X509_KEY_USAGE_encipherOnly (TPMA_X509_KEY_USAGE) (1 << 24)
1463 # define TPMA_X509_KEY_USAGE_cRLSign (TPMA_X509_KEY_USAGE) (1 << 25)
1464 # define TPMA_X509_KEY_USAGE_keyCertSign (TPMA_X509_KEY_USAGE) (1 << 26)
1465 # define TPMA_X509_KEY_USAGE_keyAgreement (TPMA_X509_KEY_USAGE) (1 << 27)
1466 # define TPMA_X509_KEY_USAGE_dataEncipherment (TPMA_X509_KEY_USAGE) (1 << 28)
1467 # define TPMA_X509_KEY_USAGE_keyEncipherment (TPMA_X509_KEY_USAGE) (1 << 29)
1468 # define TPMA_X509_KEY_USAGE_nonrepudiation (TPMA_X509_KEY_USAGE) (1 << 30)
1469 # define TPMA_X509_KEY_USAGE_digitalSignature (TPMA_X509_KEY_USAGE) (1 << 31)
1470
1471 // This is the initializer for a TPMA_X509_KEY_USAGE bit array.
1472 # define TPMA_X509_KEY_USAGE_INITIALIZER(bits_at_0, \
1473                                         decipheronly, \
1474                                         encipheronly, \
1475                                         crlsign, \
1476                                         keycertsign, \
1477                                         keyagreement, \
1478                                         dataencipherment, \
1479                                         keyencipherment, \
1480                                         nonrepudiation, \
1481                                         digitalsignature) \
1482 (TPMA_X509_KEY_USAGE) ((decipheronly << 23) + (encipheronly << 24) \
1483                        + (crlsign << 25) + (keycertsign << 26) \
1484                        + (keyagreement << 27) + (dataencipherment << 28) \
1485                        + (keyencipherment << 29) + (nonrepudiation << 30) \
1486                        + (digitalsignature << 31))
1487 #endif // USE_BIT_FIELD_STRUCTURES
1488
1489 // Table "Definition of TPMA_ACT Bits" (Part 2: Structures)
1490 #define TYPE_OF_TPMA_ACT UINT32
1491 #define TPMA_ACT_TO_UINT32(a) (*(UINT32*)&(a))
1492 #define UINT32_TO_TPMA_ACT(a) (*(TPMA_ACT*)&(a))
1493 #define TPMA_ACT_TO_BYTE_ARRAY(i, a) \
1494     UINT32_TO_BYTE_ARRAY((TPMA_ACT_TO_UINT32(i)), (a))
1495 #define BYTE_ARRAY_TO_TPMA_ACT(i, a) \
1496 { \
1497     UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
1498     i = UINT32_TO_TPMA_ACT(x); \
1499 }
1500 #if USE_BIT_FIELD_STRUCTURES
1501 typedef struct
1502 {
1503     unsigned signaled : 1;
1504     unsigned preserveSignaled : 1;
1505     unsigned Reserved_bits_at_2 : 30;

```

```

1506 } TPMA_ACT;
1507
1508 // Initializer for the bit-field structure
1509 # define TPMA_ACT_INITIALIZER(signaled, preservesignaled, bits_at_2) \
1510 { \
1511     signaled, preservesignaled, bits_at_2 \
1512 } \
1513 #else // USE_BIT_FIELD_STRUCTURES
1514
1515 // This implements Table "Definition of TPMA_ACT Bits" (Part 2: Structures) using bit
1516 // masking
1517 typedef UINT32 TPMA_ACT;
1518 # define TPMA_ACT_signaled (TPMA_ACT) (1 << 0)
1519 # define TPMA_ACT_preserveSignaled (TPMA_ACT) (1 << 1)
1520
1521 // This is the initializer for a TPMA_ACT bit array.
1522 # define TPMA_ACT_INITIALIZER(signaled, preservesignaled, bits_at_2) \
1523     (TPMA_ACT)((signaled << 0) + (preservesignaled << 1))
1524 #endif // USE_BIT_FIELD_STRUCTURES
1525
1526 typedef BYTE TPMI_YES_NO; // (Part 2: Structures)
1527 typedef TPM_HANDLE TPMI_DH_OBJECT; // (Part 2: Structures)
1528 typedef TPM_HANDLE TPMI_DH_PARENT; // (Part 2: Structures)
1529 typedef TPM_HANDLE TPMI_DH_PERSISTENT; // (Part 2: Structures)
1530 typedef TPM_HANDLE TPMI_DH_ENTITY; // (Part 2: Structures)
1531 typedef TPM_HANDLE TPMI_DH_PCR; // (Part 2: Structures)
1532 typedef TPM_HANDLE TPMI_SH_AUTH_SESSION; // (Part 2: Structures)
1533 typedef TPM_HANDLE TPMI_SH_HMAC; // (Part 2: Structures)
1534 typedef TPM_HANDLE TPMI_SH_POLICY; // (Part 2: Structures)
1535 typedef TPM_HANDLE TPMI_DH_CONTEXT; // (Part 2: Structures)
1536 typedef TPM_HANDLE TPMI_DH_SAVED; // (Part 2: Structures)
1537 typedef TPM_HANDLE TPMI_RH_HIERARCHY; // (Part 2: Structures)
1538 typedef TPM_HANDLE TPMI_RH_ENABLES; // (Part 2: Structures)
1539 typedef TPM_HANDLE TPMI_RH_HIERARCHY_AUTH; // (Part 2: Structures)
1540 typedef TPM_HANDLE TPMI_RH_HIERARCHY_POLICY; // (Part 2: Structures)
1541 typedef TPM_HANDLE TPMI_RH_BASE_HIERARCHY; // (Part 2: Structures)
1542 typedef TPM_HANDLE TPMI_RH_PLATFORM; // (Part 2: Structures)
1543 typedef TPM_HANDLE TPMI_RH_OWNER; // (Part 2: Structures)
1544 typedef TPM_HANDLE TPMI_RH_ENDORSEMENT; // (Part 2: Structures)
1545 typedef TPM_HANDLE TPMI_RH_PROVISION; // (Part 2: Structures)
1546 typedef TPM_HANDLE TPMI_RH_CLEAR; // (Part 2: Structures)
1547 typedef TPM_HANDLE TPMI_RH_NV_AUTH; // (Part 2: Structures)
1548 typedef TPM_HANDLE TPMI_RH_LOCKOUT; // (Part 2: Structures)
1549 typedef TPM_HANDLE TPMI_RH_NV_INDEX; // (Part 2: Structures)
1550 typedef TPM_HANDLE TPMI_RH_NV_DEFINED_INDEX; // (Part 2: Structures)
1551 typedef TPM_HANDLE TPMI_RH_NV_LEGACY_INDEX; // (Part 2: Structures)
1552 typedef TPM_HANDLE TPMI_RH_NV_EXP_INDEX; // (Part 2: Structures)
1553 typedef TPM_HANDLE TPMI_RH_AC; // (Part 2: Structures)
1554 typedef TPM_HANDLE TPMI_RH_ACT; // (Part 2: Structures)
1555 typedef TPM_ALG_ID TPMI_ALG_HASH; // (Part 2: Structures)
1556 typedef TPM_ALG_ID TPMI_ALG_ASYM; // (Part 2: Structures)
1557 typedef TPM_ALG_ID TPMI_ALG_SYM; // (Part 2: Structures)
1558 typedef TPM_ALG_ID TPMI_ALG_SYM_OBJECT; // (Part 2: Structures)
1559 typedef TPM_ALG_ID TPMI_ALG_SYM_MODE; // (Part 2: Structures)
1560 typedef TPM_ALG_ID TPMI_ALG_KDF; // (Part 2: Structures)
1561 typedef TPM_ALG_ID TPMI_ALG_SIG_SCHEME; // (Part 2: Structures)
1562 typedef TPM_ALG_ID TPMI_ECC_KEY_EXCHANGE; // (Part 2: Structures)
1563 typedef TPM_ST TPMI_ST_COMMAND_TAG; // (Part 2: Structures)
1564 typedef TPM_ALG_ID TPMI_ALG_MAC_SCHEME; // (Part 2: Structures)
1565 typedef TPM_ALG_ID TPMI_ALG_CIPHER_MODE; // (Part 2: Structures)
1566 typedef BYTE TPMS_EMPTY; // (Part 2: Structures)
1567
1568 typedef struct
1569 { // (Part 2: Structures)
1570     TPM_ALG_ID alg;

```

```

1571     TPMA_ALGORITHM attributes;
1572 } TPMS_ALGORITHM_DESCRIPTION;
1573
1574 typedef union
1575 { // (Part 2: Structures)
1576 #if ALG_SHA1
1577     BYTE sha1[SHA1_DIGEST_SIZE];
1578 #endif // ALG_SHA1
1579 #if ALG_SHA256
1580     BYTE sha256[SHA256_DIGEST_SIZE];
1581 #endif // ALG_SHA256
1582 #if ALG_SHA256_192
1583     BYTE sha256_192[SHA256_192_DIGEST_SIZE];
1584 #endif // ALG_SHA256_192
1585 #if ALG_SHA3_256
1586     BYTE sha3_256[SHA3_256_DIGEST_SIZE];
1587 #endif // ALG_SHA3_256
1588 #if ALG_SHA3_384
1589     BYTE sha3_384[SHA3_384_DIGEST_SIZE];
1590 #endif // ALG_SHA3_384
1591 #if ALG_SHA3_512
1592     BYTE sha3_512[SHA3_512_DIGEST_SIZE];
1593 #endif // ALG_SHA3_512
1594 #if ALG_SHA384
1595     BYTE sha384[SHA384_DIGEST_SIZE];
1596 #endif // ALG_SHA384
1597 #if ALG_SHA512
1598     BYTE sha512[SHA512_DIGEST_SIZE];
1599 #endif // ALG_SHA512
1600 #if ALG_SHAKE256_192
1601     BYTE shake256_192[SHAKE256_192_DIGEST_SIZE];
1602 #endif // ALG_SHAKE256_192
1603 #if ALG_SHAKE256_256
1604     BYTE shake256_256[SHAKE256_256_DIGEST_SIZE];
1605 #endif // ALG_SHAKE256_256
1606 #if ALG_SHAKE256_512
1607     BYTE shake256_512[SHAKE256_512_DIGEST_SIZE];
1608 #endif // ALG_SHAKE256_512
1609 #if ALG_SM3_256
1610     BYTE sm3_256[SM3_256_DIGEST_SIZE];
1611 #endif // ALG_SM3_256
1612 } TPMU_HA;
1613
1614 typedef struct
1615 { // (Part 2: Structures)
1616     TPMI_ALG_HASH hashAlg;
1617     TPMU_HA digest;
1618 } TPMT_HA;
1619
1620 typedef union
1621 { // (Part 2: Structures)
1622     struct
1623     {
1624         UINT16 size;
1625         BYTE buffer[sizeof(TPMU_HA)];
1626     } t;
1627     TPM2B b;
1628 } TPM2B_DIGEST;
1629
1630 typedef union
1631 { // (Part 2: Structures)
1632     struct
1633     {
1634         UINT16 size;
1635         BYTE buffer[sizeof(TPMT_HA)];
1636     } t;

```



```
1637     TPM2B b;
1638 } TPM2B_DATA;
1639
1640 // Table "Definition of Types for TPM2B_NONCE" (Part 2: Structures)
1641 typedef TPM2B_DIGEST TPM2B_NONCE;
1642 #define TYPE_OF_TPM2B_NONCE TPM2B_DIGEST
1643
1644 // Table "Definition of Types for TPM2B_AUTH" (Part 2: Structures)
1645 typedef TPM2B_DIGEST TPM2B_AUTH;
1646 #define TYPE_OF_TPM2B_AUTH TPM2B_DIGEST
1647
1648 // Table "Definition of Types for TPM2B_OPERAND" (Part 2: Structures)
1649 typedef TPM2B_DIGEST TPM2B_OPERAND;
1650 #define TYPE_OF_TPM2B_OPERAND TPM2B_DIGEST
1651
1652 typedef union
1653 { // (Part 2: Structures)
1654     struct
1655     {
1656         UINT16 size;
1657         BYTE    buffer[1024];
1658     } t;
1659     TPM2B b;
1660 } TPM2B_EVENT;
1661
1662 typedef union
1663 { // (Part 2: Structures)
1664     struct
1665     {
1666         UINT16 size;
1667         BYTE    buffer[MAX_DIGEST_BUFFER];
1668     } t;
1669     TPM2B b;
1670 } TPM2B_MAX_BUFFER;
1671
1672 typedef union
1673 { // (Part 2: Structures)
1674     struct
1675     {
1676         UINT16 size;
1677         BYTE    buffer[MAX_NV_BUFFER_SIZE];
1678     } t;
1679     TPM2B b;
1680 } TPM2B_MAX_NV_BUFFER;
1681
1682 typedef union
1683 { // (Part 2: Structures)
1684     struct
1685     {
1686         UINT16 size;
1687         BYTE    buffer[sizeof(UINT64)];
1688     } t;
1689     TPM2B b;
1690 } TPM2B_TIMEOUT;
1691
1692 typedef union
1693 { // (Part 2: Structures)
1694     struct
1695     {
1696         UINT16 size;
1697         BYTE    buffer[MAX_SYM_BLOCK_SIZE];
1698     } t;
1699     TPM2B b;
1700 } TPM2B_IV;
1701
1702 typedef union
```



```

1703 { // (Part 2: Structures)
1704     struct
1705     {
1706         UINT16 size;
1707         BYTE    buffer[512];
1708     } t;
1709     TPM2B b;
1710 } TPM2B_VENDOR_PROPERTY;
1711
1712 typedef union
1713 { // (Part 2: Structures)
1714     TPMT_HA    digest;
1715     TPM_HANDLE handle;
1716 } TPMU_NAME;
1717
1718 typedef union
1719 { // (Part 2: Structures)
1720     struct
1721     {
1722         UINT16 size;
1723         BYTE    name[sizeof(TPMU_NAME)];
1724     } t;
1725     TPM2B b;
1726 } TPM2B_NAME;
1727
1728 typedef struct
1729 { // (Part 2: Structures)
1730     UINT8 sizeofSelect;
1731     BYTE  pcrSelect[PCR_SELECT_MAX];
1732 } TPMS_PCR_SELECT;
1733
1734 typedef struct
1735 { // (Part 2: Structures)
1736     TPMI_ALG_HASH hash;
1737     UINT8          sizeofSelect;
1738     BYTE          pcrSelect[PCR_SELECT_MAX];
1739 } TPMS_PCR_SELECTION;
1740
1741 typedef struct
1742 { // (Part 2: Structures)
1743     TPM_ST tag;
1744     TPMI_RH_HIERARCHY hierarchy;
1745     TPM2B_DIGEST digest;
1746 } TPMT_TK_CREATION;
1747
1748 typedef struct
1749 { // (Part 2: Structures)
1750     TPM_ST tag;
1751     TPMI_RH_HIERARCHY hierarchy;
1752     TPM2B_DIGEST digest;
1753 } TPMT_TK_VERIFIED;
1754
1755 typedef struct
1756 { // (Part 2: Structures)
1757     TPM_ST tag;
1758     TPMI_RH_HIERARCHY hierarchy;
1759     TPM2B_DIGEST digest;
1760 } TPMT_TK_AUTH;
1761
1762 typedef struct
1763 { // (Part 2: Structures)
1764     TPM_ST tag;
1765     TPMI_RH_HIERARCHY hierarchy;
1766     TPM2B_DIGEST digest;
1767 } TPMT_TK_HASHCHECK;
1768

```

```

1769 typedef struct
1770 { // (Part 2: Structures)
1771     TPM_ALG_ID    alg;
1772     TPMA_ALGORITHM algProperties;
1773 } TPMS_ALG_PROPERTY;
1774
1775 typedef struct
1776 { // (Part 2: Structures)
1777     TPM_PT property;
1778     UINT32 value;
1779 } TPMS_TAGGED_PROPERTY;
1780
1781 typedef struct
1782 { // (Part 2: Structures)
1783     TPM_PT_PCR tag;
1784     UINT8    sizeofSelect;
1785     BYTE     pcrSelect[PCR_SELECT_MAX];
1786 } TPMS_TAGGED_PCR_SELECT;
1787
1788 typedef struct
1789 { // (Part 2: Structures)
1790     TPM_HANDLE handle;
1791     TPMT_HA    policyHash;
1792 } TPMS_TAGGED_POLICY;
1793
1794 typedef struct
1795 { // (Part 2: Structures)
1796     TPM_HANDLE handle;
1797     UINT32    timeout;
1798     TPMA_ACT  attributes;
1799 } TPMS_ACT_DATA;
1800
1801 typedef struct
1802 { // (Part 2: Structures)
1803     UINT32 count;
1804     TPM_CC commandCodes[MAX_CAP_CC];
1805 } TPML_CC;
1806
1807 typedef struct
1808 { // (Part 2: Structures)
1809     UINT32 count;
1810     TPMA_CC commandAttributes[MAX_CAP_CC];
1811 } TPML_CCA;
1812
1813 typedef struct
1814 { // (Part 2: Structures)
1815     UINT32 count;
1816     TPM_ALG_ID algorithms[MAX_ALG_LIST_SIZE];
1817 } TPML_ALG;
1818
1819 typedef struct
1820 { // (Part 2: Structures)
1821     UINT32 count;
1822     TPM_HANDLE handle[MAX_CAP_HANDLES];
1823 } TPML_HANDLE;
1824
1825 typedef struct
1826 { // (Part 2: Structures)
1827     UINT32 count;
1828     TPM2B_DIGEST digests[8];
1829 } TPML_DIGEST;
1830
1831 typedef struct
1832 { // (Part 2: Structures)
1833     UINT32 count;
1834     TPMT_HA digests[HASH_COUNT];

```

```

1835 } TPML_DIGEST_VALUES;
1836
1837 typedef struct
1838 { // (Part 2: Structures)
1839     UINT32 count;
1840     TPMS_PCR_SELECTION pcrSelections[HASH_COUNT];
1841 } TPML_PCR_SELECTION;
1842
1843 typedef struct
1844 { // (Part 2: Structures)
1845     UINT32 count;
1846     TPMS_ALG_PROPERTY algProperties[MAX_CAP_ALGS];
1847 } TPML_ALG_PROPERTY;
1848
1849 typedef struct
1850 { // (Part 2: Structures)
1851     UINT32 count;
1852     TPMS_TAGGED_PROPERTY tpmProperty[MAX_TPM_PROPERTIES];
1853 } TPML_TAGGED_TPM_PROPERTY;
1854
1855 typedef struct
1856 { // (Part 2: Structures)
1857     UINT32 count;
1858     TPMS_TAGGED_PCR_SELECT pcrProperty[MAX_PCR_PROPERTIES];
1859 } TPML_TAGGED_PCR_PROPERTY;
1860
1861 typedef struct
1862 { // (Part 2: Structures)
1863     UINT32 count;
1864     TPM_ECC_CURVE eccCurves[MAX_ECC_CURVES];
1865 } TPML_ECC_CURVE;
1866
1867 typedef struct
1868 { // (Part 2: Structures)
1869     UINT32 count;
1870     TPMS_TAGGED_POLICY policies[MAX_TAGGED_POLICIES];
1871 } TPML_TAGGED_POLICY;
1872
1873 typedef struct
1874 { // (Part 2: Structures)
1875     UINT32 count;
1876     TPMS_ACT_DATA actData[MAX_ACT_DATA];
1877 } TPML_ACT_DATA;
1878
1879 typedef struct
1880 { // (Part 2: Structures)
1881     UINT32 count;
1882     TPM2B_VENDOR_PROPERTY vendorData[MAX_VENDOR_PROPERTY];
1883 } TPML_VENDOR_PROPERTY;
1884
1885 typedef union
1886 { // (Part 2: Structures)
1887     TPML_ALG_PROPERTY algorithms;
1888     TPML_HANDLE handles;
1889     TPML_CCA command;
1890     TPML_CC ppCommands;
1891     TPML_CC auditCommands;
1892     TPML_PCR_SELECTION assignedPCR;
1893     TPML_TAGGED_TPM_PROPERTY tpmProperties;
1894     TPML_TAGGED_PCR_PROPERTY pcrProperties;
1895 #if ALG_ECC
1896     TPML_ECC_CURVE eccCurves;
1897 #endif // ALG_ECC
1898     TPML_TAGGED_POLICY authPolicies;
1899     TPML_ACT_DATA actData;
1900 } TPMU_CAPABILITIES;

```

```

1901
1902 typedef struct
1903 { // (Part 2: Structures)
1904     TPM_CAP                capability;
1905     TPMU_CAPABILITIES data;
1906 } TPMS_CAPABILITY_DATA;
1907
1908 typedef union
1909 { // (Part 2: Structures)
1910     // NOTE: No settable capabilities are implemented in this reference code.
1911 } TPMU_SET_CAPABILITIES;
1912
1913 typedef struct
1914 { // (Part 2: Structures)
1915     TPM_CAP                setCapability;
1916     TPMU_SET_CAPABILITIES data;
1917 } TPMS_SET_CAPABILITY_DATA;
1918
1919 typedef struct
1920 { // (Part 2: Structures)
1921     UINT16                size;
1922     TPMS_SET_CAPABILITY_DATA setCapabilityData;
1923 } TPM2B_SET_CAPABILITY_DATA;
1924
1925 typedef struct
1926 { // (Part 2: Structures)
1927     UINT64                clock;
1928     UINT32                resetCount;
1929     UINT32                restartCount;
1930     TPMI_YES_NO safe;
1931 } TPMS_CLOCK_INFO;
1932
1933 typedef struct
1934 { // (Part 2: Structures)
1935     UINT64                time;
1936     TPMS_CLOCK_INFO clockInfo;
1937 } TPMS_TIME_INFO;
1938
1939 typedef struct
1940 { // (Part 2: Structures)
1941     TPMS_TIME_INFO time;
1942     UINT64                firmwareVersion;
1943 } TPMS_TIME_ATTEST_INFO;
1944
1945 typedef struct
1946 { // (Part 2: Structures)
1947     TPM2B_NAME name;
1948     TPM2B_NAME qualifiedName;
1949 } TPMS_CERTIFY_INFO;
1950
1951 typedef struct
1952 { // (Part 2: Structures)
1953     TPML_PCR_SELECTION pcrSelect;
1954     TPM2B_DIGEST pcrDigest;
1955 } TPMS_QUOTE_INFO;
1956
1957 typedef struct
1958 { // (Part 2: Structures)
1959     UINT64                auditCounter;
1960     TPM_ALG_ID digestAlg;
1961     TPM2B_DIGEST auditDigest;
1962     TPM2B_DIGEST commandDigest;
1963 } TPMS_COMMAND_AUDIT_INFO;
1964
1965 typedef struct
1966 { // (Part 2: Structures)

```

```

1967     TPMI_YES_NO    exclusiveSession;
1968     TPM2B_DIGEST    sessionDigest;
1969 } TPMS_SESSION_AUDIT_INFO;
1970
1971 typedef struct
1972 { // (Part 2: Structures)
1973     TPM2B_NAME    objectName;
1974     TPM2B_DIGEST    creationHash;
1975 } TPMS_CREATION_INFO;
1976
1977 typedef struct
1978 { // (Part 2: Structures)
1979     TPM2B_NAME    indexName;
1980     UINT16        offset;
1981     TPM2B_MAX_NV_BUFFER nvContents;
1982 } TPMS_NV_CERTIFY_INFO;
1983
1984 typedef struct
1985 { // (Part 2: Structures)
1986     TPM2B_NAME    indexName;
1987     TPM2B_DIGEST    nvDigest;
1988 } TPMS_NV_DIGEST_CERTIFY_INFO;
1989
1990 typedef TPM_ST TPMI_ST_ATTEST; // (Part 2: Structures)
1991 typedef union
1992 { // (Part 2: Structures)
1993     TPMS_CERTIFY_INFO    certify;
1994     TPMS_CREATION_INFO    creation;
1995     TPMS_QUOTE_INFO    quote;
1996     TPMS_COMMAND_AUDIT_INFO    commandAudit;
1997     TPMS_SESSION_AUDIT_INFO    sessionAudit;
1998     TPMS_TIME_ATTEST_INFO    time;
1999     TPMS_NV_CERTIFY_INFO    nv;
2000     TPMS_NV_DIGEST_CERTIFY_INFO nvDigest;
2001 } TPMU_ATTEST;
2002
2003 typedef struct
2004 { // (Part 2: Structures)
2005     TPM_CONST32 magic;
2006     TPMI_ST_ATTEST type;
2007     TPM2B_NAME    qualifiedSigner;
2008     TPM2B_DATA    extraData;
2009     TPMS_CLOCK_INFO clockInfo;
2010     UINT64        firmwareVersion;
2011     TPMU_ATTEST    attested;
2012 } TPMS_ATTEST;
2013
2014 typedef union
2015 { // (Part 2: Structures)
2016     struct
2017     {
2018         UINT16 size;
2019         BYTE    attestationData[sizeof(TPMS_ATTEST)];
2020     } t;
2021     TPM2B b;
2022 } TPM2B_ATTEST;
2023
2024 typedef struct
2025 { // (Part 2: Structures)
2026     TPMI_SH_AUTH_SESSION sessionHandle;
2027     TPM2B_NONCE    nonce;
2028     TPMA_SESSION    sessionAttributes;
2029     TPM2B_AUTH    hmac;
2030 } TPMS_AUTH_COMMAND;
2031
2032 typedef struct

```

```

2033 { // (Part 2: Structures)
2034     TPM2B_NONCE nonce;
2035     TPMA_SESSION sessionAttributes;
2036     TPM2B_AUTH hmac;
2037 } TPMS_AUTH_RESPONSE;
2038
2039 typedef TPM_KEY_BITS TPMI_AES_KEY_BITS; // (Part 2: Structures)
2040 typedef TPM_KEY_BITS TPMI_SM4_KEY_BITS; // (Part 2: Structures)
2041 typedef TPM_KEY_BITS TPMI_CAMELLIA_KEY_BITS; // (Part 2: Structures)
2042 typedef union
2043 { // (Part 2: Structures)
2044     #if ALG_AES
2045         TPMI_AES_KEY_BITS aes;
2046     #endif // ALG_AES
2047     #if ALG_SM4
2048         TPMI_SM4_KEY_BITS sm4;
2049     #endif // ALG_SM4
2050     #if ALG_CAMELLIA
2051         TPMI_CAMELLIA_KEY_BITS camellia;
2052     #endif // ALG_CAMELLIA
2053     TPM_KEY_BITS sym;
2054     #if ALG_XOR
2055         TPMI_ALG_HASH xor ;
2056     #endif // ALG_XOR
2057 } TPMU_SYM_KEY_BITS;
2058
2059 typedef union
2060 { // (Part 2: Structures)
2061     #if ALG_AES
2062         TPMI_ALG_SYM_MODE aes;
2063     #endif // ALG_AES
2064     #if ALG_SM4
2065         TPMI_ALG_SYM_MODE sm4;
2066     #endif // ALG_SM4
2067     #if ALG_CAMELLIA
2068         TPMI_ALG_SYM_MODE camellia;
2069     #endif // ALG_CAMELLIA
2070     TPMI_ALG_SYM_MODE sym;
2071 } TPMU_SYM_MODE;
2072
2073 typedef struct
2074 { // (Part 2: Structures)
2075     TPMI_ALG_SYM algorithm;
2076     TPMU_SYM_KEY_BITS keyBits;
2077     TPMU_SYM_MODE mode;
2078 } TPMT_SYM_DEF;
2079
2080 typedef struct
2081 { // (Part 2: Structures)
2082     TPMI_ALG_SYM_OBJECT algorithm;
2083     TPMU_SYM_KEY_BITS keyBits;
2084     TPMU_SYM_MODE mode;
2085 } TPMT_SYM_DEF_OBJECT;
2086
2087 typedef union
2088 { // (Part 2: Structures)
2089     struct
2090     {
2091         UINT16 size;
2092         BYTE buffer[MAX_SYM_KEY_BYTES];
2093     } t;
2094     TPM2B b;
2095 } TPM2B_SYM_KEY;
2096
2097 typedef struct
2098 { // (Part 2: Structures)

```



```

2099     TPMT_SYM_DEF_OBJECT sym;
2100 } TPMS_SYMCIPHER_PARMS;
2101
2102 typedef union
2103 { // (Part 2: Structures)
2104     struct
2105     {
2106         UINT16 size;
2107         BYTE    buffer[LABEL_MAX_BUFFER];
2108     } t;
2109     TPM2B b;
2110 } TPM2B_LABEL;
2111
2112 typedef struct
2113 { // (Part 2: Structures)
2114     TPM2B_LABEL label;
2115     TPM2B_LABEL context;
2116 } TPMS_DERIVE;
2117
2118 typedef union
2119 { // (Part 2: Structures)
2120     struct
2121     {
2122         UINT16 size;
2123         BYTE    buffer[sizeof(TPMS_DERIVE)];
2124     } t;
2125     TPM2B b;
2126 } TPM2B_DERIVE;
2127
2128 typedef union
2129 { // (Part 2: Structures)
2130     BYTE    create[MAX_SYM_DATA];
2131     TPMS_DERIVE derive;
2132 } TPMU_SENSITIVE_CREATE;
2133
2134 typedef union
2135 { // (Part 2: Structures)
2136     struct
2137     {
2138         UINT16 size;
2139         BYTE    buffer[sizeof(TPMU_SENSITIVE_CREATE)];
2140     } t;
2141     TPM2B b;
2142 } TPM2B_SENSITIVE_DATA;
2143
2144 typedef struct
2145 { // (Part 2: Structures)
2146     TPM2B_AUTH    userAuth;
2147     TPM2B_SENSITIVE_DATA data;
2148 } TPMS_SENSITIVE_CREATE;
2149
2150 typedef struct
2151 { // (Part 2: Structures)
2152     UINT16 size;
2153     TPMS_SENSITIVE_CREATE sensitive;
2154 } TPM2B_SENSITIVE_CREATE;
2155
2156 typedef struct
2157 { // (Part 2: Structures)
2158     TPMI_ALG_HASH hashAlg;
2159 } TPMS_SCHEME_HASH;
2160
2161 typedef struct
2162 { // (Part 2: Structures)
2163     TPMI_ALG_HASH hashAlg;
2164     UINT16 count;

```

```

2165 } TPMS_SCHEME_ECDA;
2166
2167 typedef TPM_ALG_ID TPMI_ALG_KEYEDHASH_SCHEME; // (Part 2: Structures)
2168
2169 // Table "Definition of Types for HMAC_SIG_SCHEME" (Part 2: Structures)
2170 typedef TPMS_SCHEME_HASH TPMS_SCHEME_HMAC;
2171 #define TYPE_OF_TPMS_SCHEME_HMAC TPMS_SCHEME_HASH
2172
2173 typedef struct
2174 { // (Part 2: Structures)
2175     TPMI_ALG_HASH hashAlg;
2176     TPMI_ALG_KDF kdf;
2177 } TPMS_SCHEME_XOR;
2178
2179 typedef union
2180 { // (Part 2: Structures)
2181     #if ALG_HMAC
2182         TPMS_SCHEME_HMAC hmac;
2183     #endif // ALG_HMAC
2184     #if ALG_XOR
2185         TPMS_SCHEME_XOR xor ;
2186     #endif // ALG_XOR
2187 } TPMU_SCHEME_KEYEDHASH;
2188
2189 typedef struct
2190 { // (Part 2: Structures)
2191     TPMI_ALG_KEYEDHASH_SCHEME scheme;
2192     TPMU_SCHEME_KEYEDHASH details;
2193 } TPMT_KEYEDHASH_SCHEME;
2194
2195 // Table "Definition of Types for RSA Signature Schemes" (Part 2: Structures)
2196 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_RSAPSS;
2197 #define TYPE_OF_TPMS_SIG_SCHEME_RSAPSS TPMS_SCHEME_HASH
2198 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_RSASSA;
2199 #define TYPE_OF_TPMS_SIG_SCHEME_RSASSA TPMS_SCHEME_HASH
2200
2201 // Table "Definition of Types for ECC Signature Schemes" (Part 2: Structures)
2202 typedef TPMS_SCHEME_ECDA TPMS_SIG_SCHEME_ECDA;
2203 #define TYPE_OF_TPMS_SIG_SCHEME_ECDA TPMS_SCHEME_ECDA
2204 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_ECDSA;
2205 #define TYPE_OF_TPMS_SIG_SCHEME_ECDSA TPMS_SCHEME_HASH
2206 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_ECSCHNORR;
2207 #define TYPE_OF_TPMS_SIG_SCHEME_ECSCHNORR TPMS_SCHEME_HASH
2208 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_EDDSA;
2209 #define TYPE_OF_TPMS_SIG_SCHEME_EDDSA TPMS_SCHEME_HASH
2210 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_EDDSA_PH;
2211 #define TYPE_OF_TPMS_SIG_SCHEME_EDDSA_PH TPMS_SCHEME_HASH
2212 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_SM2;
2213 #define TYPE_OF_TPMS_SIG_SCHEME_SM2 TPMS_SCHEME_HASH
2214
2215 typedef union
2216 { // (Part 2: Structures)
2217     #if ALG_HMAC
2218         TPMS_SCHEME_HMAC hmac;
2219     #endif // ALG_HMAC
2220     #if ALG_RSASSA
2221         TPMS_SIG_SCHEME_RSASSA rsassa;
2222     #endif // ALG_RSASSA
2223     #if ALG_RSAPSS
2224         TPMS_SIG_SCHEME_RSAPSS rsapss;
2225     #endif // ALG_RSAPSS
2226     #if ALG_ECDSA
2227         TPMS_SIG_SCHEME_ECDSA ecdsa;
2228     #endif // ALG_ECDSA
2229     #if ALG_ECDA
2230         TPMS_SIG_SCHEME_ECDA ecda;

```

```

2231 #endif // ALG_ECDSA
2232 #if ALG_SM2
2233     TPMS_SIG_SCHEME_SM2 sm2;
2234 #endif // ALG_SM2
2235 #if ALG_ECSCHNORR
2236     TPMS_SIG_SCHEME_ECSCHNORR ecschnorr;
2237 #endif // ALG_ECSCHNORR
2238 #if ALG_EDDSA
2239     TPMS_SIG_SCHEME_EDDSA eddsa;
2240 #endif // ALG_EDDSA
2241 #if ALG_EDDSA_PH
2242     TPMS_SIG_SCHEME_EDDSA_PH eddsa_ph;
2243 #endif // ALG_EDDSA_PH
2244 #if ALG_LMS
2245     TPMS_SIG_SCHEME_LMS lms;
2246 #endif // ALG_LMS
2247 #if ALG_XMSS
2248     TPMS_SIG_SCHEME_XMSS xmss;
2249 #endif // ALG_XMSS
2250     TPMS_SCHEME_HASH any;
2251 } TPMU_SIG_SCHEME;
2252
2253 typedef struct
2254 { // (Part 2: Structures)
2255     TPMI_ALG_SIG_SCHEME scheme;
2256     TPMU_SIG_SCHEME details;
2257 } TPMT_SIG_SCHEME;
2258
2259 // Table "Definition of Types for Encryption Schemes" (Part 2: Structures)
2260 typedef TPMS_EMPTY TPMS_ENC_SCHEME_RSAES;
2261 #define TYPE_OF_TPMS_ENC_SCHEME_RSAES TPMS_EMPTY
2262 typedef TPMS_SCHEME_HASH TPMS_ENC_SCHEME_OAEP;
2263 #define TYPE_OF_TPMS_ENC_SCHEME_OAEP TPMS_SCHEME_HASH
2264
2265 // Table "Definition of Types for ECC Key Exchange" (Part 2: Structures)
2266 typedef TPMS_SCHEME_HASH TPMS_KEY_SCHEME_ECDH;
2267 #define TYPE_OF_TPMS_KEY_SCHEME_ECDH TPMS_SCHEME_HASH
2268 typedef TPMS_SCHEME_HASH TPMS_KEY_SCHEME_ECMQV;
2269 #define TYPE_OF_TPMS_KEY_SCHEME_ECMQV TPMS_SCHEME_HASH
2270 typedef TPMS_SCHEME_HASH TPMS_KEY_SCHEME_SM2;
2271 #define TYPE_OF_TPMS_KEY_SCHEME_SM2 TPMS_SCHEME_HASH
2272
2273 // Table "Definition of Types for KDF Schemes" (Part 2: Structures)
2274 typedef TPMS_SCHEME_HASH TPMS_KDF_SCHEME_KDF1_SP800_108;
2275 #define TYPE_OF_TPMS_KDF_SCHEME_KDF1_SP800_108 TPMS_SCHEME_HASH
2276 typedef TPMS_SCHEME_HASH TPMS_KDF_SCHEME_KDF1_SP800_56A;
2277 #define TYPE_OF_TPMS_KDF_SCHEME_KDF1_SP800_56A TPMS_SCHEME_HASH
2278 typedef TPMS_SCHEME_HASH TPMS_KDF_SCHEME_KDF2;
2279 #define TYPE_OF_TPMS_KDF_SCHEME_KDF2 TPMS_SCHEME_HASH
2280 typedef TPMS_SCHEME_HASH TPMS_KDF_SCHEME_MGF1;
2281 #define TYPE_OF_TPMS_KDF_SCHEME_MGF1 TPMS_SCHEME_HASH
2282
2283 typedef union
2284 { // (Part 2: Structures)
2285     TPMS_SCHEME_HASH anyKdf;
2286 #if ALG_MGF1
2287     TPMS_KDF_SCHEME_MGF1 mgf1;
2288 #endif // ALG_MGF1
2289 #if ALG_KDF1_SP800_56A
2290     TPMS_KDF_SCHEME_KDF1_SP800_56A kdf1_sp800_56a;
2291 #endif // ALG_KDF1_SP800_56A
2292 #if ALG_KDF2
2293     TPMS_KDF_SCHEME_KDF2 kdf2;
2294 #endif // ALG_KDF2
2295 #if ALG_KDF1_SP800_108
2296     TPMS_KDF_SCHEME_KDF1_SP800_108 kdf1_sp800_108;

```

```

2297 #endif // ALG_KDF1_SP800_108
2298 } TPMU_KDF_SCHEME;
2299
2300 typedef struct
2301 { // (Part 2: Structures)
2302     TPMI_ALG_KDF scheme;
2303     TPMU_KDF_SCHEME details;
2304 } TPMT_KDF_SCHEME;
2305
2306 typedef TPM_ALG_ID TPMI_ALG_ASYM_SCHEME; // (Part 2: Structures)
2307 typedef union
2308 { // (Part 2: Structures)
2309     TPMS_SCHEME_HASH anySig;
2310 #if ALG_RSASSA
2311     TPMS_SIG_SCHEME_RSASSA rsassa;
2312 #endif // ALG_RSASSA
2313 #if ALG_RSAES
2314     TPMS_ENC_SCHEME_RSAES rsaes;
2315 #endif // ALG_RSAES
2316 #if ALG_RSAPSS
2317     TPMS_SIG_SCHEME_RSAPSS rsapss;
2318 #endif // ALG_RSAPSS
2319 #if ALG_OAEP
2320     TPMS_ENC_SCHEME_OAEP oaep;
2321 #endif // ALG_OAEP
2322 #if ALG_ECDSA
2323     TPMS_SIG_SCHEME_ECDSA ecdsa;
2324 #endif // ALG_ECDSA
2325 #if ALG_ECDH
2326     TPMS_KEY_SCHEME_ECDH ecdh;
2327 #endif // ALG_ECDH
2328 #if ALG_ECDAA
2329     TPMS_SIG_SCHEME_ECDAA ecdaa;
2330 #endif // ALG_ECDAA
2331 #if ALG_SM2
2332     TPMS_KEY_SCHEME_SM2 sm2;
2333 #endif // ALG_SM2
2334 #if ALG_ECSCHNORR
2335     TPMS_SIG_SCHEME_ECSCHNORR ecschnorr;
2336 #endif // ALG_ECSCHNORR
2337 #if ALG_ECMQV
2338     TPMS_KEY_SCHEME_ECMQV ecmqv;
2339 #endif // ALG_ECMQV
2340 #if ALG_EDDSA
2341     TPMS_SIG_SCHEME_EDDSA eddsa;
2342 #endif // ALG_EDDSA
2343 #if ALG_EDDSA_PH
2344     TPMS_SIG_SCHEME_EDDSA_PH eddsa_ph;
2345 #endif // ALG_EDDSA_PH
2346 #if ALG_LMS
2347     TPMS_SIG_SCHEME_LMS lms;
2348 #endif // ALG_LMS
2349 #if ALG_XMSS
2350     TPMS_SIG_SCHEME_XMSS xmss;
2351 #endif // ALG_XMSS
2352 } TPMU_ASYM_SCHEME;
2353
2354 typedef struct
2355 { // (Part 2: Structures)
2356     TPMI_ALG_ASYM_SCHEME scheme;
2357     TPMU_ASYM_SCHEME details;
2358 } TPMT_ASYM_SCHEME;
2359
2360 typedef TPM_ALG_ID TPMI_ALG_RSA_SCHEME; // (Part 2: Structures)
2361 typedef struct
2362 { // (Part 2: Structures)

```

```

2363     TPMI_ALG_RSA_SCHEME scheme;
2364     TPMU_ASYM_SCHEME details;
2365 } TPMT_RSA_SCHEME;
2366
2367 typedef TPM_ALG_ID TPMI_ALG_RSA_DECRYPT; // (Part 2: Structures)
2368 typedef struct
2369 { // (Part 2: Structures)
2370     TPMI_ALG_RSA_DECRYPT scheme;
2371     TPMU_ASYM_SCHEME details;
2372 } TPMT_RSA_DECRYPT;
2373
2374 typedef union
2375 { // (Part 2: Structures)
2376     struct
2377     {
2378         UINT16 size;
2379         BYTE buffer[MAX_RSA_KEY_BYTES];
2380     } t;
2381     TPM2B b;
2382 } TPM2B_PUBLIC_KEY_RSA;
2383
2384 typedef TPM_KEY_BITS TPMI_RSA_KEY_BITS; // (Part 2: Structures)
2385 typedef union
2386 { // (Part 2: Structures)
2387     struct
2388     {
2389         UINT16 size;
2390         BYTE buffer[RSA_PRIVATE_SIZE];
2391     } t;
2392     TPM2B b;
2393 } TPM2B_PRIVATE_KEY_RSA;
2394
2395 typedef union
2396 { // (Part 2: Structures)
2397     struct
2398     {
2399         UINT16 size;
2400         BYTE buffer[MAX_ECC_KEY_BYTES];
2401     } t;
2402     TPM2B b;
2403 } TPM2B_ECC_PARAMETER;
2404
2405 typedef struct
2406 { // (Part 2: Structures)
2407     TPM2B_ECC_PARAMETER x;
2408     TPM2B_ECC_PARAMETER y;
2409 } TPMS_ECC_POINT;
2410
2411 typedef struct
2412 { // (Part 2: Structures)
2413     UINT16 size;
2414     TPMS_ECC_POINT point;
2415 } TPM2B_ECC_POINT;
2416
2417 typedef TPM_ALG_ID TPMI_ALG_ECC_SCHEME; // (Part 2: Structures)
2418 typedef TPM_ECC_CURVE TPMI_ECC_CURVE; // (Part 2: Structures)
2419 typedef struct
2420 { // (Part 2: Structures)
2421     TPMI_ALG_ECC_SCHEME scheme;
2422     TPMU_ASYM_SCHEME details;
2423 } TPMT_ECC_SCHEME;
2424
2425 typedef struct
2426 { // (Part 2: Structures)
2427     TPM_ECC_CURVE curveID;
2428     UINT16 keySize;

```

```

2429     TPMT_KDF_SCHEME      kdf;
2430     TPMT_ECC_SCHEME      sign;
2431     TPM2B_ECC_PARAMETER  p;
2432     TPM2B_ECC_PARAMETER  a;
2433     TPM2B_ECC_PARAMETER  b;
2434     TPM2B_ECC_PARAMETER  gX;
2435     TPM2B_ECC_PARAMETER  gY;
2436     TPM2B_ECC_PARAMETER  n;
2437     TPM2B_ECC_PARAMETER  h;
2438 } TPMS_ALGORITHM_DETAIL_ECC;
2439
2440 typedef struct
2441 { // (Part 2: Structures)
2442     TPMI_ALG_HASH      hash;
2443     TPM2B_PUBLIC_KEY_RSA sig;
2444 } TPMS_SIGNATURE_RSA;
2445
2446 // Table "Definition of Types for Signature" (Part 2: Structures)
2447 typedef TPMS_SIGNATURE_RSA TPMS_SIGNATURE_RSAPSS;
2448 #define TYPE_OF_TPMS_SIGNATURE_RSAPSS TPMS_SIGNATURE_RSA
2449 typedef TPMS_SIGNATURE_RSA TPMS_SIGNATURE_RSASSA;
2450 #define TYPE_OF_TPMS_SIGNATURE_RSASSA TPMS_SIGNATURE_RSA
2451
2452 typedef struct
2453 { // (Part 2: Structures)
2454     TPMI_ALG_HASH      hash;
2455     TPM2B_ECC_PARAMETER signatureR;
2456     TPM2B_ECC_PARAMETER signatureS;
2457 } TPMS_SIGNATURE_ECC;
2458
2459 // Table "Definition of Types for TPMS_SIGNATURE_ECC" (Part 2: Structures)
2460 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECDA;
2461 #define TYPE_OF_TPMS_SIGNATURE_ECDA TPMS_SIGNATURE_ECC
2462 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECDSA;
2463 #define TYPE_OF_TPMS_SIGNATURE_ECDSA TPMS_SIGNATURE_ECC
2464 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECSCHNORR;
2465 #define TYPE_OF_TPMS_SIGNATURE_ECSCHNORR TPMS_SIGNATURE_ECC
2466 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_EDDSA;
2467 #define TYPE_OF_TPMS_SIGNATURE_EDDSA TPMS_SIGNATURE_ECC
2468 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_EDDSA_PH;
2469 #define TYPE_OF_TPMS_SIGNATURE_EDDSA PH TPMS_SIGNATURE_ECC
2470 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_SM2;
2471 #define TYPE_OF_TPMS_SIGNATURE_SM2 TPMS_SIGNATURE_ECC
2472
2473 typedef union
2474 { // (Part 2: Structures)
2475     #if ALG_HMAC
2476         TPMT_HA hmac;
2477     #endif // ALG_HMAC
2478     #if ALG_RSASSA
2479         TPMS_SIGNATURE_RSASSA rsassa;
2480     #endif // ALG_RSASSA
2481     #if ALG_RSAPSS
2482         TPMS_SIGNATURE_RSAPSS rsapss;
2483     #endif // ALG_RSAPSS
2484     #if ALG_ECDSA
2485         TPMS_SIGNATURE_ECDSA ecdsa;
2486     #endif // ALG_ECDSA
2487     #if ALG_ECDA
2488         TPMS_SIGNATURE_ECDA ecda;
2489     #endif // ALG_ECDA
2490     #if ALG_SM2
2491         TPMS_SIGNATURE_SM2 sm2;
2492     #endif // ALG_SM2
2493     #if ALG_ECSCHNORR
2494         TPMS_SIGNATURE_ECSCHNORR ecschnorr;

```



```

2495 #endif // ALG_EC Schnorr
2496 #if ALG_EDDSA
2497     TPMS_SIGNATURE_EDDSA eddsa;
2498 #endif // ALG_EDDSA
2499 #if ALG_EDDSA_PH
2500     TPMS_SIGNATURE_EDDSA_PH eddsa_ph;
2501 #endif // ALG_EDDSA_PH
2502 #if ALG_LMS
2503     TPMS_SIGNATURE_LMS lms;
2504 #endif // ALG_LMS
2505 #if ALG_XMSS
2506     TPMS_SIGNATURE_XMSS xmss;
2507 #endif // ALG_XMSS
2508     TPMS_SCHEME_HASH any;
2509 } TPMU_SIGNATURE;
2510
2511 typedef struct
2512 { // (Part 2: Structures)
2513     TPMI_ALG_SIG_SCHEME sigAlg;
2514     TPMU_SIGNATURE signature;
2515 } TPMT_SIGNATURE;
2516
2517 typedef union
2518 { // (Part 2: Structures)
2519 #if ALG_ECC
2520     BYTE ecc[sizeof(TPMS_ECC_POINT)];
2521 #endif // ALG_ECC
2522 #if ALG_RSA
2523     BYTE rsa[MAX_RSA_KEY_BYTES];
2524 #endif // ALG_RSA
2525 #if ALG_SYMCIPHER
2526     BYTE symmetric[sizeof(TPM2B_DIGEST)];
2527 #endif // ALG_SYMCIPHER
2528 #if ALG_KEYEDHASH
2529     BYTE keyedHash[sizeof(TPM2B_DIGEST)];
2530 #endif // ALG_KEYEDHASH
2531 } TPMU_ENCRYPTED_SECRET;
2532
2533 typedef union
2534 { // (Part 2: Structures)
2535     struct
2536     {
2537         UINT16 size;
2538         BYTE secret[sizeof(TPMU_ENCRYPTED_SECRET)];
2539     } t;
2540     TPM2B b;
2541 } TPM2B_ENCRYPTED_SECRET;
2542
2543 typedef TPM_ALG_ID TPMI_ALG_PUBLIC; // (Part 2: Structures)
2544 typedef union
2545 { // (Part 2: Structures)
2546 #if ALG_KEYEDHASH
2547     TPM2B_DIGEST keyedHash;
2548 #endif // ALG_KEYEDHASH
2549 #if ALG_SYMCIPHER
2550     TPM2B_DIGEST sym;
2551 #endif // ALG_SYMCIPHER
2552 #if ALG_RSA
2553     TPM2B_PUBLIC_KEY_RSA rsa;
2554 #endif // ALG_RSA
2555 #if ALG_ECC
2556     TPMS_ECC_POINT ecc;
2557 #endif // ALG_ECC
2558     TPMS_DERIVE derive;
2559 } TPMU_PUBLIC_ID;
2560

```

```

2561 typedef struct
2562 { // (Part 2: Structures)
2563     TPMT_KEYEDHASH_SCHEME scheme;
2564 } TPMS_KEYEDHASH_PARMS;
2565
2566 typedef struct
2567 { // (Part 2: Structures)
2568     TPMT_SYM_DEF_OBJECT symmetric;
2569     TPMT_ASYM_SCHEME scheme;
2570 } TPMS_ASYM_PARMS;
2571
2572 typedef struct
2573 { // (Part 2: Structures)
2574     TPMT_SYM_DEF_OBJECT symmetric;
2575     TPMT_RSA_SCHEME scheme;
2576     TPMI_RSA_KEY_BITS keyBits;
2577     UINT32 exponent;
2578 } TPMS_RSA_PARMS;
2579
2580 typedef struct
2581 { // (Part 2: Structures)
2582     TPMT_SYM_DEF_OBJECT symmetric;
2583     TPMT_ECC_SCHEME scheme;
2584     TPMI_ECC_CURVE curveID;
2585     TPMT_KDF_SCHEME kdf;
2586 } TPMS_ECC_PARMS;
2587
2588 typedef union
2589 { // (Part 2: Structures)
2590     #if ALG_KEYEDHASH
2591         TPMS_KEYEDHASH_PARMS keyedHashDetail;
2592     #endif // ALG_KEYEDHASH
2593     #if ALG_SYMCIPHER
2594         TPMS_SYMCIPHER_PARMS symDetail;
2595     #endif // ALG_SYMCIPHER
2596     #if ALG_RSA
2597         TPMS_RSA_PARMS rsaDetail;
2598     #endif // ALG_RSA
2599     #if ALG_ECC
2600         TPMS_ECC_PARMS eccDetail;
2601     #endif // ALG_ECC
2602     TPMS_ASYM_PARMS asymDetail;
2603 } TPMU_PUBLIC_PARMS;
2604
2605 typedef struct
2606 { // (Part 2: Structures)
2607     TPMI_ALG_PUBLIC type;
2608     TPMU_PUBLIC_PARMS parameters;
2609 } TPMT_PUBLIC_PARMS;
2610
2611 typedef struct
2612 { // (Part 2: Structures)
2613     TPMI_ALG_PUBLIC type;
2614     TPMI_ALG_HASH nameAlg;
2615     TPMA_OBJECT objectAttributes;
2616     TPM2B_DIGEST authPolicy;
2617     TPMU_PUBLIC_PARMS parameters;
2618     TPMU_PUBLIC_ID unique;
2619 } TPMT_PUBLIC;
2620
2621 typedef struct
2622 { // (Part 2: Structures)
2623     UINT16 size;
2624     TPMT_PUBLIC publicArea;
2625 } TPM2B_PUBLIC;
2626

```

```

2627 typedef union
2628 { // (Part 2: Structures)
2629     struct
2630     {
2631         UINT16 size;
2632         BYTE    buffer[sizeof(TPMT_PUBLIC)];
2633     } t;
2634     TPM2B b;
2635 } TPM2B_TEMPLATE;
2636
2637 typedef union
2638 { // (Part 2: Structures)
2639     struct
2640     {
2641         UINT16 size;
2642         BYTE    buffer[PRIVATE_VENDOR_SPECIFIC_BYTES];
2643     } t;
2644     TPM2B b;
2645 } TPM2B_PRIVATE_VENDOR_SPECIFIC;
2646
2647 typedef union
2648 { // (Part 2: Structures)
2649     #if ALG_RSA
2650         TPM2B_PRIVATE_KEY_RSA rsa;
2651     #endif // ALG_RSA
2652     #if ALG_ECC
2653         TPM2B_ECC_PARAMETER ecc;
2654     #endif // ALG_ECC
2655     #if ALG_KEYEDHASH
2656         TPM2B_SENSITIVE_DATA bits;
2657     #endif // ALG_KEYEDHASH
2658     #if ALG_SYMCIPHER
2659         TPM2B_SYM_KEY sym;
2660     #endif // ALG_SYMCIPHER
2661     TPM2B_PRIVATE_VENDOR_SPECIFIC any;
2662 } TPMU_SENSITIVE_COMPOSITE;
2663
2664 typedef struct
2665 { // (Part 2: Structures)
2666     TPMI_ALG_PUBLIC sensitiveType;
2667     TPM2B_AUTH authValue;
2668     TPM2B_DIGEST seedValue;
2669     TPMU_SENSITIVE_COMPOSITE sensitive;
2670 } TPMT_SENSITIVE;
2671
2672 typedef struct
2673 { // (Part 2: Structures)
2674     UINT16 size;
2675     TPMT_SENSITIVE sensitiveArea;
2676 } TPM2B_SENSITIVE;
2677
2678 typedef struct
2679 { // (Part 2: Structures)
2680     TPM2B_DIGEST integrityOuter;
2681     TPM2B_DIGEST integrityInner;
2682     TPM2B_SENSITIVE sensitive;
2683 } _PRIVATE;
2684
2685 typedef union
2686 { // (Part 2: Structures)
2687     struct
2688     {
2689         UINT16 size;
2690         BYTE    buffer[sizeof(_PRIVATE)];
2691     } t;
2692     TPM2B b;

```

```

2693 } TPM2B_PRIVATE;
2694
2695 typedef struct
2696 { // (Part 2: Structures)
2697     TPM2B_DIGEST integrityHMAC;
2698     TPM2B_DIGEST encIdentity;
2699 } TPMS_ID_OBJECT;
2700
2701 typedef union
2702 { // (Part 2: Structures)
2703     struct
2704     {
2705         UINT16 size;
2706         BYTE credential[sizeof(TPMS_ID_OBJECT)];
2707     } t;
2708     TPM2B b;
2709 } TPM2B_ID_OBJECT;
2710
2711 // Table "Definition of TPM_NV_INDEX Bits" (Part 2: Structures)
2712 #define TYPE_OF_TPM_NV_INDEX      UINT32
2713 #define TPM_NV_INDEX_TO_UINT32(a) (*(UINT32*)&(a))
2714 #define UINT32_TO_TPM_NV_INDEX(a) (*(TPM_NV_INDEX*)&(a))
2715 #define TPM_NV_INDEX_TO_BYTE_ARRAY(i, a) \
2716     UINT32_TO_BYTE_ARRAY((TPM_NV_INDEX_TO_UINT32(i)), (a))
2717 #define BYTE_ARRAY_TO_TPM_NV_INDEX(i, a) \
2718     { \
2719         UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
2720         i = UINT32_TO_TPM_NV_INDEX(x); \
2721     }
2722 #if USE_BIT_FIELD_STRUCTURES
2723 typedef struct
2724 {
2725     unsigned index : 24;
2726     unsigned RH_NV : 8;
2727 } TPM_NV_INDEX;
2728
2729 // Initializer for the bit-field structure
2730 # define TPM_NV_INDEX_INITIALIZER(index, rh_nv) \
2731     { \
2732         index, rh_nv \
2733     }
2734 #else // USE_BIT_FIELD_STRUCTURES
2735
2736 // This implements Table "Definition of TPM_NV_INDEX Bits" (Part 2: Structures) using
2737 // bit masking
2738 typedef UINT32 TPM_NV_INDEX;
2739 # define TPM_NV_INDEX_index      (TPM_NV_INDEX) (0xFFFFF << 0)
2740 # define TPM_NV_INDEX_index_SHIFT 0
2741 # define TPM_NV_INDEX_RH_NV      (TPM_NV_INDEX) (0xFF << 24)
2742 # define TPM_NV_INDEX_RH_NV_SHIFT 24
2743
2744 // This is the initializer for a TPM_NV_INDEX bit array.
2745 # define TPM_NV_INDEX_INITIALIZER(index, rh_nv) \
2746     (TPM_NV_INDEX) ((index << 0) + (rh_nv << 24))
2747 #endif // USE_BIT_FIELD_STRUCTURES
2748
2749 // Table "Definition of TPM_NT Constants" (Part 2: Structures)
2750 typedef UINT32 TPM_NT;
2751 #define TYPE_OF_TPM_NT      UINT32
2752 #define TPM_NT_ORDINARY      (TPM_NT) (0x0)
2753 #define TPM_NT_COUNTER      (TPM_NT) (0x1)
2754 #define TPM_NT_BITS          (TPM_NT) (0x2)
2755 #define TPM_NT_EXTEND        (TPM_NT) (0x4)
2756 #define TPM_NT_PIN_FAIL      (TPM_NT) (0x8)
2757 #define TPM_NT_PIN_PASS      (TPM_NT) (0x9)

```

```

2758
2759 typedef struct
2760 { // (Part 2: Structures)
2761     UINT32 pinCount;
2762     UINT32 pinLimit;
2763 } TPMS_NV_PIN_COUNTER_PARAMETERS;
2764
2765 // Table "Definition of TPMA_NV Bits" (Part 2: Structures)
2766 #define TYPE_OF_TPMA_NV          UINT32
2767 #define TPMA_NV_TO_UINT32(a)      (*(UINT32*)&(a))
2768 #define UINT32_TO_TPMA_NV(a)      (*(TPMA_NV*)&(a))
2769 #define TPMA_NV_TO_BYTE_ARRAY(i, a)  UINT32_TO_BYTE_ARRAY((TPMA_NV_TO_UINT32(i)), (a))
2770 #define BYTE_ARRAY_TO_TPMA_NV(i, a)  \
2771     { \
2772         UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
2773         i        = UINT32_TO_TPMA_NV(x); \
2774     }
2775 #if USE_BIT_FIELD_STRUCTURES
2776 typedef struct
2777 {
2778     unsigned PPWRITE           : 1;
2779     unsigned OWNERWRITE       : 1;
2780     unsigned AUTHWRITE        : 1;
2781     unsigned POLICYWRITE      : 1;
2782     unsigned TPM_NT           : 4;
2783     unsigned Reserved_bits_at_8 : 2;
2784     unsigned POLICY_DELETE    : 1;
2785     unsigned WRITELOCKED      : 1;
2786     unsigned WRITEALL         : 1;
2787     unsigned WRITEDEFINE      : 1;
2788     unsigned WRITE_STCLEAR    : 1;
2789     unsigned GLOBALLOCK       : 1;
2790     unsigned PPREAD           : 1;
2791     unsigned OWNERREAD        : 1;
2792     unsigned AUTHREAD         : 1;
2793     unsigned POLICYREAD       : 1;
2794     unsigned Reserved_bits_at_20 : 5;
2795     unsigned NO_DA            : 1;
2796     unsigned ORDERLY          : 1;
2797     unsigned CLEAR_STCLEAR    : 1;
2798     unsigned READLOCKED      : 1;
2799     unsigned WRITTEN          : 1;
2800     unsigned PLATFORMCREATE   : 1;
2801     unsigned READ_STCLEAR     : 1;
2802 } TPMA_NV;
2803
2804 // Initializer for the bit-field structure
2805 # define TPMA_NV_INITIALIZER(ppwrite, \
2806                             ownerwrite, \
2807                             authwrite, \
2808                             policywrite, \
2809                             tpm_nt, \
2810                             bits_at_8, \
2811                             policy_delete, \
2812                             writelocked, \
2813                             writeall, \
2814                             writedefine, \
2815                             write_stclear, \
2816                             globallock, \
2817                             ppread, \
2818                             ownerread, \
2819                             authread, \
2820                             policyread, \
2821                             bits_at_20, \
2822                             no_da, \
2823                             orderly,

```

```

2824         clear_stclear,
2825         readlocked,
2826         written,
2827         platformcreate,
2828         read_stclear)
2829     {
2830         ppwrite, ownerwrite, authwrite, policywrite, tpm_nt, bits_at_8,
2831         policy_delete, writelocked, writeall, writedefine, write_stclear,
2832         globallock, ppread, ownerread, authread, policyread, bits_at_20,
2833         no_da, orderly, clear_stclear, readlocked, written, platformcreate,
2834         read_stclear
2835     }
2836 #else // USE_BIT_FIELD_STRUCTURES
2837
2838 // This implements Table "Definition of TPMA_NV Bits" (Part 2: Structures) using bit
2839 // masking
2840 typedef UINT32 TPMA_NV;
2841 # define TPMA_NV_PPWRITE (TPMA_NV) (1 << 0)
2842 # define TPMA_NV_OWNERWRITE (TPMA_NV) (1 << 1)
2843 # define TPMA_NV_AUTHWRITE (TPMA_NV) (1 << 2)
2844 # define TPMA_NV_POLICYWRITE (TPMA_NV) (1 << 3)
2845 # define TPMA_NV_TPM_NT (TPMA_NV) (0xF << 4)
2846 # define TPMA_NV_TPM_NT_SHIFT 4
2847 # define TPMA_NV_POLICY_DELETE (TPMA_NV) (1 << 10)
2848 # define TPMA_NV_WRITELOCKED (TPMA_NV) (1 << 11)
2849 # define TPMA_NV_WRITEALL (TPMA_NV) (1 << 12)
2850 # define TPMA_NV_WRITEDEFINE (TPMA_NV) (1 << 13)
2851 # define TPMA_NV_WRITE_STCLEAR (TPMA_NV) (1 << 14)
2852 # define TPMA_NV_GLOBALLOCK (TPMA_NV) (1 << 15)
2853 # define TPMA_NV_PPREAD (TPMA_NV) (1 << 16)
2854 # define TPMA_NV_OWNERREAD (TPMA_NV) (1 << 17)
2855 # define TPMA_NV_AUTHREAD (TPMA_NV) (1 << 18)
2856 # define TPMA_NV_POLICYREAD (TPMA_NV) (1 << 19)
2857 # define TPMA_NV_NO_DA (TPMA_NV) (1 << 25)
2858 # define TPMA_NV_ORDERLY (TPMA_NV) (1 << 26)
2859 # define TPMA_NV_CLEAR_STCLEAR (TPMA_NV) (1 << 27)
2860 # define TPMA_NV_READLOCKED (TPMA_NV) (1 << 28)
2861 # define TPMA_NV_WRITTEN (TPMA_NV) (1 << 29)
2862 # define TPMA_NV_PLATFORMCREATE (TPMA_NV) (1 << 30)
2863 # define TPMA_NV_READ_STCLEAR (TPMA_NV) (1 << 31)
2864
2865 // This is the initializer for a TPMA_NV bit array.
2866 # define TPMA_NV_INITIALIZER(ppwrite,
2867                             ownerwrite,
2868                             authwrite,
2869                             policywrite,
2870                             tpm_nt,
2871                             bits_at_8,
2872                             policy_delete,
2873                             writelocked,
2874                             writeall,
2875                             writedefine,
2876                             write_stclear,
2877                             globallock,
2878                             ppread,
2879                             ownerread,
2880                             authread,
2881                             policyread,
2882                             bits_at_20,
2883                             no_da,
2884                             orderly,
2885                             clear_stclear,
2886                             readlocked,
2887                             written,
2888                             platformcreate,
2889                             read_stclear)

```



```

2889         (TPMA_NV) ((ppwrite << 0) + (ownerwrite << 1) + (authwrite << 2)      \
2890                   + (policywrite << 3) + (tpm_nt << 4) + (policy_delete << 10) \
2891                   + (writelocked << 11) + (writeall << 12) + (writedefine << 13) \
2892                   + (write_stclear << 14) + (globallock << 15) + (ppread << 16)  \
2893                   + (ownerread << 17) + (authread << 18) + (policyread << 19)   \
2894                   + (no_da << 25) + (orderly << 26) + (clear_stclear << 27)     \
2895                   + (readlocked << 28) + (written << 29) + (platformcreate << 30) \
2896                   + (read_stclear << 31))
2897
2898 #endif // USE_BIT_FIELD_STRUCTURES
2899
2900 // Table "Definition of TPMA_NV_EXP Bits" (Part 2: Structures)
2901 #define TYPE_OF_TPMA_NV_EXP      UINT64
2902 #define TPMA_NV_EXP_TO_UINT64(a) (*(UINT64*)&(a))
2903 #define UINT64_TO_TPMA_NV_EXP(a) (*(TPMA_NV_EXP*)&(a))
2904 #define TPMA_NV_EXP_TO_BYTE_ARRAY(i, a) \
2905     UINT64_TO_BYTE_ARRAY((TPMA_NV_EXP_TO_UINT64(i)), (a))
2906 #define BYTE_ARRAY_TO_TPMA_NV_EXP(i, a) \
2907     { \
2908         UINT64 x = BYTE_ARRAY_TO_UINT64(a); \
2909         i = UINT64_TO_TPMA_NV_EXP(x); \
2910     }
2911 #if USE_BIT_FIELD_STRUCTURES
2912 typedef struct
2913 {
2914     unsigned TPMA_NV_PPWRITE           : 1;
2915     unsigned TPMA_NV_OWNERWRITE        : 1;
2916     unsigned TPMA_NV_AUTHWRITE         : 1;
2917     unsigned TPMA_NV_POLICYWRITE       : 1;
2918     unsigned TPM_NT                    : 4;
2919     unsigned Reserved_bits_at_8        : 2;
2920     unsigned TPMA_NV_POLICY_DELETE     : 1;
2921     unsigned TPMA_NV_WRITELOCKED       : 1;
2922     unsigned TPMA_NV_WRITEALL          : 1;
2923     unsigned TPMA_NV_WRIEDEDEFINE      : 1;
2924     unsigned TPMA_NV_WRITE_STCLEAR     : 1;
2925     unsigned TPMA_NV_GLOBALLOCK        : 1;
2926     unsigned TPMA_NV_PPREAD            : 1;
2927     unsigned TPMA_NV_OWNERREAD         : 1;
2928     unsigned TPMA_NV_AUTHREAD          : 1;
2929     unsigned TPMA_NV_POLICYREAD        : 1;
2930     unsigned Reserved_bits_at_20       : 5;
2931     unsigned TPMA_NV_NO_DA             : 1;
2932     unsigned TPMA_NV_ORDERLY           : 1;
2933     unsigned TPMA_NV_CLEAR_STCLEAR     : 1;
2934     unsigned TPMA_NV_READLOCKED        : 1;
2935     unsigned TPMA_NV_WRITTEN           : 1;
2936     unsigned TPMA_NV_PLATFORMCREATE    : 1;
2937     unsigned TPMA_NV_READ_STCLEAR      : 1;
2938     unsigned TPMA_EXTERNAL_NV_ENCRYPTION : 1;
2939     unsigned TPMA_EXTERNAL_NV_INTEGRITY : 1;
2940     unsigned TPMA_EXTERNAL_NV_ANTIROLLBACK : 1;
2941     unsigned Reserved_bits_at_35       : 29;
2942 } TPMA_NV_EXP;
2943
2944 // Initializer for the bit-field structure
2945 # define TPMA_NV_EXP_INITIALIZER(tpma_nv_ppwrite, \
2946                                 tpma_nv_ownerwrite, \
2947                                 tpma_nv_authwrite, \
2948                                 tpma_nv_policywrite, \
2949                                 tpm_nt, \
2950                                 bits_at_8, \
2951                                 tpma_nv_policy_delete, \
2952                                 tpma_nv_writelocked, \
2953                                 tpma_nv_writeall, \
2954                                 tpma_nv_writedefine, \

```

```

2955         tpma_nv_write_stclear,
2956         tpma_nv_globallock,
2957         tpma_nv_ppread,
2958         tpma_nv_ownerread,
2959         tpma_nv_authread,
2960         tpma_nv_policyread,
2961         bits_at_20,
2962         tpma_nv_no_da,
2963         tpma_nv_orderly,
2964         tpma_nv_clear_stclear,
2965         tpma_nv_readlocked,
2966         tpma_nv_written,
2967         tpma_nv_platformcreate,
2968         tpma_nv_read_stclear,
2969         tpma_external_nv_encryption,
2970         tpma_external_nv_integrity,
2971         tpma_external_nv_antirollback,
2972         bits_at_35)
2973     {
2974         tpma_nv_ppwrite, tpma_nv_ownerwrite, tpma_nv_authwrite,
2975         tpma_nv_policywrite, tpm_nt, bits_at_8, tpma_nv_policy_delete,
2976         tpma_nv_writelocked, tpma_nv_writeall, tpma_nv_writedefine,
2977         tpma_nv_write_stclear, tpma_nv_globallock, tpma_nv_ppread,
2978         tpma_nv_ownerread, tpma_nv_authread, tpma_nv_policyread, bits_at_20,
2979         tpma_nv_no_da, tpma_nv_orderly, tpma_nv_clear_stclear,
2980         tpma_nv_readlocked, tpma_nv_written, tpma_nv_platformcreate,
2981         tpma_nv_read_stclear, tpma_external_nv_encryption,
2982         tpma_external_nv_integrity, tpma_external_nv_antirollback, bits_at_35 \
2983     }
2984 #else // USE_BIT_FIELD_STRUCTURES
2985
2986 // This implements Table "Definition of TPMA_NV_EXP Bits" (Part 2: Structures) using
2987 // bit masking
2988 typedef UINT64 TPMA_NV_EXP;
2989 # define TPMA_NV_EXP_TPMA_NV_PPWRITE (TPMA_NV_EXP) (1 << 0)
2990 # define TPMA_NV_EXP_TPMA_NV_OWNERWRITE (TPMA_NV_EXP) (1 << 1)
2991 # define TPMA_NV_EXP_TPMA_NV_AUTHWRITE (TPMA_NV_EXP) (1 << 2)
2992 # define TPMA_NV_EXP_TPMA_NV_POLICYWRITE (TPMA_NV_EXP) (1 << 3)
2993 # define TPMA_NV_EXP_TPM_NT (TPMA_NV_EXP) (0xF << 4)
2994 # define TPMA_NV_EXP_TPM_NT_SHIFT 4
2995 # define TPMA_NV_EXP_TPMA_NV_POLICY_DELETE (TPMA_NV_EXP) (1 << 10)
2996 # define TPMA_NV_EXP_TPMA_NV_WRITELOCKED (TPMA_NV_EXP) (1 << 11)
2997 # define TPMA_NV_EXP_TPMA_NV_WRITEALL (TPMA_NV_EXP) (1 << 12)
2998 # define TPMA_NV_EXP_TPMA_NV_WRIEDEDEFINE (TPMA_NV_EXP) (1 << 13)
2999 # define TPMA_NV_EXP_TPMA_NV_WRITE_STCLEAR (TPMA_NV_EXP) (1 << 14)
3000 # define TPMA_NV_EXP_TPMA_NV_GLOBALLOCK (TPMA_NV_EXP) (1 << 15)
3001 # define TPMA_NV_EXP_TPMA_NV_PPREAD (TPMA_NV_EXP) (1 << 16)
3002 # define TPMA_NV_EXP_TPMA_NV_OWNERREAD (TPMA_NV_EXP) (1 << 17)
3003 # define TPMA_NV_EXP_TPMA_NV_AUTHREAD (TPMA_NV_EXP) (1 << 18)
3004 # define TPMA_NV_EXP_TPMA_NV_POLICYREAD (TPMA_NV_EXP) (1 << 19)
3005 # define TPMA_NV_EXP_TPMA_NV_NO_DA (TPMA_NV_EXP) (1 << 25)
3006 # define TPMA_NV_EXP_TPMA_NV_ORDERLY (TPMA_NV_EXP) (1 << 26)
3007 # define TPMA_NV_EXP_TPMA_NV_CLEAR_STCLEAR (TPMA_NV_EXP) (1 << 27)
3008 # define TPMA_NV_EXP_TPMA_NV_READLOCKED (TPMA_NV_EXP) (1 << 28)
3009 # define TPMA_NV_EXP_TPMA_NV_WRITTEN (TPMA_NV_EXP) (1 << 29)
3010 # define TPMA_NV_EXP_TPMA_NV_PLATFORMCREATE (TPMA_NV_EXP) (1 << 30)
3011 # define TPMA_NV_EXP_TPMA_NV_READ_STCLEAR (TPMA_NV_EXP) (1 << 31)
3012 # define TPMA_NV_EXP_TPMA_EXTERNAL_NV_ENCRYPTION (TPMA_NV_EXP) (1 << 32)
3013 # define TPMA_NV_EXP_TPMA_EXTERNAL_NV_INTEGRITY (TPMA_NV_EXP) (1 << 33)
3014 # define TPMA_NV_EXP_TPMA_EXTERNAL_NV_ANTIROLLBACK (TPMA_NV_EXP) (1 << 34)
3015
3016 // This is the initializer for a TPMA_NV_EXP bit array.
3017 # define TPMA_NV_EXP_INITIALIZER(tpma_nv_ppwrite,
3018                                tpma_nv_ownerwrite,
3019                                tpma_nv_authwrite,
3020                                tpma_nv_policywrite,

```

```

3020         tpm_nt, \
3021         bits_at_8, \
3022         tpma_nv_policy_delete, \
3023         tpma_nv_writelocked, \
3024         tpma_nv_writeall, \
3025         tpma_nv_writedefine, \
3026         tpma_nv_write_stclear, \
3027         tpma_nv_globallock, \
3028         tpma_nv_ppread, \
3029         tpma_nv_ownerread, \
3030         tpma_nv_authread, \
3031         tpma_nv_policyread, \
3032         bits_at_20, \
3033         tpma_nv_no_da, \
3034         tpma_nv_orderly, \
3035         tpma_nv_clear_stclear, \
3036         tpma_nv_readlocked, \
3037         tpma_nv_written, \
3038         tpma_nv_platformcreate, \
3039         tpma_nv_read_stclear, \
3040         tpma_external_nv_encryption, \
3041         tpma_external_nv_integrity, \
3042         tpma_external_nv_antirollback, \
3043         bits_at_35) \
3044     (TPMA_NV_EXP) ((tpma_nv_ppwrite << 0) + (tpma_nv_ownerwrite << 1) \
3045         + (tpma_nv_authwrite << 2) + (tpma_nv_policywrite << 3) \
3046         + (tpm_nt << 4) + (tpma_nv_policy_delete << 10) \
3047         + (tpma_nv_writelocked << 11) + (tpma_nv_writeall << 12) \
3048         + (tpma_nv_writedefine << 13) + (tpma_nv_write_stclear << 14) \
3049         + (tpma_nv_globallock << 15) + (tpma_nv_ppread << 16) \
3050         + (tpma_nv_ownerread << 17) + (tpma_nv_authread << 18) \
3051         + (tpma_nv_policyread << 19) + (tpma_nv_no_da << 25) \
3052         + (tpma_nv_orderly << 26) + (tpma_nv_clear_stclear << 27) \
3053         + (tpma_nv_readlocked << 28) + (tpma_nv_written << 29) \
3054         + (tpma_nv_platformcreate << 30) + (tpma_nv_read_stclear << 31) \
3055         + (tpma_external_nv_encryption << 32) \
3056         + (tpma_external_nv_integrity << 33) \
3057         + (tpma_external_nv_antirollback << 34)) \
3058 \
3059 #endif // USE_BIT_FIELD_STRUCTURES
3060
3061 typedef struct
3062 { // (Part 2: Structures)
3063     TPMI_RH_NV_LEGACY_INDEX nvIndex;
3064     TPMI_ALG_HASH            nameAlg;
3065     TPMA_NV                  attributes;
3066     TPM2B_DIGEST             authPolicy;
3067     UINT16                   dataSize;
3068 } TPMS_NV_PUBLIC;
3069
3070 typedef struct
3071 { // (Part 2: Structures)
3072     UINT16            size;
3073     TPMS_NV_PUBLIC nvPublic;
3074 } TPM2B_NV_PUBLIC;
3075
3076 typedef struct
3077 { // (Part 2: Structures)
3078     TPMI_RH_NV_EXP_INDEX nvIndex;
3079     TPMI_ALG_HASH         nameAlg;
3080     TPMA_NV_EXP           attributes;
3081     TPM2B_DIGEST          authPolicy;
3082     UINT16                dataSize;
3083 } TPMS_NV_PUBLIC_EXP_ATTR;
3084
3085 typedef union

```

```

3086 { // (Part 2: Structures)
3087     TPMS_NV_PUBLIC      nvIndex;
3088     TPMS_NV_PUBLIC_EXP_ATTR externalNV;
3089     TPMS_NV_PUBLIC      permanentNV;
3090 } TPMU_NV_PUBLIC_2;
3091
3092 typedef struct
3093 { // (Part 2: Structures)
3094     TPM_HT      handleType;
3095     TPMU_NV_PUBLIC_2 nvPublic2;
3096 } TPMT_NV_PUBLIC_2;
3097
3098 typedef struct
3099 { // (Part 2: Structures)
3100     UINT16      size;
3101     TPMT_NV_PUBLIC_2 nvPublic2;
3102 } TPM2B_NV_PUBLIC_2;
3103
3104 typedef union
3105 { // (Part 2: Structures)
3106     struct
3107     {
3108         UINT16 size;
3109         BYTE  buffer[MAX_CONTEXT_SIZE];
3110     } t;
3111     TPM2B b;
3112 } TPM2B_CONTEXT_SENSITIVE;
3113
3114 typedef struct
3115 { // (Part 2: Structures)
3116     TPM2B_DIGEST      integrity;
3117     TPM2B_CONTEXT_SENSITIVE encrypted;
3118 } TPMS_CONTEXT_DATA;
3119
3120 typedef union
3121 { // (Part 2: Structures)
3122     struct
3123     {
3124         UINT16 size;
3125         BYTE  buffer[sizeof(TPMS_CONTEXT_DATA)];
3126     } t;
3127     TPM2B b;
3128 } TPM2B_CONTEXT_DATA;
3129
3130 typedef struct
3131 { // (Part 2: Structures)
3132     UINT64      sequence;
3133     TPMI_DH_SAVED      savedHandle;
3134     TPMI_RH_HIERARCHY hierarchy;
3135     TPM2B_CONTEXT_DATA contextBlob;
3136 } TPMS_CONTEXT;
3137
3138 typedef struct
3139 { // (Part 2: Structures)
3140     TPML_PCR_SELECTION pcrSelect;
3141     TPM2B_DIGEST      pcrDigest;
3142     TPMA_LOCALITY      locality;
3143     TPM_ALG_ID      parentNameAlg;
3144     TPM2B_NAME      parentName;
3145     TPM2B_NAME      parentQualifiedName;
3146     TPM2B_DATA      outsideInfo;
3147 } TPMS_CREATION_DATA;
3148
3149 typedef struct
3150 { // (Part 2: Structures)
3151     UINT16      size;

```

```

3152     TPMS_CREATION_DATA creationData;
3153 } TPM2B_CREATION_DATA;
3154
3155 // Table "Definition of TPM_AT Constants" (Part 2: Structures)
3156 typedef UINT32 TPM_AT;
3157 #define TYPE_OF_TPM_AT UINT32
3158 #define TPM_AT_ANY      (TPM_AT) (0x00000000)
3159 #define TPM_AT_ERROR    (TPM_AT) (0x00000001)
3160 #define TPM_AT_PV1      (TPM_AT) (0x00000002)
3161 #define TPM_AT_VEND     (TPM_AT) (0x80000000)
3162
3163 // Table "Definition of TPM_AE Constants" (Part 2: Structures)
3164 typedef UINT32 TPM_AE;
3165 #define TYPE_OF_TPM_AE UINT32
3166 #define TPM_AE_NONE     (TPM_AE) (0x00000000)
3167
3168 typedef struct
3169 { // (Part 2: Structures)
3170     TPM_AT tag;
3171     UINT32 data;
3172 } TPMS_AC_OUTPUT;
3173
3174 typedef struct
3175 { // (Part 2: Structures)
3176     UINT32 count;
3177     TPMS_AC_OUTPUT acCapabilities[MAX_AC_CAPABILITIES];
3178 } TPML_AC_CAPABILITIES;
3179
3180 #endif // _TPM_INCLUDE_PRIVATE_TPMTYPES_H

```

6.251 /tpm/include/public/tpm_public.h

```

1  #include <TpmConfiguration/TpmBuildSwitches.h>
2  #include <TpmConfiguration/TpmProfile.h>
3
4  #include "VerifyConfiguration.h"
5  #include "BaseTypes.h"
6  #include "TPMB.h"
7  #include "MinMax.h"
8  #include "tpm_radix.h"
9  #include "TpmTypes.h"

```

6.252 /tpm/include/public/tpm_radix.h

```

1  /** Introduction
2  // Common defines for supporting large numbers and cryptographic buffer sizing.
3  //*****
4  #ifndef RADIX_BITS
5  #   if defined(__x86_64__) || defined(_x86_64) || defined(__amd64__) \
6      || defined(_amd64) || defined(WIN64) || defined(_M_X64) || defined(_M_ARM64) \
7      || defined(__aarch64__) || defined(_PPC64_) || defined(__s390x__) \
8      || defined(_powerpc64_) || defined(_ppc64_)
9  #       define RADIX_BITS 64
10 #   elif defined(__i386__) || defined(_i386) || defined(i386) || defined(_WIN32) \
11      || defined(_M_IX86)
12 #       define RADIX_BITS 32
13 #   elif defined(_M_ARM) || defined(__arm__) || defined(__thumb__)
14 #       define RADIX_BITS 32
15 #   elif defined(__riscv)
16 // __riscv and __riscv_xlen are standardized by the RISC-V community and should be
17 // available
18 // on any compliant compiler.
19 //
20 // https://github.com/riscv-non-isa/riscv-toolchain-conventions

```



```

20 # define RADIX_BITS __riscv_xlen
21 # else
22 # error Unable to determine RADIX_BITS from compiler environment
23 # endif
24 #endif // RADIX_BITS
25
26 #if RADIX_BITS == 64
27 # define RADIX_BYTES 8
28 # define RADIX_LOG2 6
29 #elif RADIX_BITS == 32
30 # define RADIX_BYTES 4
31 # define RADIX_LOG2 5
32 #else
33 # error "RADIX_BITS must either be 32 or 64"
34 #endif
35
36 #define HASH_ALIGNMENT RADIX_BYTES
37 #define SYMMETRIC_ALIGNMENT RADIX_BYTES
38
39 #define RADIX_MOD(x) ((x) & ((1 << RADIX_LOG2) - 1))
40 #define RADIX_DIV(x) ((x) >> RADIX_LOG2)
41 #define RADIX_MASK (((crypt_ushort_t)1) << RADIX_LOG2) - 1)
42
43 #define BITS_TO_CRYPT_WORDS(bits) RADIX_DIV((bits) + (RADIX_BITS - 1))
44 #define BYTES_TO_CRYPT_WORDS(bytes) BITS_TO_CRYPT_WORDS(bytes * 8)
45 #define SIZE_IN_CRYPT_WORDS(thing) BYTES_TO_CRYPT_WORDS(sizeof(thing))
46
47 #if RADIX_BITS == 64
48 # define SWAP_CRYPT_WORD(x) REVERSE_ENDIAN_64(x)
49 typedef uint64_t crypt_ushort_t;
50 typedef int64_t crypt_word_t;
51 # define TO_CRYPT_WORD_64 BIG_ENDIAN_BYTES_TO_UINT64
52 # define TO_CRYPT_WORD_32(a, b, c, d) TO_CRYPT_WORD_64(0, 0, 0, 0, a, b, c, d)
53 #elif RADIX_BITS == 32
54 # define SWAP_CRYPT_WORD(x) REVERSE_ENDIAN_32((x))
55 typedef uint32_t crypt_ushort_t;
56 typedef int32_t crypt_word_t;
57 # define TO_CRYPT_WORD_64(a, b, c, d, e, f, g, h) \
58     BIG_ENDIAN_BYTES_TO_UINT32(e, f, g, h), BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d)
59 #endif
60
61 #define MAX_CRYPT_UWORD (~((crypt_ushort_t)0))
62 #define MAX_CRYPT_WORD ((crypt_word_t)(MAX_CRYPT_UWORD >> 1))
63 #define MIN_CRYPT_WORD (~MAX_CRYPT_WORD)
64
65 // Avoid expanding LARGEST_NUMBER into a long expression that inlines 3 other long
66 // expressions.
67 // TODO: Decrease the size of each of the MAX_* expressions with improvements to the
68 // code generator.
69 #if ALG_RSA == ALG_YES
70 // The smallest supported RSA key (1024 bits) is larger than
71 // the largest supported ECC curve (628 bits)
72 // or the largest supported digest (512 bits)
73 # define LARGEST_NUMBER MAX_RSA_KEY_BYTES
74 #elif ALG_ECC == ALG_YES
75 # define LARGEST_NUMBER MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE)
76 #else
77 # define LARGEST_NUMBER MAX_DIGEST_SIZE
78 #endif // ALG_RSA == YES
79
80 #define LARGEST_NUMBER_BITS (LARGEST_NUMBER * 8)
81
82 #define MAX_ECC_PARAMETER_BYTES (MAX_ECC_KEY_BYTES * ALG_ECC)
83
84 // These macros use the selected libraries to get the proper include files.
85 // clang-format off

```



```

84 #define LIB_QUOTE(_STRING_)          #_STRING_
85 #define LIB_INCLUDE2(_PREFIX_, _LIB_, _TYPE_)
LIB_QUOTE(_LIB_ / _PREFIX_ ## _LIB_ ## _TYPE_.h)
86 #define LIB_INCLUDE(_PREFIX_, _LIB_, _TYPE_) LIB_INCLUDE2(_PREFIX_, _LIB_, _TYPE_)
87 // clang-format on

```

6.253 /tpm/include/public/VerifyConfiguration.h

```

1 //
2 // This verifies that information expected from the consumer's TpmConfiguration is
3 // set properly and consistently.
4 //
5 #ifndef _VERIFY_CONFIGURATION_H
6 #define _VERIFY_CONFIGURATION_H
7
8 // verify these defines are either YES or NO.
9 #define MUST_BE_0_OR_1(x) MUST_BE((x) == 0) || ((x) == 1))
10
11 // Debug Options
12 MUST_BE_0_OR_1(DEBUG);
13 MUST_BE_0_OR_1(SIMULATION);
14 MUST_BE_0_OR_1(DRBG_DEBUG_PRINT);
15 MUST_BE_0_OR_1(CERTIFYX509_DEBUG);
16 MUST_BE_0_OR_1(USE_DEBUG_RNG);
17
18 // RSA Debug Options
19 MUST_BE_0_OR_1(RSA_INSTRUMENT);
20 MUST_BE_0_OR_1(USE_RSA_KEY_CACHE);
21 MUST_BE_0_OR_1(USE_KEY_CACHE_FILE);
22
23 // Test Options
24 MUST_BE_0_OR_1(ALLOW_FORCE_FAILURE_MODE);
25
26 // Internal checks
27 MUST_BE_0_OR_1(LIBRARY_COMPATIBILITY_CHECK);
28 MUST_BE_0_OR_1(COMPILER_CHECKS);
29 MUST_BE_0_OR_1(RUNTIME_SIZE_CHECKS);
30
31 // Compliance options
32 MUST_BE_0_OR_1(FIPS_COMPLIANT);
33 MUST_BE_0_OR_1(USE_SPEC_COMPLIANT_PROOFS);
34 MUST_BE_0_OR_1(SKIP_PROOF_ERRORS);
35
36 // Implementation alternatives - should not change external behavior
37 MUST_BE_0_OR_1(TABLE_DRIVEN_DISPATCH);
38 MUST_BE_0_OR_1(TABLE_DRIVEN_MARSHAL);
39 MUST_BE_0_OR_1(USE_MARSHALING_DEFINES);
40 MUST_BE_0_OR_1(COMPRESSED_LISTS);
41 MUST_BE_0_OR_1(USE_BIT_FIELD_STRUCTURES);
42 MUST_BE_0_OR_1(RSA_KEY_SIEVE);
43
44 // Implementation alternatives - changes external behavior
45 MUST_BE_0_OR_1(_DRBG_STATE_SAVE);
46 MUST_BE_0_OR_1(USE_DA_USED);
47 MUST_BE_0_OR_1(SELF_TEST);
48 MUST_BE_0_OR_1(CLOCK_STOPS);
49 MUST_BE_0_OR_1(ACCUMULATE_SELF_HEAL_TIMER);
50 MUST_BE_0_OR_1(FAIL_TRACE);
51
52 // Vendor alternatives
53 // Check VENDOR_PERMANENT_AUTH_ENABLED & VENDOR_PERMANENT_AUTH_HANDLE are consistent
54 MUST_BE_0_OR_1(VENDOR_PERMANENT_AUTH_ENABLED);
55
56 #if VENDOR_PERMANENT_AUTH_ENABLED == YES
57 # if !defined(VENDOR_PERMANENT_AUTH_HANDLE) \

```

```

58     || VENDOR_PERMANENT_AUTH_HANDLE < TPM_RH_AUTH_00 \
59     || VENDOR_PERMANENT_AUTH_HANDLE > TPM_RH_AUTH_FF
60     # error VENDOR_PERMANENT_AUTH_ENABLED requires a valid definition for
VENDOR_PERMANENT_AUTH_HANDLE, see Part2
61     # endif
62 #else
63     # if defined(VENDOR_PERMANENT_AUTH_HANDLE)
64     # error VENDOR_PERMANENT_AUTH_HANDLE requires VENDOR_PERMANENT_AUTH_ENABLED to be
YES
65     # endif
66 #endif
67
68 // now check for inconsistent combinations of options
69 #if USE_KEY_CACHE_FILE && !USE_RSA_KEY_CACHE
70     # error cannot use USE_KEY_CACHE_FILE if not using USE_RSA_KEY_CACHE
71 #endif
72
73 #if !DEBUG
74     # if USE_KEY_CACHE_FILE || USE_RSA_KEY_CACHE || DRBG_DEBUG_PRINT \
75         || CERTIFYX509_DEBUG || USE_DEBUG_RNG
76     # error using insecure options not in DEBUG mode.
77     # endif
78 #endif
79
80 #if !SIMULATION
81     # if USE_KEY_CACHE_FILE
82     # error USE_KEY_CACHE_FILE requires SIMULATION
83     # endif
84     # if RSA_INSTRUMENT
85     # error RSA_INSTRUMENT requires SIMULATION
86     # endif
87     # if USE_DEBUG_RNG
88     # error USE_DEBUG_RNG requires SIMULATION
89     # endif
90 #endif
91
92 #endif // _VERIFY_CONFIGURATION_H

```

6.254 /tpm/include/public/prototypes/TpmFail_fp.h

```

1  /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Apr 2, 2019 Time: 03:18:00PM
4  */
5
6  #ifndef _TPM_FAIL_FP_H_
7  #define _TPM_FAIL_FP_H_
8
9  /*** SetForceFailureMode()
10 // This function is called by the simulator to enable failure mode testing.
11 #if SIMULATION
12 LIB_EXPORT void SetForceFailureMode(void);
13 #endif
14
15 /*** TpmFail()
16 // This function is called by TPM.lib when a failure occurs. It will set up the
17 // failure values to be returned on TPM2_GetTestResult().
18 NORETURN void TpmFail(
19 #if FAIL_TRACE
20     const char* function,
21     int line,
22 #else
23     uint64_t locationCode,
24 #endif
25     int failureCode);

```

```
26
27  /*** TpmFailureMode(
28  // This function is called by the interface code when the platform is in failure
29  // mode.
30  void TpmFailureMode(uint32_t      inRequestSize,    // IN: command buffer size
31                      unsigned char* inRequest,      // IN: command buffer
32                      uint32_t*     outResponseSize,  // OUT: response buffer size
33                      unsigned char** outResponse     // OUT: response buffer
34  );
35
36  /*** UnmarshalFail()
37  // This is a stub that is used to catch an attempt to unmarshal an entry
38  // that is not defined. Don't ever expect this to be called but...
39  void UnmarshalFail(void* type, BYTE** buffer, INT32* size);
40
41  #endif // _TPM_FAIL_FP_H_
42
```

7 TPM Reference Implementation Source Files

7.1 /tpm/src/command/Asymmetric/ECC_Decrypt.c

```

1  #include "Tpm.h"
2  #include "ECC_Decrypt_fp.h"
3  #include "CryptEccCrypt_fp.h"
4
5  #if CC_ECC_Decrypt // Conditional expansion of this file
6
7  // Return Type: TPM_RC
8  //     TPM_RC_ATTRIBUTES      key referenced by 'keyHandle' is restricted
9  //     TPM_RC_KEY             keyHandle does not reference an ECC key
10 //     TPM_RC_NO_RESULT       internal error in big number processing
11 //     TPM_RC_SCHEME          bad scheme
12 //     TPM_RC_VALUE           C3 did not match hash of recovered data
13 TPM_RC
14 TPM2_ECC_Decrypt(ECC_Decrypt_In* in, // IN: input parameter list
15                 ECC_Decrypt_Out* out // OUT: output parameter list
16 )
17 {
18     OBJECT* key = HandleToObject(in->keyHandle);
19     // Parameter validation
20     // Must be the correct type of key with correct attributes
21     if(key->publicArea.type != TPM_ALG_ECC)
22         return TPM_RC_KEY + RC_ECC_Decrypt_keyHandle;
23     if(IS_ATTRIBUTE(key->publicArea.objectAttributes, TPMA_OBJECT, restricted)
24        || !IS_ATTRIBUTE(key->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
25         return TPM_RCS_ATTRIBUTES + RC_ECC_Decrypt_keyHandle;
26     // Have to have a scheme selected
27     if(!CryptEccSelectScheme(key, &in->inScheme))
28         return TPM_RCS_SCHEME + RC_ECC_Decrypt_inScheme;
29     // Command Output
30     return CryptEccDecrypt(
31         key, &in->inScheme, &out->plainText, &in->C1.point, &in->C2, &in->C3);
32 }
33
34 #endif // CC_ECC_Decrypt

```

7.2 /tpm/src/command/Asymmetric/ECC_Encrypt.c

```

1  #include "Tpm.h"
2  #include "ECC_Encrypt_fp.h"
3
4  #if CC_ECC_Encrypt // Conditional expansion of this file
5
6  // Return Type: TPM_RC
7  //     TPM_RC_ATTRIBUTES      key referenced by 'keyHandle' is restricted
8  //     TPM_RC_KEY             keyHandle does not reference an ECC key
9  //     TPM_RCS_SCHEME         bad scheme
10 TPM_RC
11 TPM2_ECC_Encrypt(ECC_Encrypt_In* in, // IN: input parameter list
12                 ECC_Encrypt_Out* out // OUT: output parameter list
13 )
14 {
15     OBJECT* pubKey = HandleToObject(in->keyHandle);
16     // Parameter validation
17     if(pubKey->publicArea.type != TPM_ALG_ECC)
18         return TPM_RC_KEY + RC_ECC_Encrypt_keyHandle;
19     // Have to have a scheme selected
20     if(!CryptEccSelectScheme(pubKey, &in->inScheme))
21         return TPM_RCS_SCHEME + RC_ECC_Encrypt_inScheme;
22     // Command Output

```

```

23     return CryptEccEncrypt(
24         pubKey, &in->inScheme, &in->plainText, &out->C1.point, &out->C2, &out->C3);
25 }
26
27 #endif // CC_ECC_Encrypt

```

7.3 /tpm/src/command/Asymmetric/ECC_Parameters.c

```

1  #include "Tpm.h"
2  #include "ECC_Parameters_fp.h"
3
4  #if CC_ECC_Parameters // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command returns the parameters of an ECC curve identified by its TCG
8  // assigned curveID
9  */
10 // Return Type: TPM_RC
11 // TPM_RC_VALUE Unsupported ECC curve ID
12 TPM_RC
13 TPM2_ECC_Parameters(ECC_Parameters_In* in, // IN: input parameter list
14                    ECC_Parameters_Out* out // OUT: output parameter list
15 )
16 {
17     // Command Output
18
19     // Get ECC curve parameters
20     if(CryptEccGetParameters(in->curveID, &out->parameters))
21         return TPM_RC_SUCCESS;
22     else
23         return TPM_RCS_VALUE + RC_ECC_Parameters_curveID;
24 }
25
26 #endif // CC_ECC_Parameters

```

7.4 /tpm/src/command/Asymmetric/ECDH_KeyGen.c

```

1  #include "Tpm.h"
2  #include "ECDH_KeyGen_fp.h"
3
4  #if CC_ECDH_KeyGen // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command uses the TPM to generate an ephemeral public key and the product
8  // of the ephemeral private key and the public portion of an ECC key.
9  */
10 // Return Type: TPM_RC
11 // TPM_RC_KEY 'keyHandle' does not reference an ECC key
12 TPM_RC
13 TPM2_ECDH_KeyGen(ECDH_KeyGen_In* in, // IN: input parameter list
14                 ECDH_KeyGen_Out* out // OUT: output parameter list
15 )
16 {
17     OBJECT* eccKey;
18     TPM2B_ECC_PARAMETER sensitive;
19     TPM_RC result;
20
21     // Input Validation
22
23     eccKey = HandleToObject(in->keyHandle);
24
25     // Referenced key must be an ECC key
26     if(eccKey->publicArea.type != TPM_ALG_ECC)
27         return TPM_RCS_KEY + RC_ECDH_KeyGen_keyHandle;

```

```

28
29 // Command Output
30 do
31 {
32     TPMT_PUBLIC* keyPublic = &eccKey->publicArea;
33     // Create ephemeral ECC key
34     result = CryptEccNewKeyPair(&out->pubPoint.point,
35                               &sensitive,
36                               keyPublic->parameters.eccDetail.curveID);
37     if(result == TPM_RC_SUCCESS)
38     {
39         // Compute Z
40         result = CryptEccPointMultiply(&out->zPoint.point,
41                                       keyPublic->parameters.eccDetail.curveID,
42                                       &keyPublic->unique.ecc,
43                                       &sensitive,
44                                       NULL,
45                                       NULL);
46         // The point in the key is not on the curve. Indicate
47         // that the key is bad.
48         if(result == TPM_RC_ECC_POINT)
49             return TPM_RCS_KEY + RC_ECDH_KeyGen_keyHandle;
50         // The other possible error from CryptEccPointMultiply is
51         // TPM_RC_NO_RESULT indicating that the multiplication resulted in
52         // the point at infinity, so get a new random key and start over
53         // BTW, this never happens.
54     }
55 } while(result == TPM_RC_NO_RESULT);
56 return result;
57 }
58
59 #endif // CC_ECDH_KeyGen

```

7.5 /tpm/src/command/Asymmetric/ECDH_ZGen.c

```

1  #include "Tpm.h"
2  #include "ECDH_ZGen_fp.h"
3
4  #if CC_ECDH_ZGen // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command uses the TPM to recover the Z value from a public point
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES          key referenced by 'keyA' is restricted or
11 //                                not a decrypt key
12 //     TPM_RC_KEY                 key referenced by 'keyA' is not an ECC key
13 //     TPM_RC_NO_RESULT           multiplying 'inPoint' resulted in a
14 //                                point at infinity
15 //     TPM_RC_SCHEME              the scheme of the key referenced by 'keyA'
16 //                                is not TPM_ALG_NULL, TPM_ALG_ECDH,
17 TPM_RC
18 TPM2_ECDH_ZGen(ECDH_ZGen_In* in, // IN: input parameter list
19               ECDH_ZGen_Out* out // OUT: output parameter list
20 )
21 {
22     TPM_RC result;
23     OBJECT* eccKey;
24
25     // Input Validation
26     eccKey = HandleToObject(in->keyHandle);
27
28     // Selected key must be a non-restricted, decrypt ECC key
29     if(eccKey->publicArea.type != TPM_ALG_ECC)
30         return TPM_RCS_KEY + RC_ECDH_ZGen_keyHandle;

```



```

31 // Selected key needs to be unrestricted with the 'decrypt' attribute
32 if(IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, restricted)
33    || !IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
34     return TPM_RCS_ATTRIBUTES + RC_ECDH_ZGen_keyHandle;
35 // Make sure the scheme allows this use
36 if(eccKey->publicArea.parameters.eccDetail.scheme.scheme != TPM_ALG_ECDH
37    && eccKey->publicArea.parameters.eccDetail.scheme.scheme != TPM_ALG_NULL)
38     return TPM_RCS_SCHEME + RC_ECDH_ZGen_keyHandle;
39 // Command Output
40 // Compute Z. TPM_RC_ECC_POINT or TPM_RC_NO_RESULT may be returned here.
41 result = CryptEccPointMultiply(&out->outPoint.point,
42                                eccKey->publicArea.parameters.eccDetail.curveID,
43                                &in->inPoint.point,
44                                &eccKey->sensitive.sensitive.ecc,
45                                NULL,
46                                NULL);
47 if(result != TPM_RC_SUCCESS)
48     return RcSafeAddToResult(result, RC_ECDH_ZGen_inPoint);
49 return result;
50 }
51
52 #endif // CC_ECDH_ZGen

```

7.6 /tpm/src/command/Asymmetric/EC_Ephemeral.c

```

1  #include "Tpm.h"
2  #include "EC_Ephemeral_fp.h"
3
4  #if CC_EC_Ephemeral // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command creates an ephemeral key using the commit mechanism
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_NO_RESULT the TPM is not able to generate an 'r' value
11 TPM_RC
12 TPM2_EC_Ephemeral(EC_Ephemeral_In* in, // IN: input parameter list
13                  EC_Ephemeral_Out* out // OUT: output parameter list
14 )
15 {
16     TPM2B_ECC_PARAMETER r;
17     TPM_RC result;
18     //
19     do
20     {
21         // Get the random value that will be used in the point multiplications
22         // Note: this does not commit the count.
23         if(!CryptGenerateR(&r, NULL, in->curveID, NULL))
24             return TPM_RC_NO_RESULT;
25         // do a point multiply
26         result =
27             CryptEccPointMultiply(&out->Q.point, in->curveID, NULL, &r, NULL, NULL);
28         // commit the count value if either the r value results in the point at
29         // infinity or if the value is good. The commit on the r value for infinity
30         // is so that the r value will be skipped.
31         if((result == TPM_RC_SUCCESS) || (result == TPM_RC_NO_RESULT))
32             out->counter = CryptCommit();
33     } while(result == TPM_RC_NO_RESULT);
34
35     return TPM_RC_SUCCESS;
36 }
37
38 #endif // CC_EC_Ephemeral

```

7.7 /tpm/src/command/Asymmetric/RSA_Decrypt.c

```

1  #include "Tpm.h"
2  #include "RSA_Decrypt_fp.h"
3
4  #if CC_RSA_Decrypt // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // decrypts the provided data block and removes the padding if applicable
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES      'decrypt' is not SET or if 'restricted' is SET in
11 //                             the key referenced by 'keyHandle'
12 //     TPM_RC_BINDING         The public and private parts of the key are not
13 //                             properly bound
14 //     TPM_RC_KEY             'keyHandle' does not reference an unrestricted
15 //                             decrypt key
16 //     TPM_RC_SCHEME          incorrect input scheme, or the chosen
17 //                             'scheme' is not a valid RSA decrypt scheme
18 //     TPM_RC_SIZE            'cipherText' is not the size of the modulus
19 //                             of key referenced by 'keyHandle'
20 //     TPM_RC_VALUE           'label' is not a null terminated string or the value
21 //                             of 'cipherText' is greater than the modulus of
22 //                             'keyHandle' or the encoding of the data is not
23 //                             valid
24
25 TPM_RC
26 TPM2_RSA_Decrypt(RSA_Decrypt_In* in, // IN: input parameter list
27                 RSA_Decrypt_Out* out // OUT: output parameter list
28 )
29 {
30     TPM_RC      result;
31     OBJECT*     rsaKey;
32     TPMT_RSA_DECRYPT* scheme;
33
34     // Input Validation
35
36     rsaKey = HandleToObject(in->keyHandle);
37
38     // The selected key must be an RSA key
39     if(rsaKey->publicArea.type != TPM_ALG_RSA)
40         return TPM_RCS_KEY + RC_RSA_Decrypt_keyHandle;
41
42     // The selected key must be an unrestricted decryption key
43     if(!IS_ATTRIBUTE(rsaKey->publicArea.objectAttributes, TPMA_OBJECT, restricted)
44        || !IS_ATTRIBUTE(rsaKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
45         return TPM_RCS_ATTRIBUTES + RC_RSA_Decrypt_keyHandle;
46
47     // NOTE: Proper operation of this command requires that the sensitive area
48     // of the key is loaded. This is assured because authorization is required
49     // to use the sensitive area of the key. In order to check the authorization,
50     // the sensitive area has to be loaded, even if authorization is with policy.
51
52     // If label is present, make sure that it is a NULL-terminated string
53     if(!IsLabelProperlyFormatted(&in->label.b))
54         return TPM_RCS_VALUE + RC_RSA_Decrypt_label;
55     // Command Output
56     // Select a scheme for decrypt.
57     scheme = CryptRsaSelectScheme(in->keyHandle, &in->inScheme);
58     if(scheme == NULL)
59         return TPM_RCS_SCHEME + RC_RSA_Decrypt_inScheme;
60
61     // Decryption. TPM_RC_VALUE, TPM_RC_SIZE, and TPM_RC_KEY error may be
62     // returned by CryptRsaDecrypt.
63     // NOTE: CryptRsaDecrypt can also return TPM_RC_ATTRIBUTES or TPM_RC_BINDING
64     // when the key is not a decryption key but that was checked above.

```

```

65     out->message.t.size = sizeof(out->message.t.buffer);
66     result              = CryptRsaDecrypt(
67         &out->message.b, &in->cipherText.b, rsaKey, scheme, &in->label.b);
68     return result;
69 }
70
71 #endif // CC_RSA_Decrypt

```

7.8 /tpm/src/command/Asymmetric/RSA_Encrypt.c

```

1  #include "Tpm.h"
2  #include "RSA_Encrypt_fp.h"
3
4  #if CC_RSA_Encrypt // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command performs the padding and encryption of a data block
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES      'decrypt' attribute is not SET in key referenced
11 //                             by 'keyHandle'
12 //     TPM_RC_KEY              'keyHandle' does not reference an RSA key
13 //     TPM_RC_SCHEME            incorrect input scheme, or the chosen
14 //                             scheme is not a valid RSA decrypt scheme
15 //     TPM_RC_VALUE            the numeric value of 'message' is greater than
16 //                             the public modulus of the key referenced by
17 //                             'keyHandle', or 'label' is not a null-terminated
18 //                             string
19 TPM_RC
20 TPM2_RSA_Encrypt(RSA_Encrypt_In* in, // IN: input parameter list
21                 RSA_Encrypt_Out* out // OUT: output parameter list
22 )
23 {
24     TPM_RC      result;
25     OBJECT*     rsaKey;
26     TPMT_RSA_DECRYPT* scheme;
27     // Input Validation
28     rsaKey = HandleToObject(in->keyHandle);
29
30     // selected key must be an RSA key
31     if(rsaKey->publicArea.type != TPM_ALG_RSA)
32         return TPM_RCS_KEY + RC_RSA_Encrypt_keyHandle;
33     // selected key must have the decryption attribute
34     if(!IS_ATTRIBUTE(rsaKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
35         return TPM_RCS_ATTRIBUTES + RC_RSA_Encrypt_keyHandle;
36
37     // Is there a label?
38     if(!IsLabelProperlyFormatted(&in->label.b))
39         return TPM_RCS_VALUE + RC_RSA_Encrypt_label;
40     // Command Output
41     // Select a scheme for encryption
42     scheme = CryptRsaSelectScheme(in->keyHandle, &in->inScheme);
43     if(scheme == NULL)
44         return TPM_RCS_SCHEME + RC_RSA_Encrypt_inScheme;
45
46     // Encryption. TPM_RC_VALUE, or TPM_RC_SCHEME errors may be returned by
47     // CryptEncryptRSA.
48     out->outData.t.size = sizeof(out->outData.t.buffer);
49
50     result              = CryptRsaEncrypt(
51         &out->outData, &in->message.b, rsaKey, scheme, &in->label.b, NULL);
52     return result;
53 }
54
55 #endif // CC_RSA_Encrypt

```

7.9 /tpm/src/command/Asymmetric/ZGen_2Phase.c

```

1  #include "Tpm.h"
2  #include "ZGen_2Phase_fp.h"
3
4  #if CC_ZGen_2Phase // Conditional expansion of this file
5
6  // This command uses the TPM to recover one or two Z values in a two phase key
7  // exchange protocol
8  // Return Type: TPM_RC
9  //     TPM_RC_ATTRIBUTES          key referenced by 'keyA' is restricted or
10 //                                not a decrypt key
11 //     TPM_RC_ECC_POINT           'inQsB' or 'inQeB' is not on the curve of
12 //                                the key reference by 'keyA'
13 //     TPM_RC_KEY                 key referenced by 'keyA' is not an ECC key
14 //     TPM_RC_SCHEME              the scheme of the key referenced by 'keyA'
15 //                                is not TPM_ALG_NULL, TPM_ALG_ECDH,
16 //                                TPM_ALG_ECMQV or TPM_ALG_SM2
17 TPM_RC
18 TPM2_ZGen_2Phase(ZGen_2Phase_In* in, // IN: input parameter list
19                 ZGen_2Phase_Out* out // OUT: output parameter list
20 )
21 {
22     TPM_RC          result;
23     OBJECT*         eccKey;
24     TPM2B_ECC_PARAMETER r;
25     TPM_ALG_ID      scheme;
26
27     // Input Validation
28
29     eccKey = HandleToObject(in->keyA);
30
31     // keyA must be an ECC key
32     if(eccKey->publicArea.type != TPM_ALG_ECC)
33         return TPM_RCS_KEY + RC_ZGen_2Phase_keyA;
34
35     // keyA must not be restricted and must be a decrypt key
36     if(IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, restricted)
37        || !IS_ATTRIBUTE(eccKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
38         return TPM_RCS_ATTRIBUTES + RC_ZGen_2Phase_keyA;
39
40     // if the scheme of keyA is TPM_ALG_NULL, then use the input scheme; otherwise
41     // the input scheme must be the same as the scheme of keyA
42     scheme = eccKey->publicArea.parameters.asymDetail.scheme.scheme;
43     if(scheme != TPM_ALG_NULL)
44     {
45         if(scheme != in->inScheme)
46             return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
47     }
48     else
49     {
50         scheme = in->inScheme;
51         if(scheme == TPM_ALG_NULL)
52             return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
53     }
54
55     // Input points must be on the curve of keyA
56     if(!CryptEccIsPointOnCurve(eccKey->publicArea.parameters.eccDetail.curveID,
57                                &in->inQsB.point))
58         return TPM_RCS_ECC_POINT + RC_ZGen_2Phase_inQsB;
59
60     if(!CryptEccIsPointOnCurve(eccKey->publicArea.parameters.eccDetail.curveID,
61                                &in->inQeB.point))
62         return TPM_RCS_ECC_POINT + RC_ZGen_2Phase_inQeB;
63
64     if(!CryptGenerator(
65         &r, &in->counter, eccKey->publicArea.parameters.eccDetail.curveID, NULL))
66         return TPM_RCS_VALUE + RC_ZGen_2Phase_counter;

```

```

65
66 // Command Output
67
68 result =
69     CryptEcc2PhaseKeyExchange(&out->outZ1.point,
70                               &out->outZ2.point,
71                               eccKey->publicArea.parameters.eccDetail.curveID,
72                               scheme,
73                               &eccKey->sensitive.sensitive.ecc,
74                               &r,
75                               &in->inQsB.point,
76                               &in->inQeB.point);
77
78 if(result == TPM_RC_SCHEME)
79     return TPM_RCS_SCHEME + RC_ZGen_2Phase_inScheme;
80
81 if(result == TPM_RC_SUCCESS)
82     CryptEndCommit(in->counter);
83
84 return result;
85 }
86 #endif // CC_ZGen_2Phase

```

7.10 /tpm/src/command/AttachedComponent/AC_GetCapability.c

```

1  #include "Tpm.h"
2  #include "AC_GetCapability_fp.h"
3  #include "AC_spt_fp.h"
4
5  #if CC_AC_GetCapability // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // This command returns various information regarding Attached Components
9  */
10 TPM_RC
11 TPM2_AC_GetCapability(AC_GetCapability_In* in, // IN: input parameter list
12                      AC_GetCapability_Out* out // OUT: output parameter list
13 )
14 {
15     // Command Output
16     out->moreData =
17         AcCapabilitiesGet(in->ac, in->capability, in->count, &out->capabilitiesData);
18
19     return TPM_RC_SUCCESS;
20 }
21
22 #endif // CC_AC_GetCapability

```

7.11 /tpm/src/command/AttachedComponent/AC_Send.c

```

1  #include "Tpm.h"
2  #include "AC_Send_fp.h"
3  #include "AC_spt_fp.h"
4
5  #if CC_AC_Send // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // Duplicate a loaded object
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_ATTRIBUTES key to duplicate has 'fixedParent' SET
12 //     TPM_RC_HASH       for an RSA key, the nameAlg digest size for the
13 //                       newParent is not compatible with the key size
14 //     TPM_RC_HIERARCHY  'encryptedDuplication' is SET and 'newParentHandle'
15 //                       specifies Null Hierarchy

```

```

16 //      TPM_RC_KEY          'newParentHandle' references invalid ECC key (public
17 //                          point not on the curve)
18 //      TPM_RC_SIZE         input encryption key size does not match the
19 //                          size specified in symmetric algorithm
20 //      TPM_RC_SYMMETRIC    'encryptedDuplication' is SET but no symmetric
21 //                          algorithm is provided
22 //      TPM_RC_TYPE         'newParentHandle' is neither a storage key nor
23 //                          TPM_RH_NULL; or the object has a NULL nameAlg
24 //      TPM_RC_VALUE        for an RSA newParent, the sizes of the digest and
25 //                          the encryption key are too large to be OAEP encoded
26 TPM_RC
27 TPM2_AC_Send(AC_Send_In* in, // IN: input parameter list
28              AC_Send_Out* out // OUT: output parameter list
29 )
30 {
31     NV_REF locator;
32     TPM_HANDLE nvAlias = ((in->ac - AC_FIRST) + NV_AC_FIRST);
33     NV_INDEX* nvIndex = NvGetIndexInfo(nvAlias, &locator);
34     OBJECT* object = HandleToObject(in->sendObject);
35     TPM_RC result;
36     // Input validation
37     // If there is an NV alias, then the index must allow the authorization provided
38     if(nvIndex != NULL)
39     {
40         // Common access checks, NvWriteAccessCheck() may return
41         // TPM_RC_NV_AUTHORIZATION or TPM_RC_NV_LOCKED
42         result = NvWriteAccessChecks(
43             in->authHandle, nvAlias, nvIndex->publicArea.attributes);
44         if(result != TPM_RC_SUCCESS)
45             return result;
46     }
47     // If 'ac' did not have an alias then the authorization had to be with either
48     // platform or owner authorization. The type of TPMT_RH_NV_AUTH only allows
49     // owner or platform or an NV index. If it was a valid index, it would have had
50     // an alias and be processed above, so only success here is if this is a
51     // permanent handle.
52     else if(HandleGetType(in->authHandle) != TPM_HT_PERMANENT)
53         return TPM_RCS_HANDLE + RC_AC_Send_authHandle;
54     // Make sure that the object to be duplicated has the right attributes
55     if(IS_ATTRIBUTE(
56         object->publicArea.objectAttributes, TPMA_OBJECT, encryptedDuplication)
57         || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, fixedParent)
58         || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, fixedTPM))
59         return TPM_RCS_ATTRIBUTES + RC_AC_Send_sendObject;
60     // Command output
61     // Do the implementation dependent send
62     return AcSendObject(in->ac, object, &out->acDataOut);
63 }
64
65 #endif // TPM_CC_AC_Send

```

7.12 /tpm/src/command/AttachedComponent/AC_spt.c

```

1 /** Introduction
2  * This code in this clause is provided for testing of the TPM's command interface.
3  * The implementation of Attached Components is not expected to be as shown in this
4  * code.
5
6  /** Includes
7  #include "Tpm.h"
8  #include "AC_spt_fp.h"
9
10 // This is the simulated AC data. This should be present in an actual implementation.
11 #if 1
12

```



```

13 typedef struct
14 {
15     TPMI_RH_AC          ac;
16     TPML_AC_CAPABILITIES* acData;
17 } acCapabilities;
18
19
20 TPML_AC_CAPABILITIES acData0001 = {1, {{TPM_AT_PV1, 0x01234567}}};
21
22 acCapabilities      ac[1]      = {{0x0001, &acData0001}};
23
24 # define NUM_AC (sizeof(ac) / sizeof(acCapabilities))
25
26 #endif // 1 The simulated AC data
27
28 /** Functions
29
30 **** AcToCapabilities()
31 // This function returns a pointer to a list of AC capabilities.
32 TPML_AC_CAPABILITIES* AcToCapabilities(TPMI_RH_AC component // IN: component
33 )
34 {
35     UINT32 index;
36     //
37     for(index = 0; index < NUM_AC; index++)
38     {
39         if(ac[index].ac == component)
40             return ac[index].acData;
41     }
42     return NULL;
43 }
44
45 **** AcIsAccessible()
46 // Function to determine if an AC handle references an actual AC
47 // Return Type: BOOL
48 BOOL AcIsAccessible(TPM_HANDLE acHandle)
49 {
50     // In this implementation, the AC exists if there are some capabilities to go
51     // with the handle
52     return AcToCapabilities(acHandle) != NULL;
53 }
54
55 **** AcCapabilitiesGet()
56 // This function returns a list of capabilities associated with an AC
57 // Return Type: TPMI_YES_NO
58 //     YES      if there are more handles available
59 //     NO       all the available handles has been returned
60 TPMI_YES_NO
61 AcCapabilitiesGet(TPMI_RH_AC          component, // IN: the component
62                  TPM_AT              type,      // IN: start capability type
63                  UINT32              count,     // IN: requested number
64                  TPML_AC_CAPABILITIES* capabilityList // OUT: list of handle
65 )
66 {
67     TPMI_YES_NO more = NO;
68     UINT32      i;
69     // Get the list of capabilities and their values associated with the AC
70     TPML_AC_CAPABILITIES* capabilities;
71
72     pAssert(HandleGetType(component) == TPM_HT_AC);
73     capabilities = AcToCapabilities(component);
74
75     // Initialize output handle list
76     capabilityList->count = 0;
77     if(count > MAX_AC_CAPABILITIES)
78         count = MAX_AC_CAPABILITIES;

```

```

79
80     if(capabilities != NULL)
81     {
82         // Find the first capability less than or equal to type
83         for(i = 0; i < capabilities->count; i++)
84         {
85             if(capabilities->acCapabilities[i].tag >= type)
86             {
87                 // copy the capabilities until we run out or fill the list
88                 for(; (capabilityList->count < count) && (i < capabilities->count);
89                     i++)
90                 {
91                     capabilityList->acCapabilities[capabilityList->count] =
92                         capabilities->acCapabilities[i];
93                     capabilityList->count++;
94                 }
95                 more = i < capabilities->count;
96             }
97         }
98     }
99     return more;
100 }
101
102 /*** AcSendObject()
103 // Stub to handle sending of an AC object
104 // Return Type: TPM_RC
105 TPM_RC
106 AcSendObject(TPM_HANDLE      acHandle, // IN: Handle of AC receiving object
107              OBJECT*         object,   // IN: object structure to send
108              TPMS_AC_OUTPUT*  acDataOut // OUT: results of operation
109 )
110 {
111     NOT_REFERENCED(object);
112     NOT_REFERENCED(acHandle);
113     acDataOut->tag = TPM_AT_ERROR; // indicate that the response contains an
114                                   // error code
115     acDataOut->data = TPM_AE_NONE; // but there is no error.
116
117     return TPM_RC_SUCCESS;
118 }

```

7.13 /tpm/src/command/AttachedComponent/Policy_AC_SendSelect.c

```

1  #include "Tpm.h"
2  #include "Policy_AC_SendSelect_fp.h"
3
4  #if CC_Policy_AC_SendSelect // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // allows qualification of attached component and object to be sent.
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_COMMAND_CODE 'commandCode' of 'policySession' is not empty
11 //     TPM_RC_CPHASH       'cpHash' of 'policySession' is not empty
12 TPM_RC
13 TPM2_Policy_AC_SendSelect(Policy_AC_SendSelect_In* in // IN: input parameter list
14 )
15 {
16     SESSION* session;
17     HASH_STATE hashState;
18     TPM_CC commandCode = TPM_CC_Policy_AC_SendSelect;
19
20     // Input Validation
21
22     // Get pointer to the session structure

```

```

23     session = SessionGet(in->policySession);
24
25     // cpHash in session context must be empty
26     if(session->u1.cpHash.t.size != 0)
27         return TPM_RC_CPHASH;
28     // commandCode in session context must be empty
29     if(session->commandCode != 0)
30         return TPM_RC_COMMAND_CODE;
31     // Internal Data Update
32     // Update name hash
33     session->u1.cpHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
34
35     // add objectName
36     CryptDigestUpdate2B(&hashState, &in->objectName.b);
37
38     // add authHandleName
39     CryptDigestUpdate2B(&hashState, &in->authHandleName.b);
40
41     // add ac name
42     CryptDigestUpdate2B(&hashState, &in->acName.b);
43
44     // complete hash
45     CryptHashEnd2B(&hashState, &session->u1.cpHash.b);
46
47     // update policy hash
48     // Old policyDigest size should be the same as the new policyDigest size since
49     // they are using the same hash algorithm
50     session->u2.policyDigest.t.size =
51         CryptHashStart(&hashState, session->authHashAlg);
52     // add old policy
53     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
54
55     // add command code
56     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
57
58     // add objectName
59     if(in->includeObject == YES)
60         CryptDigestUpdate2B(&hashState, &in->objectName.b);
61
62     // add authHandleName
63     CryptDigestUpdate2B(&hashState, &in->authHandleName.b);
64
65     // add acName
66     CryptDigestUpdate2B(&hashState, &in->acName.b);
67
68     // add includeObject
69     CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->includeObject);
70
71     // complete digest
72     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
73
74     // set commandCode in session context
75     session->commandCode = TPM_CC_AC_Send;
76
77     return TPM_RC_SUCCESS;
78 }
79
80 #endif // CC_Policy_AC_SendSelect

```

7.14 /tpm/src/command/Attestation/Attest_spt.c

```

1  /** Includes
2  #include "Tpm.h"
3  #include "Attest_spt_fp.h"
4  #include "Marshal.h"

```

```

5
6 /** Functions
7
8 /** FillInAttestInfo()
9 // Fill in common fields of TPMS_ATTEST structure.
10 void FillInAttestInfo(
11     TPMI_DH_OBJECT    signHandle, // IN: handle of signing object
12     TPMT_SIG_SCHEME*  scheme,     // IN/OUT: scheme to be used for signing
13     TPM2B_DATA*       data,       // IN: qualifying data
14     TPMS_ATTEST*      attest      // OUT: attest structure
15 )
16 {
17     OBJECT* signObject = HandleToObject(signHandle);
18
19     // Magic number
20     attest->magic = TPM_GENERATED_VALUE;
21
22     if(signObject == NULL)
23     {
24         // The name for a null handle is TPM_RH_NULL
25         // This is defined because UINT32_TO_BYTE_ARRAY does a cast. If the
26         // size of the cast is smaller than a constant, the compiler warns
27         // about the truncation of a constant value.
28         TPM_HANDLE nullHandle = TPM_RH_NULL;
29         attest->qualifiedSigner.t.size = sizeof(TPM_HANDLE);
30         UINT32_TO_BYTE_ARRAY(nullHandle, attest->qualifiedSigner.t.name);
31     }
32     else
33     {
34         // Certifying object qualified name
35         // if the scheme is anonymous, this is an empty buffer
36         if(CryptIsSchemeAnonymous(scheme->scheme))
37             attest->qualifiedSigner.t.size = 0;
38         else
39             attest->qualifiedSigner = signObject->qualifiedName;
40     }
41     // current clock in plain text
42     TimeFillInfo(&attest->clockInfo);
43
44     // Firmware version in plain text
45     attest->firmwareVersion = ((UINT64)gp.firmwareV1 << (sizeof(UINT32) * 8));
46     attest->firmwareVersion += gp.firmwareV2;
47
48     // Check the hierarchy of sign object. For NULL sign handle, the hierarchy
49     // will be TPM_RH_NULL
50     if((signObject == NULL)
51        || (!signObject->attributes.epsHierarchy
52            && !signObject->attributes.ppsHierarchy))
53     {
54         // For signing key that is not in platform or endorsement hierarchy,
55         // obfuscate the reset, restart and firmware version information
56         UINT64 obfuscation[2];
57         CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG,
58                 &gp.shProof.b,
59                 OBFUSCATE_STRING,
60                 &attest->qualifiedSigner.b,
61                 NULL,
62                 128,
63                 (BYTE*)&obfuscation[0],
64                 NULL,
65                 FALSE);
66         // Obfuscate data
67         attest->firmwareVersion += obfuscation[0];
68         attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
69         attest->clockInfo.restartCount += (UINT32)obfuscation[1];
70     }

```

```

71 // External data
72 if(CryptIsSchemeAnonymous(scheme->scheme))
73     attest->extraData.t.size = 0;
74 else
75 {
76     // If we move the data to the attestation structure, then it is not
77     // used in the signing operation except as part of the signed data
78     attest->extraData = *data;
79     data->t.size      = 0;
80 }
81 }
82
83 /***SignAttestInfo()
84 // Sign a TPMS_ATTEST structure. If signHandle is TPM_RH_NULL, a null signature
85 // is returned.
86 //
87 // Return Type: TPM_RC
88 //     TPM_RC_ATTRIBUTES 'signHandle' references not a signing key
89 //     TPM_RC_SCHEME      'scheme' is not compatible with 'signHandle' type
90 //     TPM_RC_VALUE       digest generated for the given 'scheme' is greater than
91 //                         the modulus of 'signHandle' (for an RSA key);
92 //                         invalid commit status or failed to generate "r" value
93 //                         (for an ECC key)
94 TPM_RC
95 SignAttestInfo(OBJECT*      signKey,          // IN: sign object
96                TPMT_SIG_SCHEME* scheme,       // IN: sign scheme
97                TPMS_ATTEST*  certifyInfo,     // IN: the data to be signed
98                TPM2B_DATA*   qualifyingData,  // IN: extra data for the signing
99                                     // process
100                TPM2B_ATTEST* attest,         // OUT: marshaled attest blob to be
101                                     // signed
102                TPMT_SIGNATURE* signature     // OUT: signature
103 )
104 {
105     BYTE*      buffer;
106     HASH_STATE hashState;
107     TPM2B_DIGEST digest;
108     TPM_RC      result;
109
110     // Marshal TPMS_ATTEST structure for hash
111     buffer = attest->t.attestationData;
112     attest->t.size = TPMS_ATTEST_Marshal(certifyInfo, &buffer, NULL);
113
114     if(signKey == NULL)
115     {
116         signature->sigAlg = TPM_ALG_NULL;
117         result            = TPM_RC_SUCCESS;
118     }
119     else
120     {
121         TPMI_ALG_HASH hashAlg;
122         // Compute hash
123         hashAlg = scheme->details.any.hashAlg;
124         // need to set the receive buffer to get something put in it
125         digest.t.size = sizeof(digest.t.buffer);
126         digest.t.size = CryptHashBlock(hashAlg,
127                                     attest->t.size,
128                                     attest->t.attestationData,
129                                     digest.t.size,
130                                     digest.t.buffer);
131         // If there is qualifying data, need to rehash the data
132         // hash(qualifyingData || hash(attestationData))
133         if(qualifyingData->t.size != 0)
134         {
135             CryptHashStart(&hashState, hashAlg);
136             CryptDigestUpdate2B(&hashState, &qualifyingData->b);

```

```

137         CryptDigestUpdate2B(&hashState, &digest.b);
138         CryptHashEnd2B(&hashState, &digest.b);
139     }
140     // Sign the hash. A TPM_RC_VALUE, TPM_RC_SCHEME, or
141     // TPM_RC_ATTRIBUTES error may be returned at this point
142     result = CryptSign(signKey, scheme, &digest, signature);
143
144     // Since the clock is used in an attestation, the state in NV is no longer
145     // "orderly" with respect to the data in RAM if the signature is valid
146     if(result == TPM_RC_SUCCESS)
147     {
148         // Command uses the clock so need to clear the orderly state if it is
149         // set.
150         result = NvClearOrderly();
151     }
152 }
153 return result;
154 }
155
156 /*** IsSigningObject()
157 // Checks to see if the object is OK for signing. This is here rather than in
158 // Object_spt.c because all the attestation commands use this file but not
159 // Object_spt.c.
160 // Return Type: BOOL
161 //     TRUE(1)      object may sign
162 //     FALSE(0)     object may not sign
163 BOOL IsSigningObject(OBJECT* object // IN:
164 )
165 {
166     return ((object == NULL)
167         || ((IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, sign)
168             && object->publicArea.type != TPM_ALG_SYMCIPHER)));
169 }

```

7.15 /tpm/src/command/Attestation/Certify.c

```

1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "Certify_fp.h"
4
5  #if CC_Certify // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // prove an object with a specific Name is loaded in the TPM
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_KEY      key referenced by 'signHandle' is not a signing key
12 //     TPM_RC_SCHEME   'inScheme' is not compatible with 'signHandle'
13 //     TPM_RC_VALUE    digest generated for 'inScheme' is greater or has larger
14 //                     size than the modulus of 'signHandle', or the buffer for
15 //                     the result in 'signature' is too small (for an RSA key);
16 //                     invalid commit status (for an ECC key with a split scheme)
17 TPM_RC
18 TPM2_Certify(Certify_In* in, // IN: input parameter list
19             Certify_Out* out // OUT: output parameter list
20 )
21 {
22     TPMS_ATTEST certifyInfo;
23     OBJECT*      signObject      = HandleToObject(in->signHandle);
24     OBJECT*      certifiedObject = HandleToObject(in->objectHandle);
25     // Input validation
26     if(!IsSigningObject(signObject))
27         return TPM_RCS_KEY + RC_Certify_signHandle;
28     if(!CryptSelectSignScheme(signObject, &in->inScheme))
29         return TPM_RCS_SCHEME + RC_Certify_inScheme;

```



```

30
31 // Command Output
32 // Filling in attest information
33 // Common fields
34 FillInAttestInfo(
35     in->signHandle, &in->inScheme, &in->qualifyingData, &certifyInfo);
36
37 // Certify specific fields
38 certifyInfo.type = TPM_ST_ATTEST_CERTIFY;
39 // NOTE: the certified object is not allowed to be TPM_ALG_NULL so
40 // 'certifiedObject' will never be NULL
41 certifyInfo.attested.certify.name = certifiedObject->name;
42
43 // When using an anonymous signing scheme, need to set the qualified Name to the
44 // empty buffer to avoid correlation between keys
45 if(CryptIsSchemeAnonymous(in->inScheme.scheme))
46     certifyInfo.attested.certify.qualifiedName.t.size = 0;
47 else
48     certifyInfo.attested.certify.qualifiedName = certifiedObject->qualifiedName;
49
50 // Sign attestation structure. A NULL signature will be returned if
51 // signHandle is TPM_RH_NULL. A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
52 // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned
53 // by SignAttestInfo()
54 return SignAttestInfo(signObject,
55                       &in->inScheme,
56                       &certifyInfo,
57                       &in->qualifyingData,
58                       &out->certifyInfo,
59                       &out->signature);
60 }
61
62 #endif // CC_Certify

```

7.16 /tpm/src/command/Attestation/CertifyCreation.c

```

1 #include "Tpm.h"
2 #include "Attest_spt_fp.h"
3 #include "CertifyCreation_fp.h"
4
5 #if CC_CertifyCreation // Conditional expansion of this file
6
7 /*(See part 3 specification)
8 // Prove the association between an object and its creation data
9 */
10 // Return Type: TPM_RC
11 //     TPM_RC_KEY          key referenced by 'signHandle' is not a signing key
12 //     TPM_RC_SCHEME       'inScheme' is not compatible with 'signHandle'
13 //     TPM_RC_TICKET       'creationTicket' does not match 'objectHandle'
14 //     TPM_RC_VALUE        digest generated for 'inScheme' is greater or has larger
15 //                          size than the modulus of 'signHandle', or the buffer for
16 //                          the result in 'signature' is too small (for an RSA key);
17 //                          invalid commit status (for an ECC key with a split
18 //                          scheme).
19 TPM_RC
20 TPM2_CertifyCreation(CertifyCreation_In* in, // IN: input parameter list
21                     CertifyCreation_Out* out // OUT: output parameter list
22 )
23 {
24     TPM_RC          result = TPM_RC_SUCCESS;
25     TPMT_TK_CREATION ticket;
26     TPMS_ATTEST     certifyInfo;
27     OBJECT*         certified = HandleToObject(in->objectHandle);
28     OBJECT*         signObject = HandleToObject(in->signHandle);
29     // Input Validation

```

```

29     if(!IsSigningObject(signObject))
30         return TPM_RCS_KEY + RC_CertifyCreation_signHandle;
31     if(!CryptSelectSignScheme(signObject, &in->inScheme))
32         return TPM_RCS_SCHEME + RC_CertifyCreation_inScheme;
33
34     // CertifyCreation specific input validation
35     // Re-compute ticket
36     result = TicketComputeCreation(
37         in->creationTicket.hierarchy, &certified->name, &in->creationHash, &ticket);
38     if(result != TPM_RC_SUCCESS)
39         return result;
40
41     // Compare ticket
42     if(!MemoryEqual2B(&ticket.digest.b, &in->creationTicket.digest.b))
43         return TPM_RCS_TICKET + RC_CertifyCreation_creationTicket;
44
45     // Command Output
46     // Common fields
47     FillInAttestInfo(
48         in->signHandle, &in->inScheme, &in->qualifyingData, &certifyInfo);
49
50     // CertifyCreation specific fields
51     // Attestation type
52     certifyInfo.type = TPM_ST_ATTEST_CREATION;
53     certifyInfo.attested.creation.objectName = certified->name;
54
55     // Copy the creationHash
56     certifyInfo.attested.creation.creationHash = in->creationHash;
57
58     // Sign attestation structure. A NULL signature will be returned if
59     // signObject is TPM_RH_NULL. A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
60     // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned at
61     // this point
62     return SignAttestInfo(signObject,
63                           &in->inScheme,
64                           &certifyInfo,
65                           &in->qualifyingData,
66                           &out->certifyInfo,
67                           &out->signature);
68 }
69
70 #endif // CC_CertifyCreation

```

7.17 /tpm/src/command/Attestation/CertifyX509.c

```

1  #include "Tpm.h"
2  #include "CertifyX509_fp.h"
3  #include "X509.h"
4  #include "TpmASN1_fp.h"
5  #include "X509_spt_fp.h"
6  #include "Attest_spt_fp.h"
7  #if CERTIFYX509_DEBUG
8  // TODO_RENAME_INC_FOLDER:platform_interface refers to the TPM_CoreLib platform
9  // interface
10 #include <platform_interface/tpm_to_platform_interface.h>
11 #endif
12 #if CC_CertifyX509 // Conditional expansion of this file
13
14 /*(See part 3 specification)
15 // Certify using an X509-formatted certificate
16 */
17 // return type: TPM_RC
18 // TPM_RC_ATTRIBUTES the attributes of 'objectHandle' are not compatible
19 // with the KeyUsage or TPMA_OBJECT values in the

```

```

20 // extensions fields
21 // TPM_RC_BINDING the public and private portions of the key are not
22 // properly bound.
23 // TPM_RC_HASH the hash algorithm in the scheme is not supported
24 // TPM_RC_KEY 'signHandle' does not reference a signing key;
25 // TPM_RC_SCHEME the scheme is not compatible with sign key type,
26 // or input scheme is not compatible with default
27 // scheme, or the chosen scheme is not a valid
28 // sign scheme
29 // TPM_RC_VALUE most likely a problem with the format of
30 // 'partialCertificate'
31 TPM_RC
32 TPM2_CertifyX509(CertifyX509_In* in, // IN: input parameter list
33 CertifyX509_Out* out // OUT: output parameter list
34 )
35 {
36     TPM_RC result;
37     OBJECT* signKey = HandleToObject(in->signHandle);
38     OBJECT* object = HandleToObject(in->objectHandle);
39     HASH_STATE hash;
40     INT16 length; // length for a tagged element
41     ASN1UnmarshalContext ctx;
42     ASN1MarshalContext ctxOut;
43     // certTBS holds an array of pointers and lengths. Each entry references the
44     // corresponding value in a TBSCertificate structure. For example, the 1th
45     // element references the version number
46     stringRef certTBS[REF_COUNT] = {{0}};
47     # define ALLOWED_SEQUENCES (SUBJECT PUBLIC KEY REF - SIGNATURE_REF)
48     stringRef partial[ALLOWED_SEQUENCES] = {{0}};
49     INT16 countOfSequences = 0;
50     INT16 i;
51     //
52     # if CERTIFYX509_DEBUG
53     DebugFileInit();
54     DebugDumpBuffer(in->partialCertificate.t.size,
55 in->partialCertificate.t.buffer,
56 "partialCertificate");
57     # endif
58
59     // Input Validation
60     if(in->reserved.b.size != 0)
61         return TPM_RC_SIZE + RC_CertifyX509_reserved;
62     // signing key must be able to sign
63     if(!IsSigningObject(signKey))
64         return TPM_RCS_KEY + RC_CertifyX509_signHandle;
65     // Pick a scheme for sign. If the input sign scheme is not compatible with
66     // the default scheme, return an error.
67     if(!CryptSelectSignScheme(signKey, &in->inScheme))
68         return TPM_RCS_SCHEME + RC_CertifyX509_inScheme;
69     // Make sure that the public Key encoding is known
70     if(X509AddPublicKey(NULL, object) == 0)
71         return TPM_RCS_ASYMMETRIC + RC_CertifyX509_objectHandle;
72     // Unbundle 'partialCertificate'.
73     // Initialize the unmarshaling context
74     if(!ASN1UnmarshalContextInitialize(
75 &ctx, in->partialCertificate.t.size, in->partialCertificate.t.buffer))
76         return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
77     // Make sure that this is a constructed SEQUENCE
78     length = ASN1NextTag(&ctx);
79     // Must be a constructed SEQUENCE that uses all of the input parameter
80     if((ctx.tag != (ASN1_CONSTRUCTED_SEQUENCE))
81 || ((ctx.offset + length) != in->partialCertificate.t.size))
82         return TPM_RC_SIZE + RC_CertifyX509_partialCertificate;
83
84     // This scans through the contents of the outermost SEQUENCE. This would be the
85     // 'issuer', 'validity', 'subject', 'issuerUniqueID' (optional),

```

```

86 // 'subjectUniqueID' (optional), and 'extensions.'
87 while(ctx.offset < ctx.size)
88 {
89     INT16 startOfElement = ctx.offset;
90     //
91     // Read the next tag and length field.
92     length = ASN1NextTag(&ctx);
93     if(length < 0)
94         break;
95     if(ctx.tag == ASN1_CONSTRUCTED_SEQUENCE)
96     {
97         if(countOfSequences < ALLOWED_SEQUENCES)
98         {
99             partial[countOfSequences].buf = &ctx.buffer[startOfElement];
100             ctx.offset += length;
101             partial[countOfSequences].len = (INT16)ctx.offset - startOfElement;
102         }
103         countOfSequences++;
104         if(countOfSequences > ALLOWED_SEQUENCES)
105             break;
106     }
107     else if(ctx.tag == X509_EXTENSIONS)
108     {
109         if(certTBS[EXTENSIONS_REF].len != 0)
110             return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
111         certTBS[EXTENSIONS_REF].buf = &ctx.buffer[startOfElement];
112         ctx.offset += length;
113         certTBS[EXTENSIONS_REF].len = (INT16)ctx.offset - startOfElement;
114     }
115     else
116         return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
117 }
118 // Make sure that we used all of the data and found at least the required
119 // number of elements.
120 if((ctx.offset != ctx.size) || (countOfSequences < 3) || (countOfSequences > 4)
121 || (certTBS[EXTENSIONS_REF].buf == NULL))
122     return TPM_RCS_VALUE + RC_CertifyX509_partialCertificate;
123 // Now that we know how many sequences there were, we can put them where they
124 // belong
125 for(i = 0; i < countOfSequences; i++)
126     certTBS[SUBJECT_KEY_REF - i] = partial[countOfSequences - 1 - i];
127
128 // If only three SEQUENCES, then the TPM needs to produce the signature algorithm.
129 // See if it can
130 if((countOfSequences == 3)
131 && (X509AddSigningAlgorithm(NULL, signKey, &in->inScheme) == 0))
132     return TPM_RCS_SCHEME + RC_CertifyX509_signHandle;
133
134 // Process the extensions
135 result = X509ProcessExtensions(object, &certTBS[EXTENSIONS_REF]);
136 if(result != TPM_RC_SUCCESS)
137     // If the extension has the TPMA_OBJECT extension and the attributes don't
138     // match, then the error code will be TPM_RCS_ATTRIBUTES. Otherwise, the error
139     // indicates a malformed partialCertificate.
140     return result
141         + ((result == TPM_RCS_ATTRIBUTES) ? RC_CertifyX509_objectHandle
142         : RC_CertifyX509_partialCertificate);
143
144 // Command Output
145 // Create the addedToCertificate values
146
147 // Build the addedToCertificate from the bottom up.
148 // Initialize the context structure
149 ASN1InitializeMarshalContext(&ctxOut,
150                             sizeof(out->addedToCertificate.t.buffer),
151                             out->addedToCertificate.t.buffer);
152 // Place a marker for the overall context

```

```

152     ASN1StartMarshalContext(&ctxOut); // SEQUENCE for addedToCertificate
153
154     // Add the subject public key descriptor
155     certTBS[SUBJECT_PUBLIC_KEY_REF].len = X509AddPublicKey(&ctxOut, object);
156     certTBS[SUBJECT_PUBLIC_KEY_REF].buf = ctxOut.buffer + ctxOut.offset;
157     // If the caller didn't provide the algorithm identifier, create it
158     if(certTBS[SIGNATURE_REF].len == 0)
159     {
160         certTBS[SIGNATURE_REF].len =
161             X509AddSigningAlgorithm(&ctxOut, signKey, &in->inScheme);
162         certTBS[SIGNATURE_REF].buf = ctxOut.buffer + ctxOut.offset;
163     }
164     // Create the serial number value. Use the out->tbsDigest as scratch.
165     {
166         TPM2B* digest = &out->tbsDigest.b;
167         //
168         digest->size = (INT16)CryptHashStart(&hash, signKey->publicArea.nameAlg);
169         pAssert(digest->size != 0);
170
171         // The serial number size is the smaller of the digest and the vendor-defined
172         // value
173         digest->size = MIN(digest->size, SIZE_OF_X509_SERIAL_NUMBER);
174         // Add all the parts of the certificate other than the serial number
175         // and version number
176         for(i = SIGNATURE_REF; i < REF_COUNT; i++)
177             CryptDigestUpdate(&hash, certTBS[i].len, certTBS[i].buf);
178         // throw in the Name of the signing key...
179         CryptDigestUpdate2B(&hash, &signKey->name.b);
180         // ...and the Name of the signed key.
181         CryptDigestUpdate2B(&hash, &object->name.b);
182         // Done
183         CryptHashEnd2B(&hash, digest);
184     }
185
186     // Add the serial number
187     certTBS[SERIAL_NUMBER_REF].len =
188         ASN1PushInteger(&ctxOut, out->tbsDigest.t.size, out->tbsDigest.t.buffer);
189     certTBS[SERIAL_NUMBER_REF].buf = ctxOut.buffer + ctxOut.offset;
190
191     // Add the static version number
192     ASN1StartMarshalContext(&ctxOut);
193     ASN1PushUINT(&ctxOut, 2);
194     certTBS[VERSION_REF].len =
195         ASN1EndEncapsulation(&ctxOut, ASN1_APPLICATION_SPECIFIC);
196     certTBS[VERSION_REF].buf = ctxOut.buffer + ctxOut.offset;
197
198     // Create a fake tag and length for the TBS in the space used for
199     // 'addedToCertificate'
200     {
201         for(length = 0, i = 0; i < REF_COUNT; i++)
202             length += certTBS[i].len;
203         // Put a fake tag and length into the buffer for use in the tbsDigest
204         certTBS[ENCODED_SIZE_REF].len =
205             ASN1PushTagAndLength(&ctxOut, ASN1_CONSTRUCTED_SEQUENCE, length);
206         certTBS[ENCODED_SIZE_REF].buf = ctxOut.buffer + ctxOut.offset;
207         // Restore the buffer pointer to add back the number of octets used for the
208         // tag and length
209         ctxOut.offset += certTBS[ENCODED_SIZE_REF].len;
210     }
211     // sanity check
212     if(ctxOut.offset < 0)
213         return TPM_RC_FAILURE;
214     // Create the tbsDigest to sign
215     out->tbsDigest.t.size = CryptHashStart(&hash, in->inScheme.details.any.hashAlg);
216     for(i = 0; i < REF_COUNT; i++)
217         CryptDigestUpdate(&hash, certTBS[i].len, certTBS[i].buf);

```



```

218     CryptHashEnd2B(&hash, &out->tbsDigest.b);
219
220 # if CERTIFYX509_DEBUG
221 {
222     BYTE    fullTBS[4096];
223     BYTE*   fill = fullTBS;
224     int     j;
225     for(j = 0; j < REF_COUNT; j++)
226     {
227         MemoryCopy(fill, certTBS[j].buf, certTBS[j].len);
228         fill += certTBS[j].len;
229     }
230     DebugDumpBuffer((int) (fill - &fullTBS[0]), fullTBS, "\nfull TBS");
231 }
232 # endif
233
234 // Finish up the processing of addedToCertificate
235 // Create the actual tag and length for the addedToCertificate structure
236 out->addedToCertificate.t.size =
237     ASN1EndEncapsulation(&ctxOut, ASN1_CONSTRUCTED_SEQUENCE);
238 // Now move all the addedToContext to the start of the buffer
239 MemoryCopy(out->addedToCertificate.t.buffer,
240           ctxOut.buffer + ctxOut.offset,
241           out->addedToCertificate.t.size);
242 # if CERTIFYX509_DEBUG
243     DebugDumpBuffer(out->addedToCertificate.t.size,
244                     out->addedToCertificate.t.buffer,
245                     "\naddedToCertificate");
246 # endif
247 // only thing missing is the signature
248 result = CryptSign(signKey, &in->inScheme, &out->tbsDigest, &out->signature);
249
250 return result;
251 }
252
253 #endif // CC_CertifyX509

```

7.18 /tpm/src/command/Attestation/GetCommandAuditDigest.c

```

1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "GetCommandAuditDigest_fp.h"
4
5  #if CC_GetCommandAuditDigest // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // Get current value of command audit log
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_KEY          key referenced by 'signHandle' is not a signing key
12 //     TPM_RC_SCHEME       'inScheme' is incompatible with 'signHandle' type; or
13 //                         both 'scheme' and key's default scheme are empty; or
14 //                         'scheme' is empty while key's default scheme requires
15 //                         explicit input scheme (split signing); or
16 //                         non-empty default key scheme differs from 'scheme'
17 //     TPM_RC_VALUE        digest generated for the given 'scheme' is greater than
18 //                         the modulus of 'signHandle' (for an RSA key);
19 //                         invalid commit status or failed to generate "r" value
20 //                         (for an ECC key)
21 TPM_RC
22 TPM2_GetCommandAuditDigest(
23     GetCommandAuditDigest_In* in, // IN: input parameter list
24     GetCommandAuditDigest_Out* out // OUT: output parameter list
25 )
26 {

```



```

27     TPM_RC      result;
28     TPMS_ATTEST auditInfo;
29     OBJECT*     signObject = HandleToObject(in->signHandle);
30     // Input validation
31     if(!IsSigningObject(signObject))
32         return TPM_RC_KEY + RC_GetCommandAuditDigest_signHandle;
33     if(!CryptSelectSignScheme(signObject, &in->inScheme))
34         return TPM_RC_SCHEME + RC_GetCommandAuditDigest_inScheme;
35
36     // Command Output
37     // Fill in attest information common fields
38     FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &auditInfo);
39
40     // CommandAuditDigest specific fields
41     auditInfo.type = TPM_ST_ATTEST_COMMAND_AUDIT;
42     auditInfo.attested.commandAudit.digestAlg = gp.auditHashAlg;
43     auditInfo.attested.commandAudit.auditCounter = gp.auditCounter;
44
45     // Copy command audit log
46     auditInfo.attested.commandAudit.auditDigest = gr.commandAuditDigest;
47     CommandAuditGetDigest(&auditInfo.attested.commandAudit.commandDigest);
48
49     // Sign attestation structure. A NULL signature will be returned if
50     // signHandle is TPM_RH_NULL. A TPM_RC_NV_UNAVAILABLE, TPM_RC_NV_RATE,
51     // TPM_RC_VALUE, TPM_RC_SCHEME or TPM_RC_ATTRIBUTES error may be returned at
52     // this point
53     result = SignAttestInfo(signObject,
54                             &in->inScheme,
55                             &auditInfo,
56                             &in->qualifyingData,
57                             &out->auditInfo,
58                             &out->signature);
59
60     // Internal Data Update
61     if(result == TPM_RC_SUCCESS && in->signHandle != TPM_RH_NULL)
62         // Reset log
63         gr.commandAuditDigest.t.size = 0;
64
65     return result;
66 }
67 #endif // CC_GetCommandAuditDigest

```

7.19 /tpm/src/command/Attestation/GetSessionAuditDigest.c

```

1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "GetSessionAuditDigest_fp.h"
4
5  #if CC_GetSessionAuditDigest // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // Get audit session digest
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_KEY          key referenced by 'signHandle' is not a signing key
12 //     TPM_RC_SCHEME       'inScheme' is incompatible with 'signHandle' type; or
13 //                         both 'scheme' and key's default scheme are empty; or
14 //                         'scheme' is empty while key's default scheme requires
15 //                         explicit input scheme (split signing); or
16 //                         non-empty default key scheme differs from 'scheme'
17 //     TPM_RC_TYPE         'sessionHandle' does not reference an audit session
18 //     TPM_RC_VALUE        digest generated for the given 'scheme' is greater than
19 //                         the modulus of 'signHandle' (for an RSA key);
20 //                         invalid commit status or failed to generate "r" value
21 //                         (for an ECC key)

```

```

22 TPM_RC
23 TPM2_GetSessionAuditDigest(
24     GetSessionAuditDigest_In* in, // IN: input parameter list
25     GetSessionAuditDigest_Out* out // OUT: output parameter list
26 )
27 {
28     SESSION* session = SessionGet(in->sessionHandle);
29     TPMS_ATTEST auditInfo;
30     OBJECT* signObject = HandleToObject(in->signHandle);
31     // Input Validation
32     if(!IsSigningObject(signObject))
33         return TPM_RCS_KEY + RC_GetSessionAuditDigest_signHandle;
34     if(!CryptSelectSignScheme(signObject, &in->inScheme))
35         return TPM_RCS_SCHEME + RC_GetSessionAuditDigest_inScheme;
36
37     // session must be an audit session
38     if(session->attributes.isAudit == CLEAR)
39         return TPM_RCS_TYPE + RC_GetSessionAuditDigest_sessionHandle;
40
41     // Command Output
42     // Fill in attest information common fields
43     FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &auditInfo);
44
45     // SessionAuditDigest specific fields
46     auditInfo.type = TPM_ST_ATTEST_SESSION_AUDIT;
47     auditInfo.attested.sessionAudit.sessionDigest = session->u2.auditDigest;
48
49     // Exclusive audit session
50     auditInfo.attested.sessionAudit.exclusiveSession =
51         (g_exclusiveAuditSession == in->sessionHandle);
52
53     // Sign attestation structure. A NULL signature will be returned if
54     // signObject is NULL.
55     return SignAttestInfo(signObject,
56                           &in->inScheme,
57                           &auditInfo,
58                           &in->qualifyingData,
59                           &out->auditInfo,
60                           &out->signature);
61 }
62
63 #endif // CC_GetSessionAuditDigest

```

7.20 /tpm/src/command/Attestation/GetTime.c

```

1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "GetTime_fp.h"
4
5  #if CC_GetTime // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // Applies a time stamp to the passed blob (qualifyingData).
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_KEY          key referenced by 'signHandle' is not a signing key
12 //     TPM_RC_SCHEME       'inScheme' is incompatible with 'signHandle' type; or
13 //                         both 'scheme' and key's default scheme are empty; or
14 //                         'scheme' is empty while key's default scheme requires
15 //                         explicit input scheme (split signing); or
16 //                         non-empty default key scheme differs from 'scheme'
17 //     TPM_RC_VALUE        digest generated for the given 'scheme' is greater than
18 //                         the modulus of 'signHandle' (for an RSA key);
19 //                         invalid commit status or failed to generate "r" value
20 //                         (for an ECC key)

```

```

21 TPM_RC
22 TPM2_GetTime(GetTime_In* in, // IN: input parameter list
23             GetTime_Out* out // OUT: output parameter list
24 )
25 {
26     TPMS_ATTEST timeInfo;
27     OBJECT*      signObject = HandleToObject(in->signHandle);
28     // Input Validation
29     if(!IsSigningObject(signObject))
30         return TPM_RCS_KEY + RC_GetTime_signHandle;
31     if(!CryptSelectSignScheme(signObject, &in->inScheme))
32         return TPM_RCS_SCHEME + RC_GetTime_inScheme;
33
34     // Command Output
35     // Fill in attest common fields
36     FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &timeInfo);
37
38     // GetClock specific fields
39     timeInfo.type = TPM_ST_ATTEST_TIME;
40     timeInfo.attested.time.time.time = g_time;
41     TimeFillInfo(&timeInfo.attested.time.time.clockInfo);
42
43     // Firmware version in plain text
44     timeInfo.attested.time.firmwareVersion =
45         ((UINT64)gp.firmwareV1) << 32) + gp.firmwareV2;
46
47     // Sign attestation structure. A NULL signature will be returned if
48     // signObject is NULL.
49     return SignAttestInfo(signObject,
50                          &in->inScheme,
51                          &timeInfo,
52                          &in->qualifyingData,
53                          &out->timeInfo,
54                          &out->signature);
55 }
56
57 #endif // CC_GetTime

```

7.21 /tpm/src/command/Attestation/Quote.c

```

1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "Quote_fp.h"
4
5  #if CC_Quote // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // quote PCR values
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_KEY 'signHandle' does not reference a signing key;
12 //     TPM_RC_SCHEME the scheme is not compatible with sign key type,
13 // or input scheme is not compatible with default
14 // scheme, or the chosen scheme is not a valid
15 // sign scheme
16 TPM_RC
17 TPM2_Quote(Quote_In* in, // IN: input parameter list
18          Quote_Out* out // OUT: output parameter list
19 )
20 {
21     TPMT_ALG HASH hashAlg;
22     TPMS_ATTEST quoted;
23     OBJECT*      signObject = HandleToObject(in->signHandle);
24     // Input Validation
25     if(!IsSigningObject(signObject))

```

```

26     return TPM_RCS_KEY + RC_Quote_signHandle;
27     if(!CryptSelectSignScheme(signObject, &in->inScheme))
28         return TPM_RCS_SCHEME + RC_Quote_inScheme;
29
30     // Command Output
31
32     // Filling in attest information
33     // Common fields
34     // FillInAttestInfo may return TPM_RC_SCHEME or TPM_RC_KEY
35     FillInAttestInfo(in->signHandle, &in->inScheme, &in->qualifyingData, &quoted);
36
37     // Quote specific fields
38     // Attestation type
39     quoted.type = TPM_ST_ATTEST_QUOTE;
40
41     // Get hash algorithm in sign scheme. This hash algorithm is used to
42     // compute PCR digest. If there is no algorithm, then the PCR cannot
43     // be digested and this command returns TPM_RC_SCHEME
44     hashAlg = in->inScheme.details.any.hashAlg;
45
46     if(hashAlg == TPM_ALG_NULL)
47         return TPM_RCS_SCHEME + RC_Quote_inScheme;
48
49     // Compute PCR digest
50     PCRComputeCurrentDigest(
51         hashAlg, &in->PCRselect, &quoted.attested.quote.pcrDigest);
52
53     // Copy PCR select. "PCRselect" is modified in PCRComputeCurrentDigest
54     // function
55     quoted.attested.quote.pcrSelect = in->PCRselect;
56
57     // Sign attestation structure. A NULL signature will be returned if
58     // signObject is NULL.
59     return SignAttestInfo(signObject,
60                           &in->inScheme,
61                           &quoted,
62                           &in->qualifyingData,
63                           &out->quoted,
64                           &out->signature);
65 }
66
67 #endif // CC_Quote

```

7.22 /tpm/src/command/Capability/GetCapability.c

```

1  #include "Tpm.h"
2  #include "GetCapability_fp.h"
3
4  #if CC_GetCapability // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command returns various information regarding the TPM and its current
8  // state
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_HANDLE      value of 'property' is in an unsupported handle range
12 //     TPM_RC_VALUE       for the TPM_CAP_HANDLES 'capability' value
13 //     TPM_RC_VALUE       invalid 'capability'; or 'property' is not 0 for the
14 //     TPM_CAP_PCRS 'capability' value
15 TPM_RC
16 TPM2_GetCapability(GetCapability_In* in, // IN: input parameter list
17                  GetCapability_Out* out // OUT: output parameter list
18 )
19 {
20     TPMU_CAPABILITIES* data = &out->capabilityData.data;

```

```

21 // Command Output
22
23 // Set output capability type the same as input type
24 out->capabilityData.capability = in->capability;
25
26 switch(in->capability)
27 {
28     case TPM_CAP_ALGS:
29         out->moreData = AlgorithmCapGetImplemented(
30             (TPM_ALG_ID)in->property, in->propertyCount, &data->algorithms);
31         break;
32     case TPM_CAP_HANDLES:
33         switch(HandleGetType((TPM_HANDLE)in->property))
34         {
35             case TPM_HT_TRANSIENT:
36                 // Get list of handles of loaded transient objects
37                 out->moreData = ObjectCapGetLoaded(
38                     (TPM_HANDLE)in->property, in->propertyCount, &data->handles);
39                 break;
40             case TPM_HT_PERSISTENT:
41                 // Get list of handles of persistent objects
42                 out->moreData = NvCapGetPersistent(
43                     (TPM_HANDLE)in->property, in->propertyCount, &data->handles);
44                 break;
45             case TPM_HT_NV_INDEX:
46                 // Get list of defined NV index
47                 out->moreData = NvCapGetIndex(
48                     (TPM_HANDLE)in->property, in->propertyCount, &data->handles);
49                 break;
50             case TPM_HT_LOADED_SESSION:
51                 // Get list of handles of loaded sessions
52                 out->moreData = SessionCapGetLoaded(
53                     (TPM_HANDLE)in->property, in->propertyCount, &data->handles);
54                 break;
55             case TPM_HT_SAVED_SESSION:
56                 // Get list of handles of
57                 out->moreData = SessionCapGetSaved(
58                     (TPM_HANDLE)in->property, in->propertyCount, &data->handles);
59                 break;
60             case TPM_HT_PCR:
61                 // Get list of handles of PCR
62                 out->moreData = PCRCapGetHandles(
63                     (TPM_HANDLE)in->property, in->propertyCount, &data->handles);
64                 break;
65             case TPM_HT_PERMANENT:
66                 // Get list of permanent handles
67                 out->moreData = PermanentCapGetHandles(
68                     (TPM_HANDLE)in->property, in->propertyCount, &data->handles);
69                 break;
70             default:
71                 // Unsupported input handle type
72                 return TPM_RCS_HANDLE + RC_GetCapability_property;
73                 break;
74         }
75         break;
76     case TPM_CAP_COMMANDS:
77         out->moreData = CommandCapGetCCList(
78             (TPM_CC)in->property, in->propertyCount, &data->command);
79         break;
80     case TPM_CAP_PP_COMMANDS:
81         out->moreData = PhysicalPresenceCapGetCCList(
82             (TPM_CC)in->property, in->propertyCount, &data->ppCommands);
83         break;
84     case TPM_CAP_AUDIT_COMMANDS:
85         out->moreData = CommandAuditCapGetCCList(
86             (TPM_CC)in->property, in->propertyCount, &data->auditCommands);

```

```

87         break;
88     case TPM_CAP_PCRS:
89         // Input property must be 0
90         if(in->property != 0)
91             return TPM_RCS_VALUE + RC_GetCapability_property;
92         out->moreData =
93             PCRCapGetAllocation(in->propertyCount, &data->assignedPCR);
94         break;
95     case TPM_CAP_PCR_PROPERTIES:
96         out->moreData = PCRCapGetProperties(
97             (TPM_PT_PCR)in->property, in->propertyCount, &data->pcrProperties);
98         break;
99     case TPM_CAP_TPM_PROPERTIES:
100         out->moreData = TPMCapGetProperties(
101             (TPM_PT)in->property, in->propertyCount, &data->tpmProperties);
102         break;
103 # if ALG_ECC
104     case TPM_CAP_ECC_CURVES:
105         out->moreData = CryptCapGetECCCurve(
106             (TPM_ECC_CURVE)in->property, in->propertyCount, &data->eccCurves);
107         break;
108 # endif // ALG_ECC
109     case TPM_CAP_AUTH_POLICIES:
110         if(HandleGetType((TPM_HANDLE)in->property) != TPM_HT_PERMANENT)
111             return TPM_RCS_VALUE + RC_GetCapability_property;
112         out->moreData = PermanentHandleGetPolicy(
113             (TPM_HANDLE)in->property, in->propertyCount, &data->authPolicies);
114         break;
115     case TPM_CAP_ACT:
116 # if ACT_SUPPORT
117         if(((TPM_RH)in->property < TPM_RH_ACT_0)
118             || ((TPM_RH)in->property > TPM_RH_ACT_F))
119             return TPM_RCS_VALUE + RC_GetCapability_property;
120         out->moreData = ActGetCapabilityData(
121             (TPM_HANDLE)in->property, in->propertyCount, &data->actData);
122         break;
123 # else
124         return TPM_RCS_VALUE + RC_GetCapability_property;
125 # endif // ACT_SUPPORT
126     case TPM_CAP_VENDOR_PROPERTY:
127         // vendor property is not implemented
128     default:
129         // Unsupported TPM_CAP value
130         return TPM_RCS_VALUE + RC_GetCapability_capability;
131         break;
132     }
133
134     return TPM_RC_SUCCESS;
135 }
136
137 #endif // CC_GetCapability

```

7.23 /tpm/src/command/Capability/SetCapability.c

```

1  #include "Tpm.h"
2  #include "SetCapability_fp.h"
3
4  #if CC_SetCapability // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command allows configuration of the TPM's capabilities.
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_HANDLE     value of 'property' is in an unsupported handle range
11 //                        for the TPM_CAP_HANDLES 'capability' value

```



```

12 //      TPM_RC_VALUE      invalid 'capability'
13 TPM_RC
14 TPM2_SetCapability(SetCapability_In* in // IN: input parameter list
15 )
16 {
17     // This reference implementation does not implement any settable capabilities.
18     return TPM_RC_VALUE + SetCapability_setCapabilityData;
19 }
20
21 #endif // CC_SetCapability

```

7.24 /tpm/src/command/Capability/TestParms.c

```

1 #include "Tpm.h"
2 #include "TestParms_fp.h"
3
4 #if CC_TestParms // Conditional expansion of this file
5
6 /*(See part 3 specification)
7 // TestParms
8 */
9 TPM_RC
10 TPM2_TestParms(TestParms_In* in // IN: input parameter list
11 )
12 {
13     // Input parameter is not reference in command action
14     NOT_REFERENCED(in);
15
16     // The parameters are tested at unmarshal process. We do nothing in command
17     // action
18     return TPM_RC_SUCCESS;
19 }
20
21 #endif // CC_TestParms

```

7.25 /tpm/src/command/ClockTimer/ACT_SetTimeout.c

```

1 #include "Tpm.h"
2 #include "ACT_SetTimeout_fp.h"
3
4 #if CC_ACT_SetTimeout // Conditional expansion of this file
5
6 /*(See part 3 specification)
7 // prove an object with a specific Name is loaded in the TPM
8 */
9 // Return Type: TPM_RC
10 //      TPM_RC_RETRY      returned when an update for the selected ACT is
11 //                        already pending
12 //      TPM_RC_VALUE      attempt to disable signaling from an ACT that has
13 //                        not expired
14 TPM_RC
15 TPM2_ACT_SetTimeout(ACT_SetTimeout_In* in // IN: input parameter list
16 )
17 {
18     // If 'startTimeout' is UINT32_MAX, then this is an attempt to disable the ACT
19     // and turn off the signaling for the ACT. This is only valid if the ACT
20     // is signaling.
21     # if ACT_SUPPORT
22         if((in->startTimeout == UINT32_MAX) && !ActGetSignaled(in->actHandle))
23             return TPM_RC_VALUE + RC_ACT_SetTimeout_startTimeout;
24         return ActCounterUpdate(in->actHandle, in->startTimeout);
25     # else // ACT_SUPPORT
26         NOT_REFERENCED(in);
27         return TPM_RC_VALUE + RC_ACT_SetTimeout_startTimeout;

```

```

28 # endif // ACT_SUPPORT
29 }
30
31 #endif // CC_ACT_SetTimeout

```

7.26 /tpm/src/command/ClockTimer/ACT_spt.c

```

1  /** Introduction
2  // This code implements the ACT update code. It does not use a mutex. This code uses
3  // a platform service (_plat_ACT_UpdateCounter()) that returns 'false' if the update
4  // is not accepted. If this occurs, then TPM_RC_RETRY should be sent to the caller so
5  // that they can retry the operation later. The implementation of this is platform
6  // dependent but the reference uses a simple flag to indicate that an update is
7  // pending and the only process that can clear that flag is the process that does the
8  // actual update.
9
10 /** Includes
11 #include "Tpm.h"
12 #include "ACT_spt_fp.h"
13 // TODO_RENAME_INC_FOLDER:platform_interface refers to the TPM_CoreLib platform
14 // interface
15 #include <platform_interface/tpm_to_platform_interface.h>
16
17 /** Functions
18
19 #if ACT_SUPPORT
20
21 /***_ _ActResume()
22 // This function does the resume processing for an ACT. It updates the saved count
23 // and turns signaling back on if necessary.
24 static void _ActResume(UINT32 act, //IN: the act number
25 ACT_STATE* actData //IN: pointer to the saved ACT data
26 )
27 {
28 // If the act was non-zero, then restore the counter value.
29 if(actData->remaining > 0)
30 _plat_ACT_UpdateCounter(act, actData->remaining);
31 // if the counter was zero and the ACT signaling, enable the signaling.
32 else if(go.signaledACT & (1 << act))
33 _plat_ACT_SetSignaled(act, TRUE);
34 }
35
36 /***_ ActStartup()
37 // This function is called by TPM2_Startup() to initialize the ACT counter values.
38 BOOL ActStartup(STARTUP_TYPE type)
39 {
40 // Reset all the ACT hardware
41 _plat_ACT_Initialize();
42
43 // If this not a cold start, copy all the current 'signaled' settings to
44 // 'preservedSignaled'.
45 if(g_powerWasLost)
46 go.preservedSignaled = 0;
47 else
48 go.preservedSignaled |= go.signaledACT;
49
50 // For TPM_RESET or TPM_RESTART, the ACTs will all be disabled and the output
51 // de-asserted.
52 if(type != SU_RESUME)
53 {
54 go.signaledACT = 0;
55 # define CLEAR_ACT_POLICY(N) \
56 go.ACT_##N.hashAlg = TPM_ALG_NULL; \
57 go.ACT_##N.authPolicy.b.size = 0; \
58 FOR_EACH_ACT(CLEAR_ACT_POLICY)

```

```

58     }
59     else
60     {
61         // Resume each of the implemented ACT
62 # define RESUME_ACT(N) _ActResume(0x##N, &go.ACT_##N);
63
64         FOR_EACH_ACT(RESUME_ACT)
65     }
66     // set no ACT updated since last startup. This is to enable the halving of the
67     // timeout value
68     s_ActUpdated = 0;
69     _plat__ACT_EnableTicks(TRUE);
70     return TRUE;
71 }
72
73 /***_ActSaveState()
74 // Get the counter state and the signaled state for an ACT. If the ACT has not been
75 // updated since the last time it was saved, then divide the count by 2.
76 static void _ActSaveState(UINT32 act, P_ACT_STATE actData)
77 {
78     actData->remaining = _plat__ACT_GetRemaining(act);
79     // If the ACT hasn't been updated since the last startup, then it should be
80     // be halved.
81     if((s_ActUpdated & (1 << act)) == 0)
82     {
83         // Don't halve if the count is set to max or if halving would make it zero
84         if((actData->remaining != UINT32_MAX) && (actData->remaining > 1))
85             actData->remaining /= 2;
86     }
87     if(_plat__ACT_GetSignaled(act))
88         go.signaledACT |= (1 << act);
89 }
90
91 /***_ActGetSignaled()
92 // This function returns the state of the signaled flag associated with an ACT.
93 BOOL ActGetSignaled(TPM_RH actHandle)
94 {
95     UINT32 act = actHandle - TPM_RH_ACT_0;
96     //
97     return _plat__ACT_GetSignaled(act);
98 }
99
100 /***_ActShutdown()
101 // This function saves the current state of the counters
102 BOOL ActShutdown(TPM_SU state //IN: the type of the shutdown.
103 )
104 {
105     // if this is not shutdown state, then the only type of startup is TPM RESTART
106     // so the timer values will be cleared. If this is shutdown state, get the current
107     // countdown and signaled values. Plus, if the counter has not been updated
108     // since the last restart, divide the time by 2 so that there is no attack on the
109     // countdown by saving the countdown state early and then not using the TPM.
110     if(state == TPM_SU_STATE)
111     {
112         // This will be populated as each of the ACT is queried
113         go.signaledACT = 0;
114         // Get the current count and the signaled state
115 # define SAVE_ACT_STATE(N) _ActSaveState(0x##N, &go.ACT_##N);
116
117         FOR_EACH_ACT(SAVE_ACT_STATE);
118     }
119     return TRUE;
120 }
121
122 /***_ActIsImplemented()
123 // This function determines if an ACT is implemented in both the TPM and the platform

```

```

124 // code.
125 BOOL ActIsImplemented(UINT32 act)
126 {
127     // This switch accounts for the TPM implemented values.
128     switch(act)
129     {
130         FOR_EACH_ACT(CASE_ACT_NUMBER)
131         // This ensures that the platform implements the values implemented by
132         // the TPM
133         return _plat__ACT_GetImplemented(act);
134         default:
135             break;
136     }
137     return FALSE;
138 }
139
140 /***ActCounterUpdate()
141 // This function updates the ACT counter. If the counter already has a pending update,
142 // it returns TPM_RC_RETRY so that the update can be tried again later.
143 TPM_RC
144 ActCounterUpdate(TPM_RH handle, //IN: the handle of the act
145                 UINT32 newValue //IN: the value to set in the ACT
146 )
147 {
148     UINT32 act;
149     TPM_RC result;
150     //
151     act = handle - TPM_RH_ACT_0;
152     // This should never fail, but...
153     if(!_plat__ACT_GetImplemented(act))
154         result = TPM_RC_VALUE;
155     else
156     {
157         // Will need to clear orderly so fail if we are orderly and NV is
158         // not available
159         if(NV_IS_ORDERLY)
160             RETURN_IF_NV_IS_NOT_AVAILABLE;
161         // if the attempt to update the counter fails, it means that there is an
162         // update pending so wait until it has occurred and then do an update.
163         if(!_plat__ACT_UpdateCounter(act, newValue))
164             result = TPM_RC_RETRY;
165         else
166         {
167             // Indicate that the ACT has been updated since last TPM2_Startup().
168             s_ActUpdated |= (UINT16)(1 << act);
169
170             // Clear the preservedSignaled attribute.
171             go.preservedSignaled &= ~((UINT16)(1 << act));
172
173             // Need to clear the orderly flag
174             g_clearOrderly = TRUE;
175
176             result = TPM_RC_SUCCESS;
177         }
178     }
179     return result;
180 }
181
182 /*** ActGetCapabilityData()
183 // This function returns the list of ACT data
184 // Return Type: TPMTI_YES_NO
185 // YES if more ACT data is available
186 // NO if no more ACT data to
187 TPMTI_YES_NO
188 ActGetCapabilityData(TPM_HANDLE actHandle, // IN: the handle for the starting ACT
189                     UINT32 maxCount, // IN: maximum allowed return values

```

```

190         TPML_ACT_DATA* actList      // OUT: ACT data list
191     )
192 {
193     // Initialize output property list
194     actList->count = 0;
195
196     // Make sure that the starting handle value is in range (again)
197     if((actHandle < TPM_RH_ACT_0) || (actHandle > TPM_RH_ACT_F))
198         return FALSE;
199     // The maximum count of curves we may return is MAX_ECC_CURVES
200     if(maxCount > MAX_ACT_DATA)
201         maxCount = MAX_ACT_DATA;
202     // Scan the ACT data from the starting ACT
203     for(; actHandle <= TPM_RH_ACT_F; actHandle++)
204     {
205         UINT32 act = actHandle - TPM_RH_ACT_0;
206         if(actList->count < maxCount)
207         {
208             if(ActIsImplemented(act))
209             {
210                 TPMS_ACT_DATA* actData = &actList->actData[actList->count];
211                 //
212                 memset(&actData->attributes, 0, sizeof(actData->attributes));
213                 actData->handle = actHandle;
214                 actData->timeout = _plat_ACT_GetRemaining(act);
215                 if(_plat_ACT_GetSignaled(act))
216                     SET_ATTRIBUTE(actData->attributes, TPMA_ACT, signaled);
217                 else
218                     CLEAR_ATTRIBUTE(actData->attributes, TPMA_ACT, signaled);
219                 if(go.preservedSignaled & (1 << act))
220                     SET_ATTRIBUTE(actData->attributes, TPMA_ACT, preserveSignaled);
221                 actList->count++;
222             }
223         }
224         else
225         {
226             if(_plat_ACT_GetImplemented(act))
227                 return YES;
228         }
229     }
230     // If we get here, either all of the ACT values were put in the list, or the list
231     // was filled and there are no more ACT values to return
232     return NO;
233 }
234
235 /*** ActGetOneCapability()
236 // This function returns an ACT's capability, if present.
237 BOOL ActGetOneCapability(TPM_HANDLE actHandle, // IN: the handle for the ACT
238                         TPMS_ACT_DATA* actData // OUT: ACT data
239 )
240 {
241     UINT32 act = actHandle - TPM_RH_ACT_0;
242
243     if(ActIsImplemented(actHandle - TPM_RH_ACT_0))
244     {
245         memset(&actData->attributes, 0, sizeof(actData->attributes));
246         actData->handle = actHandle;
247         actData->timeout = _plat_ACT_GetRemaining(act);
248         if(_plat_ACT_GetSignaled(act))
249             SET_ATTRIBUTE(actData->attributes, TPMA_ACT, signaled);
250         else
251             CLEAR_ATTRIBUTE(actData->attributes, TPMA_ACT, signaled);
252         if(go.preservedSignaled & (1 << act))
253             SET_ATTRIBUTE(actData->attributes, TPMA_ACT, preserveSignaled);
254         return TRUE;
255     }

```

```

256     return FALSE;
257 }
258
259 #endif // ACT_SUPPORT

```

7.27 /tpm/src/command/ClockTimer/ClockRateAdjust.c

```

1  #include "Tpm.h"
2  #include "ClockRateAdjust_fp.h"
3
4  #if CC_ClockRateAdjust // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // adjusts the rate of advance of Clock and Timer to provide a better
8  // approximation to real time.
9  */
10 TPM_RC
11 TPM2_ClockRateAdjust(ClockRateAdjust_In* in // IN: input parameter list
12 )
13 {
14     // Internal Data Update
15     TimeSetAdjustRate(in->rateAdjust);
16
17     return TPM_RC_SUCCESS;
18 }
19
20 #endif // CC_ClockRateAdjust

```

7.28 /tpm/src/command/ClockTimer/ClockSet.c

```

1  #include "Tpm.h"
2  #include "ClockSet_fp.h"
3
4  #if CC_ClockSet // Conditional expansion of this file
5
6  // Read the current TPMS_TIMER_INFO structure settings
7  // Return Type: TPM_RC
8  // TPM_RC_NV_RATE NV is unavailable because of rate limit
9  // TPM_RC_NV_UNAVAILABLE NV is inaccessible
10 // TPM_RC_VALUE invalid new clock
11
12 TPM_RC
13 TPM2_ClockSet(ClockSet_In* in // IN: input parameter list
14 )
15 {
16     // Input Validation
17     // new time can not be bigger than 0xFFFF000000000000 or smaller than
18     // current clock
19     if(in->newTime > 0xFFFF000000000000ULL || in->newTime < go.clock)
20         return TPM_RCS_VALUE + RC_ClockSet_newTime;
21
22     // Internal Data Update
23     // Can't modify the clock if NV is not available.
24     RETURN_IF_NV_IS_NOT_AVAILABLE;
25
26     TimeClockUpdate(in->newTime);
27     return TPM_RC_SUCCESS;
28 }
29
30 #endif // CC_ClockSet

```


7.29 /tpm/src/command/ClockTimer/ReadClock.c

```

1  #include "Tpm.h"
2  #include "ReadClock_fp.h"
3
4  #if CC_ReadClock // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // read the current TPMS_TIMER_INFO structure settings
8  */
9  TPM_RC
10 TPM2_ReadClock(ReadClock_Out* out // OUT: output parameter list
11 )
12 {
13     // Command Output
14
15     out->currentTime.time = g_time;
16     TimeFillInfo(&out->currentTime.clockInfo);
17
18     return TPM_RC_SUCCESS;
19 }
20
21 #endif // CC_ReadClock

```

7.30 /tpm/src/command/CommandAudit/SetCommandCodeAuditStatus.c

```

1  #include "Tpm.h"
2  #include "SetCommandCodeAuditStatus_fp.h"
3
4  #if CC_SetCommandCodeAuditStatus // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // change the audit status of a command or to set the hash algorithm used for
8  // the audit digest.
9  */
10 TPM_RC
11 TPM2_SetCommandCodeAuditStatus(
12     SetCommandCodeAuditStatus_In* in // IN: input parameter list
13 )
14 {
15
16     // The command needs NV update. Check if NV is available.
17     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
18     // this point
19     RETURN_IF_NV_IS_NOT_AVAILABLE;
20
21     // Internal Data Update
22
23     // Update hash algorithm
24     if(in->auditAlg != TPM_ALG_NULL && in->auditAlg != gp.auditHashAlg)
25     {
26         // Can't change the algorithm and command list at the same time
27         if(in->setList.count != 0 || in->clearList.count != 0)
28             return TPM_RCS_VALUE + RC_SetCommandCodeAuditStatus_auditAlg;
29
30         // Change the hash algorithm for audit
31         gp.auditHashAlg = in->auditAlg;
32
33         // Set the digest size to a unique value that indicates that the digest
34         // algorithm has been changed. The size will be cleared to zero in the
35         // command audit processing on exit.
36         gr.commandAuditDigest.t.size = 1;
37
38         // Save the change of command audit data (this sets g_updateNV so that NV
39         // will be updated on exit.)

```

```

40     NV_SYNC_PERSISTENT(auditHashAlg);
41 }
42 else
43 {
44     UINT32 i;
45     BOOL    changed = FALSE;
46
47     // Process set list
48     for(i = 0; i < in->setList.count; i++)
49
50         // If change is made in CommandAuditSet, set changed flag
51         if(CommandAuditSet(in->setList.commandCodes[i]))
52             changed = TRUE;
53
54     // Process clear list
55     for(i = 0; i < in->clearList.count; i++)
56         // If change is made in CommandAuditClear, set changed flag
57         if(CommandAuditClear(in->clearList.commandCodes[i]))
58             changed = TRUE;
59
60     // if change was made to command list, update NV
61     if(changed)
62         // this sets g_updateNV so that NV will be updated on exit.
63         NV_SYNC_PERSISTENT(auditCommands);
64 }
65
66 return TPM_RC_SUCCESS;
67 }
68
69 #endif // CC_SetCommandCodeAuditStatus

```

7.31 /tpm/src/command/Context/ContextLoad.c

```

1  #include "Tpm.h"
2
3  #if CC_ContextLoad // Conditional expansion of this file
4
5  # include "ContextLoad_fp.h"
6  # include "Marshal.h"
7  # include "Context_spt_fp.h"
8
9  /*(See part 3 specification)
10 // Load context
11 */
12
13 // Return Type: TPM_RC
14 //     TPM_RC_CONTEXT_GAP      there is only one available slot and this is not
15 //                               the oldest saved session context
16 //     TPM_RC_HANDLE           'context.savedHandle' does not reference a saved
17 //                               session
18 //     TPM_RC_HIERARCHY        'context.hierarchy' is disabled
19 //     TPM_RC_INTEGRITY         'context' integrity check fail
20 //     TPM_RC_OBJECT_MEMORY     no free slot for an object
21 //     TPM_RC_SESSION_MEMORY    no free session slots
22 //     TPM_RC_SIZE              incorrect context blob size
23 TPM_RC
24 TPM2_ContextLoad(ContextLoad_In* in, // IN: input parameter list
25                  ContextLoad_Out* out // OUT: output parameter list
26 )
27 {
28     TPM_RC    result;
29     TPM2B_DIGEST integrityToCompare;
30     TPM2B_DIGEST integrity;
31     BYTE*      buffer; // defined to save some typing
32     INT32      size;    // defined to save some typing

```

```

33     TPM_HT      handleType;
34     TPM2B_SYM_KEY symKey;
35     TPM2B_IV     iv;
36
37     // Input Validation
38
39     // See discussion about the context format in TPM2_ContextSave Detailed Actions
40
41     // IF this is a session context, make sure that the sequence number is
42     // consistent with the version in the slot
43
44     // Check context blob size
45     handleType = HandleGetType(in->context.savedHandle);
46
47     // Get integrity from context blob
48     buffer = in->context.contextBlob.t.buffer;
49     size   = (INT32)in->context.contextBlob.t.size;
50     result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
51     if(result != TPM_RC_SUCCESS)
52         return result;
53
54     // the size of the integrity value has to match the size of digest produced
55     // by the integrity hash
56     if(integrity.t.size != CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG))
57         return TPM_RCS_SIZE + RC_ContextLoad_context;
58
59     // Make sure that the context blob has enough space for the fingerprint. This
60     // is elastic pants to go with the belt and suspenders we already have to make
61     // sure that the context is complete and untampered.
62     if((unsigned)size < sizeof(in->context.sequence))
63         return TPM_RCS_SIZE + RC_ContextLoad_context;
64
65     // After unmarshaling the integrity value, 'buffer' is pointing at the first
66     // byte of the integrity protected and encrypted buffer and 'size' is the number
67     // of integrity protected and encrypted bytes.
68
69     // Compute context integrity
70     result = ComputeContextIntegrity(&in->context, &integrityToCompare);
71     if(result != TPM_RC_SUCCESS)
72         return result;
73
74     // Compare integrity
75     if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
76         return TPM_RCS_INTEGRITY + RC_ContextLoad_context;
77     // Compute context encryption key
78     result = ComputeContextProtectionKey(&in->context, &symKey, &iv);
79     if(result != TPM_RC_SUCCESS)
80         return result;
81
82     // Decrypt context data in place
83     CryptSymmetricDecrypt(buffer,
84                           CONTEXT_ENCRYPT_ALG,
85                           CONTEXT_ENCRYPT_KEY_BITS,
86                           symKey.t.buffer,
87                           &iv,
88                           TPM_ALG_CFB,
89                           size,
90                           buffer);
91
92     // See if the fingerprint value matches. If not, it is symptomatic of either
93     // a broken TPM or that the TPM is under attack so go into failure mode.
94     if(!MemoryEqual(buffer, &in->context.sequence, sizeof(in->context.sequence)))
95         FAIL(FATAL_ERROR_INTERNAL);
96
97     // step over fingerprint
98     buffer += sizeof(in->context.sequence);

```

```

99     // set the remaining size of the context
100     size -= sizeof(in->context.sequence);
101
102     // Perform object or session specific input check
103     switch(handleType)
104     {
105         case TPM_HT_TRANSIENT:
106         {
107             OBJECT* outObject;
108
109             if(size > (INT32)sizeof(OBJECT))
110                 FAIL(FATAL_ERROR_INTERNAL);
111
112             // Discard any changes to the handle that the TRM might have made
113             in->context.savedHandle = TRANSIENT_FIRST;
114
115             // If hierarchy is disabled, no object context can be loaded in this
116             // hierarchy
117             if(!HierarchyIsEnabled(in->context.hierarchy))
118                 return TPM_RCS_HIERARCHY + RC_ContextLoad_context;
119
120             // Restore object. If there is no empty space, indicate as much
121             outObject =
122                 ObjectContextLoad((ANY_OBJECT_BUFFER*)buffer, &out->loadedHandle);
123             if(outObject == NULL)
124                 return TPM_RC_OBJECT_MEMORY;
125
126             break;
127         }
128         case TPM_HT_POLICY_SESSION:
129         case TPM_HT_HMAC_SESSION:
130         {
131             if(size != sizeof(SESSION))
132                 FAIL(FATAL_ERROR_INTERNAL);
133
134             // This command may cause the orderlyState to be cleared due to
135             // the update of state reset data. If this is the case, check if NV is
136             // available first
137             RETURN_IF_ORDERLY;
138
139             // Check if input handle points to a valid saved session and that the
140             // sequence number makes sense
141             if(!SequenceNumberForSavedContextIsValid(&in->context))
142                 return TPM_RCS_HANDLE + RC_ContextLoad_context;
143
144             // Restore session. A TPM_RC_SESSION_MEMORY, TPM_RC_CONTEXT_GAP error
145             // may be returned at this point
146             result =
147                 SessionContextLoad((SESSION_BUF*)buffer, &in->context.savedHandle);
148             if(result != TPM_RC_SUCCESS)
149                 return result;
150
151             out->loadedHandle = in->context.savedHandle;
152
153             // orderly state should be cleared because of the update of state
154             // reset and state clear data
155             g_clearOrderly = TRUE;
156
157             break;
158         }
159         default:
160             // Context blob may only have an object handle or a session handle.
161             // All the other handle type should be filtered out at unmarshal
162             FAIL(FATAL_ERROR_INTERNAL);
163             break;
164     }

```

```

165
166     return TPM_RC_SUCCESS;
167 }
168
169 #endif // CC_ContextLoad

```

7.32 /tpm/src/command/Context/ContextSave.c

```

1  #include "Tpm.h"
2
3  #if CC_ContextSave // Conditional expansion of this file
4
5  # include "ContextSave_fp.h"
6  # include "Marshal.h"
7  # include "Context_spt_fp.h"
8
9  /*(See part 3 specification)
10  Save context
11  */
12  // Return Type: TPM_RC
13  //     TPM_RC_CONTEXT_GAP          a contextID could not be assigned for a session
14  //                                     context save
15  //     TPM_RC_TOO_MANY_CONTEXTS    no more contexts can be saved as the counter has
16  //                                     maxed out
17  TPM_RC
18  TPM2_ContextSave(ContextSave_In* in, // IN: input parameter list
19                  ContextSave_Out* out // OUT: output parameter list
20  )
21  {
22      TPM_RC result = TPM_RC_SUCCESS;
23      UINT16 fingerprintSize; // The size of fingerprint in context
24      // blob.
25      UINT64 contextID = 0; // session context ID
26      TPM2B_SYM_KEY symKey;
27      TPM2B_IV iv;
28
29      TPM2B_DIGEST integrity;
30      UINT16 integritySize;
31      BYTE* buffer;
32
33      // This command may cause the orderlyState to be cleared due to
34      // the update of state reset data. If the state is orderly and
35      // cannot be changed, exit early.
36      RETURN_IF_ORDERLY;
37
38      // Internal Data Update
39
40      // This implementation does not do things in quite the same way as described in
41      // Part 2 of the specification. In Part 2, it indicates that the
42      // TPMS_CONTEXT_DATA contains two TPM2B values. That is not how this is
43      // implemented. Rather, the size field of the TPM2B_CONTEXT_DATA is used to
44      // determine the amount of data in the encrypted data. That part is not
45      // independently sized. This makes the actual size 2 bytes smaller than
46      // calculated using Part 2. Since this is opaque to the caller, it is not
47      // necessary to fix. The actual size is returned by TPM2_GetCapabilities().
48
49      // Initialize output handle. At the end of command action, the output
50      // handle of an object will be replaced, while the output handle
51      // for a session will be the same as input
52      out->context.savedHandle = in->saveHandle;
53
54      // Get the size of fingerprint in context blob. The sequence value in
55      // TPMS_CONTEXT structure is used as the fingerprint
56      fingerprintSize = sizeof(out->context.sequence);
57

```

```

58 // Compute the integrity size at the beginning of context blob
59 integritySize =
60     sizeof(integrity.t.size) + CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
61
62 // Perform object or session specific context save
63 switch(HandleGetType(in->saveHandle))
64 {
65     case TPM_HT_TRANSIENT:
66     {
67         OBJECT* object = HandleToObject(in->saveHandle);
68         ANY_OBJECT_BUFFER* outObject;
69         UINT16 objectSize = ObjectIsSequence(object) ? sizeof(HASH_OBJECT)
70                                                         : sizeof(OBJECT);
71
72         outObject = (ANY_OBJECT_BUFFER*)(out->context.contextBlob.t.buffer
73                                         + integritySize + fingerprintSize);
74
75         // Set size of the context data. The contents of context blob is vendor
76         // defined. In this implementation, the size is size of integrity
77         // plus fingerprint plus the whole internal OBJECT structure
78         out->context.contextBlob.t.size =
79             integritySize + fingerprintSize + objectSize;
80 # if ALG_RSA
81     // For an RSA key, make sure that the key has had the private exponent
82     // computed before saving.
83     if(object->publicArea.type == TPM_ALG_RSA
84         && !(object->attributes.publicOnly))
85         CryptRsaLoadPrivateExponent(&object->publicArea, &object->sensitive);
86 # endif
87     // Make sure things fit
88     pAssert(out->context.contextBlob.t.size
89             <= sizeof(out->context.contextBlob.t.buffer));
90     // Copy the whole internal OBJECT structure to context blob
91     MemoryCopy(outObject, object, objectSize);
92
93     // Increment object context ID
94     gr.objectContextID++;
95     // If object context ID overflows, TPM should be put in failure mode
96     if(gr.objectContextID == 0)
97         FAIL(FATAL_ERROR_INTERNAL);
98
99     // Fill in other return values for an object.
100    out->context.sequence = gr.objectContextID;
101    // For regular object, savedHandle is 0x80000000. For sequence object,
102    // savedHandle is 0x80000001. For object with stClear, savedHandle
103    // is 0x80000002
104    if(ObjectIsSequence(object))
105    {
106        out->context.savedHandle = 0x80000001;
107        SequenceDataExport((HASH_OBJECT*)object,
108                           (HASH_OBJECT_BUFFER*)outObject);
109    }
110    else
111        out->context.savedHandle =
112            (object->attributes.stClear == SET) ? 0x80000002 : 0x80000000;
113    // Get object hierarchy
114    out->context.hierarchy = object->hierarchy;
115
116    break;
117 }
118 case TPM_HT_HMAC_SESSION:
119 case TPM_HT_POLICY_SESSION:
120 {
121     SESSION* session = SessionGet(in->saveHandle);
122
123     // Set size of the context data. The contents of context blob is vendor

```



```

124 // defined. In this implementation, the size of context blob is the
125 // size of a internal session structure plus the size of
126 // fingerprint plus the size of integrity
127 out->context.contextBlob.t.size =
128     integritySize + fingerprintSize + sizeof(*session);
129
130 // Make sure things fit
131 pAssert(out->context.contextBlob.t.size
132     < sizeof(out->context.contextBlob.t.buffer));
133
134 // Copy the whole internal SESSION structure to context blob.
135 // Save space for fingerprint at the beginning of the buffer
136 // This is done before anything else so that the actual context
137 // can be reclaimed after this call
138 pAssert(sizeof(*session) <= sizeof(out->context.contextBlob.t.buffer)
139     - integritySize - fingerprintSize);
140
141 MemoryCopy(
142     out->context.contextBlob.t.buffer + integritySize + fingerprintSize,
143     session,
144     sizeof(*session));
145 // Fill in the other return parameters for a session
146 // Get a context ID and set the session tracking values appropriately
147 // TPM_RC_CONTEXT_GAP is a possible error.
148 // SessionContextSave() will flush the in-memory context
149 // so no additional errors may occur after this call.
150 result = SessionContextSave(out->context.savedHandle, &contextID);
151 if(result != TPM_RC_SUCCESS)
152     return result;
153 // sequence number is the current session contextID
154 out->context.sequence = contextID;
155
156 // use TPM_RH_NULL as hierarchy for session context
157 out->context.hierarchy = TPM_RH_NULL;
158
159 break;
160 }
161 default:
162     // SaveContext may only take an object handle or a session handle.
163     // All the other handle type should be filtered out at unmarshal
164     FAIL(FATAL_ERROR_INTERNAL);
165     break;
166 }
167
168 // Save fingerprint at the beginning of encrypted area of context blob.
169 // Reserve the integrity space
170 pAssert(sizeof(out->context.sequence)
171     <= sizeof(out->context.contextBlob.t.buffer) - integritySize);
172 MemoryCopy(out->context.contextBlob.t.buffer + integritySize,
173     &out->context.sequence,
174     sizeof(out->context.sequence));
175
176 // Compute context encryption key
177 result = ComputeContextProtectionKey(&out->context, &symKey, &iv);
178 if(result != TPM_RC_SUCCESS)
179     return result;
180
181 // Encrypt context blob
182 CryptSymmetricEncrypt(out->context.contextBlob.t.buffer + integritySize,
183     CONTEXT_ENCRYPT_ALG,
184     CONTEXT_ENCRYPT_KEY_BITS,
185     symKey.t.buffer,
186     &iv,
187     TPM_ALG_CFB,
188     out->context.contextBlob.t.size - integritySize,
189     out->context.contextBlob.t.buffer + integritySize);

```

```

190     // Compute integrity hash for the object
191     // In this implementation, the same routine is used for both sessions
192     // and objects.
193     result = ComputeContextIntegrity(&out->context, &integrity);
194     if(result != TPM_RC_SUCCESS)
195         return result;
196
197     // add integrity at the beginning of context blob
198     buffer = out->context.contextBlob.t.buffer;
199     TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
200
201     // orderly state should be cleared because of the update of state reset and
202     // state clear data
203     g_clearOrderly = TRUE;
204
205     return result;
206 }
207
208 #endif // CC_ContextSave

```

7.33 /tpm/src/command/Context/Context_spt.c

```

1  /** Includes
2
3  #include "Tpm.h"
4  #include "Context_spt_fp.h"
5
6  /** Functions
7
8  /** ComputeContextProtectionKey()
9  // This function retrieves the symmetric protection key for context encryption
10 // It is used by TPM2_ContextSave and TPM2_ContextLoad to create the symmetric
11 // encryption key and iv
12 /*(See part 1 specification)
13     KDFa is used to generate the symmetric encryption key and IV. The parameters
14     of the call are:
15         Symkey = KDFa(hashAlg, hProof, vendorString, sequence, handle, bits)
16     where
17     hashAlg      a vendor-defined hash algorithm
18     hProof        the hierarchy proof as selected by the hierarchy parameter
19                   of the TPMS_CONTEXT
20     vendorString  a value used to differentiate the uses of the KDF
21     sequence      the sequence parameter of the TPMS_CONTEXT
22     handle        the handle parameter of the TPMS_CONTEXT
23     bits          the number of bits needed for a symmetric key and IV for
24                   the context encryption
25 */
26 // Return Type: TPM_RC
27 //     TPM_RC_FW_LIMITED      The requested hierarchy is FW-limited, but the TPM
28 //                             does not support FW-limited objects or the TPM failed
29 //                             to derive the Firmware Secret.
30 //     TPM_RC_SVN_LIMITED     The requested hierarchy is SVN-limited, but the TPM
31 //                             does not support SVN-limited objects or the TPM
32 //                             failed to derive the Firmware SVN Secret for the
33 //                             requested SVN.
34 TPM_RC ComputeContextProtectionKey(TPMS_CONTEXT* contextBlob, // IN: context blob
35     TPM2B_SYM_KEY* symKey, // OUT: the symmetric key
36     TPM2B_IV* iv // OUT: the IV.
37 )
38 {
39     TPM_RC result = TPM_RC_SUCCESS;
40     UINT16 symKeyBits; // number of bits in the parent's
41                       // symmetric key
42     TPM2B_PROOF proof; // the proof value to use
43

```

```

44     BYTE          kdfResult[sizeof(TPMU_HA) * 2]; // Value produced by the KDF
45
46     TPM2B_DATA    sequence2B, handle2B;
47
48     // Get sequence value in 2B format
49     sequence2B.t.size = sizeof(contextBlob->sequence);
50     MUST_BE(sizeof(contextBlob->sequence) <= sizeof(sequence2B.t.buffer));
51     MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence, sequence2B.t.size);
52
53     // Get handle value in 2B format
54     handle2B.t.size = sizeof(contextBlob->savedHandle);
55     MUST_BE(sizeof(contextBlob->savedHandle) <= sizeof(handle2B.t.buffer));
56     MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle, handle2B.t.size);
57
58     // Get the symmetric encryption key size
59     symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
60     symKeyBits    = CONTEXT_ENCRYPT_KEY_BITS;
61     // Get the size of the IV for the algorithm
62     iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);
63
64     // Get proof value
65     result = HierarchyGetProof(contextBlob->hierarchy, &proof);
66     if(result != TPM_RC_SUCCESS)
67         return result;
68
69     // KDFa to generate symmetric key and IV value
70     CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG,
71              &proof.b,
72              CONTEXT_KEY,
73              &sequence2B.b,
74              &handle2B.b,
75              (symKey->t.size + iv->t.size) * 8,
76              kdfResult,
77              NULL,
78              FALSE);
79
80     MemorySet(proof.b.buffer, 0, proof.b.size);
81
82     // Copy part of the returned value as the key
83     pAssert(symKey->t.size <= sizeof(symKey->t.buffer));
84     MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size);
85
86     // Copy the rest as the IV
87     pAssert(iv->t.size <= sizeof(iv->t.buffer));
88     MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size);
89
90     return TPM_RC_SUCCESS;
91 }
92
93 /*** ComputeContextIntegrity()
94 // Generate the integrity hash for a context
95 //     It is used by TPM2_ContextSave to create an integrity hash
96 //     and by TPM2_ContextLoad to compare an integrity hash
97 /*(See part 1 specification)
98     The HMAC integrity computation for a saved context is:
99     HMACvendorAlg(hProof, resetValue {|| clearCount} || sequence || handle ||
100                  encContext)
101
102     where
103     HMACvendorAlg    HMAC using a vendor-defined hash algorithm
104     hProof           the hierarchy proof as selected by the hierarchy
105                      parameter of the TPMS_CONTEXT
106     resetValue       either a counter value that increments on each TPM Reset
107                      and is not reset over the lifetime of the TPM or a random
108                      value that changes on each TPM Reset and has the size of
109                      the digest produced by vendorAlg
110     clearCount       a counter value that is incremented on each TPM Reset

```

```

110                                     or TPM Restart. This value is only included if the handle
111                                     value is 0x80000002.
112     sequence                         the sequence parameter of the TPMS_CONTEXT
113     handle                           the handle parameter of the TPMS_CONTEXT
114     encContext                       the encrypted context blob
115 */
116 // Return Type: TPM_RC
117 //     TPM_RC_FW_LIMITED             The requested hierarchy is FW-limited, but the TPM
118 //                                   does not support FW-limited objects or the TPM failed
119 //                                   to derive the Firmware Secret.
120 //     TPM_RC_SVN_LIMITED            The requested hierarchy is SVN-limited, but the TPM
121 //                                   does not support SVN-limited objects or the TPM
122 //                                   failed to derive the Firmware SVN Secret for the
123 //                                   requested SVN.
124 TPM_RC ComputeContextIntegrity(TPMS_CONTEXT* contextBlob, // IN: context blob
125                                TPM2B_DIGEST* integrity    // OUT: integrity
126 )
127 {
128     TPM_RC      result = TPM_RC_SUCCESS;
129     HMAC_STATE  hmacState;
130     TPM2B_PROOF proof;
131     UINT16      integritySize;
132
133     // Get proof value
134     result = HierarchyGetProof(contextBlob->hierarchy, &proof);
135     if(result != TPM_RC_SUCCESS)
136         return result;
137
138     // Start HMAC
139     integrity->t.size =
140         CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG, &proof.b);
141
142     MemorySet(proof.b.buffer, 0, proof.b.size);
143
144     // Compute integrity size at the beginning of context blob
145     integritySize = sizeof(integrity->t.size) + integrity->t.size;
146
147     // Adding total reset counter so that the context cannot be
148     // used after a TPM Reset
149     CryptDigestUpdateInt(
150         &hmacState.hashState, sizeof(gp.totalResetCount), gp.totalResetCount);
151
152     // If this is a ST_CLEAR object, add the clear count
153     // so that this context cannot be loaded after a TPM Restart
154     if(contextBlob->savedHandle == 0x80000002)
155         CryptDigestUpdateInt(
156             &hmacState.hashState, sizeof(gr.clearCount), gr.clearCount);
157
158     // Adding sequence number to the HMAC to make sure that it doesn't
159     // get changed
160     CryptDigestUpdateInt(
161         &hmacState.hashState, sizeof(contextBlob->sequence), contextBlob->sequence);
162
163     // Protect the handle
164     CryptDigestUpdateInt(&hmacState.hashState,
165                         sizeof(contextBlob->savedHandle),
166                         contextBlob->savedHandle);
167
168     // Adding sensitive contextData, skip the leading integrity area
169     CryptDigestUpdate(&hmacState.hashState,
170                     contextBlob->contextBlob.t.size - integritySize,
171                     contextBlob->contextBlob.t.buffer + integritySize);
172
173     // Complete HMAC
174     CryptHmacEnd2B(&hmacState, &integrity->b);
175

```

```

176     return TPM_RC_SUCCESS;
177 }
178
179 /*** SequenceDataExport();
180 // This function is used scan through the sequence object and
181 // either modify the hash state data for export (contextSave) or to
182 // import it into the internal format (contextLoad).
183 // This function should only be called after the sequence object has been copied
184 // to the context buffer (contextSave) or from the context buffer into the sequence
185 // object. The presumption is that the context buffer version of the data is the
186 // same size as the internal representation so nothing outside of the hash context
187 // area gets modified.
188 void SequenceDataExport(
189     HASH_OBJECT* object,           // IN: an internal hash object
190     HASH_OBJECT_BUFFER* exportObject // OUT: a sequence context in a buffer
191 )
192 {
193     // If the hash object is not an event, then only one hash context is needed
194     int count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
195
196     for(count--; count >= 0; count--)
197     {
198         HASH_STATE* hash      = &object->state.hashState[count];
199         size_t      offset    = (BYTE*)hash - (BYTE*)object;
200         BYTE*       exportHash = &((BYTE*)exportObject)[offset];
201
202         CryptHashExportState(hash, (EXPORT_HASH_STATE*)exportHash);
203     }
204 }
205
206 /*** SequenceDataImport();
207 // This function is used scan through the sequence object and
208 // either modify the hash state data for export (contextSave) or to
209 // import it into the internal format (contextLoad).
210 // This function should only be called after the sequence object has been copied
211 // to the context buffer (contextSave) or from the context buffer into the sequence
212 // object. The presumption is that the context buffer version of the data is the
213 // same size as the internal representation so nothing outside of the hash context
214 // area gets modified.
215 void SequenceDataImport(
216     HASH_OBJECT* object,           // IN/OUT: an internal hash object
217     HASH_OBJECT_BUFFER* exportObject // IN/OUT: a sequence context in a buffer
218 )
219 {
220     // If the hash object is not an event, then only one hash context is needed
221     int count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
222
223     for(count--; count >= 0; count--)
224     {
225         HASH_STATE* hash      = &object->state.hashState[count];
226         size_t      offset    = (BYTE*)hash - (BYTE*)object;
227         BYTE*       importHash = &((BYTE*)exportObject)[offset];
228         //
229         CryptHashImportState(hash, (EXPORT_HASH_STATE*)importHash);
230     }
231 }

```

7.34 /tpm/src/command/Context/EvictControl.c

```

1  #include "Tpm.h"
2  #include "EvictControl_fp.h"
3
4  #if CC_EvictControl // Conditional expansion of this file
5
6  /*(See part 3 specification)

```



```

7  // Make a transient object persistent or evict a persistent object
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES    an object with 'temporary', 'stClear' or 'publicOnly'
11 //                          attribute SET cannot be made persistent
12 //     TPM_RC_HIERARCHY     'auth' cannot authorize the operation in the hierarchy
13 //                          of 'evictObject';
14 //                          an object in a firmware-bound or SVN-bound hierarchy
15 //                          cannot be made persistent.
16 //     TPM_RC_HANDLE        'evictHandle' of the persistent object to be evicted is
17 //                          not the same as the 'persistentHandle' argument
18 //     TPM_RC_NV_HANDLE     'persistentHandle' is unavailable
19 //     TPM_RC_NV_SPACE      no space in NV to make 'evictHandle' persistent
20 //     TPM_RC_RANGE         'persistentHandle' is not in the range corresponding to
21 //                          the hierarchy of 'evictObject'
22 TPM_RC
23 TPM2_EvictControl(EvictControl_In* in // IN: input parameter list
24 )
25 {
26     TPM_RC result;
27     OBJECT* evictObject;
28
29     // Input Validation
30
31     // Get internal object pointer
32     evictObject = HandleToObject(in->objectHandle);
33
34     // Objects in a firmware-limited or SVN-limited hierarchy cannot be made
35     // persistent.
36     if(HierarchyIsFirmwareLimited(evictObject->hierarchy)
37        || HierarchyIsSvnLimited(evictObject->hierarchy))
38         return TPM_RCS_HIERARCHY + RC_EvictControl_objectHandle;
39
40     // Temporary, stClear or public only objects can not be made persistent
41     if(evictObject->attributes.temporary == SET
42        || evictObject->attributes.stClear == SET
43        || evictObject->attributes.publicOnly == SET)
44         return TPM_RCS_ATTRIBUTES + RC_EvictControl_objectHandle;
45
46     // If objectHandle refers to a persistent object, it should be the same as
47     // input persistentHandle
48     if(evictObject->attributes.evict == SET
49        && evictObject->evictHandle != in->persistentHandle)
50         return TPM_RCS_HANDLE + RC_EvictControl_objectHandle;
51
52     // Additional authorization validation
53     if(in->auth == TPM_RH_PLATFORM)
54     {
55         // To make persistent
56         if(evictObject->attributes.evict == CLEAR)
57         {
58             // PlatformAuth can not set evict object in storage or endorsement
59             // hierarchy
60             if(evictObject->attributes.ppsHierarchy == CLEAR)
61                 return TPM_RCS_HIERARCHY + RC_EvictControl_objectHandle;
62             // Platform cannot use a handle outside of platform persistent range.
63             if(!NvIsPlatformPersistentHandle(in->persistentHandle))
64                 return TPM_RCS_RANGE + RC_EvictControl_persistentHandle;
65         }
66         // PlatformAuth can delete any persistent object
67     }
68     else if(in->auth == TPM_RH_OWNER)
69     {
70         // OwnerAuth can not set or clear evict object in platform hierarchy
71         if(evictObject->attributes.ppsHierarchy == SET)
72             return TPM_RCS_HIERARCHY + RC_EvictControl_objectHandle;

```



```

73
74     // Owner cannot use a handle outside of owner persistent range.
75     if(evictObject->attributes.evict == CLEAR
76         && !NvIsOwnerPersistentHandle(in->persistentHandle))
77         return TPM_RCS_RANGE + RC_EvictControl_persistentHandle;
78     }
79     else
80     {
81         // Other authorization is not allowed in this command and should have been
82         // filtered out in unmarshal process
83         FAIL(FATAL_ERROR_INTERNAL);
84     }
85     // Internal Data Update
86     // Change evict state
87     if(evictObject->attributes.evict == CLEAR)
88     {
89         // Make object persistent
90         if(NvFindHandle(in->persistentHandle) != 0)
91             return TPM_RC_NV_DEFINED;
92         // A TPM_RC_NV_HANDLE or TPM_RC_NV_SPACE error may be returned at this
93         // point
94         result = NvAddEvictObject(in->persistentHandle, evictObject);
95     }
96     else
97     {
98         // Delete the persistent object in NV
99         result = NvDeleteEvict(evictObject->evictHandle);
100    }
101    return result;
102 }
103
104 #endif // CC_EvictControl

```

7.35 /tpm/src/command/Context/FlushContext.c

```

1  #include "Tpm.h"
2  #include "FlushContext_fp.h"
3
4  #if CC_FlushContext // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Flush a specific object or session
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_HANDLE 'flushHandle' does not reference a loaded object or session
11 TPM_RC
12 TPM2_FlushContext(FlushContext_In* in // IN: input parameter list
13 )
14 {
15     // Internal Data Update
16
17     // Call object or session specific routine to flush
18     switch(HandleGetType(in->flushHandle))
19     {
20         case TPM_HT_TRANSIENT:
21             if(!IsObjectPresent(in->flushHandle))
22                 return TPM_RCS_HANDLE + RC_FlushContext_flushHandle;
23             // Flush object
24             FlushObject(in->flushHandle);
25             break;
26         case TPM_HT_HMAC_SESSION:
27         case TPM_HT_POLICY_SESSION:
28             if(!SessionIsLoaded(in->flushHandle) && !SessionIsSaved(in->flushHandle))
29                 return TPM_RCS_HANDLE + RC_FlushContext_flushHandle;
30     }

```

```

31         // If the session to be flushed is the exclusive audit session, then
32         // indicate that there is no exclusive audit session any longer.
33         if(in->flushHandle == g_exclusiveAuditSession)
34             g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
35
36         // Flush session
37         SessionFlush(in->flushHandle);
38         break;
39     default:
40         // This command only takes object or session handle. Other handles
41         // should be filtered out at handle unmarshal
42         FAIL(FATAL_ERROR_INTERNAL);
43         break;
44     }
45
46     return TPM_RC_SUCCESS;
47 }
48
49 #endif // CC_FlushContext

```

7.36 /tpm/src/command/DA/DictionaryAttackLockReset.c

```

1  #include "Tpm.h"
2  #include "DictionaryAttackLockReset_fp.h"
3
4  #if CC_DictionaryAttackLockReset // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command cancels the effect of a TPM lockout due to a number of
8  // successive authorization failures. If this command is properly authorized,
9  // the lockout counter is set to 0.
10 */
11 TPM_RC
12 TPM2_DictionaryAttackLockReset(
13     DictionaryAttackLockReset_In* in // IN: input parameter list
14 )
15 {
16     // Input parameter is not reference in command action
17     NOT_REFERENCED(in);
18
19     // The command needs NV update.
20     RETURN_IF_NV_IS_NOT_AVAILABLE;
21
22     // Internal Data Update
23
24     // Set failed tries to 0
25     gp.failedTries = 0;
26
27     // Record the changes to NV
28     NV_SYNC_PERSISTENT(failedTries);
29
30     return TPM_RC_SUCCESS;
31 }
32
33 #endif // CC_DictionaryAttackLockReset

```

7.37 /tpm/src/command/DA/DictionaryAttackParameters.c

```

1  #include "Tpm.h"
2  #include "DictionaryAttackParameters_fp.h"
3
4  #if CC_DictionaryAttackParameters // Conditional expansion of this file
5
6  /*(See part 3 specification)

```

```

7  // change the lockout parameters
8  */
9  TPM_RC
10 TPM2_DictionaryAttackParameters(
11     DictionaryAttackParameters_In* in  // IN: input parameter list
12 )
13 {
14     // The command needs NV update.
15     RETURN_IF_NV_IS_NOT_AVAILABLE;
16
17     // Internal Data Update
18
19     // Set dictionary attack parameters
20     gp.maxTries      = in->newMaxTries;
21     gp.recoveryTime  = in->newRecoveryTime;
22     gp.lockoutRecovery = in->lockoutRecovery;
23
24     # if 0
25     // Errata eliminates this code
26     // This functionality has been disabled. The preferred implementation is now
27     // to leave failedTries unchanged when the parameters are changed. This could
28     // have the effect of putting the TPM into DA lockout if in->newMaxTries is
29     // not greater than the current value of gp.failedTries.
30     // Set failed tries to 0
31     gp.failedTries = 0;
32     # endif
33
34     // Record the changes to NV
35     NV_SYNC_PERSISTENT(failedTries);
36     NV_SYNC_PERSISTENT(maxTries);
37     NV_SYNC_PERSISTENT(recoveryTime);
38     NV_SYNC_PERSISTENT(lockoutRecovery);
39
40     return TPM_RC_SUCCESS;
41 }
42
43 #endif // CC_DictionaryAttackParameters

```

7.38 /tpm/src/command/Duplication/Duplicate.c

```

1  #include "Tpm.h"
2  #include "Duplicate_fp.h"
3
4  #if CC_Duplicate // Conditional expansion of this file
5
6  # include "Object_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Duplicate a loaded object
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_ATTRIBUTES key to duplicate has 'fixedParent' SET
13 //     TPM_RC_HASH        for an RSA key, the nameAlg digest size for the
14 //                         newParent is not compatible with the key size
15 //     TPM_RC_HIERARCHY   'encryptedDuplication' is SET and 'newParentHandle'
16 //                         specifies Null Hierarchy
17 //     TPM_RC_KEY         'newParentHandle' references invalid ECC key (public
18 //                         point not on the curve)
19 //     TPM_RC_SIZE        input encryption key size does not match the
20 //                         size specified in symmetric algorithm
21 //     TPM_RC_SYMMETRIC   'encryptedDuplication' is SET but no symmetric
22 //                         algorithm is provided
23 //     TPM_RC_TYPE        'newParentHandle' is neither a storage key nor
24 //                         TPM_RH_NULL; or the object has a NULL nameAlg
25 //     TPM_RC_VALUE       for an RSA newParent, the sizes of the digest and

```

```

26 // the encryption key are too large to be OAEP encoded
27 TPM_RC
28 TPM2_Duplicate(Duplicate_In* in, // IN: input parameter list
29               Duplicate_Out* out // OUT: output parameter list
30 )
31 {
32     TPM_RC result = TPM_RC_SUCCESS;
33     TPMT_SENSITIVE sensitive;
34
35     UINT16 innerKeySize = 0; // encrypt key size for inner wrap
36
37     OBJECT* object;
38     OBJECT* newParent;
39     TPM2B_DATA data;
40
41     // Input Validation
42
43     // Get duplicate object pointer
44     object = HandleToObject(in->objectHandle);
45     // Get new parent
46     newParent = HandleToObject(in->newParentHandle);
47
48     // duplicate key must have fixParent bit CLEAR.
49     if(IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, fixedParent))
50         return TPM_RCS_ATTRIBUTES + RC_Duplicate_objectHandle;
51
52     // Do not duplicate object with NULL nameAlg
53     if(object->publicArea.nameAlg == TPM_ALG_NULL)
54         return TPM_RCS_TYPE + RC_Duplicate_objectHandle;
55
56     // new parent key must be a storage object or TPM_RH_NULL
57     if(in->newParentHandle != TPM_RH_NULL && !ObjectIsStorage(in->newParentHandle))
58         return TPM_RCS_TYPE + RC_Duplicate_newParentHandle;
59
60     // If the duplicated object has encryptedDuplication SET, then there must be
61     // an inner wrapper and the new parent may not be TPM_RH_NULL
62     if(IS_ATTRIBUTE(
63         object->publicArea.objectAttributes, TPMA_OBJECT, encryptedDuplication))
64     {
65         if(in->symmetricAlg.algorithm == TPM_ALG_NULL)
66             return TPM_RCS_SYMMETRIC + RC_Duplicate_symmetricAlg;
67         if(in->newParentHandle == TPM_RH_NULL)
68             return TPM_RCS_HIERARCHY + RC_Duplicate_newParentHandle;
69     }
70
71     if(in->symmetricAlg.algorithm == TPM_ALG_NULL)
72     {
73         // if algorithm is TPM_ALG_NULL, input key size must be 0
74         if(in->encryptionKeyIn.t.size != 0)
75             return TPM_RCS_SIZE + RC_Duplicate_encryptionKeyIn;
76     }
77     else
78     {
79         // Get inner wrap key size
80         innerKeySize = in->symmetricAlg.keyBits.sym;
81
82         // If provided the input symmetric key must match the size of the algorithm
83         if(in->encryptionKeyIn.t.size != 0
84            && in->encryptionKeyIn.t.size != (innerKeySize + 7) / 8)
85             return TPM_RCS_SIZE + RC_Duplicate_encryptionKeyIn;
86     }
87
88     // Command Output
89
90     if(in->newParentHandle != TPM_RH_NULL)
91     {

```

```

92     // Make encrypt key and its associated secret structure.  A TPM_RC_KEY
93     // error may be returned at this point
94     out->outSymSeed.t.size = sizeof(out->outSymSeed.t.secret);
95     result =
96         CryptSecretEncrypt(newParent, DUPLICATE_STRING, &data, &out->outSymSeed);
97     if(result != TPM_RC_SUCCESS)
98         return result;
99 }
100 else
101 {
102     // Do not apply outer wrapper
103     data.t.size = 0;
104     out->outSymSeed.t.size = 0;
105 }
106
107 // Copy sensitive area
108 sensitive = object->sensitive;
109
110 // Prepare output private data from sensitive.
111 // Note: If there is no encryption key, one will be provided by
112 // SensitiveToDuplicate(). This is why the assignment of encryptionKeyIn to
113 // encryptionKeyOut will work properly and is not conditional.
114 SensitiveToDuplicate(&sensitive,
115                     &object->name.b,
116                     newParent,
117                     object->publicArea.nameAlg,
118                     &data.b,
119                     &in->symmetricAlg,
120                     &in->encryptionKeyIn,
121                     &out->duplicate);
122
123 out->encryptionKeyOut = in->encryptionKeyIn;
124
125 return TPM_RC_SUCCESS;
126 }
127
128 #endif // CC_Duplicate

```

7.39 /tpm/src/command/Duplication/Import.c

```

1  #include "Tpm.h"
2  #include "Import_fp.h"
3
4  #if CC_Import // Conditional expansion of this file
5
6  # include "Object_spt_fp.h"
7
8  /*(See part 3 specification)
9  // This command allows an asymmetrically encrypted blob, containing a duplicated
10 // object to be re-encrypted using the group symmetric key associated with the
11 // parent.
12 */
13 // Return Type: TPM_RC
14 //     TPM_RC_ATTRIBUTES      'FixedTPM' and 'fixedParent' of 'objectPublic' are not
15 //                             both CLEAR; or 'inSymSeed' is nonempty and
16 //                             'parentHandle' does not reference a decryption key; or
17 //                             'objectPublic' and 'parentHandle' have incompatible
18 //                             or inconsistent attributes; or
19 //                             encryptedDuplication is SET in 'objectPublic' but the
20 //                             inner or outer wrapper is missing.
21 //                             Note that if the TPM provides parameter values, the
22 //                             parameter number will indicate 'symmetricKey' (missing
23 //                             inner wrapper) or 'inSymSeed' (missing outer wrapper)
24 //     TPM_RC_BINDING          'duplicate' and 'objectPublic' are not
25 //                             cryptographically bound

```

```

26 //      TPM_RC_ECC_POINT      'inSymSeed' is nonempty and ECC point in 'inSymSeed'
27 //      is not on the curve
28 //      TPM_RC_HASH           'objectPublic' does not have a valid nameAlg
29 //      TPM_RC_INSUFFICIENT    'inSymSeed' is nonempty and failed to retrieve ECC
30 //                             point from the secret; or unmarshaling sensitive value
31 //                             from 'duplicate' failed the result of 'inSymSeed'
32 //                             decryption
33 //      TPM_RC_INTEGRITY       'duplicate' integrity is broken
34 //      TPM_RC_KDF             'objectPublic' representing decrypting keyed hash
35 //                             object specifies invalid KDF
36 //      TPM_RC_KEY             inconsistent parameters of 'objectPublic'; or
37 //                             'inSymSeed' is nonempty and 'parentHandle' does not
38 //                             reference a key of supported type; or
39 //                             invalid key size in 'objectPublic' representing an
40 //                             asymmetric key
41 //      TPM_RC_NO_RESULT       'inSymSeed' is nonempty and multiplication resulted in
42 //                             ECC point at infinity
43 //      TPM_RC_OBJECT_MEMORY   no available object slot
44 //      TPM_RC_SCHEME          inconsistent attributes 'decrypt', 'sign',
45 //                             'restricted' and key's scheme ID in 'objectPublic';
46 //                             or hash algorithm is inconsistent with the scheme ID
47 //                             for keyed hash object
48 //      TPM_RC_SIZE            'authPolicy' size does not match digest size of the
49 //                             name algorithm in 'objectPublic'; or
50 //                             'symmetricAlg' and 'encryptionKey' have different
51 //                             sizes; or
52 //                             'inSymSeed' is nonempty and its size is not
53 //                             consistent with the type of 'parentHandle'; or
54 //                             unmarshaling sensitive value from 'duplicate' failed
55 //      TPM_RC_SYMMETRIC       'objectPublic' is either a storage key with no
56 //                             symmetric algorithm or a non-storage key with
57 //                             symmetric algorithm different from TPM_ALG_NULL
58 //      TPM_RC_TYPE            unsupported type of 'objectPublic'; or
59 //                             'parentHandle' is not a storage key; or
60 //                             only the public portion of 'parentHandle' is loaded;
61 //                             or 'objectPublic' and 'duplicate' are of different
62 //                             types
63 //      TPM_RC_VALUE           nonempty 'inSymSeed' and its numeric value is
64 //                             greater than the modulus of the key referenced by
65 //                             'parentHandle' or 'inSymSeed' is larger than the
66 //                             size of the digest produced by the name algorithm of
67 //                             the symmetric key referenced by 'parentHandle'
68 TPM_RC
69 TPM2_Import(Import_In* in, // IN: input parameter list
70             Import_Out* out // OUT: output parameter list
71 )
72 {
73     TPM_RC      result = TPM_RC_SUCCESS;
74     OBJECT*     parentObject;
75     TPM2B_DATA   data; // symmetric key
76     TPMT_SENSITIVE sensitive;
77     TPM2B_NAME   name;
78     TPMA_OBJECT  attributes;
79     UINT16       innerKeySize = 0; // encrypt key size for inner
80                                     // wrapper
81
82     // Input Validation
83     // to save typing
84     attributes = in->objectPublic.publicArea.objectAttributes;
85     // FixedTPM and fixedParent must be CLEAR
86     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
87        || IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent))
88         return TPM_RCS_ATTRIBUTES + RC_Import_objectPublic;
89
90     // Get parent pointer
91     parentObject = HandleToObject(in->parentHandle);

```



```

92
93     if(!ObjectIsParent(parentObject))
94         return TPM_RCS_TYPE + RC_Import_parentHandle;
95
96     if(in->symmetricAlg.algorithm != TPM_ALG_NULL)
97     {
98         // Get inner wrap key size
99         innerKeySize = in->symmetricAlg.keyBits.sym;
100         // Input symmetric key must match the size of algorithm.
101         if(in->encryptionKey.t.size != (innerKeySize + 7) / 8)
102             return TPM_RCS_SIZE + RC_Import_encryptionKey;
103     }
104     else
105     {
106         // If input symmetric algorithm is NULL, input symmetric key size must
107         // be 0 as well
108         if(in->encryptionKey.t.size != 0)
109             return TPM_RCS_SIZE + RC_Import_encryptionKey;
110         // If encryptedDuplication is SET, then the object must have an inner
111         // wrapper
112         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
113             return TPM_RCS_ATTRIBUTES + RC_Import_encryptionKey;
114     }
115     // See if there is an outer wrapper
116     if(in->inSymSeed.t.size != 0)
117     {
118         // in->inParentHandle is a parent, but in order to decrypt an outer wrapper,
119         // it must be able to do key exchange and a symmetric key can't do that.
120         if(parentObject->publicArea.type == TPM_ALG_SYMCIPHER)
121             return TPM_RCS_TYPE + RC_Import_parentHandle;
122
123         // Decrypt input secret data via asymmetric decryption. TPM_RC_ATTRIBUTES,
124         // TPM_RC_ECC_POINT, TPM_RC_INSUFFICIENT, TPM_RC_KEY, TPM_RC_NO_RESULT,
125         // TPM_RC_SIZE, TPM_RC_VALUE may be returned at this point
126         result = CryptSecretDecrypt(
127             parentObject, NULL, DUPLICATE_STRING, &in->inSymSeed, &data);
128         pAssert(result != TPM_RC_BINDING);
129         if(result != TPM_RC_SUCCESS)
130             return RcSafeAddToResult(result, RC_Import_inSymSeed);
131     }
132     else
133     {
134         // If encryptedDuplication is set, then the object must have an outer
135         // wrapper
136         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
137             return TPM_RCS_ATTRIBUTES + RC_Import_inSymSeed;
138         data.t.size = 0;
139     }
140     // Compute name of object
141     PublicMarshalAndComputeName(&(in->objectPublic.publicArea), &name);
142     if(name.t.size == 0)
143         return TPM_RCS_HASH + RC_Import_objectPublic;
144
145     // Retrieve sensitive from private.
146     // TPM_RC_INSUFFICIENT, TPM_RC_INTEGRITY, TPM_RC_SIZE may be returned here.
147     result = DuplicateToSensitive(&in->duplicate.b,
148                                  &name.b,
149                                  parentObject,
150                                  in->objectPublic.publicArea.nameAlg,
151                                  &data.b,
152                                  &in->symmetricAlg,
153                                  &in->encryptionKey.b,
154                                  &sensitive);
155     if(result != TPM_RC_SUCCESS)
156         return RcSafeAddToResult(result, RC_Import_duplicate);
157

```

```

158 // If the parent of this object has fixedTPM SET, then validate this
159 // object as if it were being loaded so that validation can be skipped
160 // when it is actually loaded.
161 if(IS_ATTRIBUTE(parentObject->publicArea.objectAttributes, TPMA_OBJECT, fixedTPM))
162 {
163     result = ObjectLoad(NULL,
164                         NULL,
165                         &in->objectPublic.publicArea,
166                         &sensitive,
167                         RC_Import_objectPublic,
168                         RC_Import_duplicate,
169                         NULL);
170 }
171 // Command output
172 if(result == TPM_RC_SUCCESS)
173 {
174     // Prepare output private data from sensitive
175     SensitiveToPrivate(&sensitive,
176                       &name,
177                       parentObject,
178                       in->objectPublic.publicArea.nameAlg,
179                       &out->outPrivate);
180 }
181 return result;
182 }
183
184 #endif // CC_Import

```

7.40 /tpm/src/command/Duplication/Rewrap.c

```

1  #include "Tpm.h"
2  #include "Rewrap_fp.h"
3
4  #if CC_Rewrap // Conditional expansion of this file
5
6  # include "Object_spt_fp.h"
7
8  /*(See part 3 specification)
9  // This command allows the TPM to serve in the role as an MA.
10 */
11 // Return Type: TPM_RC
12 // TPM_RC_ATTRIBUTES 'newParent' is not a decryption key
13 // TPM_RC_HANDLE 'oldParent' is not consistent with inSymSeed
14 // TPM_RC_INTEGRITY the integrity check of 'inDuplicate' failed
15 // TPM_RC_KEY for an ECC key, the public key is not on the curve
16 // of the curve ID
17 // TPM_RC_KEY_SIZE the decrypted input symmetric key size
18 // does not match the symmetric algorithm
19 // key size of 'oldParent'
20 // TPM_RC_TYPE 'oldParent' is not a storage key, or 'newParent'
21 // is not a storage key
22 // TPM_RC_VALUE for an 'oldParent'; RSA key, the data to be decrypted
23 // is greater than the public exponent
24 // Unmarshal errors errors during unmarshaling the input
25 // encrypted buffer to a ECC public key, or
26 // unmarshal the private buffer to 'sensitive'
27 TPM_RC
28 TPM2_Rewrap(Rewrap_In* in, // IN: input parameter list
29            Rewrap_Out* out // OUT: output parameter list
30 )
31 {
32     TPM_RC result = TPM_RC_SUCCESS;
33     TPM2B_DATA data; // symmetric key
34     UINT16 hashSize = 0;
35     TPM2B_PRIVATE privateBlob; // A temporary private blob

```

```

36                                     // to transit between old
37                                     // and new wrappers
38                                     // Input Validation
39 if((in->inSymSeed.t.size == 0 && in->oldParent != TPM_RH_NULL)
40    || (in->inSymSeed.t.size != 0 && in->oldParent == TPM_RH_NULL))
41     return TPM_RCS_HANDLE + RC_Rewrap_oldParent;
42 if(in->oldParent != TPM_RH_NULL)
43 {
44     OBJECT* oldParent = HandleToObject(in->oldParent);
45
46     // old parent key must be a storage object
47     if(!ObjectIsStorage(in->oldParent))
48         return TPM_RCS_TYPE + RC_Rewrap_oldParent;
49     // Decrypt input secret data via asymmetric decryption. A
50     // TPM_RC_VALUE, TPM_RC_KEY or unmarshal errors may be returned at this
51     // point
52     result = CryptSecretDecrypt(
53         oldParent, NULL, DUPLICATE_STRING, &in->inSymSeed, &data);
54     if(result != TPM_RC_SUCCESS)
55         return TPM_RCS_VALUE + RC_Rewrap_inSymSeed;
56     // Unwrap Outer
57     result = UnwrapOuter(oldParent,
58                          &in->name.b,
59                          oldParent->publicArea.nameAlg,
60                          &data.b,
61                          FALSE,
62                          in->inDuplicate.t.size,
63                          in->inDuplicate.t.buffer);
64     if(result != TPM_RC_SUCCESS)
65         return RcSafeAddToResult(result, RC_Rewrap_inDuplicate);
66     // Copy unwrapped data to temporary variable, remove the integrity field
67     hashSize =
68         sizeof(UINT16) + CryptHashGetDigestSize(oldParent->publicArea.nameAlg);
69     privateBlob.t.size = in->inDuplicate.t.size - hashSize;
70     pAssert(privateBlob.t.size <= sizeof(privateBlob.t.buffer));
71     MemoryCopy(privateBlob.t.buffer,
72               in->inDuplicate.t.buffer + hashSize,
73               privateBlob.t.size);
74 }
75 else
76 {
77     // No outer wrap from input blob. Direct copy.
78     privateBlob = in->inDuplicate;
79 }
80 if(in->newParent != TPM_RH_NULL)
81 {
82     OBJECT* newParent;
83     newParent = HandleToObject(in->newParent);
84
85     // New parent must be a storage object
86     if(!ObjectIsStorage(in->newParent))
87         return TPM_RCS_TYPE + RC_Rewrap_newParent;
88     // Make new encrypt key and its associated secret structure. A
89     // TPM_RC_VALUE error may be returned at this point if RSA algorithm is
90     // enabled in TPM
91     out->outSymSeed.t.size = sizeof(out->outSymSeed.t.secret);
92     result =
93         CryptSecretEncrypt(newParent, DUPLICATE_STRING, &data, &out->outSymSeed);
94     if(result != TPM_RC_SUCCESS)
95         return result;
96     // Copy temporary variable to output, reserve the space for integrity
97     hashSize =
98         sizeof(UINT16) + CryptHashGetDigestSize(newParent->publicArea.nameAlg);
99     // Make sure that everything fits into the output buffer
100    // Note: this is mostly only an issue if there was no outer wrapper on
101    // 'inDuplicate'. It could be as large as a TPM2B_PRIVATE buffer. If we add

```

```

102     // a digest for an outer wrapper, it won't fit anymore.
103     if((privateBlob.t.size + hashSize) > sizeof(out->outDuplicate.t.buffer))
104         return TPM_RCS_VALUE + RC_Rewrap_inDuplicate;
105     // Command output
106     out->outDuplicate.t.size = privateBlob.t.size;
107     pAssert(privateBlob.t.size <= sizeof(out->outDuplicate.t.buffer) - hashSize);
108     MemoryCopy(out->outDuplicate.t.buffer + hashSize,
109               privateBlob.t.buffer,
110               privateBlob.t.size);
111     // Produce outer wrapper for output
112     out->outDuplicate.t.size = ProduceOuterWrap(newParent,
113                                               &in->name.b,
114                                               newParent->publicArea.nameAlg,
115                                               &data.b,
116                                               FALSE,
117                                               out->outDuplicate.t.size,
118                                               out->outDuplicate.t.buffer);
119 }
120 else // New parent is a null key so there is no seed
121 {
122     out->outSymSeed.t.size = 0;
123
124     // Copy privateBlob directly
125     out->outDuplicate = privateBlob;
126 }
127 return TPM_RC_SUCCESS;
128 }
129
130 #endif // CC_Rewrap

```

7.41 /tpm/src/command/EA/PolicyAuthorize.c

```

1  #include "Tpm.h"
2  #include "PolicyAuthorize_fp.h"
3
4  #if CC_PolicyAuthorize // Conditional expansion of this file
5
6  # include "Policy_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Change policy by a signature from authority
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_HASH      hash algorithm in 'keyName' is not supported
13 //     TPM_RC_SIZE      'keyName' is not the correct size for its hash algorithm
14 //     TPM_RC_VALUE     the current policyDigest of 'policySession' does not
15 //                      match 'approvedPolicy'; or 'checkTicket' doesn't match
16 //                      the provided values
17 TPM_RC
18 TPM2_PolicyAuthorize(PolicyAuthorize_In* in // IN: input parameter list
19 )
20 {
21     TPM_RC      result = TPM_RC_SUCCESS;
22     SESSION*    session;
23     TPM2B_DIGEST authHash;
24     HASH_STATE  hashState;
25     TPMT_TK_VERIFIED ticket;
26     TPM_ALG_ID  hashAlg;
27     UINT16      digestSize;
28
29     // Input Validation
30
31     // Get pointer to the session structure
32     session = SessionGet(in->policySession);
33

```

```

34     if(in->keySign.t.size < 2)
35     {
36         return TPM_RCS_SIZE + RC_PolicyAuthorize_keySign;
37     }
38
39     // Extract from the Name of the key, the algorithm used to compute its Name
40     hashAlg = BYTE_ARRAY_TO_UINT16(in->keySign.t.name);
41
42     // 'keySign' parameter needs to use a supported hash algorithm, otherwise
43     // can't tell how large the digest should be
44     if(!CryptHashIsValidAlg(hashAlg, FALSE))
45         return TPM_RCS_HASH + RC_PolicyAuthorize_keySign;
46
47     digestSize = CryptHashGetDigestSize(hashAlg);
48     if(digestSize != (in->keySign.t.size - 2))
49         return TPM_RCS_SIZE + RC_PolicyAuthorize_keySign;
50
51     //If this is a trial policy, skip all validations
52     if(session->attributes.isTrialPolicy == CLEAR)
53     {
54         // Check that "approvedPolicy" matches the current value of the
55         // policyDigest in policy session
56         if(!MemoryEqual2B(&session->u2.policyDigest.b, &in->approvedPolicy.b))
57             return TPM_RCS_VALUE + RC_PolicyAuthorize_approvedPolicy;
58
59         // Validate ticket TPMT_TK_VERIFIED
60         // Compute aHash. The authorizing object sign a digest
61         // aHash := hash(approvedPolicy || policyRef).
62         // Start hash
63         authHash.t.size = CryptHashStart(&hashState, hashAlg);
64
65         // add approvedPolicy
66         CryptDigestUpdate2B(&hashState, &in->approvedPolicy.b);
67
68         // add policyRef
69         CryptDigestUpdate2B(&hashState, &in->policyRef.b);
70
71         // complete hash
72         CryptHashEnd2B(&hashState, &authHash.b);
73
74         // re-compute TPMT_TK_VERIFIED
75         result = TicketComputeVerified(
76             in->checkTicket.hierarchy, &authHash, &in->keySign, &ticket);
77         if(result != TPM_RC_SUCCESS)
78             return result;
79
80         // Compare ticket digest. If not match, return error
81         if(!MemoryEqual2B(&in->checkTicket.digest.b, &ticket.digest.b))
82             return TPM_RCS_VALUE + RC_PolicyAuthorize_checkTicket;
83     }
84
85     // Internal Data Update
86
87     // Set policyDigest to zero digest
88     PolicyDigestClear(session);
89
90     // Update policyDigest
91     PolicyContextUpdate(
92         TPM_CC_PolicyAuthorize, &in->keySign, &in->policyRef, NULL, 0, session);
93
94     return TPM_RC_SUCCESS;
95 }
96
97 #endif // CC_PolicyAuthorize

```

7.42 /tpm/src/command/EA/PolicyAuthorizeNV.c

```

1  #include "Tpm.h"
2
3  #if CC_PolicyAuthorizeNV // Conditional expansion of this file
4
5  # include "PolicyAuthorizeNV_fp.h"
6  # include "Policy_spt_fp.h"
7  # include "Marshal.h"
8
9  /*(See part 3 specification)
10 // Change policy by a signature from authority
11 */
12 // Return Type: TPM_RC
13 //     TPM_RC_HASH      hash algorithm in 'keyName' is not supported or is not
14 //                      the same as the hash algorithm of the policy session
15 //     TPM_RC_SIZE      'keyName' is not the correct size for its hash algorithm
16 //     TPM_RC_VALUE      the current policyDigest of 'policySession' does not
17 //                      match 'approvedPolicy'; or 'checkTicket' doesn't match
18 //                      the provided values
19 TPM_RC
20 TPM2_PolicyAuthorizeNV(PolicyAuthorizeNV_In* in)
21 {
22     SESSION*   session;
23     TPM_RC     result;
24     NV_REF     locator;
25     NV_INDEX*  nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
26     TPM2B_NAME name;
27     TPMT_HA    policyInNv;
28     BYTE       nvTemp[sizeof(TPMT_HA)];
29     BYTE*      buffer = nvTemp;
30     INT32      size;
31
32     // Input Validation
33     // Get pointer to the session structure
34     session = SessionGet(in->policySession);
35
36     // Skip checks if this is a trial policy
37     if(!session->attributes.isTrialPolicy)
38     {
39         // Check the authorizations for reading
40         // Common read access checks. NvReadAccessChecks() returns
41         // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
42         // error may be returned at this point
43         result = NvReadAccessChecks(
44             in->authHandle, in->nvIndex, nvIndex->publicArea.attributes);
45         if(result != TPM_RC_SUCCESS)
46             return result;
47
48         // Read the contents of the index into a temp buffer
49         size = MIN(nvIndex->publicArea.dataSize, sizeof(TPMT_HA));
50         NvGetIndexData(nvIndex, locator, 0, (UINT16)size, nvTemp);
51
52         // Unmarshal the contents of the buffer into the internal format of a
53         // TPMT_HA so that the hash and digest elements can be accessed from the
54         // structure rather than the byte array that is in the Index (written by
55         // user of the Index).
56         result = TPMT_HA_Unmarshal(&policyInNv, &buffer, &size, FALSE);
57         if(result != TPM_RC_SUCCESS)
58             return result;
59
60         // Verify that the hash is the same
61         if(policyInNv.hashAlg != session->authHashAlg)
62             return TPM_RC_HASH;
63
64         // See if the contents of the digest in the Index matches the value

```



```

65     // in the policy
66     if(!MemoryEqual(&policyInNv.digest,
67                     &session->u2.policyDigest.t.buffer,
68                     session->u2.policyDigest.t.size))
69         return TPM_RC_VALUE;
70 }
71
72 // Internal Data Update
73
74 // Set policyDigest to zero digest
75 PolicyDigestClear(session);
76
77 // Update policyDigest
78 PolicyContextUpdate(TPM_CC_PolicyAuthorizeNV,
79                     EntityGetName(in->nvIndex, &name),
80                     NULL,
81                     NULL,
82                     0,
83                     session);
84
85 return TPM_RC_SUCCESS;
86 }
87
88 #endif // CC_PolicyAuthorize

```

7.43 /tpm/src/command/EA/PolicyAuthValue.c

```

1  #include "Tpm.h"
2  #include "PolicyAuthValue_fp.h"
3
4  #if CC_PolicyAuthValue // Conditional expansion of this file
5
6  # include "Policy_spt_fp.h"
7
8  /*(See part 3 specification)
9  // allows a policy to be bound to the authorization value of the authorized
10 // object
11 */
12 TPM_RC
13 TPM2_PolicyAuthValue(PolicyAuthValue_In* in // IN: input parameter list
14 )
15 {
16     SESSION* session;
17     TPM_CC commandCode = TPM_CC_PolicyAuthValue;
18     HASH_STATE hashState;
19
20     // Internal Data Update
21
22     // Get pointer to the session structure
23     session = SessionGet(in->policySession);
24
25     // Update policy hash
26     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyAuthValue)
27     // Start hash
28     CryptHashStart(&hashState, session->authHashAlg);
29
30     // add old digest
31     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
32
33     // add commandCode
34     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
35
36     // complete the hash and get the results
37     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
38

```

```

39     // update isAuthValueNeeded bit in the session context
40     session->attributes.isAuthValueNeeded = SET;
41     session->attributes.isPasswordNeeded = CLEAR;
42
43     return TPM_RC_SUCCESS;
44 }
45
46 #endif // CC_PolicyAuthValue

```

7.44 /tpm/src/command/EA/PolicyCapability.c

```

1  #include "Tpm.h"
2  #include "PolicyCapability_fp.h"
3  #include "Policy_spt_fp.h"
4  #include "ACT_spt_fp.h"
5  #include "AlgorithmCap_fp.h"
6  #include "CommandAudit_fp.h"
7  #include "CommandCodeAttributes_fp.h"
8  #include "CryptEccMain_fp.h"
9  #include "Handle_fp.h"
10 #include "NvDynamic_fp.h"
11 #include "Object_fp.h"
12 #include "PCR_fp.h"
13 #include "PP_fp.h"
14 #include "PropertyCap_fp.h"
15 #include "Session_fp.h"
16
17 #if CC_PolicyCapability // Conditional expansion of this file
18
19 /*(See part 3 specification)
20 // This command performs an immediate policy assertion against the current
21 // value of a TPM Capability.
22 */
23 // Return Type: TPM_RC
24 //     TPM_RC_HANDLE      value of 'property' is in an unsupported handle range
25 //                         for the TPM_CAP_HANDLES 'capability' value
26 //     TPM_RC_VALUE       invalid 'capability'; or 'property' is not 0 for the
27 //                         TPM_CAP_PCERS 'capability' value
28 //     TPM_RC_SIZE        'operandB' is larger than the size of the capability
29 //                         data minus 'offset'.
30 TPM_RC
31 TPM2_PolicyCapability(PolicyCapability_In* in // IN: input parameter list
32 )
33 {
34     union
35     {
36         TPMS_ALG_PROPERTY      alg;
37         TPM_HANDLE             handle;
38         TPMA_CC                commandAttributes;
39         TPM_CC                 command;
40         TPMS_TAGGED_PCR_SELECT pcrSelect;
41         TPMS_TAGGED_PROPERTY   tpmProperty;
42     # if ALG_ECC
43         TPM_ECC_CURVE curve;
44     # endif // ALG_ECC
45         TPMS_TAGGED_POLICY policy;
46     # if ACT_SUPPORT
47         TPMS_ACT_DATA act;
48     # endif // ACT_SUPPORT
49     } propertyUnion;
50
51     SESSION*      session;
52     BYTE          propertyData[sizeof(propertyUnion)];
53     UINT16        propertySize = 0;
54     BYTE*         buffer       = propertyData;

```

```

55     INT32      bufferSize   = sizeof(propertyData);
56     TPM_CC     commandCode  = TPM_CC_PolicyCapability;
57     HASH_STATE hashState;
58     TPM2B_DIGEST argHash;
59
60     // Get pointer to the session structure
61     session = SessionGet(in->policySession);
62
63     if(session->attributes.isTrialPolicy == CLEAR)
64     {
65         switch(in->capability)
66         {
67             case TPM_CAP_ALGS:
68                 if(AlgorithmCapGetOneImplemented((TPM_ALG_ID)in->property,
69                                                     &propertyUnion.alg))
70                 {
71                     propertySize = TPMS_ALG_PROPERTY_Marshal(
72                                     &propertyUnion.alg, &buffer, &bufferSize);
73                 }
74                 break;
75             case TPM_CAP_HANDLES:
76                 BOOL foundHandle = FALSE;
77                 switch(HandleGetType((TPM_HANDLE)in->property))
78                 {
79                     case TPM_HT_TRANSIENT:
80                         foundHandle = ObjectCapGetOneLoaded((TPM_HANDLE)in->property);
81                         break;
82                     case TPM_HT_PERSISTENT:
83                         foundHandle = NvCapGetOnePersistent((TPM_HANDLE)in->property);
84                         break;
85                     case TPM_HT_NV_INDEX:
86                         foundHandle = NvCapGetOneIndex((TPM_HANDLE)in->property);
87                         break;
88                     case TPM_HT_LOADED_SESSION:
89                         foundHandle =
90                             SessionCapGetOneLoaded((TPM_HANDLE)in->property);
91                         break;
92                     case TPM_HT_SAVED_SESSION:
93                         foundHandle = SessionCapGetOneSaved((TPM_HANDLE)in->property);
94                         break;
95                     case TPM_HT_PCR:
96                         foundHandle = PCRCapGetOneHandle((TPM_HANDLE)in->property);
97                         break;
98                     case TPM_HT_PERMANENT:
99                         foundHandle =
100                             PermanentCapGetOneHandle((TPM_HANDLE)in->property);
101                         break;
102                     default:
103                         // Unsupported input handle type
104                         return TPM_RCS_HANDLE + RC_PolicyCapability_property;
105                         break;
106                 }
107                 if(foundHandle)
108                 {
109                     TPM_HANDLE handle = (TPM_HANDLE)in->property;
110                     propertySize = TPM_HANDLE_Marshal(&handle, &buffer, &bufferSize);
111                 }
112                 break;
113             case TPM_CAP_COMMANDS:
114                 if(CommandCapGetOneCC((TPM_CC)in->property,
115                                         &propertyUnion.commandAttributes))
116                 {
117                     propertySize = TPMA_CC_Marshal(
118                                     &propertyUnion.commandAttributes, &buffer, &bufferSize);
119                 }
120                 break;

```

```

121     case TPM_CAP_PP_COMMANDS:
122         if (PhysicalPresenceCapGetOneCC((TPM_CC)in->property))
123         {
124             TPM_CC cc = (TPM_CC)in->property;
125             propertySize = TPM_CC_Marshal(&cc, &buffer, &bufferSize);
126         }
127         break;
128     case TPM_CAP_AUDIT_COMMANDS:
129         if (CommandAuditCapGetOneCC((TPM_CC)in->property))
130         {
131             TPM_CC cc = (TPM_CC)in->property;
132             propertySize = TPM_CC_Marshal(&cc, &buffer, &bufferSize);
133         }
134         break;
135     // NOTE: TPM_CAP_PCERS can't work for PolicyCapability since CAP_PCERS
136     // requires property to be 0 and always returns all the PCR banks.
137     case TPM_CAP_PCR_PROPERTIES:
138         if (PCRGetProperty((TPM_PT_PCR)in->property, &propertyUnion.pcrSelect))
139         {
140             propertySize = TPMS_TAGGED_PCR_SELECT_Marshal(
141                 &propertyUnion.pcrSelect, &buffer, &bufferSize);
142         }
143         break;
144     case TPM_CAP_TPM_PROPERTIES:
145         if (TPMCapGetOneProperty((TPM_PT)in->property,
146             &propertyUnion.tpmProperty))
147         {
148             propertySize = TPMS_TAGGED_PROPERTY_Marshal(
149                 &propertyUnion.tpmProperty, &buffer, &bufferSize);
150         }
151         break;
152 # if ALG_ECC
153     case TPM_CAP_ECC_CURVES:
154         TPM_ECC_CURVE curve = (TPM_ECC_CURVE)in->property;
155         if (CryptCapGetOneECCCurve(curve))
156         {
157             propertySize =
158                 TPM_ECC_CURVE_Marshal(&curve, &buffer, &bufferSize);
159         }
160         break;
161 # endif // ALG_ECC
162     case TPM_CAP_AUTH_POLICIES:
163         if (HandleGetType((TPM_HANDLE)in->property) != TPM_HT_PERMANENT)
164             return TPM_RCS_VALUE + RC_PolicyCapability_property;
165         if (PermanentHandleGetOnePolicy((TPM_HANDLE)in->property,
166             &propertyUnion.policy))
167         {
168             propertySize = TPMS_TAGGED_POLICY_Marshal(
169                 &propertyUnion.policy, &buffer, &bufferSize);
170         }
171         break;
172 # if ACT_SUPPORT
173     case TPM_CAP_ACT:
174         if (((TPM_RH)in->property < TPM_RH_ACT_0)
175             || ((TPM_RH)in->property > TPM_RH_ACT_F))
176             return TPM_RCS_VALUE + RC_PolicyCapability_property;
177         if (ActGetOneCapability((TPM_HANDLE)in->property, &propertyUnion.act))
178         {
179             propertySize = TPMS_ACT_DATA_Marshal(
180                 &propertyUnion.act, &buffer, &bufferSize);
181         }
182         break;
183 # endif // ACT_SUPPORT
184     case TPM_CAP_VENDOR_PROPERTY:
185         // vendor property is not implemented
186     default:

```

```

187         // Unsupported TPM_CAP value
188         return TPM_RCS_VALUE + RC_PolicyCapability_capability;
189         break;
190     }
191
192     if(propertySize == 0)
193     {
194         // A property that doesn't exist trivially satisfies NEQ, and
195         // trivially can't satisfy any other operation.
196         if(in->operation != TPM_EO_NEQ)
197         {
198             return TPM_RC_POLICY;
199         }
200     }
201     else
202     {
203         // The property was found, so we need to perform the comparison.
204
205         // Make sure that offset is within range
206         if(in->offset > propertySize)
207         {
208             return TPM_RCS_VALUE + RC_PolicyCapability_offset;
209         }
210
211         // Property data size should not be smaller than input operandB size
212         if((propertySize - in->offset) < in->operandB.t.size)
213         {
214             return TPM_RCS_SIZE + RC_PolicyCapability_operandB;
215         }
216
217         if(!PolicySptCheckCondition(in->operation,
218                                     propertyData + in->offset,
219                                     in->operandB.t.buffer,
220                                     in->operandB.t.size))
221         {
222             return TPM_RC_POLICY;
223         }
224     }
225 }
226 // Internal Data Update
227
228 // Start argument hash
229 argHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
230
231 // add operandB
232 CryptDigestUpdate2B(&hashState, &in->operandB.b);
233
234 // add offset
235 CryptDigestUpdateInt(&hashState, sizeof(UINT16), in->offset);
236
237 // add operation
238 CryptDigestUpdateInt(&hashState, sizeof(TPM_EO), in->operation);
239
240 // add capability
241 CryptDigestUpdateInt(&hashState, sizeof(TPM_CAP), in->capability);
242
243 // add property
244 CryptDigestUpdateInt(&hashState, sizeof(UINT32), in->property);
245
246 // complete argument digest
247 CryptHashEnd2B(&hashState, &argHash.b);
248
249 // Update policyDigest
250 // Start digest
251 CryptHashStart(&hashState, session->authHashAlg);
252

```

```

253     // add old digest
254     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
255
256     // add commandCode
257     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
258
259     // add argument digest
260     CryptDigestUpdate2B(&hashState, &argHash.b);
261
262     // complete the digest
263     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
264
265     return TPM_RC_SUCCESS;
266 }
267
268 #endif // CC_PolicyCapability

```

7.45 /tpm/src/command/EA/PolicyCommandCode.c

```

1  #include "Tpm.h"
2  #include "PolicyCommandCode_fp.h"
3
4  #if CC_PolicyCommandCode // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Add a Command Code restriction to the policyDigest
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_VALUE      'commandCode' of 'policySession' previously set to
11 //                        a different value
12
13 TPM_RC
14 TPM2_PolicyCommandCode(PolicyCommandCode_In* in // IN: input parameter list
15 )
16 {
17     SESSION* session;
18     TPM_CC commandCode = TPM_CC_PolicyCommandCode;
19     HASH_STATE hashState;
20
21     // Input validation
22
23     // Get pointer to the session structure
24     session = SessionGet(in->policySession);
25
26     if(session->commandCode != 0 && session->commandCode != in->code)
27         return TPM_RCS_VALUE + RC_PolicyCommandCode_code;
28     if(CommandCodeToCommandIndex(in->code) == UNIMPLEMENTED_COMMAND_INDEX)
29         return TPM_RCS_POLICY_CC + RC_PolicyCommandCode_code;
30
31     // Internal Data Update
32     // Update policy hash
33     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCommandCode || code)
34     // Start hash
35     CryptHashStart(&hashState, session->authHashAlg);
36
37     // add old digest
38     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
39
40     // add commandCode
41     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
42
43     // add input commandCode
44     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), in->code);
45
46     // complete the hash and get the results

```



```

47     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
48
49     // update commandCode value in session context
50     session->commandCode = in->code;
51
52     return TPM_RC_SUCCESS;
53 }
54
55 #endif // CC_PolicyCommandCode

```

7.46 /tpm/src/command/EA/PolicyCounterTimer.c

```

1  #include "Tpm.h"
2  #include "PolicyCounterTimer_fp.h"
3
4  #if CC_PolicyCounterTimer // Conditional expansion of this file
5
6  # include "Policy_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Add a conditional gating of a policy based on the contents of the
10 // TPMS_TIME_INFO structure.
11 */
12 // Return Type: TPM_RC
13 //     TPM_RC_POLICY          the comparison of the selected portion of the
14 //                             TPMS_TIME_INFO with 'operandB' failed
15 //     TPM_RC_RANGE          'offset' + 'size' exceed size of TPMS_TIME_INFO
16 //                             structure
17 TPM_RC
18 TPM2_PolicyCounterTimer(PolicyCounterTimer_In* in // IN: input parameter list
19 )
20 {
21     SESSION*      session;
22     TIME_INFO     infoData; // data buffer of TPMS_TIME_INFO
23     BYTE*         pInfoData = (BYTE*)&infoData;
24     UINT16         infoDataSize;
25     TPM_CC         commandCode = TPM_CC_PolicyCounterTimer;
26     HASH_STATE     hashState;
27     TPM2B_DIGEST   argHash;
28
29     // Input Validation
30     // Get a marshaled time structure
31     infoDataSize = TimeGetMarshaled(&infoData);
32     // Make sure that the referenced stays within the bounds of the structure.
33     // NOTE: the offset checks are made even for a trial policy because the policy
34     // will not make any sense if the references are out of bounds of the timer
35     // structure.
36     if(in->offset > infoDataSize)
37         return TPM_RCS_VALUE + RC_PolicyCounterTimer_offset;
38     if((UINT32)in->offset + (UINT32)in->operandB.t.size > infoDataSize)
39         return TPM_RCS_RANGE;
40     // Get pointer to the session structure
41     session = SessionGet(in->policySession);
42
43     //If this is a trial policy, skip the check to see if the condition is met.
44     if(session->attributes.isTrialPolicy == CLEAR)
45     {
46         // If the command is going to use any part of the counter or timer, need
47         // to verify that time is advancing.
48         // The time and clock vales are the first two 64-bit values in the clock
49         if(in->offset < sizeof(UINT64) + sizeof(UINT64))
50         {
51             // Using Clock or Time so see if clock is running. Clock doesn't
52             // run while NV is unavailable.
53             // TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned here.

```

```

54     RETURN_IF_NV_IS_NOT_AVAILABLE;
55 }
56 // offset to the starting position
57 pInfoData = (BYTE*)infoData;
58 // Check to see if the condition is valid
59 if(!PolicySptCheckCondition(in->operation,
60                             pInfoData + in->offset,
61                             in->operandB.t.buffer,
62                             in->operandB.t.size))
63     return TPM_RC_POLICY;
64 }
65 // Internal Data Update
66 // Start argument list hash
67 argHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
68 // add operandB
69 CryptDigestUpdate2B(&hashState, &in->operandB.b);
70 // add offset
71 CryptDigestUpdateInt(&hashState, sizeof(UINT16), in->offset);
72 // add operation
73 CryptDigestUpdateInt(&hashState, sizeof(TPM_EO), in->operation);
74 // complete argument hash
75 CryptHashEnd2B(&hashState, &argHash.b);
76
77 // update policyDigest
78 // start hash
79 CryptHashStart(&hashState, session->authHashAlg);
80
81 // add old digest
82 CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
83
84 // add commandCode
85 CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
86
87 // add argument digest
88 CryptDigestUpdate2B(&hashState, &argHash.b);
89
90 // complete the digest
91 CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
92
93 return TPM_RC_SUCCESS;
94 }
95
96 #endif // CC_PolicyCounterTimer

```

7.47 /tpm/src/command/EA/PolicyCpHash.c

```

1  #include "Tpm.h"
2  #include "PolicyCpHash_fp.h"
3
4  #if CC_PolicyCpHash // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Add a cpHash restriction to the policyDigest
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_CPHASH           cpHash of 'policySession' has previously been set
11 //                               to a different value
12 //     TPM_RC_SIZE             'cpHashA' is not the size of a digest produced
13 //                               by the hash algorithm associated with
14 //                               'policySession'
15 TPM_RC
16 TPM2_PolicyCpHash(PolicyCpHash_In* in // IN: input parameter list
17 )
18 {
19     SESSION* session;

```

```

20     TPM_CC      commandCode = TPM_CC_PolicyCpHash;
21     HASH_STATE hashState;
22
23     // Input Validation
24
25     // Get pointer to the session structure
26     session = SessionGet(in->policySession);
27
28     // A valid cpHash must have the same size as session hash digest
29     // NOTE: the size of the digest can't be zero because TPM_ALG_NULL
30     // can't be used for the authHashAlg.
31     if(in->cpHashA.t.size != CryptHashGetDigestSize(session->authHashAlg))
32         return TPM_RCS_SIZE + RC_PolicyCpHash_cpHashA;
33
34     // error if the cpHash in session context is not empty and is not the same
35     // as the input or is not a cpHash
36     if((IsCpHashUnionOccupied(session->attributes)
37         && (!session->attributes.isCpHashDefined
38             || !MemoryEqual2B(&in->cpHashA.b, &session->u1.cpHash.b)))
39         return TPM_RC_CPHASH;
40
41     // Internal Data Update
42
43     // Update policy hash
44     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCpHash || cpHashA)
45     // Start hash
46     CryptHashStart(&hashState, session->authHashAlg);
47
48     // add old digest
49     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
50
51     // add commandCode
52     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
53
54     // add cpHashA
55     CryptDigestUpdate2B(&hashState, &in->cpHashA.b);
56
57     // complete the digest and get the results
58     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
59
60     // update cpHash in session context
61     session->u1.cpHash = in->cpHashA;
62     session->attributes.isCpHashDefined = SET;
63
64     return TPM_RC_SUCCESS;
65 }
66
67 #endif // CC_PolicyCpHash

```

7.48 /tpm/src/command/EA/PolicyDuplicationSelect.c

```

1  #include "Tpm.h"
2  #include "PolicyDuplicationSelect_fp.h"
3
4  #if CC_PolicyDuplicationSelect // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // allows qualification of duplication so that it a specific new parent may be
8  // selected or a new parent selected for a specific object.
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_COMMAND_CODE 'commandCode' of 'policySession' is not empty
12 //     TPM_RC_CPHASH       'nameHash' of 'policySession' is not empty
13 TPM_RC
14 TPM2_PolicyDuplicationSelect(

```

```

15     PolicyDuplicationSelect_In* in  // IN: input parameter list
16 )
17 {
18     SESSION*    session;
19     HASH_STATE hashState;
20     TPM_CC      commandCode = TPM_CC_PolicyDuplicationSelect;
21
22     // Input Validation
23
24     // Get pointer to the session structure
25     session = SessionGet(in->policySession);
26
27     // nameHash in session context must be empty
28     if(session->u1.nameHash.t.size != 0)
29         return TPM_RC_CPHASH;
30
31     // commandCode in session context must be empty
32     if(session->commandCode != 0)
33         return TPM_RC_COMMAND_CODE;
34
35     // Internal Data Update
36
37     // Update name hash
38     session->u1.nameHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
39
40     // add objectName
41     CryptDigestUpdate2B(&hashState, &in->objectName.b);
42
43     // add new parent name
44     CryptDigestUpdate2B(&hashState, &in->newParentName.b);
45
46     // complete hash
47     CryptHashEnd2B(&hashState, &session->u1.nameHash.b);
48     session->attributes.isNameHashDefined = SET;
49
50     // update policy hash
51     // Old policyDigest size should be the same as the new policyDigest size since
52     // they are using the same hash algorithm
53     session->u2.policyDigest.t.size =
54         CryptHashStart(&hashState, session->authHashAlg);
55     // add old policy
56     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
57
58     // add command code
59     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
60
61     // add objectName
62     if(in->includeObject == YES)
63         CryptDigestUpdate2B(&hashState, &in->objectName.b);
64
65     // add new parent name
66     CryptDigestUpdate2B(&hashState, &in->newParentName.b);
67
68     // add includeObject
69     CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->includeObject);
70
71     // complete digest
72     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
73
74     // set commandCode in session context
75     session->commandCode = TPM_CC_Duplicate;
76
77     return TPM_RC_SUCCESS;
78 }
79
80 #endif // CC_PolicyDuplicationSelect

```

7.49 /tpm/src/command/EA/PolicyGetDigest.c

```

1  #include "Tpm.h"
2  #include "PolicyGetDigest_fp.h"
3
4  #if CC_PolicyGetDigest // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // returns the current policyDigest of the session
8  */
9  TPM_RC
10 TPM2_PolicyGetDigest(PolicyGetDigest_In* in, // IN: input parameter list
11                     PolicyGetDigest_Out* out // OUT: output parameter list
12 )
13 {
14     SESSION* session;
15
16     // Command Output
17
18     // Get pointer to the session structure
19     session = SessionGet(in->policySession);
20
21     out->policyDigest = session->u2.policyDigest;
22
23     return TPM_RC_SUCCESS;
24 }
25
26 #endif // CC_PolicyGetDigest

```

7.50 /tpm/src/command/EA/PolicyLocality.c

```

1  #include "Tpm.h"
2  #include "PolicyLocality_fp.h"
3  #include "Marshal.h"
4
5  #if CC_PolicyLocality // Conditional expansion of this file
6
7  // Return Type: TPM_RC
8  // TPM_RC_RANGE      all the locality values selected by
9  //                    'locality' have been disabled
10 //                    by previous TPM2_PolicyLocality() calls.
11 TPM_RC
12 TPM2_PolicyLocality(PolicyLocality_In* in // IN: input parameter list
13 )
14 {
15     SESSION* session;
16     BYTE     marshalBuffer[sizeof(TPMA_LOCALITY)];
17     BYTE     prevSetting[sizeof(TPMA_LOCALITY)];
18     UINT32   marshalSize;
19     BYTE*    buffer;
20     TPM_CC   commandCode = TPM_CC_PolicyLocality;
21     HASH_STATE hashState;
22
23     // Input Validation
24
25     // Get pointer to the session structure
26     session = SessionGet(in->policySession);
27
28     // Get new locality setting in canonical form
29     marshalBuffer[0] = 0; // Code analysis says that this is not initialized
30     buffer = marshalBuffer;
31     marshalSize = TPMA_LOCALITY_Marshal(&in->locality, &buffer, NULL);
32
33     // Its an error if the locality parameter is zero
34     if(marshalBuffer[0] == 0)

```

```

35     return TPM_RCS_RANGE + RC_PolicyLocality_locality;
36
37     // Get existing locality setting in canonical form
38     prevSetting[0] = 0; // Code analysis says that this is not initialized
39     buffer         = prevSetting;
40     TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
41
42     // If the locality has previously been set
43     if(prevSetting[0] != 0
44        // then the current locality setting and the requested have to be the same
45        // type (that is, either both normal or both extended
46        && ((prevSetting[0] < 32) != (marshalBuffer[0] < 32)))
47        return TPM_RCS_RANGE + RC_PolicyLocality_locality;
48
49     // See if the input is a regular or extended locality
50     if(marshalBuffer[0] < 32)
51     {
52         // if there was no previous setting, start with all normal localities
53         // enabled
54         if(prevSetting[0] == 0)
55             prevSetting[0] = 0x1F;
56
57         // AND the new setting with the previous setting and store it in prevSetting
58         prevSetting[0] &= marshalBuffer[0];
59
60         // The result setting can not be 0
61         if(prevSetting[0] == 0)
62             return TPM_RCS_RANGE + RC_PolicyLocality_locality;
63     }
64     else
65     {
66         // for extended locality
67         // if the locality has already been set, then it must match the
68         if(prevSetting[0] != 0 && prevSetting[0] != marshalBuffer[0])
69             return TPM_RCS_RANGE + RC_PolicyLocality_locality;
70
71         // Setting is OK
72         prevSetting[0] = marshalBuffer[0];
73     }
74
75     // Internal Data Update
76
77     // Update policy hash
78     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyLocality || locality)
79     // Start hash
80     CryptHashStart(&hashState, session->authHashAlg);
81
82     // add old digest
83     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
84
85     // add commandCode
86     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
87
88     // add input locality
89     CryptDigestUpdate(&hashState, marshalSize, marshalBuffer);
90
91     // complete the digest
92     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
93
94     // update session locality by unmarshal function. The function must succeed
95     // because both input and existing locality setting have been validated.
96     buffer = prevSetting;
97     TPMA_LOCALITY_Unmarshal(&session->commandLocality, &buffer, (INT32*)&marshalSize);
98
99     return TPM_RC_SUCCESS;
100 }

```



```

101
102 #endif // CC_PolicyLocality

```

7.51 /tpm/src/command/EA/PolicyNameHash.c

```

1  #include "Tpm.h"
2  #include "PolicyNameHash_fp.h"
3
4  #if CC_PolicyNameHash // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Add a nameHash restriction to the policyDigest
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_CPHASH      'nameHash' has been previously set to a different value
11 //     TPM_RC_SIZE        'nameHash' is not the size of the digest produced by the
12 //                        hash algorithm associated with 'policySession'
13 TPM_RC
14 TPM2_PolicyNameHash(PolicyNameHash_In* in // IN: input parameter list
15 )
16 {
17     SESSION* session;
18     TPM_CC commandCode = TPM_CC_PolicyNameHash;
19     HASH_STATE hashState;
20
21     // Input Validation
22
23     // Get pointer to the session structure
24     session = SessionGet(in->policySession);
25
26     // A valid nameHash must have the same size as session hash digest
27     // Since the authHashAlg for a session cannot be TPM_ALG_NULL, the digest size
28     // is always non-zero.
29     if(in->nameHash.t.size != CryptHashGetDigestSize(session->authHashAlg))
30         return TPM_RC_SIZE + RC_PolicyNameHash_nameHash;
31
32     // error if the nameHash in session context is not empty
33     if(IsCpHashUnionOccupied(session->attributes))
34         return TPM_RC_CPHASH;
35
36     // Internal Data Update
37
38     // Update policy hash
39     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyNameHash || nameHash)
40     // Start hash
41     CryptHashStart(&hashState, session->authHashAlg);
42
43     // add old digest
44     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
45
46     // add commandCode
47     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
48
49     // add nameHash
50     CryptDigestUpdate2B(&hashState, &in->nameHash.b);
51
52     // complete the digest
53     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
54
55     // update nameHash in session context
56     session->u1.nameHash = in->nameHash;
57     session->attributes.isNameHashDefined = SET;
58
59     return TPM_RC_SUCCESS;
60 }

```

```

61
62 #endif // CC_PolicyNameHash

```

7.52 /tpm/src/command/EA/PolicyNV.c

```

1  #include "Tpm.h"
2  #include "PolicyNV_fp.h"
3
4  #if CC_PolicyNV // Conditional expansion of this file
5
6  # include "Policy_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Do comparison to NV location
10 */
11 // Return Type: TPM_RC
12 // TPM_RC_AUTH_TYPE NV index authorization type is not correct
13 // TPM_RC_NV_LOCKED NV index read locked
14 // TPM_RC_NV_UNINITIALIZED the NV index has not been initialized
15 // TPM_RC_POLICY the comparison to the NV contents failed
16 // TPM_RC_SIZE the size of 'nvIndex' data starting at 'offset'
17 // is less than the size of 'operandB'
18 // TPM_RC_VALUE 'offset' is too large
19 TPM_RC
20 TPM2_PolicyNV(PolicyNV_In* in // IN: input parameter list
21 )
22 {
23     TPM_RC result;
24     SESSION* session;
25     NV_REF locator;
26     NV_INDEX* nvIndex;
27     BYTE nvBuffer[sizeof(in->operandB.t.buffer)];
28     TPM2B_NAME nvName;
29     TPM_CC commandCode = TPM_CC_PolicyNV;
30     HASH_STATE hashState;
31     TPM2B_DIGEST argHash;
32
33     // Input Validation
34
35     // Get pointer to the session structure
36     session = SessionGet(in->policySession);
37
38     //If this is a trial policy, skip all validations and the operation
39     if(session->attributes.isTrialPolicy == CLEAR)
40     {
41         // No need to access the actual NV index information for a trial policy.
42         nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
43
44         // Common read access checks. NvReadAccessChecks() may return
45         // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
46         result = NvReadAccessChecks(
47             in->authHandle, in->nvIndex, nvIndex->publicArea.attributes);
48         if(result != TPM_RC_SUCCESS)
49             return result;
50
51         // Make sure that offset is within range
52         if(in->offset > nvIndex->publicArea.dataSize)
53             return TPM_RCS_VALUE + RC_PolicyNV_offset;
54
55         // Valid NV data size should not be smaller than input operandB size
56         if((nvIndex->publicArea.dataSize - in->offset) < in->operandB.t.size)
57             return TPM_RCS_SIZE + RC_PolicyNV_operandB;
58
59         // Get NV data. The size of NV data equals the input operand B size
60         NvGetIndexData(nvIndex, locator, in->offset, in->operandB.t.size, nvBuffer);

```

```

61
62     // Check to see if the condition is valid
63     if(!PolicySptCheckCondition(
64         in->operation, nvBuffer, in->operandB.t.buffer, in->operandB.t.size))
65         return TPM_RC_POLICY;
66     }
67     // Internal Data Update
68
69     // Start argument hash
70     argHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
71
72     // add operandB
73     CryptDigestUpdate2B(&hashState, &in->operandB.b);
74
75     // add offset
76     CryptDigestUpdateInt(&hashState, sizeof(UINT16), in->offset);
77
78     // add operation
79     CryptDigestUpdateInt(&hashState, sizeof(TPM_EO), in->operation);
80
81     // complete argument digest
82     CryptHashEnd2B(&hashState, &argHash.b);
83
84     // Update policyDigest
85     // Start digest
86     CryptHashStart(&hashState, session->authHashAlg);
87
88     // add old digest
89     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
90
91     // add commandCode
92     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
93
94     // add argument digest
95     CryptDigestUpdate2B(&hashState, &argHash.b);
96
97     // Adding nvName
98     CryptDigestUpdate2B(&hashState, &EntityGetName(in->nvIndex, &nvName)->b);
99
100    // complete the digest
101    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
102
103    return TPM_RC_SUCCESS;
104 }
105
106 #endif // CC_PolicyNV

```

7.53 /tpm/src/command/EA/PolicyNvWritten.c

```

1  #include "Tpm.h"
2  #include "PolicyNvWritten_fp.h"
3
4  #if CC_PolicyNvWritten // Conditional expansion of this file
5
6  // Make an NV Index policy dependent on the state of the TPMA_NV_WRITTEN
7  // attribute of the index.
8  // Return Type: TPM_RC
9  //     TPM_RC_VALUE          a conflicting request for the attribute has
10 //                          already been processed
11 TPM_RC
12 TPM2_PolicyNvWritten(PolicyNvWritten_In* in // IN: input parameter list
13 )
14 {
15     SESSION* session;
16     TPM_CC commandCode = TPM_CC_PolicyNvWritten;

```

```

17     HASH_STATE hashState;
18
19     // Input Validation
20
21     // Get pointer to the session structure
22     session = SessionGet(in->policySession);
23
24     // If already set is this a duplicate (the same setting)? If it
25     // is a conflicting setting, it is an error
26     if(session->attributes.checkNvWritten == SET)
27     {
28         if(((session->attributes.nvWrittenState == SET) != (in->writtenSet == YES)))
29             return TPM_RCS_VALUE + RC_PolicyNvWritten_writtenSet;
30     }
31
32     // Internal Data Update
33
34     // Set session attributes so that the NV Index needs to be checked
35     session->attributes.checkNvWritten = SET;
36     session->attributes.nvWrittenState = (in->writtenSet == YES);
37
38     // Update policy hash
39     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyNvWritten
40     //                        || writtenSet)
41     // Start hash
42     CryptHashStart(&hashState, session->authHashAlg);
43
44     // add old digest
45     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
46
47     // add commandCode
48     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
49
50     // add the byte of writtenState
51     CryptDigestUpdateInt(&hashState, sizeof(TPMI_YES_NO), in->writtenSet);
52
53     // complete the digest
54     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
55
56     return TPM_RC_SUCCESS;
57 }
58
59 #endif // CC_PolicyNvWritten

```

7.54 /tpm/src/command/EA/PolicyOR.c

```

1  #include "Tpm.h"
2  #include "PolicyOR_fp.h"
3
4  #if CC_PolicyOR // Conditional expansion of this file
5
6  # include "Policy_spt_fp.h"
7
8  /*(See part 3 specification)
9  // PolicyOR command
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_VALUE           no digest in 'pHashList' matched the current
13 //                             value of policyDigest for 'policySession'
14 TPM_RC
15 TPM2_PolicyOR(PolicyOR_In* in // IN: input parameter list
16 )
17 {
18     SESSION* session;
19     UINT32 i;

```

```

20
21 // Input Validation and Update
22
23 // Get pointer to the session structure
24 session = SessionGet(in->policySession);
25
26 // Compare and Update Internal Session policy if match
27 for(i = 0; i < in->pHashList.count; i++)
28 {
29     if(session->attributes.isTrialPolicy == SET
30        || (MemoryEqual2B(&session->u2.policyDigest.b,
31                          &in->pHashList.digests[i].b)))
32     {
33         // Found a match
34         HASH_STATE hashState;
35         TPM_CC      commandCode = TPM_CC_PolicyOR;
36
37         // Start hash
38         session->u2.policyDigest.t.size =
39             CryptHashStart(&hashState, session->authHashAlg);
40         // Set policyDigest to 0 string and add it to hash
41         MemorySet(session->u2.policyDigest.t.buffer,
42                   0,
43                   session->u2.policyDigest.t.size);
44         CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
45
46         // add command code
47         CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
48
49         // Add each of the hashes in the list
50         for(i = 0; i < in->pHashList.count; i++)
51         {
52             // Extend policyDigest
53             CryptDigestUpdate2B(&hashState, &in->pHashList.digests[i].b);
54         }
55         // Complete digest
56         CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
57         return TPM_RC_SUCCESS;
58     }
59 }
60
61 // None of the values in the list matched the current policyDigest
62 return TPM_RCS_VALUE + RC_PolicyOR_pHashList;
63 }
64
65 #endif // CC_PolicyOR

```

7.55 /tpm/src/command/EA/PolicyParameters.c

```

1  #include "Tpm.h"
2  #include "PolicyParameters_fp.h"
3
4  #if CC_PolicyParameters // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Add a parameters restriction to the policyDigest
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_CPHASH      cpHash of 'policySession' has previously been set
11 //                        to a different value
12 //     TPM_RC_SIZE        'pHash' is not the size of the digest produced by the
13 //                        hash algorithm associated with 'policySession'
14 TPM_RC
15 TPM2_PolicyParameters(PolicyParameters_In* in // IN: input parameter list
16 )

```

```

17 {
18     SESSION*    session;
19     TPM_CC      commandCode = TPM_CC_PolicyParameters;
20     HASH_STATE  hashState;
21
22     // Input Validation
23
24     // Get pointer to the session structure
25     session = SessionGet(in->policySession);
26
27     // A valid pHash must have the same size as session hash digest
28     // Since the authHashAlg for a session cannot be TPM_ALG_NULL, the digest size
29     // is always non-zero.
30     if(in->pHash.t.size != CryptHashGetDigestSize(session->authHashAlg))
31         return TPM_RCS_SIZE + RC_PolicyParameters_pHash;
32
33     // error if the pHash in session context is not empty
34     if(IsCpHashUnionOccupied(session->attributes))
35         return TPM_RC_CPHASH;
36
37     // Internal Data Update
38
39     // Update policy hash
40     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyParameters || pHash)
41     // Start hash
42     CryptHashStart(&hashState, session->authHashAlg);
43
44     // add old digest
45     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
46
47     // add commandCode
48     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
49
50     // add pHash
51     CryptDigestUpdate2B(&hashState, &in->pHash.b);
52
53     // complete the digest
54     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
55
56     // update pHash in session context
57     session->u1.pHash = in->pHash;
58     session->attributes.isParametersHashDefined = SET;
59
60     return TPM_RC_SUCCESS;
61 }
62
63 #endif // CC_PolicyParameters

```

7.56 /tpm/src/command/EA/PolicyPassword.c

```

1  #include "Tpm.h"
2  #include "PolicyPassword_fp.h"
3
4  #if CC_PolicyPassword // Conditional expansion of this file
5
6  # include "Policy_spt_fp.h"
7
8  /*(See part 3 specification)
9  // allows a policy to be bound to the authorization value of the authorized
10 // object
11 */
12 TPM_RC
13 TPM2_PolicyPassword(PolicyPassword_In* in // IN: input parameter list
14 )
15 {

```



```

16     SESSION*    session;
17     TPM_CC      commandCode = TPM_CC_PolicyAuthValue;
18     HASH_STATE  hashState;
19
20     // Internal Data Update
21
22     // Get pointer to the session structure
23     session = SessionGet(in->policySession);
24
25     // Update policy hash
26     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyAuthValue)
27     // Start hash
28     CryptHashStart(&hashState, session->authHashAlg);
29
30     // add old digest
31     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
32
33     // add commandCode
34     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
35
36     // complete the digest
37     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
38
39     // Update isPasswordNeeded bit
40     session->attributes.isPasswordNeeded = SET;
41     session->attributes.isAuthValueNeeded = CLEAR;
42
43     return TPM_RC_SUCCESS;
44 }
45
46 #endif // CC_PolicyPassword

```

7.57 /tpm/src/command/EA/PolicyPCR.c

```

1  #include "Tpm.h"
2
3  #if CC_PolicyPCR // Conditional expansion of this file
4
5  # include "PolicyPCR_fp.h"
6  # include "Marshal.h"
7
8  /*(See part 3 specification)
9  // Add a PCR gate for a policy session
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_VALUE      if provided, 'pcrDigest' does not match the
13 //                       current PCR settings
14 //     TPM_RC_PCR_CHANGED a previous TPM2_PolicyPCR() set
15 //                       pcrCounter and it has changed
16 TPM_RC
17 TPM2_PolicyPCR(PolicyPCR_In* in // IN: input parameter list
18 )
19 {
20     SESSION*    session;
21     TPM2B_DIGEST pcrDigest;
22     BYTE        pcrc[sizeof(TPML_PCR_SELECTION)];
23     UINT32      pcrSize;
24     BYTE*       buffer;
25     TPM_CC      commandCode = TPM_CC_PolicyPCR;
26     HASH_STATE  hashState;
27
28     // Input Validation
29
30     // Get pointer to the session structure
31     session = SessionGet(in->policySession);

```

```

32
33 // Compute current PCR digest
34 PCRComputeCurrentDigest(session->authHashAlg, &in->pcrs, &pcrDigest);
35
36 // Do validation for non trial session
37 if(session->attributes.isTrialPolicy == CLEAR)
38 {
39     // Make sure that this is not going to invalidate a previous PCR check
40     if(session->pcrCounter != 0 && session->pcrCounter != gr.pcrCounter)
41         return TPM_RC_PCR_CHANGED;
42
43     // If the caller specified the PCR digest and it does not
44     // match the current PCR settings, return an error..
45     if(in->pcrDigest.t.size != 0)
46     {
47         if(!MemoryEqual2B(&in->pcrDigest.b, &pcrDigest.b))
48             return TPM_RCS_VALUE + RC_PolicyPCR_pcrDigest;
49     }
50 }
51 else
52 {
53     // For trial session, just use the input PCR digest if one provided
54     // Note: It can't be too big because it is a TPM2B_DIGEST and the size
55     // would have been checked during unmarshaling
56     if(in->pcrDigest.t.size != 0)
57         pcrDigest = in->pcrDigest;
58 }
59 // Internal Data Update
60 // Update policy hash
61 // policyDigestnew = hash( policyDigestold || TPM_CC_PolicyPCR
62 //                          || PCRS || pcrDigest)
63 // Start hash
64 CryptHashStart(&hashState, session->authHashAlg);
65
66 // add old digest
67 CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
68
69 // add commandCode
70 CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
71
72 // add PCRS
73 buffer = pcrs;
74 pcrSize = TPML_PCR_SELECTION_Marshal(&in->pcrs, &buffer, NULL);
75 CryptDigestUpdate(&hashState, pcrSize, pcrs);
76
77 // add PCR digest
78 CryptDigestUpdate2B(&hashState, &pcrDigest.b);
79
80 // complete the hash and get the results
81 CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
82
83 // update pcrCounter in session context for non trial session
84 if(session->attributes.isTrialPolicy == CLEAR)
85 {
86     session->pcrCounter = gr.pcrCounter;
87 }
88
89 return TPM_RC_SUCCESS;
90 }
91
92 #endif // CC_PolicyPCR

```

7.58 /tpm/src/command/EA/PolicyPhysicalPresence.c

```
1 #include "Tpm.h"
```

```

2  #include "PolicyPhysicalPresence_fp.h"
3
4  #if CC_PolicyPhysicalPresence // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // indicate that physical presence will need to be asserted at the time the
8  // authorization is performed
9  */
10 TPM_RC
11 TPM2_PolicyPhysicalPresence(PolicyPhysicalPresence_In* in // IN: input parameter list
12 )
13 {
14     SESSION* session;
15     TPM_CC commandCode = TPM_CC_PolicyPhysicalPresence;
16     HASH_STATE hashState;
17
18     // Internal Data Update
19
20     // Get pointer to the session structure
21     session = SessionGet(in->policySession);
22
23     // Update policy hash
24     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyPhysicalPresence)
25     // Start hash
26     CryptHashStart(&hashState, session->authHashAlg);
27
28     // add old digest
29     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
30
31     // add commandCode
32     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
33
34     // complete the digest
35     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
36
37     // update session attribute
38     session->attributes.isPPRequired = SET;
39
40     return TPM_RC_SUCCESS;
41 }
42
43 #endif // CC_PolicyPhysicalPresence

```

7.59 /tpm/src/command/EA/PolicySecret.c

```

1  #include "Tpm.h"
2  #include "PolicySecret_fp.h"
3
4  #if CC_PolicySecret // Conditional expansion of this file
5
6  # include "Policy_spt_fp.h"
7  # include "NV_spt_fp.h"
8
9  /*(See part 3 specification)
10 // Add a secret-based authorization to the policy evaluation
11 */
12 // Return Type: TPM_RC
13 //     TPM_RC_CPHASH           cpHash for policy was previously set to a
14 //                               value that is not the same as 'cpHashA'
15 //     TPM_RC_EXPIRED          'expiration' indicates a time in the past
16 //     TPM_RC_NONCE            'nonceTPM' does not match the nonce associated
17 //                               with 'policySession'
18 //     TPM_RC_SIZE             'cpHashA' is not the size of a digest for the
19 //                               hash associated with 'policySession'
20 TPM_RC

```

```

21 TPM2_PolicySecret(PolicySecret_In* in, // IN: input parameter list
22                  PolicySecret_Out* out // OUT: output parameter list
23 )
24 {
25     TPM_RC      result;
26     SESSION*    session;
27     TPM2B_NAME  entityName;
28     UINT64      authTimeout = 0;
29     // Input Validation
30     // Get pointer to the session structure
31     session = SessionGet(in->policySession);
32
33     //Only do input validation if this is not a trial policy session
34     if(session->attributes.isTrialPolicy == CLEAR)
35     {
36         authTimeout = ComputeAuthTimeout(session, in->expiration, &in->nonceTPM);
37
38         result      = PolicyParameterChecks(session,
39                                             authTimeout,
40                                             &in->cpHashA,
41                                             &in->nonceTPM,
42                                             RC_PolicySecret_nonceTPM,
43                                             RC_PolicySecret_cpHashA,
44                                             RC_PolicySecret_expiration);
45
46         if(result != TPM_RC_SUCCESS)
47             return result;
48     }
49     // Internal Data Update
50     // Update policy context with input policyRef and name of authorizing key
51     // This value is computed even for trial sessions. Possibly update the cpHash
52     PolicyContextUpdate(TPM_CC_PolicySecret,
53                        EntityGetName(in->authHandle, &entityName),
54                        &in->policyRef,
55                        &in->cpHashA,
56                        authTimeout,
57                        session);
58     // Command Output
59     // Create ticket and timeout buffer if in->expiration < 0 and this is not
60     // a trial session.
61     // NOTE: PolicyParameterChecks() makes sure that nonceTPM is present
62     // when expiration is non-zero.
63     if(in->expiration < 0 && session->attributes.isTrialPolicy == CLEAR
64        && !NvIsPinPassIndex(in->authHandle))
65     {
66         BOOL expiresOnReset = (in->nonceTPM.t.size == 0);
67         // Compute policy ticket
68         authTimeout &= ~EXPIRATION_BIT;
69         result = TicketComputeAuth(TPM_ST_AUTH_SECRET,
70                                  EntityGetHierarchy(in->authHandle),
71                                  authTimeout,
72                                  expiresOnReset,
73                                  &in->cpHashA,
74                                  &in->policyRef,
75                                  &entityName,
76                                  &out->policyTicket);
77
78         if(result != TPM_RC_SUCCESS)
79             return result;
80
81         // Generate timeout buffer. The format of output timeout buffer is
82         // TPM-specific.
83         // Note: In this implementation, the timeout buffer value is computed after
84         // the ticket is produced so, when the ticket is checked, the expiration
85         // flag needs to be extracted before the ticket is checked.
86         out->timeout.t.size = sizeof(authTimeout);
87         // In the Windows compatible version, the least-significant bit of the
88         // timeout value is used as a flag to indicate if the authorization expires

```

```

87         // on reset. The flag is the MSb.
88         if(expiresOnReset)
89             authTimeout |= EXPIRATION_BIT;
90         UINT64_TO_BYTE_ARRAY(authTimeout, out->timeout.t.buffer);
91     }
92     else
93     {
94         // timeout buffer is null
95         out->timeout.t.size = 0;
96
97         // authorization ticket is null
98         out->policyTicket.tag          = TPM_ST_AUTH_SECRET;
99         out->policyTicket.hierarchy    = TPM_RH_NULL;
100        out->policyTicket.digest.t.size = 0;
101    }
102    return TPM_RC_SUCCESS;
103 }
104
105 #endif // CC_PolicySecret

```

7.60 /tpm/src/command/EA/PolicySigned.c

```

1  #include "Tpm.h"
2  #include "Policy_spt_fp.h"
3  #include "PolicySigned_fp.h"
4
5  #if CC_PolicySigned // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // Include an asymmetrically signed authorization to the policy evaluation
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_CPHASH           cpHash was previously set to a different value
12 //     TPM_RC_EXPIRED          'expiration' indicates a time in the past or
13 //                             'expiration' is non-zero but no nonceTPM is present
14 //     TPM_RC_NONCE            'nonceTPM' is not the nonce associated with the
15 //                             'policySession'
16 //     TPM_RC_SCHEME           the signing scheme of 'auth' is not supported by the
17 //                             TPM
18 //     TPM_RC_SIGNATURE        the signature is not genuine
19 //     TPM_RC_SIZE             input cpHash has wrong size
20 TPM_RC
21 TPM2_PolicySigned(PolicySigned_In* in, // IN: input parameter list
22                 PolicySigned_Out* out // OUT: output parameter list
23 )
24 {
25     TPM_RC      result = TPM_RC_SUCCESS;
26     SESSION*    session;
27     TPM2B_NAME  entityName;
28     TPM2B_DIGEST authHash;
29     HASH_STATE  hashState;
30     UINT64      authTimeout = 0;
31     // Input Validation
32     // Set up local pointers
33     session = SessionGet(in->policySession); // the session structure
34
35     // Only do input validation if this is not a trial policy session
36     if(session->attributes.isTrialPolicy == CLEAR)
37     {
38         authTimeout = ComputeAuthTimeout(session, in->expiration, &in->nonceTPM);
39
40         result      = PolicyParameterChecks(session,
41                                             authTimeout,
42                                             &in->cpHashA,
43                                             &in->nonceTPM,

```

```

44         RC_PolicySigned_nonceTPM,
45         RC_PolicySigned_cpHashA,
46         RC_PolicySigned_expiration);
47     if(result != TPM_RC_SUCCESS)
48         return result;
49     // Re-compute the digest being signed
50     /*(See part 3 specification)
51     // The digest is computed as:
52     //     aHash := hash ( nonceTPM | expiration | cpHashA | policyRef)
53     // where:
54     //     hash()      the hash associated with the signed authorization
55     //     nonceTPM    the nonceTPM value from the TPM2_StartAuthSession .
56     //                 response If the authorization is not limited to this
57     //                 session, the size of this value is zero.
58     //     expiration  time limit on authorization set by authorizing object.
59     //                 This 32-bit value is set to zero if the expiration
60     //                 time is not being set.
61     //     cpHashA     hash of the command parameters for the command being
62     //                 approved using the hash algorithm of the PSAP session.
63     //                 Set to NULLauth if the authorization is not limited
64     //                 to a specific command.
65     //     policyRef   hash of an opaque value determined by the authorizing
66     //                 object. Set to the NULLdigest if no hash is present.
67     */
68     // Start hash
69     authHash.t.size = CryptHashStart(&hashState, CryptGetSignHashAlg(&in->auth));
70     // If there is no digest size, then we don't have a verification function
71     // for this algorithm (e.g. TPM_ALG_ECDSA) so indicate that it is a
72     // bad scheme.
73     if(authHash.t.size == 0)
74         return TPM_RCS_SCHEME + RC_PolicySigned_auth;
75
76     // nonceTPM
77     CryptDigestUpdate2B(&hashState, &in->nonceTPM.b);
78
79     // expiration
80     CryptDigestUpdateInt(&hashState, sizeof(UINT32), in->expiration);
81
82     // cpHashA
83     CryptDigestUpdate2B(&hashState, &in->cpHashA.b);
84
85     // policyRef
86     CryptDigestUpdate2B(&hashState, &in->policyRef.b);
87
88     // Complete digest
89     CryptHashEnd2B(&hashState, &authHash.b);
90
91     // Validate Signature. A TPM_RC_SCHEME, TPM_RC_HANDLE or TPM_RC_SIGNATURE
92     // error may be returned at this point
93     result = CryptValidateSignature(in->authObject, &authHash, &in->auth);
94     if(result != TPM_RC_SUCCESS)
95         return RcSafeAddToResult(result, RC_PolicySigned_auth);
96 }
97 // Internal Data Update
98 // Update policy with input policyRef and name of authorization key
99 // These values are updated even if the session is a trial session
100 PolicyContextUpdate(TPM_CC_PolicySigned,
101                    EntityGetName(in->authObject, &entityName),
102                    &in->policyRef,
103                    &in->cpHashA,
104                    authTimeout,
105                    session);
106 // Command Output
107 // Create ticket and timeout buffer if in->expiration < 0 and this is not
108 // a trial session.
109 // NOTE: PolicyParameterChecks() makes sure that nonceTPM is present

```



```

110 // when expiration is non-zero.
111 if(in->expiration < 0 && session->attributes.isTrialPolicy == CLEAR)
112 {
113     BOOL expiresOnReset = (in->nonceTPM.t.size == 0);
114     // Compute policy ticket
115     authTimeout &= ~EXPIRATION_BIT;
116
117     result = TicketComputeAuth(TPM_ST_AUTH_SIGNED,
118                               EntityGetHierarchy(in->authObject),
119                               authTimeout,
120                               expiresOnReset,
121                               &in->cpHashA,
122                               &in->policyRef,
123                               &entityName,
124                               &out->policyTicket);
125     if(result != TPM_RC_SUCCESS)
126         return result;
127
128     // Generate timeout buffer. The format of output timeout buffer is
129     // TPM-specific.
130     // Note: In this implementation, the timeout buffer value is computed after
131     // the ticket is produced so, when the ticket is checked, the expiration
132     // flag needs to be extracted before the ticket is checked.
133     // In the Windows compatible version, the least-significant bit of the
134     // timeout value is used as a flag to indicate if the authorization expires
135     // on reset. The flag is the MSb.
136     out->timeout.t.size = sizeof(authTimeout);
137     if(expiresOnReset)
138         authTimeout |= EXPIRATION_BIT;
139     UINT64_TO_BYTE_ARRAY(authTimeout, out->timeout.t.buffer);
140 }
141 else
142 {
143     // Generate a null ticket.
144     // timeout buffer is null
145     out->timeout.t.size = 0;
146
147     // authorization ticket is null
148     out->policyTicket.tag = TPM_ST_AUTH_SIGNED;
149     out->policyTicket.hierarchy = TPM_RH_NULL;
150     out->policyTicket.digest.t.size = 0;
151 }
152 return TPM_RC_SUCCESS;
153 }
154
155 #endif // CC_PolicySigned

```

7.61 /tpm/src/command/EA/PolicyTemplate.c

```

1 #include "Tpm.h"
2 #include "PolicyTemplate_fp.h"
3
4 #if CC_PolicyTemplate // Conditional expansion of this file
5
6 /*(See part 3 specification)
7 // Add a cpHash restriction to the policyDigest
8 */
9 // Return Type: TPM_RC
10 // TPM_RC_CPHASH cpHash of 'policySession' has previously been set
11 // to a different value
12 // TPM_RC_SIZE 'templateHash' is not the size of a digest produced
13 // by the hash algorithm associated with
14 // 'policySession'
15 TPM_RC
16 TPM2_PolicyTemplate(PolicyTemplate_In* in // IN: input parameter list

```

```

17 )
18 {
19     SESSION*    session;
20     TPM_CC      commandCode = TPM_CC_PolicyTemplate;
21     HASH_STATE  hashState;
22
23     // Input Validation
24
25     // Get pointer to the session structure
26     session = SessionGet(in->policySession);
27
28     // error if the templateHash in session context is not empty and is not the
29     // same as the input or is not a template
30     if((IsCpHashUnionOccupied(session->attributes)
31         && (!session->attributes.isTemplateHashDefined
32            || !MemoryEqual2B(&in->templateHash.b, &session->u1.templateHash.b)))
33         return TPM_RC_CPHASH;
34
35     // A valid templateHash must have the same size as session hash digest
36     if(in->templateHash.t.size != CryptHashGetDigestSize(session->authHashAlg))
37         return TPM_RC_SIZE + RC_PolicyTemplate_templateHash;
38
39     // Internal Data Update
40     // Update policy hash
41     // policyDigestnew = hash(policyDigestold || TPM_CC_PolicyCpHash
42     // || cpHashA.buffer)
43     // Start hash
44     CryptHashStart(&hashState, session->authHashAlg);
45
46     // add old digest
47     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
48
49     // add commandCode
50     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), commandCode);
51
52     // add cpHashA
53     CryptDigestUpdate2B(&hashState, &in->templateHash.b);
54
55     // complete the digest and get the results
56     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
57
58     // update templateHash in session context
59     session->u1.templateHash = in->templateHash;
60     session->attributes.isTemplateHashDefined = SET;
61
62     return TPM_RC_SUCCESS;
63 }
64
65 #endif // CC_PolicyTemplateHash

```

7.62 /tpm/src/command/EA/PolicyTicket.c

```

1  #include "Tpm.h"
2  #include "PolicyTicket_fp.h"
3
4  #if CC_PolicyTicket // Conditional expansion of this file
5
6  # include "Policy_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Include ticket to the policy evaluation
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_CPHASH           policy's cpHash was previously set to a different
13 //                             value

```

```

14 //      TPM_RC_EXPIRED      'timeout' value in the ticket is in the past and the
15 //      ticket has expired
16 //      TPM_RC_SIZE        'timeout' or 'cpHash' has invalid size for the
17 //      TPM_RC_TICKET      'ticket' is not valid
18 TPM_RC
19 TPM2_PolicyTicket(PolicyTicket_In* in // IN: input parameter list
20 )
21 {
22     TPM_RC      result;
23     SESSION*    session;
24     UINT64      authTimeout;
25     TPMT_TK_AUTH ticketToCompare;
26     TPM_CC      commandCode = TPM_CC_PolicySecret;
27     BOOL        expiresOnReset;
28
29     // Input Validation
30
31     // Get pointer to the session structure
32     session = SessionGet(in->policySession);
33
34     // NOTE: A trial policy session is not allowed to use this command.
35     // A ticket is used in place of a previously given authorization. Since
36     // a trial policy doesn't actually authenticate, the validated
37     // ticket is not necessary and, in place of using a ticket, one
38     // should use the intended authorization for which the ticket
39     // would be a substitute.
40     if(session->attributes.isTrialPolicy)
41         return TPM_RCS_ATTRIBUTES + RC_PolicyTicket_policySession;
42     // Restore timeout data. The format of timeout buffer is TPM-specific.
43     // In this implementation, the most significant bit of the timeout value is
44     // used as the flag to indicate that the ticket expires on TPM Reset or
45     // TPM Restart. The flag has to be removed before the parameters and ticket
46     // are checked.
47     if(in->timeout.t.size != sizeof(UINT64))
48         return TPM_RCS_SIZE + RC_PolicyTicket_timeout;
49     authTimeout = BYTE_ARRAY_TO_UINT64(in->timeout.t.buffer);
50
51     // extract the flag
52     expiresOnReset = (authTimeout & EXPIRATION_BIT) != 0;
53     authTimeout &= ~EXPIRATION_BIT;
54
55     // Do the normal checks on the cpHashA and timeout values
56     result = PolicyParameterChecks(session,
57                                     authTimeout,
58                                     &in->cpHashA,
59                                     NULL, // no nonce
60                                     0,    // no bad nonce return
61                                     RC_PolicyTicket_cpHashA,
62                                     RC_PolicyTicket_timeout);
63     if(result != TPM_RC_SUCCESS)
64         return result;
65     // Validate Ticket
66     // Re-generate policy ticket by input parameters
67     result = TicketComputeAuth(in->ticket.tag,
68                               in->ticket.hierarchy,
69                               authTimeout,
70                               expiresOnReset,
71                               &in->cpHashA,
72                               &in->policyRef,
73                               &in->authName,
74                               &ticketToCompare);
75     if(result != TPM_RC_SUCCESS)
76         return result;
77
78     // Compare generated digest with input ticket digest
79     if(!MemoryEqual2B(&in->ticket.digest.b, &ticketToCompare.digest.b))

```

```

80         return TPM_RCS_TICKET + RC_PolicyTicket_ticket;
81
82     // Internal Data Update
83
84     // Is this ticket to take the place of a TPM2_PolicySigned() or
85     // a TPM2_PolicySecret()?
86     if(in->ticket.tag == TPM_ST_AUTH_SIGNED)
87         commandCode = TPM_CC_PolicySigned;
88     else if(in->ticket.tag == TPM_ST_AUTH_SECRET)
89         commandCode = TPM_CC_PolicySecret;
90     else
91         // There could only be two possible tag values. Any other value should
92         // be caught by the ticket validation process.
93         FAIL(FATAL_ERROR_INTERNAL);
94
95     // Update policy context
96     PolicyContextUpdate(commandCode,
97                         &in->authName,
98                         &in->policyRef,
99                         &in->cpHashA,
100                        authTimeout,
101                        session);
102
103     return TPM_RC_SUCCESS;
104 }
105
106 #endif // CC_PolicyTicket

```

7.63 /tpm/src/command/EA/Policy_spt.c

```

1  /** Includes
2  #include "Tpm.h"
3  #include "Policy_spt_fp.h"
4  #include "PolicySigned_fp.h"
5  #include "PolicySecret_fp.h"
6  #include "PolicyTicket_fp.h"
7
8  /** Functions
9  /** PolicyParameterChecks()
10 // This function validates the common parameters of TPM2_PolicySigned()
11 // and TPM2_PolicySecret(). The common parameters are 'nonceTPM',
12 // 'expiration', and 'cpHashA'.
13 TPM_RC
14 PolicyParameterChecks(SESSION* session,
15                      UINT64 authTimeout,
16                      TPM2B_DIGEST* cpHashA,
17                      TPM2B_NONCE* nonce,
18                      TPM_RC blameNonce,
19                      TPM_RC blameCpHash,
20                      TPM_RC blameExpiration)
21 {
22     // Validate that input nonceTPM is correct if present
23     if(nonce != NULL && nonce->t.size != 0)
24     {
25         if(!MemoryEqual2B(&nonce->b, &session->nonceTPM.b))
26             return TPM_RCS_NONCE + blameNonce;
27     }
28     // If authTimeout is set (expiration != 0...
29     if(authTimeout != 0)
30     {
31         // Validate input expiration.
32         // Cannot compare time if clock stop advancing. A TPM_RC_NV_UNAVAILABLE
33         // or TPM_RC_NV_RATE error may be returned here.
34         RETURN_IF_NV_IS_NOT_AVAILABLE;
35     }

```

```

36     // if the time has already passed or the time epoch has changed then the
37     // time value is no longer good.
38     if((authTimeout < g_time) || (session->epoch != g_timeEpoch))
39         return TPM_RCS_EXPIRED + blameExpiration;
40 }
41 // If the cpHash is present, then check it
42 if(cpHashA != NULL && cpHashA->t.size != 0)
43 {
44     // The cpHash input has to have the correct size
45     if(cpHashA->t.size != session->u2.policyDigest.t.size)
46         return TPM_RCS_SIZE + blameCpHash;
47
48     // If the cpHash has already been set, then this input value
49     // must match the current value.
50     if(session->u1.cpHash.b.size != 0
51         && !MemoryEqual2B(&cpHashA->b, &session->u1.cpHash.b))
52         return TPM_RC_CPHASH;
53 }
54 return TPM_RC_SUCCESS;
55 }
56
57 /*** PolicyContextUpdate()
58 // Update policy hash
59 // Update the policyDigest in policy session by extending policyRef and
60 // objectName to it. This will also update the cpHash if it is present.
61 //
62 // Return Type: void
63 void PolicyContextUpdate(
64     TPM_CC      commandCode, // IN: command code
65     TPM2B_NAME* name,        // IN: name of entity
66     TPM2B_NONCE* ref,        // IN: the reference data
67     TPM2B_DIGEST* cpHash,    // IN: the cpHash (optional)
68     UINT64      policyTimeout, // IN: the timeout value for the policy
69     SESSION*     session      // IN/OUT: policy session to be updated
70 )
71 {
72     HASH_STATE hashState;
73
74     // Start hash
75     CryptHashStart(&hashState, session->authHashAlg);
76
77     // policyDigest size should always be the digest size of session hash algorithm.
78     pAssert(session->u2.policyDigest.t.size
79         == CryptHashGetDigestSize(session->authHashAlg));
80
81     // add old digest
82     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
83
84     // add commandCode
85     CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
86
87     // add name if applicable
88     if(name != NULL)
89         CryptDigestUpdate2B(&hashState, &name->b);
90
91     // Complete the digest and get the results
92     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
93
94     // If the policy reference is not null, do a second update to the digest.
95     if(ref != NULL)
96     {
97
98         // Start second hash computation
99         CryptHashStart(&hashState, session->authHashAlg);
100
101         // add policyDigest

```

```

102     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
103
104     // add policyRef
105     CryptDigestUpdate2B(&hashState, &ref->b);
106
107     // Complete second digest
108     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
109 }
110 // Deal with the cpHash. If the cpHash value is present
111 // then it would have already been checked to make sure that
112 // it is compatible with the current value so all we need
113 // to do here is copy it and set the isCpHashDefined attribute
114 if(cpHash != NULL && cpHash->t.size != 0)
115 {
116     session->u1.cpHash = *cpHash;
117     session->attributes.isCpHashDefined = SET;
118 }
119
120 // update the timeout if it is specified
121 if(policyTimeout != 0)
122 {
123     // If the timeout has not been set, then set it to the new value
124     // than the current timeout then set it to the new value
125     if(session->timeout == 0 || session->timeout > policyTimeout)
126         session->timeout = policyTimeout;
127 }
128 return;
129 }
130 /*** ComputeAuthTimeout()
131 // This function is used to determine what the authorization timeout value for
132 // the session should be.
133 UINT64
134 ComputeAuthTimeout(SESSION* session, // IN: the session containing the time
135                    // values
136                    INT32 expiration, // IN: either the number of seconds from
137                    // the start of the session or the
138                    // time in g_timer;
139                    TPM2B_NONCE* nonce // IN: indicator of the time base
140 )
141 {
142     UINT64 policyTime;
143     // If no expiration, policy time is 0
144     if(expiration == 0)
145         policyTime = 0;
146     else
147     {
148         if(expiration < 0)
149             expiration = -expiration;
150         if(nonce->t.size == 0)
151             // The input time is absolute Time (not Clock), but it is expressed
152             // in seconds. To make sure that we don't time out too early, take the
153             // current value of milliseconds in g_time and add that to the input
154             // seconds value.
155             policyTime = (((UINT64)expiration) * 1000) + g_time % 1000;
156         else
157             // The policy timeout is the absolute value of the expiration in seconds
158             // added to the start time of the policy.
159             policyTime = session->startTime + (((UINT64)expiration) * 1000);
160     }
161     return policyTime;
162 }
163
164 /*** PolicyDigestClear()
165 // Function to reset the policyDigest of a session
166 void PolicyDigestClear(SESSION* session)
167 {

```



```

168     session->u2.policyDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
169     MemorySet(session->u2.policyDigest.t.buffer, 0, session->u2.policyDigest.t.size);
170 }
171
172 /*** PolicySptCheckCondition()
173 // Checks to see if the condition in the policy is satisfied.
174 BOOL PolicySptCheckCondition(TPM_EO operation, BYTE* opA, BYTE* opB, UINT16 size)
175 {
176     // Arithmetic Comparison
177     switch(operation)
178     {
179     case TPM_EO_EQ:
180         // compare A = B
181         return (UnsignedCompareB(size, opA, size, opB) == 0);
182         break;
183     case TPM_EO_NEQ:
184         // compare A != B
185         return (UnsignedCompareB(size, opA, size, opB) != 0);
186         break;
187     case TPM_EO_SIGNED_GT:
188         // compare A > B signed
189         return (SignedCompareB(size, opA, size, opB) > 0);
190         break;
191     case TPM_EO_UNSIGNED_GT:
192         // compare A > B unsigned
193         return (UnsignedCompareB(size, opA, size, opB) > 0);
194         break;
195     case TPM_EO_SIGNED_LT:
196         // compare A < B signed
197         return (SignedCompareB(size, opA, size, opB) < 0);
198         break;
199     case TPM_EO_UNSIGNED_LT:
200         // compare A < B unsigned
201         return (UnsignedCompareB(size, opA, size, opB) < 0);
202         break;
203     case TPM_EO_SIGNED_GE:
204         // compare A >= B signed
205         return (SignedCompareB(size, opA, size, opB) >= 0);
206         break;
207     case TPM_EO_UNSIGNED_GE:
208         // compare A >= B unsigned
209         return (UnsignedCompareB(size, opA, size, opB) >= 0);
210         break;
211     case TPM_EO_SIGNED_LE:
212         // compare A <= B signed
213         return (SignedCompareB(size, opA, size, opB) <= 0);
214         break;
215     case TPM_EO_UNSIGNED_LE:
216         // compare A <= B unsigned
217         return (UnsignedCompareB(size, opA, size, opB) <= 0);
218         break;
219     case TPM_EO_BITSET:
220         // All bits SET in B are SET in A. ((A&B)=B)
221         {
222             UINT32 i;
223             for(i = 0; i < size; i++)
224                 if((opA[i] & opB[i]) != opB[i])
225                     return FALSE;
226         }
227         break;
228     case TPM_EO_BITCLEAR:
229         // All bits SET in B are CLEAR in A. ((A&B)=0)
230         {
231             UINT32 i;
232             for(i = 0; i < size; i++)
233                 if((opA[i] & opB[i]) != 0)

```

```

234         return FALSE;
235     }
236     break;
237     default:
238         FAIL(FATAL_ERROR_INTERNAL);
239         break;
240 }
241 return TRUE;
242 }

```

7.64 /tpm/src/command/Ecdaa/Commit.c

```

1  #include "Tpm.h"
2  #include "Commit_fp.h"
3  #include "TpmMath_Util_fp.h"
4
5  #if CC_Commit // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // This command performs the point multiply operations for anonymous signing
9  // scheme.
10 */
11 // Return Type: TPM_RC
12 // TPM_RC_ATTRIBUTES 'keyHandle' references a restricted key that is not a
13 // signing key
14 // TPM_RC_ECC_POINT either 'P1' or the point derived from 's2' is not on
15 // the curve of 'keyHandle'
16 // TPM_RC_HASH invalid name algorithm in 'keyHandle'
17 // TPM_RC_KEY 'keyHandle' does not reference an ECC key
18 // TPM_RC_SCHEME the scheme of 'keyHandle' is not an anonymous scheme
19 // TPM_RC_NO_RESULT 'K', 'L' or 'E' was a point at infinity; or
20 // failed to generate "r" value
21 // TPM_RC_SIZE 's2' is empty but 'y2' is not or 's2' provided but
22 // 'y2' is not
23 TPM_RC
24 TPM2_Commit(Commit_In* in, // IN: input parameter list
25             Commit_Out* out // OUT: output parameter list
26 )
27 {
28     OBJECT* eccKey;
29     TPMS_ECC_POINT P2;
30     TPMS_ECC_POINT* pP2 = NULL;
31     TPMS_ECC_POINT* pP1 = NULL;
32     TPM2B_ECC_PARAMETER r;
33     TPM2B_ECC_PARAMETER p;
34     TPM_RC result;
35     TPMS_ECC_PARMS* parms;
36
37     // Input Validation
38
39     eccKey = HandleToObject(in->signHandle);
40     parms = &eccKey->publicArea.parameters.eccDetail;
41
42     // Input key must be an ECC key
43     if(eccKey->publicArea.type != TPM_ALG_ECC)
44         return TPM_RCS_KEY + RC_Commit_signHandle;
45
46     // This command may only be used with a sign-only key using an anonymous
47     // scheme.
48     // NOTE: a sign + decrypt key has no scheme so it will not be an anonymous one
49     // and an unrestricted sign key might not have a signing scheme but it can't
50     // be use in Commit()
51     if(!CryptIsSchemeAnonymous(parms->scheme.scheme))
52         return TPM_RCS_SCHEME + RC_Commit_signHandle;
53

```

```

54 // Make sure that both parts of P2 are present if either is present
55 if((in->s2.t.size == 0) != (in->y2.t.size == 0))
56     return TPM_RCS_SIZE + RC_Commit_y2;
57
58 // Get prime modulus for the curve. This is needed later but getting this now
59 // allows confirmation that the curve exists.
60 if(!TpmMath_IntTo2B(ExtEcc_CurveGetPrime(parms->curveID), &p.b, 0))
61     return TPM_RCS_KEY + RC_Commit_signHandle;
62
63 // Get the random value that will be used in the point multiplications
64 // Note: this does not commit the count.
65 if(!CryptGeneratorR(&r, NULL, parms->curveID, &eccKey->name))
66     return TPM_RC_NO_RESULT;
67
68 // Set up P2 if s2 and Y2 are provided
69 if(in->s2.t.size != 0)
70 {
71     TPM2B_DIGEST x2;
72
73     pP2 = &P2;
74
75     // copy y2 for P2
76     P2.y = in->y2;
77
78     // Compute x2 HnameAlg(s2) mod p
79     // do the hash operation on s2 with the size of curve 'p'
80     x2.t.size = CryptHashBlock(eccKey->publicArea.nameAlg,
81                               in->s2.t.size,
82                               in->s2.t.buffer,
83                               sizeof(x2.t.buffer),
84                               x2.t.buffer);
85
86     // If there were error returns in the hash routine, indicate a problem
87     // with the hash algorithm selection
88     if(x2.t.size == 0)
89         return TPM_RCS_HASH + RC_Commit_signHandle;
90     // The size of the remainder will be same as the size of p. DivideB() will
91     // pad the results (leading zeros) if necessary to make the size the same
92     P2.x.t.size = p.t.size;
93     // set p2.x = hash(s2) mod p
94     if(DivideB(&x2.b, &p.b, NULL, &P2.x.b) != TPM_RC_SUCCESS)
95         return TPM_RC_NO_RESULT;
96
97     if(!CryptEccIsPointOnCurve(parms->curveID, pP2))
98         return TPM_RCS_ECC_POINT + RC_Commit_s2;
99
100     if(eccKey->attributes.publicOnly == SET)
101         return TPM_RCS_KEY + RC_Commit_signHandle;
102 }
103 // If there is a P1, make sure that it is on the curve
104 // NOTE: an "empty" point has two UINT16 values which are the size values
105 // for each of the coordinates.
106 if(in->P1.size > 4)
107 {
108     pP1 = &in->P1.point;
109     if(!CryptEccIsPointOnCurve(parms->curveID, pP1))
110         return TPM_RCS_ECC_POINT + RC_Commit_P1;
111 }
112
113 // Pass the parameters to CryptCommit.
114 // The work is not done in-line because it does several point multiplies
115 // with the same curve. It saves work by not having to reload the curve
116 // parameters multiple times.
117 result = CryptEccCommitCompute(&out->K.point,
118                               &out->L.point,
119                               &out->E.point,

```

```

120         parms->curveID,
121         pP1,
122         pP2,
123         &eccKey->sensitive.sensitive.ecc,
124         &r);
125     if(result != TPM_RC_SUCCESS)
126         return result;
127
128     // The commit computation was successful so complete the commit by setting
129     // the bit
130     out->counter = CryptCommit();
131
132     return TPM_RC_SUCCESS;
133 }
134
135 #endif // CC_Commit

```

7.65 /tpm/src/command/FieldUpgrade/FieldUpgradeData.c

```

1  #include "Tpm.h"
2  #include "FieldUpgradeData_fp.h"
3  #if CC_FieldUpgradeData // Conditional expansion of this file
4
5  /*(See part 3 specification)
6  // FieldUpgradeData
7  */
8  TPM_RC
9  TPM2_FieldUpgradeData(FieldUpgradeData_In* in, // IN: input parameter list
10                      FieldUpgradeData_Out* out // OUT: output parameter list
11  )
12  {
13      // Not implemented
14      UNUSED_PARAMETER(in);
15      UNUSED_PARAMETER(out);
16      return TPM_RC_SUCCESS;
17  }
18  #endif

```

7.66 /tpm/src/command/FieldUpgrade/FieldUpgradeStart.c

```

1  #include "Tpm.h"
2  #include "FieldUpgradeStart_fp.h"
3  #if CC_FieldUpgradeStart // Conditional expansion of this file
4
5  /*(See part 3 specification)
6  // FieldUpgradeStart
7  */
8  TPM_RC
9  TPM2_FieldUpgradeStart(FieldUpgradeStart_In* in // IN: input parameter list
10  )
11  {
12      // Not implemented
13      UNUSED_PARAMETER(in);
14      return TPM_RC_SUCCESS;
15  }
16  #endif

```

7.67 /tpm/src/command/FieldUpgrade/FirmwareRead.c

```

1  #include "Tpm.h"
2  #include "FirmwareRead_fp.h"
3
4  #if CC_FirmwareRead // Conditional expansion of this file

```

```

5
6  /*(See part 3 specification)
7  // FirmwareRead
8  */
9  TPM_RC
10 TPM2_FirmwareRead(FirmwareRead_In* in, // IN: input parameter list
11                  FirmwareRead_Out* out // OUT: output parameter list
12 )
13 {
14     // Not implemented
15     UNUSED_PARAMETER(in);
16     UNUSED_PARAMETER(out);
17     return TPM_RC_SUCCESS;
18 }
19
20 #endif // CC_FirmwareRead

```

7.68 /tpm/src/command/HashHMAC/EventSequenceComplete.c

```

1  #include "Tpm.h"
2  #include "EventSequenceComplete_fp.h"
3
4  #if CC_EventSequenceComplete // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  Complete an event sequence and flush the object.
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_LOCALITY    PCR extension is not allowed at the current locality
11 //     TPM_RC_MODE        input handle is not a valid event sequence object
12 TPM_RC
13 TPM2_EventSequenceComplete(
14     EventSequenceComplete_In* in, // IN: input parameter list
15     EventSequenceComplete_Out* out // OUT: output parameter list
16 )
17 {
18     HASH_OBJECT* hashObject;
19     UINT32 i;
20     TPM_ALG_ID hashAlg;
21     // Input validation
22     // get the event sequence object pointer
23     hashObject = (HASH_OBJECT*)HandleToObject(in->sequenceHandle);
24
25     // input handle must reference an event sequence object
26     if(hashObject->attributes.eventSeq != SET)
27         return TPM_RCS_MODE + RC_EventSequenceComplete_sequenceHandle;
28
29     // see if a PCR extend is requested in call
30     if(in->pcrHandle != TPM_RH_NULL)
31     {
32         // see if extend of the PCR is allowed at the locality of the command,
33         if(!PCRIsExtendAllowed(in->pcrHandle))
34             return TPM_RC_LOCALITY;
35         // if an extend is going to take place, then check to see if there has
36         // been an orderly shutdown. If so, and the selected PCR is one of the
37         // state saved PCR, then the orderly state has to change. The orderly state
38         // does not change for PCR that are not preserved.
39         // NOTE: This doesn't just check for Shutdown(STATE) because the orderly
40         // state will have to change if this is a state-saved PCR regardless
41         // of the current state. This is because a subsequent Shutdown(STATE) will
42         // check to see if there was an orderly shutdown and not do anything if
43         // there was. So, this must indicate that a future Shutdown(STATE) has
44         // something to do.
45         if(PCRIsStateSaved(in->pcrHandle))
46             RETURN_IF_ORDERLY;

```

```

47     }
48     // Command Output
49     out->results.count = 0;
50
51     for(i = 0; i < HASH_COUNT; i++)
52     {
53         hashAlg = CryptHashGetAlgByIndex(i);
54         // Update last piece of data
55         CryptDigestUpdate2B(&hashObject->state.hashState[i], &in->buffer.b);
56         // Complete hash
57         out->results.digests[out->results.count].hashAlg = hashAlg;
58         CryptHashEnd(&hashObject->state.hashState[i],
59                     CryptHashGetDigestSize(hashAlg),
60                     (BYTE*)&out->results.digests[out->results.count].digest);
61         // Extend PCR
62         if(in->pcrHandle != TPM_RH_NULL)
63             PCRExtend(in->pcrHandle,
64                     hashAlg,
65                     CryptHashGetDigestSize(hashAlg),
66                     (BYTE*)&out->results.digests[out->results.count].digest);
67         out->results.count++;
68     }
69     // Internal Data Update
70     // mark sequence object as evict so it will be flushed on the way out
71     hashObject->attributes.evict = SET;
72
73     return TPM_RC_SUCCESS;
74 }
75
76 #endif // CC_EventSequenceComplete

```

7.69 /tpm/src/command/HashHMAC/HashSequenceStart.c

```

1  #include "Tpm.h"
2  #include "HashSequenceStart_fp.h"
3
4  #if CC_HashSequenceStart // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Start a hash or an event sequence
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_OBJECT_MEMORY    no space to create an internal object
11 TPM_RC
12 TPM2_HashSequenceStart(HashSequenceStart_In* in, // IN: input parameter list
13                       HashSequenceStart_Out* out // OUT: output parameter list
14 )
15 {
16     // Internal Data Update
17
18     if(in->hashAlg == TPM_ALG_NULL)
19         // Start a event sequence. A TPM_RC_OBJECT_MEMORY error may be
20         // returned at this point
21         return ObjectCreateEventSequence(&in->auth, &out->sequenceHandle);
22
23     // Start a hash sequence. A TPM_RC_OBJECT_MEMORY error may be
24     // returned at this point
25     return ObjectCreateHashSequence(in->hashAlg, &in->auth, &out->sequenceHandle);
26 }
27
28 #endif // CC_HashSequenceStart

```


7.70 /tpm/src/command/HashHMAC/HMAC_Start.c

```

1  #include "Tpm.h"
2  #include "HMAC_Start_fp.h"
3
4  #if CC_HMAC_Start // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Initialize a HMAC sequence and create a sequence object
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES      key referenced by 'handle' is not a signing key
11 //                             or is restricted
12 //     TPM_RC_OBJECT_MEMORY   no space to create an internal object
13 //     TPM_RC_KEY             key referenced by 'handle' is not an HMAC key
14 //     TPM_RC_VALUE           'hashAlg' is not compatible with the hash algorithm
15 //                             of the scheme of the object referenced by 'handle'
16 TPM_RC
17 TPM2_HMAC_Start(HMAC_Start_In* in, // IN: input parameter list
18               HMAC_Start_Out* out // OUT: output parameter list
19 )
20 {
21     OBJECT*      keyObject;
22     TPMT_PUBLIC* publicArea;
23     TPM_ALG_ID   hashAlg;
24
25     // Input Validation
26
27     // Get HMAC key object and public area pointers
28     keyObject = HandleToObject(in->handle);
29     publicArea = &keyObject->publicArea;
30
31     // Make sure that the key is an HMAC key
32     if(publicArea->type != TPM_ALG_KEYEDHASH)
33         return TPM_RCS_TYPE + RC_HMAC_Start_handle;
34
35     // and that it is unrestricted
36     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
37         return TPM_RCS_ATTRIBUTES + RC_HMAC_Start_handle;
38
39     // and that it is a signing key
40     if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
41         return TPM_RCS_KEY + RC_HMAC_Start_handle;
42
43     // See if the key has a default
44     if(publicArea->parameters.keyedHashDetail.scheme.scheme == TPM_ALG_NULL)
45         // it doesn't so use the input value
46         hashAlg = in->hashAlg;
47     else
48     {
49         // key has a default so use it
50         hashAlg = publicArea->parameters.keyedHashDetail.scheme.details.hmac.hashAlg;
51         // and verify that the input was either the TPM_ALG_NULL or the default
52         if(in->hashAlg != TPM_ALG_NULL && in->hashAlg != hashAlg)
53             hashAlg = TPM_ALG_NULL;
54     }
55     // if we ended up without a hash algorithm then return an error
56     if(hashAlg == TPM_ALG_NULL)
57         return TPM_RCS_VALUE + RC_HMAC_Start_hashAlg;
58
59     // Internal Data Update
60
61     // Create a HMAC sequence object. A TPM_RC_OBJECT_MEMORY error may be
62     // returned at this point
63     return ObjectCreateHMACSequence(
64         hashAlg, keyObject, &in->auth, &out->sequenceHandle);

```

```

65 }
66
67 #endif // CC_HMAC_Start

```

7.71 /tpm/src/command/HashHMAC/MAC_Start.c

```

1  #include "Tpm.h"
2  #include "MAC_Start_fp.h"
3
4  #if CC_MAC_Start // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Initialize a HMAC sequence and create a sequence object
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES      key referenced by 'handle' is not a signing key
11 //                             or is restricted
12 //     TPM_RC_OBJECT_MEMORY   no space to create an internal object
13 //     TPM_RC_KEY             key referenced by 'handle' is not an HMAC key
14 //     TPM_RC_VALUE           'hashAlg' is not compatible with the hash algorithm
15 //                             of the scheme of the object referenced by 'handle'
16 TPM_RC
17 TPM2_MAC_Start(MAC_Start_In* in, // IN: input parameter list
18               MAC_Start_Out* out // OUT: output parameter list
19 )
20 {
21     OBJECT*      keyObject;
22     TPMT_PUBLIC* publicArea;
23     TPM_RC       result;
24
25     // Input Validation
26
27     // Get HMAC key object and public area pointers
28     keyObject = HandleToObject(in->handle);
29     publicArea = &keyObject->publicArea;
30
31     // Make sure that the key can do what is required
32     result = CryptSelectMac(publicArea, &in->inScheme);
33     // If the key is not able to do a MAC, indicate that the handle selects an
34     // object that can't do a MAC
35     if(result == TPM_RCS_TYPE)
36         return TPM_RCS_TYPE + RC_MAC_Start_handle;
37     // If there is another error type, indicate that the scheme and key are not
38     // compatible
39     if(result != TPM_RC_SUCCESS)
40         return RcSafeAddToResult(result, RC_MAC_Start_inScheme);
41     // Make sure that the key is not restricted
42     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
43         return TPM_RCS_ATTRIBUTES + RC_MAC_Start_handle;
44     // and that it is a signing key
45     if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
46         return TPM_RCS_KEY + RC_MAC_Start_handle;
47
48     // Internal Data Update
49     // Create a HMAC sequence object. A TPM_RC_OBJECT_MEMORY error may be
50     // returned at this point
51     return ObjectCreateHMACSequence(
52         in->inScheme, keyObject, &in->auth, &out->sequenceHandle);
53 }
54
55 #endif // CC_MAC_Start

```

7.72 /tpm/src/command/HashHMAC/SequenceComplete.c

```

1  #include "Tpm.h"
2  #include "SequenceComplete_fp.h"
3
4  #if CC_SequenceComplete // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Complete a sequence and flush the object.
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_MODE 'sequenceHandle' does not reference a hash or HMAC
11 // sequence object
12 TPM_RC
13 TPM2_SequenceComplete(SequenceComplete_In* in, // IN: input parameter list
14                      SequenceComplete_Out* out // OUT: output parameter list
15 )
16 {
17     HASH_OBJECT* hashObject;
18     // Input validation
19     // Get hash object pointer
20     hashObject = (HASH_OBJECT*)HandleToObject(in->sequenceHandle);
21
22     // input handle must be a hash or HMAC sequence object.
23     if(hashObject->attributes.hashSeq == CLEAR
24        && hashObject->attributes.hmacSeq == CLEAR)
25         return TPM_RCS_MODE + RC_SequenceComplete_sequenceHandle;
26     // Command Output
27     if(hashObject->attributes.hashSeq == SET) // sequence object for hash
28     {
29         // Get the hash algorithm before the algorithm is lost in CryptHashEnd
30         TPM_ALG_ID hashAlg = hashObject->state.hashState[0].hashAlg;
31
32         // Update last piece of the data
33         CryptDigestUpdate2B(&hashObject->state.hashState[0], &in->buffer.b);
34
35         // Complete hash
36         out->result.t.size = CryptHashEnd(&hashObject->state.hashState[0],
37                                         sizeof(out->result.t.buffer),
38                                         out->result.t.buffer);
39         // Check if the first block of the sequence has been received
40         if(hashObject->attributes.firstBlock == CLEAR)
41         {
42             // If not, then this is the first block so see if it is 'safe'
43             // to sign.
44             if(TicketIsSafe(&in->buffer.b))
45                 hashObject->attributes.ticketSafe = SET;
46         }
47         // Output ticket
48         out->validation.tag = TPM_ST_HASHCHECK;
49         out->validation.hierarchy = in->hierarchy;
50
51         if(in->hierarchy == TPM_RH_NULL)
52         {
53             // Ticket is not required
54             out->validation.digest.t.size = 0;
55         }
56         else if(hashObject->attributes.ticketSafe == CLEAR)
57         {
58             // Ticket is not safe to generate
59             out->validation.hierarchy = TPM_RH_NULL;
60             out->validation.digest.t.size = 0;
61         }
62         else
63         {
64             TPM_RC result;

```

```

65         // Compute ticket
66         result = TicketComputeHashCheck(
67             out->validation.hierarchy, hashAlg, &out->result, &out->validation);
68         if(result != TPM_RC_SUCCESS)
69             return result;
70     }
71 }
72 else
73 {
74     // Update last piece of data
75     CryptDigestUpdate2B(&hashObject->state.hmacState.hashState, &in->buffer.b);
76 # if !SMAC_IMPLEMENTED
77     // Complete HMAC
78     out->result.t.size = CryptHmacEnd(&(hashObject->state.hmacState),
79                                     sizeof(out->result.t.buffer),
80                                     out->result.t.buffer);
81 # else
82     // Complete the MAC
83     out->result.t.size = CryptMacEnd(&(hashObject->state.hmacState),
84                                     sizeof(out->result.t.buffer),
85                                     out->result.t.buffer);
86 # endif
87     // No ticket is generated for HMAC sequence
88     out->validation.tag = TPM_ST_HASHCHECK;
89     out->validation.hierarchy = TPM_RH_NULL;
90     out->validation.digest.t.size = 0;
91 }
92 // Internal Data Update
93 // mark sequence object as evict so it will be flushed on the way out
94 hashObject->attributes.evict = SET;
95
96 return TPM_RC_SUCCESS;
97 }
98
99 #endif // CC_SequenceComplete

```

7.73 /tpm/src/command/HashHMAC/SequenceUpdate.c

```

1  #include "Tpm.h"
2  #include "SequenceUpdate_fp.h"
3
4  #if CC_SequenceUpdate // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This function is used to add data to a sequence object.
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_MODE 'sequenceHandle' does not reference a hash or HMAC
11 // sequence object
12 TPM_RC
13 TPM2_SequenceUpdate(SequenceUpdate_In* in // IN: input parameter list
14 )
15 {
16     OBJECT* object;
17     HASH_OBJECT* hashObject;
18
19     // Input Validation
20
21     // Get sequence object pointer
22     object = HandleToObject(in->sequenceHandle);
23     hashObject = (HASH_OBJECT*)object;
24
25     // Check that referenced object is a sequence object.
26     if(!ObjectIsSequence(object))
27         return TPM_RCS_MODE + RC_SequenceUpdate_sequenceHandle;

```

```

28
29 // Internal Data Update
30
31 if(object->attributes.eventSeq == SET)
32 {
33     // Update event sequence object
34     UINT32 i;
35     for(i = 0; i < HASH_COUNT; i++)
36     {
37         // Update sequence object
38         CryptDigestUpdate2B(&hashObject->state.hashState[i], &in->buffer.b);
39     }
40 }
41 else
42 {
43     // Update hash/HMAC sequence object
44     if(hashObject->attributes.hashSeq == SET)
45     {
46         // Is this the first block of the sequence
47         if(hashObject->attributes.firstBlock == CLEAR)
48         {
49             // If so, indicate that first block was received
50             hashObject->attributes.firstBlock = SET;
51
52             // Check the first block to see if the first block can contain
53             // the TPM_GENERATED_VALUE. If it does, it is not safe for
54             // a ticket.
55             if(TicketIsSafe(&in->buffer.b))
56                 hashObject->attributes.ticketSafe = SET;
57         }
58         // Update sequence object hash/HMAC stack
59         CryptDigestUpdate2B(&hashObject->state.hashState[0], &in->buffer.b);
60     }
61     else if(object->attributes.hmacSeq == SET)
62     {
63         // Update sequence object HMAC stack
64         CryptDigestUpdate2B(&hashObject->state.hmacState.hashState,
65                             &in->buffer.b);
66     }
67 }
68 return TPM_RC_SUCCESS;
69 }
70
71 #endif // CC_SequenceUpdate

```

7.74 /tpm/src/command/Hierarchy/ChangeEPS.c

```

1 #include "Tpm.h"
2 #include "ChangeEPS_fp.h"
3
4 #if CC_ChangeEPS // Conditional expansion of this file
5
6 /*(See part 3 specification)
7 // Reset current EPS value
8 */
9 TPM_RC
10 TPM2_ChangeEPS(ChangeEPS_In* in // IN: input parameter list
11 )
12 {
13     // The command needs NV update. Check if NV is available.
14     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
15     // this point
16     RETURN_IF_NV_IS_NOT_AVAILABLE;
17
18     // Input parameter is not reference in command action

```

```

19     NOT_REFERENCED(in);
20
21     // Internal Data Update
22
23     // Reset endorsement hierarchy seed from RNG
24     CryptRandomGenerate(sizeof(gp.EPSeed.t.buffer), gp.EPSeed.t.buffer);
25
26     // Create new ehProof value from RNG
27     CryptRandomGenerate(sizeof(gp.ehProof.t.buffer), gp.ehProof.t.buffer);
28
29     // Enable endorsement hierarchy
30     gc.ehEnable = TRUE;
31
32     // set authValue buffer to zeros
33     MemorySet(gp.endorsementAuth.t.buffer, 0, gp.endorsementAuth.t.size);
34     // Set endorsement authValue to null
35     gp.endorsementAuth.t.size = 0;
36
37     // Set endorsement authPolicy to null
38     gp.endorsementAlg = TPM_ALG_NULL;
39     gp.endorsementPolicy.t.size = 0;
40
41     // Flush loaded object in endorsement hierarchy
42     ObjectFlushHierarchy(TPM_RH_ENDORSEMENT);
43
44     // Flush evict object of endorsement hierarchy stored in NV
45     NvFlushHierarchy(TPM_RH_ENDORSEMENT);
46
47     // Save hierarchy changes to NV
48     NV_SYNC_PERSISTENT(EPSeed);
49     NV_SYNC_PERSISTENT(ehProof);
50     NV_SYNC_PERSISTENT(endorsementAuth);
51     NV_SYNC_PERSISTENT(endorsementAlg);
52     NV_SYNC_PERSISTENT(endorsementPolicy);
53
54     // orderly state should be cleared because of the update to state clear data
55     g_clearOrderly = TRUE;
56
57     return TPM_RC_SUCCESS;
58 }
59
60 #endif // CC_ChangeEPS

```

7.75 /tpm/src/command/Hierarchy/ChangePPS.c

```

1  #include "Tpm.h"
2  #include "ChangePPS_fp.h"
3
4  #if CC_ChangePPS // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Reset current PPS value
8  */
9  TPM_RC
10 TPM2_ChangePPS(ChangePPS_In* in // IN: input parameter list
11 )
12 {
13     UINT32 i;
14
15     // Check if NV is available. A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE
16     // error may be returned at this point
17     RETURN_IF_NV_IS_NOT_AVAILABLE;
18
19     // Input parameter is not reference in command action
20     NOT_REFERENCED(in);

```



```

21
22     // Internal Data Update
23
24     // Reset platform hierarchy seed from RNG
25     CryptRandomGenerate(sizeof(gp.PPSeed.t.buffer), gp.PPSeed.t.buffer);
26
27     // Create a new phProof value from RNG to prevent the saved platform
28     // hierarchy contexts being loaded
29     CryptRandomGenerate(sizeof(gp.phProof.t.buffer), gp.phProof.t.buffer);
30
31     // Set platform authPolicy to null
32     gc.platformAlg = TPM_ALG_NULL;
33     gc.platformPolicy.t.size = 0;
34
35     // Flush loaded object in platform hierarchy
36     ObjectFlushHierarchy(TPM_RH_PLATFORM);
37
38     // Flush platform evict object and index in NV
39     NvFlushHierarchy(TPM_RH_PLATFORM);
40
41     // Save hierarchy changes to NV
42     NV_SYNC_PERSISTENT(PPSeed);
43     NV_SYNC_PERSISTENT(phProof);
44
45     // Re-initialize PCR policies
46 # if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
47     for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
48     {
49         gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
50         gp.pcrPolicies.policy[i].t.size = 0;
51     }
52     NV_SYNC_PERSISTENT(pcrPolicies);
53 # endif
54
55     // orderly state should be cleared because of the update to state clear data
56     g_clearOrderly = TRUE;
57
58     return TPM_RC_SUCCESS;
59 }
60
61 #endif // CC_ChangePPS

```

7.76 /tpm/src/command/Hierarchy/Clear.c

```

1  #include "Tpm.h"
2  #include "Clear_fp.h"
3
4  #if CC_Clear // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Clear owner
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_DISABLED Clear command has been disabled
11 TPM_RC
12 TPM2_Clear(Clear_In* in // IN: input parameter list
13 )
14 {
15     // Input parameter is not reference in command action
16     NOT_REFERENCED(in);
17
18     // The command needs NV update. Check if NV is available.
19     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
20     // this point
21     RETURN_IF_NV_IS_NOT_AVAILABLE;

```

```

22
23 // Input Validation
24
25 // If Clear command is disabled, return an error
26 if(gp.disableClear)
27     return TPM_RC_DISABLED;
28
29 // Internal Data Update
30
31 // Reset storage hierarchy seed from RNG
32 CryptRandomGenerate(sizeof(gp.SPSeed.t.buffer), gp.SPSeed.t.buffer);
33
34 // Create new shProof and ehProof value from RNG
35 CryptRandomGenerate(sizeof(gp.shProof.t.buffer), gp.shProof.t.buffer);
36 CryptRandomGenerate(sizeof(gp.ehProof.t.buffer), gp.ehProof.t.buffer);
37
38 // Enable storage and endorsement hierarchy
39 gc.shEnable = gc.ehEnable = TRUE;
40
41 // set the authValue buffers to zero
42 MemorySet(&gp.ownerAuth, 0, sizeof(gp.ownerAuth));
43 MemorySet(&gp.endorsementAuth, 0, sizeof(gp.endorsementAuth));
44 MemorySet(&gp.lockoutAuth, 0, sizeof(gp.lockoutAuth));
45
46 // Set storage, endorsement, and lockout authPolicy to null
47 gp.ownerAlg = gp.endorsementAlg = gp.lockoutAlg = TPM_ALG_NULL;
48 MemorySet(&gp.ownerPolicy, 0, sizeof(gp.ownerPolicy));
49 MemorySet(&gp.endorsementPolicy, 0, sizeof(gp.endorsementPolicy));
50 MemorySet(&gp.lockoutPolicy, 0, sizeof(gp.lockoutPolicy));
51
52 // Flush loaded object in storage and endorsement hierarchy
53 ObjectFlushHierarchy(TPM_RH_OWNER);
54 ObjectFlushHierarchy(TPM_RH_ENDORSEMENT);
55
56 // Flush owner and endorsement object and owner index in NV
57 NvFlushHierarchy(TPM_RH_OWNER);
58 NvFlushHierarchy(TPM_RH_ENDORSEMENT);
59
60 // Initialize dictionary attack parameters
61 DAPreInstall_Init();
62
63 // Reset clock
64 go.clock = 0;
65 go.clockSafe = YES;
66 NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
67
68 // Reset counters
69 gp.resetCount = gr.restartCount = gr.clearCount = 0;
70 gp.auditCounter = 0;
71
72 // Save persistent data changes to NV
73 // Note: since there are so many changes to the persistent data structure, the
74 // entire PERSISTENT_DATA structure is written as a unit
75 NvWrite(NV_PERSISTENT_DATA, sizeof(PERSISTENT_DATA), &gp);
76
77 // Reset the PCR authValues (this does not change the PCRs)
78 PCR_ClearAuth();
79
80 // Bump the PCR counter
81 PCRChanged(0);
82
83 // orderly state should be cleared because of the update to state clear data
84 g_clearOrderly = TRUE;
85
86 return TPM_RC_SUCCESS;
87 }

```

```

88
89 #endif // CC_Clear

```

7.77 /tpm/src/command/Hierarchy/ClearControl.c

```

1  #include "Tpm.h"
2  #include "ClearControl_fp.h"
3
4  #if CC_ClearControl // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Enable or disable the execution of TPM2_Clear command
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_AUTH_FAIL          authorization is not properly given
11 TPM_RC
12 TPM2_ClearControl(ClearControl_In* in // IN: input parameter list
13 )
14 {
15     // The command needs NV update.
16     RETURN_IF_NV_IS_NOT_AVAILABLE;
17
18     // Input Validation
19
20     // LockoutAuth may be used to set disableLockoutClear to TRUE but not to FALSE
21     if(in->auth == TPM_RH_LOCKOUT && in->disable == NO)
22         return TPM_RC_AUTH_FAIL;
23
24     // Internal Data Update
25
26     if(in->disable == YES)
27         gp.disableClear = TRUE;
28     else
29         gp.disableClear = FALSE;
30
31     // Record the change to NV
32     NV_SYNC_PERSISTENT(disableClear);
33
34     return TPM_RC_SUCCESS;
35 }
36
37 #endif // CC_ClearControl

```

7.78 /tpm/src/command/Hierarchy/CreatePrimary.c

```

1  #include "Tpm.h"
2  #include "CreatePrimary_fp.h"
3
4  #if CC_CreatePrimary // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Creates a primary or temporary object from a primary seed.
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES          sensitiveDataOrigin is CLEAR when sensitive.data is an
11 //                                Empty Buffer; 'fixedTPM', 'fixedParent', or
12 //                                'encryptedDuplication' attributes are inconsistent
13 //                                between themselves or with those of the parent object;
14 //                                inconsistent 'restricted', 'decrypt', 'sign',
15 //                                'firmwareLimited', or 'svnLimited' attributes;
16 //                                attempt to inject sensitive data for an asymmetric
17 //                                key;
18 //     TPM_RC_FW_LIMITED          The requested hierarchy is FW-limited, but the TPM
19 //                                does not support FW-limited objects or the TPM failed

```

```

20 // to derive the Firmware Secret.
21 // TPM_RC_SVN_LIMITED The requested hierarchy is SVN-limited, but the TPM
22 // does not support SVN-limited objects or the TPM failed
23 // to derive the Firmware SVN Secret for the requested
24 // SVN.
25 // TPM_RC_KDF incorrect KDF specified for decrypting keyed hash
26 // object
27 // TPM_RC_KEY a provided symmetric key value is not allowed
28 // TPM_RC_OBJECT_MEMORY there is no free slot for the object
29 // TPM_RC_SCHEME inconsistent attributes 'decrypt', 'sign',
30 // 'restricted' and key's scheme ID; or hash algorithm is
31 // inconsistent with the scheme ID for keyed hash object
32 // TPM_RC_SIZE size of public authorization policy or sensitive
33 // authorization value does not match digest size of the
34 // name algorithm; or sensitive data size for the keyed
35 // hash object is larger than is allowed for the scheme
36 // TPM_RC_SYMMETRIC a storage key with no symmetric algorithm specified;
37 // or non-storage key with symmetric algorithm different
38 // from TPM_ALG_NULL
39 // TPM_RC_TYPE unknown object type
40 TPM_RC
41 TPM2_CreatePrimary(CreatePrimary_In* in, // IN: input parameter list
42                   CreatePrimary_Out* out // OUT: output parameter list
43 )
44 {
45     TPM_RC result = TPM_RC_SUCCESS;
46     TPMT_PUBLIC* publicArea;
47     DRBG_STATE rand;
48     OBJECT* newObject;
49     TPM2B_NAME name;
50     TPM2B_SEED primary_seed;
51
52     // Input Validation
53     // Will need a place to put the result
54     newObject = FindEmptyObjectSlot(&out->objectHandle);
55     if(newObject == NULL)
56         return TPM_RC_OBJECT_MEMORY;
57     // Get the address of the public area in the new object
58     // (this is just to save typing)
59     publicArea = &newObject->publicArea;
60
61     *publicArea = in->inPublic.publicArea;
62
63     // Check attributes in input public area. CreateChecks() checks the things that
64     // are unique to creation and then validates the attributes and values that are
65     // common to create and load.
66     result = CreateChecks(
67         NULL, in->primaryHandle, publicArea, in->inSensitive.sensitive.data.t.size);
68     if(result != TPM_RC_SUCCESS)
69         return RcSafeAddToResult(result, RC_CreatePrimary_inPublic);
70     // Validate the sensitive area values
71     if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth, publicArea->nameAlg))
72         return TPM_RC_SIZE + RC_CreatePrimary_inSensitive;
73     // Command output
74     // Compute the name using out->name as a scratch area (this is not the value
75     // that ultimately will be returned, then instantiate the state that will be
76     // used as a random number generator during the object creation.
77     // The caller does not know the seed values so the actual name does not have
78     // to be over the input, it can be over the unmarshaled structure.
79
80     result = HierarchyGetPrimarySeed(in->primaryHandle, &primary_seed);
81     if(result != TPM_RC_SUCCESS)
82         return result;
83
84     result =
85         DRBG_InstantiateSeeded(&rand,

```

```

86         &primary_seed.b,
87         PRIMARY_OBJECT_CREATION,
88         (TPM2B*)PublicMarshalAndComputeName(publicArea, &name),
89         &in->inSensitive.sensitive.data.b);
90     MemorySet(primary_seed.b.buffer, 0, primary_seed.b.size);
91
92     if(result == TPM_RC_SUCCESS)
93     {
94         newObject->attributes.primary = SET;
95         if(HierarchyNormalizeHandle(in->primaryHandle) == TPM_RH_ENDORSEMENT)
96             newObject->attributes.epsHierarchy = SET;
97
98         // Create the primary object.
99         result = CryptCreateObject(
100             newObject, &in->inSensitive.sensitive, (RAND_STATE*)&rand);
101         DRBG_Uninstantiate(&rand);
102     }
103     if(result != TPM_RC_SUCCESS)
104         return result;
105
106     // Set the publicArea and name from the computed values
107     out->outPublic.publicArea = newObject->publicArea;
108     out->name = newObject->name;
109
110     // Fill in creation data
111     FillInCreationData(in->primaryHandle,
112         publicArea->nameAlg,
113         &in->creationPCR,
114         &in->outsideInfo,
115         &out->creationData,
116         &out->creationHash);
117
118     // Compute creation ticket
119     result = TicketComputeCreation(EntityGetHierarchy(in->primaryHandle),
120         &out->name,
121         &out->creationHash,
122         &out->creationTicket);
123
124     if(result != TPM_RC_SUCCESS)
125         return result;
126
127     // Set the remaining attributes for a loaded object
128     ObjectSetLoadedAttributes(newObject, in->primaryHandle);
129     return result;
130 }
131 #endif // CC_CreatePrimary

```

7.79 /tpm/src/command/Hierarchy/HierarchyChangeAuth.c

```

1  #include "Tpm.h"
2  #include "HierarchyChangeAuth_fp.h"
3
4  #if CC_HierarchyChangeAuth // Conditional expansion of this file
5
6  # include "Object_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Set a hierarchy authValue
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_SIZE      'newAuth' size is greater than that of integrity hash
13 //                        digest
14 TPM_RC
15 TPM2_HierarchyChangeAuth(HierarchyChangeAuth_In* in // IN: input parameter list
16 )

```

```

17 {
18     // The command needs NV update.
19     RETURN_IF_NV_IS_NOT_AVAILABLE;
20
21     // Make sure that the authorization value is a reasonable size (not larger than
22     // the size of the digest produced by the integrity hash. The integrity
23     // hash is assumed to produce the longest digest of any hash implemented
24     // on the TPM. This will also remove trailing zeros from the authValue.
25     if (MemoryRemoveTrailingZeros(&in->newAuth) > CONTEXT_INTEGRITY_HASH_SIZE)
26         return TPM_RCS_SIZE + RC_HierarchyChangeAuth_newAuth;
27
28     // Set hierarchy authValue
29     switch(in->authHandle)
30     {
31         case TPM_RH_OWNER:
32             gp.ownerAuth = in->newAuth;
33             NV_SYNC_PERSISTENT(ownerAuth);
34             break;
35         case TPM_RH_ENDORSEMENT:
36             gp.endorsementAuth = in->newAuth;
37             NV_SYNC_PERSISTENT(endorsementAuth);
38             break;
39         case TPM_RH_PLATFORM:
40             gc.platformAuth = in->newAuth;
41             // orderly state should be cleared
42             g_clearOrderly = TRUE;
43             break;
44         case TPM_RH_LOCKOUT:
45             gp.lockoutAuth = in->newAuth;
46             NV_SYNC_PERSISTENT(lockoutAuth);
47             break;
48         default:
49             FAIL(FATAL_ERROR_INTERNAL);
50             break;
51     }
52
53     return TPM_RC_SUCCESS;
54 }
55
56 #endif // CC_HierarchyChangeAuth

```

7.80 /tpm/src/command/Hierarchy/HierarchyControl.c

```

1  #include "Tpm.h"
2  #include "HierarchyControl_fp.h"
3
4  #if CC_HierarchyControl // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Enable or disable use of a hierarchy
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_AUTH_TYPE 'authHandle' is not applicable to 'hierarchy' in its
11 //     current state
12 TPM_RC
13 TPM2_HierarchyControl(HierarchyControl_In* in // IN: input parameter list
14 )
15 {
16     BOOL select = (in->state == YES);
17     BOOL* selected = NULL;
18
19     // Input Validation
20     switch(in->enable)
21     {
22         // Platform hierarchy has to be disabled by PlatformAuth

```



```

23     // If the platform hierarchy has already been disabled, only a reboot
24     // can enable it again
25     case TPM_RH_PLATFORM:
26     case TPM_RH_PLATFORM_NV:
27         if(in->authHandle != TPM_RH_PLATFORM)
28             return TPM_RC_AUTH_TYPE;
29         break;
30
31     // ShEnable may be disabled if PlatformAuth/PlatformPolicy or
32     // OwnerAuth/OwnerPolicy is provided. If ShEnable is disabled, then it
33     // may only be enabled if PlatformAuth/PlatformPolicy is provided.
34     case TPM_RH_OWNER:
35         if(in->authHandle != TPM_RH_PLATFORM && in->authHandle != TPM_RH_OWNER)
36             return TPM_RC_AUTH_TYPE;
37         if(gc.shEnable == FALSE && in->state == YES
38            && in->authHandle != TPM_RH_PLATFORM)
39             return TPM_RC_AUTH_TYPE;
40         break;
41
42     // EhEnable may be disabled if either PlatformAuth/PlatformPolicy or
43     // EndorsementAuth/EndorsementPolicy is provided. If EhEnable is disabled,
44     // then it may only be enabled if PlatformAuth/PlatformPolicy is
45     // provided.
46     case TPM_RH_ENDORSEMENT:
47         if(in->authHandle != TPM_RH_PLATFORM
48            && in->authHandle != TPM_RH_ENDORSEMENT)
49             return TPM_RC_AUTH_TYPE;
50         if(gc.ehEnable == FALSE && in->state == YES
51            && in->authHandle != TPM_RH_PLATFORM)
52             return TPM_RC_AUTH_TYPE;
53         break;
54     default:
55         FAIL(FATAL_ERROR_INTERNAL);
56         break;
57 }
58
59 // Internal Data Update
60
61 // Enable or disable the selected hierarchy
62 // Note: the authorization processing for this command may keep these
63 // command actions from being executed. For example, if phEnable is
64 // CLEAR, then platformAuth cannot be used for authorization. This
65 // means that would not be possible to use platformAuth to change the
66 // state of phEnable from CLEAR to SET.
67 // If it is decided that platformPolicy can still be used when phEnable
68 // is CLEAR, then this code could SET phEnable when proper platform
69 // policy is provided.
70 switch(in->enable)
71 {
72     case TPM_RH_OWNER:
73         selected = &gc.shEnable;
74         break;
75     case TPM_RH_ENDORSEMENT:
76         selected = &gc.ehEnable;
77         break;
78     case TPM_RH_PLATFORM:
79         selected = &g_phEnable;
80         break;
81     case TPM_RH_PLATFORM_NV:
82         selected = &gc.phEnableNV;
83         break;
84     default:
85         FAIL(FATAL_ERROR_INTERNAL);
86         break;
87 }
88 if(selected != NULL && *selected != select)

```

```

89     {
90         // Before changing the internal state, make sure that NV is available.
91         // Only need to update NV if changing the orderly state
92         RETURN_IF_ORDERLY;
93
94         // state is changing and NV is available so modify
95         *selected = select;
96         // If a hierarchy was just disabled, flush it
97         if(select == CLEAR && in->enable != TPM_RH_PLATFORM_NV)
98             // Flush hierarchy
99             ObjectFlushHierarchy(in->enable);
100
101         // orderly state should be cleared because of the update to state clear data
102         // This gets processed in ExecuteCommand() on the way out.
103         g_clearOrderly = TRUE;
104     }
105     return TPM_RC_SUCCESS;
106 }
107
108 #endif // CC_HierarchyControl

```

7.81 /tpm/src/command/Hierarchy/SetPrimaryPolicy.c

```

1  #include "Tpm.h"
2  #include "SetPrimaryPolicy_fp.h"
3
4  #if CC_SetPrimaryPolicy // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Set a hierarchy policy
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_SIZE size of input authPolicy is not consistent with
11 // input hash algorithm
12 TPM_RC
13 TPM2_SetPrimaryPolicy(SetPrimaryPolicy_In* in // IN: input parameter list
14 )
15 {
16     // Input Validation
17
18     // Check the authPolicy consistent with hash algorithm. If the policy size is
19     // zero, then the algorithm is required to be TPM_ALG_NULL
20     if(in->authPolicy.t.size != CryptHashGetDigestSize(in->hashAlg))
21         return TPM_RCS_SIZE + RC_SetPrimaryPolicy_authPolicy;
22
23     // The command need NV update for OWNER and ENDORSEMENT hierarchy, and
24     // might need orderlyState update for PLATFORM hierarchy.
25     // Check if NV is available. A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE
26     // error may be returned at this point
27     RETURN_IF_NV_IS_NOT_AVAILABLE;
28
29     // Internal Data Update
30
31     // Set hierarchy policy
32     switch(in->authHandle)
33     {
34         case TPM_RH_OWNER:
35             gp.ownerAlg = in->hashAlg;
36             gp.ownerPolicy = in->authPolicy;
37             NV_SYNC_PERSISTENT(ownerAlg);
38             NV_SYNC_PERSISTENT(ownerPolicy);
39             break;
40         case TPM_RH_ENDORSEMENT:
41             gp.endorsementAlg = in->hashAlg;
42             gp.endorsementPolicy = in->authPolicy;

```

```

43     NV_SYNC_PERSISTENT(endorsementAlg);
44     NV_SYNC_PERSISTENT(endorsementPolicy);
45     break;
46     case TPM_RH_PLATFORM:
47         gc.platformAlg = in->hashAlg;
48         gc.platformPolicy = in->authPolicy;
49         // need to update orderly state
50         g_clearOrderly = TRUE;
51         break;
52     case TPM_RH_LOCKOUT:
53         gp.lockoutAlg = in->hashAlg;
54         gp.lockoutPolicy = in->authPolicy;
55         NV_SYNC_PERSISTENT(lockoutAlg);
56         NV_SYNC_PERSISTENT(lockoutPolicy);
57         break;
58
59 # if ACT_SUPPORT
60 #     define SET_ACT_POLICY(N) \
61     case TPM_RH_ACT_##N: \
62         go.ACT_##N.hashAlg = in->hashAlg; \
63         go.ACT_##N.authPolicy = in->authPolicy; \
64         g_clearOrderly = TRUE; \
65         break;
66
67     FOR_EACH_ACT(SET_ACT_POLICY)
68 # endif // ACT_SUPPORT
69
70     default:
71         FAIL(FATAL_ERROR_INTERNAL);
72         break;
73 }
74
75 return TPM_RC_SUCCESS;
76 }
77
78 #endif // CC_SetPrimaryPolicy

```

7.82 /tpm/src/command/Misc/PP_Commands.c

```

1  #include "Tpm.h"
2  #include "PP_Commands_fp.h"
3
4  #if CC_PP_Commands // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command is used to determine which commands require assertion of
8  // Physical Presence in addition to platformAuth/platformPolicy.
9  */
10 TPM_RC
11 TPM2_PP_Commands(PP_Commands_In* in // IN: input parameter list
12 )
13 {
14     UINT32 i;
15
16     // The command needs NV update. Check if NV is available.
17     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
18     // this point
19     RETURN_IF_NV_IS_NOT_AVAILABLE;
20
21     // Internal Data Update
22
23     // Process set list
24     for(i = 0; i < in->setList.count; i++)
25         // If command is implemented, set it as PP required. If the input
26         // command is not a PP command, it will be ignored at

```

```

27     // PhysicalPresenceCommandSet().
28     // Note: PhysicalPresenceCommandSet() checks if the command is implemented.
29     PhysicalPresenceCommandSet(in->setList.commandCodes[i]);
30
31     // Process clear list
32     for(i = 0; i < in->clearList.count; i++)
33         // If command is implemented, clear it as PP required. If the input
34         // command is not a PP command, it will be ignored at
35         // PhysicalPresenceCommandClear(). If the input command is
36         // TPM2_PP_Commands, it will be ignored as well
37         PhysicalPresenceCommandClear(in->clearList.commandCodes[i]);
38
39     // Save the change of PP list
40     NV_SYNC_PERSISTENT(ppList);
41
42     return TPM_RC_SUCCESS;
43 }
44
45 #endif // CC_PP_Commands

```

7.83 /tpm/src/command/Misc/SetAlgorithmSet.c

```

1  #include "Tpm.h"
2  #include "SetAlgorithmSet_fp.h"
3
4  #if CC_SetAlgorithmSet // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command allows the platform to change the algorithm set setting of the TPM
8  */
9  TPM_RC
10 TPM2_SetAlgorithmSet(SetAlgorithmSet_In* in // IN: input parameter list
11 )
12 {
13     // The command needs NV update. Check if NV is available.
14     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
15     // this point
16     RETURN_IF_NV_IS_NOT_AVAILABLE;
17
18     // Internal Data Update
19     gp.algorithmSet = in->algorithmSet;
20
21     // Write the algorithm set changes to NV
22     NV_SYNC_PERSISTENT(algorithmSet);
23
24     return TPM_RC_SUCCESS;
25 }
26
27 #endif // CC_SetAlgorithmSet

```

7.84 /tpm/src/command/NVStorage/NV_Certify.c

```

1  #include "Tpm.h"
2  #include "Attest_spt_fp.h"
3  #include "NV_Certify_fp.h"
4
5  #if CC_NV_Certify // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // certify the contents of an NV index or portion of an NV index
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_NV_AUTHORIZATION           the authorization was valid but the
12 //                                         authorizing entity ('authHandle')

```

```

13 // is not allowed to read from the Index
14 // referenced by 'nvIndex'
15 // TPM_RC_KEY 'signHandle' does not reference a signing
16 // key
17 // TPM_RC_NV_LOCKED Index referenced by 'nvIndex' is locked
18 // for reading
19 // TPM_RC_NV_RANGE 'offset' plus 'size' extends outside of the
20 // data range of the Index referenced by
21 // 'nvIndex'
22 // TPM_RC_NV_UNINITIALIZED Index referenced by 'nvIndex' has not been
23 // written
24 // TPM_RC_SCHEME 'inScheme' is not an allowed value for the
25 // key definition
26 TPM_RC
27 TPM2_NV_Certify(NV_Certify_In* in, // IN: input parameter list
28                 NV_Certify_Out* out // OUT: output parameter list
29 )
30 {
31     TPM_RC result;
32     NV_REF locator;
33     NV_INDEX* nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
34     TPMS_ATTEST certifyInfo;
35     OBJECT* signObject = HandleToObject(in->signHandle);
36     // Input Validation
37     if(!IsSigningObject(signObject))
38         return TPM_RCS_KEY + RC_NV_Certify_signHandle;
39     if(!CryptSelectSignScheme(signObject, &in->inScheme))
40         return TPM_RCS_SCHEME + RC_NV_Certify_inScheme;
41
42     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
43     // or TPM_RC_NV_LOCKED
44     result = NvReadAccessChecks(
45         in->authHandle, in->nvIndex, nvIndex->publicArea.attributes);
46     if(result != TPM_RC_SUCCESS)
47         return result;
48
49     // make sure that the selection is within the range of the Index (cast to avoid
50     // any wrap issues with addition)
51     if((UINT32)in->size + (UINT32)in->offset > (UINT32)nvIndex->publicArea.dataSize)
52         return TPM_RC_NV_RANGE;
53     // Make sure the data will fit the return buffer.
54     // NOTE: This check may be modified if the output buffer will not hold the
55     // maximum sized NV buffer as part of the certified data. The difference in
56     // size could be substantial if the signature scheme was produced a large
57     // signature (e.g., RSA 4096).
58     if(in->size > MAX_NV_BUFFER_SIZE)
59         return TPM_RCS_VALUE + RC_NV_Certify_size;
60
61     // Command Output
62
63     // Fill in attest information common fields
64     FillInAttestInfo(
65         in->signHandle, &in->inScheme, &in->qualifyingData, &certifyInfo);
66
67     // Get the name of the index
68     NvGetIndexName(nvIndex, &certifyInfo.attested.nv.indexName);
69
70     // See if this is old format or new format
71     if((in->size != 0) || (in->offset != 0))
72     {
73         // NV certify specific fields
74         // Attestation type
75         certifyInfo.type = TPM_ST_ATTEST_NV;
76
77         // Set the return size
78         certifyInfo.attested.nv.nvContents.t.size = in->size;

```

```

79
80     // Set the offset
81     certifyInfo.attested.nv.offset = in->offset;
82
83     // Perform the read
84     NvGetIndexData(nvIndex,
85                   locator,
86                   in->offset,
87                   in->size,
88                   certifyInfo.attested.nv.nvContents.t.buffer);
89 }
90 else
91 {
92     HASH_STATE hashState;
93     // This is to sign a digest of the data
94     certifyInfo.type = TPM_ST_ATTEST_NV_DIGEST;
95     // Initialize the hash before calling the function to add the Index data to
96     // the hash.
97     certifyInfo.attested.nvDigest.nvDigest.t.size =
98         CryptHashStart(&hashState, in->inScheme.details.any.hashAlg);
99     NvHashIndexData(
100         &hashState, nvIndex, locator, 0, nvIndex->publicArea.dataSize);
101     CryptHashEnd2B(&hashState, &certifyInfo.attested.nvDigest.nvDigest.b);
102 }
103 // Sign attestation structure. A NULL signature will be returned if
104 // signObject is NULL.
105 return SignAttestInfo(signObject,
106                      &in->inScheme,
107                      &certifyInfo,
108                      &in->qualifyingData,
109                      &out->certifyInfo,
110                      &out->signature);
111 }
112
113 #endif // CC_NV_Certify

```

7.85 /tpm/src/command/NVStorage/NV_ChangeAuth.c

```

1  #include "Tpm.h"
2  #include "NV_ChangeAuth_fp.h"
3
4  #if CC_NV_ChangeAuth // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // change authorization value of a NV index
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_SIZE 'newAuth' size is larger than the digest
11 // size of the Name algorithm for the Index
12 // referenced by 'nvIndex'
13 TPM_RC
14 TPM2_NV_ChangeAuth(NV_ChangeAuth_In* in // IN: input parameter list
15 )
16 {
17     NV_REF locator;
18     NV_INDEX* nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
19
20     // Input Validation
21
22     // Remove trailing zeros and make sure that the result is not larger than the
23     // digest of the nameAlg.
24     if(MemoryRemoveTrailingZeros(&in->newAuth)
25        > CryptHashGetDigestSize(nvIndex->publicArea.nameAlg))
26         return TPM_RCS_SIZE + RC_NV_ChangeAuth_newAuth;
27

```



```

28     // Internal Data Update
29     // Change authValue
30     return NvWriteIndexAuth(locator, &in->newAuth);
31 }
32
33 #endif // CC_NV_ChangeAuth

```

7.86 /tpm/src/command/NVStorage/NV_DefineSpace.c

```

1  #include "Tpm.h"
2  #include "NV_DefineSpace_fp.h"
3
4  #if CC_NV_DefineSpace // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Define a NV index space
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_HIERARCHY           for authorizations using TPM_RH_PLATFORM
11 //                               phEnable_NV is clear preventing access to NV
12 //                               data in the platform hierarchy.
13 //     TPM_RC_ATTRIBUTES         attributes of the index are not consistent
14 //     TPM_RC_NV_DEFINED          index already exists
15 //     TPM_RC_NV_SPACE           insufficient space for the index
16 //     TPM_RC_SIZE                'auth->size' or 'publicInfo->authPolicy.size' is
17 //                               larger than the digest size of
18 //                               'publicInfo->nameAlg'; or 'publicInfo->dataSize'
19 //                               is not consistent with 'publicInfo->attributes'
20 //                               (this includes the case when the index is
21 //                               larger than a MAX_NV_BUFFER_SIZE but the
22 //                               TPMA_NV_WRITEALL attribute is SET)
23 TPM_RC
24 TPM2_NV_DefineSpace(NV_DefineSpace_In* in // IN: input parameter list
25 )
26 {
27     // This command only supports TPM_HT_NV_INDEX-typed NV indices.
28     if(HandleGetType(in->publicInfo.nvPublic.nvIndex) != TPM_HT_NV_INDEX)
29     {
30         return TPM_RCS_HANDLE + RC_NV_DefineSpace_publicInfo;
31     }
32
33     return NvDefineSpace(in->authHandle,
34                          &in->auth,
35                          &in->publicInfo.nvPublic,
36                          RC_NV_DefineSpace_authHandle,
37                          RC_NV_DefineSpace_auth,
38                          RC_NV_DefineSpace_publicInfo);
39 }
40
41 #endif // CC_NV_DefineSpace

```

7.87 /tpm/src/command/NVStorage/NV_DefineSpace2.c

```

1  #include "Tpm.h"
2  #include "NV_DefineSpace2_fp.h"
3
4  #if CC_NV_DefineSpace2 // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Define a NV index space
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_HIERARCHY           for authorizations using TPM_RH_PLATFORM
11 //                               phEnable_NV is clear preventing access to NV

```

```

12 // data in the platform hierarchy.
13 // attributes of the index are not consistent
14 // TPM_RC_NV_DEFINED index already exists
15 // TPM_RC_NV_SPACE insufficient space for the index
16 // TPM_RC_SIZE 'auth->size' or 'publicInfo->authPolicy.size' is
17 // larger than the digest size of
18 // 'publicInfo->nameAlg'; or 'publicInfo->dataSize'
19 // is not consistent with 'publicInfo->attributes'
20 // (this includes the case when the index is
21 // larger than a MAX_NV_BUFFER_SIZE but the
22 // TPMA_NV_WRITEALL attribute is SET)
23 TPM_RC
24 TPM2_NV_DefineSpace2(NV_DefineSpace2_In* in // IN: input parameter list
25 )
26 {
27     TPM_RC result;
28     TPMS_NV_PUBLIC legacyPublic;
29
30     // Input Validation
31
32     // Validate the handle type and the (handle-type-specific) attributes.
33     switch(in->publicInfo.nvPublic2.handleType)
34     {
35         case TPM_HT_NV_INDEX:
36             break;
37     # if EXTERNAL_NV
38         case TPM_HT_EXTERNAL_NV:
39             // The reference implementation may let you define an "external" NV
40             // index, but it doesn't currently support setting any of the extended
41             // bits for customizing the behavior of external NV.
42             if((TPMA_NV_EXP_TO_UINT64(
43                 in->publicInfo.nvPublic2.nvPublic2.externalNV.attributes)
44                 & 0xffffffff00000000)
45                 != 0)
46             {
47                 return TPM_RCS_ATTRIBUTES + RC_NV_DefineSpace2_publicInfo;
48             }
49             break;
50     # endif
51         default:
52             return TPM_RCS_HANDLE + RC_NV_DefineSpace2_publicInfo;
53     }
54
55     result = NvPublicFromNvPublic2(&in->publicInfo.nvPublic2, &legacyPublic);
56     if(result != TPM_RC_SUCCESS)
57     {
58         return RcSafeAddToResult(result, RC_NV_DefineSpace2_publicInfo);
59     }
60
61     return NvDefineSpace(in->authHandle,
62                         &in->auth,
63                         &legacyPublic,
64                         RC_NV_DefineSpace2_authHandle,
65                         RC_NV_DefineSpace2_auth,
66                         RC_NV_DefineSpace2_publicInfo);
67 }
68
69 #endif // CC_NV_DefineSpace

```

7.88 /tpm/src/command/NVStorage/NV_Extend.c

```

1 #include "Tpm.h"
2 #include "NV_Extend_fp.h"
3
4 #if CC_NV_Extend // Conditional expansion of this file

```

```

5
6  /*(See part 3 specification)
7  // Write to a NV index
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES           the TPMA_NV_EXTEND attribute is not SET in
11 //                                the Index referenced by 'nvIndex'
12 //     TPM_RC_NV_AUTHORIZATION     the authorization was valid but the
13 //                                authorizing entity ('authHandle')
14 //                                is not allowed to write to the Index
15 //                                referenced by 'nvIndex'
16 //     TPM_RC_NV_LOCKED            the Index referenced by 'nvIndex' is locked
17 //                                for writing
18 TPM_RC
19 TPM2_NV_Extend(NV_Extend_In* in // IN: input parameter list
20 )
21 {
22     TPM_RC      result;
23     NV_REF      locator;
24     NV_INDEX*   nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
25
26     TPM2B_DIGEST oldDigest;
27     TPM2B_DIGEST newDigest;
28     HASH_STATE   hashState;
29
30     // Input Validation
31
32     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
33     // or TPM_RC_NV_LOCKED
34     result = NvWriteAccessChecks(
35         in->authHandle, in->nvIndex, nvIndex->publicArea.attributes);
36     if(result != TPM_RC_SUCCESS)
37         return result;
38
39     // Make sure that this is an extend index
40     if(!IsNvExtendIndex(nvIndex->publicArea.attributes))
41         return TPM_RCS_ATTRIBUTES + RC_NV_Extend_nvIndex;
42
43     // Internal Data Update
44
45     // Perform the write.
46     oldDigest.t.size = CryptHashGetDigestSize(nvIndex->publicArea.nameAlg);
47     pAssert(oldDigest.t.size <= sizeof(oldDigest.t.buffer));
48     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
49     {
50         NvGetIndexData(nvIndex, locator, 0, oldDigest.t.size, oldDigest.t.buffer);
51     }
52     else
53     {
54         MemorySet(oldDigest.t.buffer, 0, oldDigest.t.size);
55     }
56     // Start hash
57     newDigest.t.size = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
58
59     // Adding old digest
60     CryptDigestUpdate2B(&hashState, &oldDigest.b);
61
62     // Adding new data
63     CryptDigestUpdate2B(&hashState, &in->data.b);
64
65     // Complete hash
66     CryptHashEnd2B(&hashState, &newDigest.b);
67
68     // Write extended hash back.
69     // Note, this routine will SET the TPMA_NV_WRITTEN attribute if necessary
70     return NvWriteIndexData(nvIndex, 0, newDigest.t.size, newDigest.t.buffer);

```

```

71 }
72
73 #endif // CC_NV_Extend

```

7.89 /tpm/src/command/NVStorage/NV_GlobalWriteLock.c

```

1  #include "Tpm.h"
2  #include "NV_GlobalWriteLock_fp.h"
3
4  #if CC_NV_GlobalWriteLock // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Set global write lock for NV index
8  */
9  TPM_RC
10 TPM2_NV_GlobalWriteLock(NV_GlobalWriteLock_In* in // IN: input parameter list
11 )
12 {
13     // Input parameter (the authorization handle) is not reference in command action.
14     NOT_REFERENCED(in);
15
16     // Internal Data Update
17
18     // Implementation dependent method of setting the global lock
19     return NvSetGlobalLock();
20 }
21
22 #endif // CC_NV_GlobalWriteLock

```

7.90 /tpm/src/command/NVStorage/NV_Increment.c

```

1  #include "Tpm.h"
2  #include "NV_Increment_fp.h"
3
4  #if CC_NV_Increment // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Increment a NV counter
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES          NV index is not a counter
11 //     TPM_RC_NV_AUTHORIZATION    authorization failure
12 //     TPM_RC_NV_LOCKED          Index is write locked
13 TPM_RC
14 TPM2_NV_Increment(NV_Increment_In* in // IN: input parameter list
15 )
16 {
17     TPM_RC    result;
18     NV_REF    locator;
19     NV_INDEX* nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
20     UINT64    countValue;
21
22     // Input Validation
23
24     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
25     // or TPM_RC_NV_LOCKED
26     result = NvWriteAccessChecks(
27         in->authHandle, in->nvIndex, nvIndex->publicArea.attributes);
28     if(result != TPM_RC_SUCCESS)
29         return result;
30
31     // Make sure that this is a counter
32     if(!IsNvCounterIndex(nvIndex->publicArea.attributes))
33         return TPM_RCS_ATTRIBUTES + RC_NV_Increment_nvIndex;

```

```

34
35 // Internal Data Update
36
37 // If counter index is not been written, initialize it
38 if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
39     countValue = NvReadMaxCount();
40 else
41     // Read NV data in native format for TPM CPU.
42     countValue = NvGetUINT64Data(nvIndex, locator);
43
44 // Do the increment
45 countValue++;
46
47 // Write NV data back. A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may
48 // be returned at this point. If necessary, this function will set the
49 // TPMA_NV_WRITTEN attribute
50 result = NvWriteUINT64Data(nvIndex, countValue);
51 if(result == TPM_RC_SUCCESS)
52 {
53     // If a counter just rolled over, then force the NV update.
54     // Note, if this is an orderly counter, then the write-back needs to be
55     // forced, for other counters, the write-back will happen anyway
56     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY)
57        && (countValue & MAX_ORDERLY_COUNT) == 0)
58     {
59         // Need to force an NV update of orderly data
60         SET_NV_UPDATE(UT_ORDERLY);
61     }
62 }
63 return result;
64 }
65
66 #endif // CC_NV_Increment

```

7.91 /tpm/src/command/NVStorage/NV_Read.c

```

1  #include "Tpm.h"
2  #include "NV_Read_fp.h"
3
4  #if CC_NV_Read // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Read of an NV index
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_NV_AUTHORIZATION    the authorization was valid but the
11 //                                authorizing entity ('authHandle')
12 //                                is not allowed to read from the Index
13 //                                referenced by 'nvIndex'
14 //     TPM_RC_NV_LOCKED           the Index referenced by 'nvIndex' is
15 //                                read locked
16 //     TPM_RC_NV_RANGE            read range defined by 'size' and 'offset'
17 //                                is outside the range of the Index referenced
18 //                                by 'nvIndex'
19 //     TPM_RC_NV_UNINITIALIZED    the Index referenced by 'nvIndex' has
20 //                                not been initialized (written)
21 //     TPM_RC_VALUE               the read size is larger than the
22 //                                MAX_NV_BUFFER_SIZE
23 TPM_RC
24 TPM2_NV_Read(NV_Read_In* in, // IN: input parameter list
25             NV_Read_Out* out // OUT: output parameter list
26 )
27 {
28     NV_REF locator;
29     NV_INDEX* nvIndex = NvGetIndexInfo(in->nvIndex, &locator);

```

```

30     TPM_RC    result;
31
32     // Input Validation
33     // Common read access checks. NvReadAccessChecks() may return
34     // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
35     result = NvReadAccessChecks(
36         in->authHandle, in->nvIndex, nvIndex->publicArea.attributes);
37     if(result != TPM_RC_SUCCESS)
38         return result;
39
40     // Make sure the data will fit the return buffer
41     if(in->size > MAX_NV_BUFFER_SIZE)
42         return TPM_RCS_VALUE + RC_NV_Read_size;
43
44     // Verify that the offset is not too large
45     if(in->offset > nvIndex->publicArea.dataSize)
46         return TPM_RCS_VALUE + RC_NV_Read_offset;
47
48     // Make sure that the selection is within the range of the Index
49     if(in->size > (nvIndex->publicArea.dataSize - in->offset))
50         return TPM_RC_NV_RANGE;
51
52     // Command Output
53     // Set the return size
54     out->data.t.size = in->size;
55
56     // Perform the read
57     NvGetIndexData(nvIndex, locator, in->offset, in->size, out->data.t.buffer);
58
59     return TPM_RC_SUCCESS;
60 }
61
62 #endif // CC_NV_Read

```

7.92 /tpm/src/command/NVStorage/NV_ReadLock.c

```

1  #include "Tpm.h"
2  #include "NV_ReadLock_fp.h"
3
4  #if CC_NV_ReadLock // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Set read lock on a NV index
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES          TPMA_NV_READ_STCLEAR is not SET so
11 //                                Index referenced by 'nvIndex' may not be
12 //                                write locked
13 //     TPM_RC_NV_AUTHORIZATION    the authorization was valid but the
14 //                                authorizing entity ('authHandle')
15 //                                is not allowed to read from the Index
16 //                                referenced by 'nvIndex'
17 TPM_RC
18 TPM2_NV_ReadLock(NV_ReadLock_In* in // IN: input parameter list
19 )
20 {
21     TPM_RC result;
22     NV_REF locator;
23     // The referenced index has been checked multiple times before this is called
24     // so it must be present and will be loaded into cache
25     NV_INDEX* nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
26     TPMA_NV nvAttributes = nvIndex->publicArea.attributes;
27
28     // Input Validation
29     // Common read access checks. NvReadAccessChecks() may return

```



```

30 // TPM_RC_NV_AUTHORIZATION, TPM_RC_NV_LOCKED, or TPM_RC_NV_UNINITIALIZED
31 result = NvReadAccessChecks(in->authHandle, in->nvIndex, nvAttributes);
32 if(result == TPM_RC_NV_AUTHORIZATION)
33     return TPM_RC_NV_AUTHORIZATION;
34 // Index is already locked for write
35 else if(result == TPM_RC_NV_LOCKED)
36     return TPM_RC_SUCCESS;
37
38 // If NvReadAccessChecks return TPM_RC_NV_UNINITIALIZED, then continue.
39 // It is not an error to read lock an uninitialized Index.
40
41 // if TPMA_NV_READ_STCLEAR is not set, the index can not be read-locked
42 if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, READ_STCLEAR))
43     return TPM_RCS_ATTRIBUTES + RC_NV_ReadLock_nvIndex;
44
45 // Internal Data Update
46
47 // Set the READLOCK attribute
48 SET_ATTRIBUTE(nvAttributes, TPMA_NV, READLOCKED);
49
50 // Write NV info back
51 return NvWriteIndexAttributes(nvIndex->publicArea.nvIndex, locator, nvAttributes);
52 }
53
54 #endif // CC_NV_ReadLock

```

7.93 /tpm/src/command/NVStorage/NV_ReadPublic.c

```

1  #include "Tpm.h"
2  #include "NV_ReadPublic_fp.h"
3
4  #if CC_NV_ReadPublic // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Read the public information of a NV index
8  */
9  TPM_RC
10 TPM2_NV_ReadPublic(NV_ReadPublic_In* in, // IN: input parameter list
11                   NV_ReadPublic_Out* out // OUT: output parameter list
12 )
13 {
14     NV_INDEX* nvIndex;
15
16     // This command only supports TPM_HT_NV_INDEX-typed NV indices.
17     if(HandleGetType(in->nvIndex) != TPM_HT_NV_INDEX)
18     {
19         return TPM_RCS_HANDLE + RC_NV_ReadPublic_nvIndex;
20     }
21
22     nvIndex = NvGetIndexInfo(in->nvIndex, NULL);
23
24     // Command Output
25
26     // Copy index public data to output
27     out->nvPublic.nvPublic = nvIndex->publicArea;
28
29     // Compute NV name
30     NvGetIndexName(nvIndex, &out->nvName);
31
32     return TPM_RC_SUCCESS;
33 }
34
35 #endif // CC_NV_ReadPublic

```

7.94 /tpm/src/command/NVStorage/NV_ReadPublic2.c

```

1  #include "Tpm.h"
2  #include "NV_ReadPublic2_fp.h"
3
4  #if CC_NV_ReadPublic2 // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Read the public information of a NV index
8  */
9  TPM_RC
10 TPM2_NV_ReadPublic2(NV_ReadPublic2_In* in, // IN: input parameter list
11                    NV_ReadPublic2_Out* out // OUT: output parameter list
12 )
13 {
14     TPM_RC    result;
15     NV_INDEX* nvIndex;
16
17     nvIndex = NvGetIndexInfo(in->nvIndex, NULL);
18
19     // Command Output
20
21     // The reference code stores its NV indices in the legacy form, because
22     // it doesn't support any extended attributes.
23     // Translate the legacy form to the general form.
24     result = NvPublic2FromNvPublic(&nvIndex->publicArea, &out->nvPublic.nvPublic2);
25     if(result != TPM_RC_SUCCESS)
26     {
27         return RcSafeAddToResult(result, RC_NV_ReadPublic2_nvIndex);
28     }
29
30     // Compute NV name
31     NvGetIndexName(nvIndex, &out->nvName);
32
33     return TPM_RC_SUCCESS;
34 }
35
36 #endif // CC_NV_ReadPublic2

```

7.95 /tpm/src/command/NVStorage/NV_SetBits.c

```

1  #include "Tpm.h"
2  #include "NV_SetBits_fp.h"
3
4  #if CC_NV_SetBits // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Set bits in a NV index
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES          the TPMA_NV_BITS attribute is not SET in the
11 //                                Index referenced by 'nvIndex'
12 //     TPM_RC_NV_AUTHORIZATION   the authorization was valid but the
13 //                                authorizing entity ('authHandle')
14 //                                is not allowed to write to the Index
15 //                                referenced by 'nvIndex'
16 //     TPM_RC_NV_LOCKED          the Index referenced by 'nvIndex' is locked
17 //                                for writing
18 TPM_RC
19 TPM2_NV_SetBits(NV_SetBits_In* in // IN: input parameter list
20 )
21 {
22     TPM_RC    result;
23     NV_REF    locator;
24     NV_INDEX* nvIndex = NvGetIndexInfo(in->nvIndex, &locator);

```

```

25     UINT64    oldValue;
26     UINT64    newValue;
27
28     // Input Validation
29
30     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
31     // or TPM_RC_NV_LOCKED
32     result = NvWriteAccessChecks(
33         in->authHandle, in->nvIndex, nvIndex->publicArea.attributes);
34     if(result != TPM_RC_SUCCESS)
35         return result;
36
37     // Make sure that this is a bit field
38     if(!IsNvBitsIndex(nvIndex->publicArea.attributes))
39         return TPM_RCS_ATTRIBUTES + RC_NV_SetBits_nvIndex;
40
41     // If index is not been written, initialize it
42     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
43         oldValue = 0;
44     else
45         // Read index data
46         oldValue = NvGetUINT64Data(nvIndex, locator);
47
48     // Figure out what the new value is going to be
49     newValue = oldValue | in->bits;
50
51     // Internal Data Update
52     return NvWriteUINT64Data(nvIndex, newValue);
53 }
54
55 #endif // CC_NV_SetBits

```

7.96 /tpm/src/command/NVStorage/NV_spt.c

```

1  /** Includes
2  #include "Tpm.h"
3  #include "NV_spt_fp.h"
4
5  /** Functions
6
7  /*** NvReadAccessChecks()
8  // Common routine for validating a read
9  // Used by TPM2_NV_Read, TPM2_NV_ReadLock and TPM2_PolicyNV
10 // Return Type: TPM_RC
11 //     TPM_RC_NV_AUTHORIZATION    authHandle is not allowed to authorize read
12 //                                of the index
13 //     TPM_RC_NV_LOCKED          Read locked
14 //     TPM_RC_NV_UNINITIALIZED    Try to read an uninitialized index
15 //
16 TPM_RC
17 NvReadAccessChecks(TPM_HANDLE authHandle, // IN: the handle that provided the
18                                     // authorization
19                 TPM_HANDLE nvHandle,    // IN: the handle of the NV index to be read
20                 TPMA_NV attributes      // IN: the attributes of 'nvHandle'
21 )
22 {
23     // If data is read locked, returns an error
24     if(IS_ATTRIBUTE(attributes, TPMA_NV, READLOCKED))
25         return TPM_RC_NV_LOCKED;
26     // If the authorization was provided by the owner or platform, then check
27     // that the attributes allow the read. If the authorization handle
28     // is the same as the index, then the checks were made when the authorization
29     // was checked..
30     if(authHandle == TPM_RH_OWNER)
31     {

```

```

32     // If Owner provided authorization then OWNERWRITE must be SET
33     if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERREAD))
34         return TPM_RC_NV_AUTHORIZATION;
35 }
36 else if(authHandle == TPM_RH_PLATFORM)
37 {
38     // If Platform provided authorization then PPWRITE must be SET
39     if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPREAD))
40         return TPM_RC_NV_AUTHORIZATION;
41 }
42 // If neither Owner nor Platform provided authorization, make sure that it was
43 // provided by this index.
44 else if(authHandle != nvHandle)
45     return TPM_RC_NV_AUTHORIZATION;
46
47 // If the index has not been written, then the value cannot be read
48 // NOTE: This has to come after other access checks to make sure that
49 // the proper authorization is given to TPM2_NV_ReadLock()
50 if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN))
51     return TPM_RC_NV_UNINITIALIZED;
52
53 return TPM_RC_SUCCESS;
54 }
55
56 /*** NvWriteAccessChecks()
57 // Common routine for validating a write
58 // Used by TPM2_NV_Write, TPM2_NV_Increment, TPM2_SetBits, and TPM2_NV_WriteLock
59 // Return Type: TPM_RC
60 //     TPM_RC_NV_AUTHORIZATION    Authorization fails
61 //     TPM_RC_NV_LOCKED           Write locked
62 //
63 TPM_RC
64 NvWriteAccessChecks(
65     TPM_HANDLE authHandle, // IN: the handle that provided the
66                           // authorization
67     TPM_HANDLE nvHandle,   // IN: the handle of the NV index to be written
68     TPMA_NV attributes    // IN: the attributes of 'nvHandle'
69 )
70 {
71     // If data is write locked, returns an error
72     if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED))
73         return TPM_RC_NV_LOCKED;
74     // If the authorization was provided by the owner or platform, then check
75     // that the attributes allow the write. If the authorization handle
76     // is the same as the index, then the checks were made when the authorization
77     // was checked..
78     if(authHandle == TPM_RH_OWNER)
79     {
80         // If Owner provided authorization then OWNERWRITE must be SET
81         if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERWRITE))
82             return TPM_RC_NV_AUTHORIZATION;
83     }
84     else if(authHandle == TPM_RH_PLATFORM)
85     {
86         // If Platform provided authorization then PPWRITE must be SET
87         if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPWRITE))
88             return TPM_RC_NV_AUTHORIZATION;
89     }
90     // If neither Owner nor Platform provided authorization, make sure that it was
91     // provided by this index.
92     else if(authHandle != nvHandle)
93         return TPM_RC_NV_AUTHORIZATION;
94     return TPM_RC_SUCCESS;
95 }
96
97 /*** NvClearOrderly()

```

```

98 // This function is used to cause gp.orderlyState to be cleared to the
99 // non-orderly state.
100 TPM_RC
101 NvClearOrderly(void)
102 {
103     if(gp.orderlyState < SU_DA_USED_VALUE)
104         RETURN_IF_NV_IS_NOT_AVAILABLE;
105     g_clearOrderly = TRUE;
106     return TPM_RC_SUCCESS;
107 }
108
109 /*** NvIsPinPassIndex()
110 // Function to check to see if an NV index is a PIN Pass Index
111 // Return Type: BOOL
112 //     TRUE(1)         is pin pass
113 //     FALSE(0)        is not pin pass
114 BOOL NvIsPinPassIndex(TPM_HANDLE index // IN: Handle to check
115 )
116 {
117     if(HandleGetType(index) == TPM_HT_NV_INDEX)
118     {
119         NV_INDEX* nvIndex = NvGetIndexInfo(index, NULL);
120
121         return IsNvPinPassIndex(nvIndex->publicArea.attributes);
122     }
123     return FALSE;
124 }
125
126 /*** NvGetIndexName()
127 // This function computes the Name of an index
128 // The 'name' buffer receives the bytes of the Name and the return value
129 // is the number of octets in the Name.
130 //
131 // This function requires that the NV Index is defined.
132 TPM2B_NAME* NvGetIndexName(
133     NV_INDEX* nvIndex, // IN: the index over which the name is to be
134                       // computed
135     TPM2B_NAME* name // OUT: name of the index
136 )
137 {
138     UINT16      dataSize, digestSize;
139     BYTE        marshalBuffer[sizeof(TPMU_NV_PUBLIC_2)];
140     BYTE*       buffer;
141     INT32       bufferSize = sizeof(marshalBuffer);
142     HASH_STATE  hashState;
143     TPMT_NV_PUBLIC_2 public2;
144
145     // Convert the legacy representation into the tagged-union representation.
146     NvPublic2FromNvPublic(&nvIndex->publicArea, &public2);
147
148     // Marshal the whole public area, but not the TPM HT selector:
149     // This is safe, because the TPM_HT is the first byte of the handle value,
150     // which is already in every element of TPMT_NV_PUBLIC_2.
151     // This allows the Name of an NV index calculated based on the
152     // TPMT_NV_PUBLIC_2 to be consistent with the Name of the same index if it
153     // has a TPMS_NV_PUBLIC representation.
154     buffer = marshalBuffer;
155     dataSize =
156         TPMU_NV_PUBLIC_2_Marshal(&public2.nvPublic2,
157                                 &buffer,
158                                 &bufferSize,
159                                 (UINT32)HandleGetType(nvIndex->publicArea.nvIndex));
160
161     // hash public area
162     digestSize = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
163     CryptDigestUpdate(&hashState, dataSize, marshalBuffer);

```

```

164
165     // Complete digest leaving room for the nameAlg
166     CryptHashEnd(&hashState, digestSize, &name->b.buffer[2]);
167
168     // Include the nameAlg
169     UINT16_TO_BYTE_ARRAY(nvIndex->publicArea.nameAlg, name->b.buffer);
170     name->t.size = digestSize + 2;
171     return name;
172 }
173
174 // NOTE: This is a lossy conversion: any expanded attributes are lost here.
175 // Calling code should return an error to the user, instead of dropping their
176 // data, if any of the expanded attributes are SET.
177 static TPMA_NV LegacyAttributesFromExpanded(TPMA_NV_EXP attributes)
178 {
179     UINT64 attributes64;
180     UINT32 attributes32;
181
182     attributes64 = TPMA_NV_EXP_TO_UINT64(attributes);
183     attributes32 = (UINT32)attributes64;
184
185     return UINT32_TO_TPMA_NV(attributes32);
186 }
187
188 static TPMA_NV_EXP ExpandedAttributesFromLegacy(TPMA_NV attributes)
189 {
190     UINT32 attributes32;
191     UINT64 attributes64;
192
193     attributes32 = TPMA_NV_TO_UINT32(attributes);
194     attributes64 = (UINT64)attributes32;
195
196     return UINT64_TO_TPMA_NV_EXP(attributes64);
197 }
198
199 /*** NvPublic2FromNvPublic()
200 // This function converts a legacy-form NV public (TPMS_NV_PUBLIC) into the
201 // generalized TPMT_NV_PUBLIC_2 tagged-union representation.
202 TPM_RC NvPublic2FromNvPublic(
203     TPMS_NV_PUBLIC* nvPublic, // IN: the source S-form NV public area
204     TPMT_NV_PUBLIC_2* nvPublic2 // OUT: the T-form NV public area to populate
205 )
206 {
207     TPM_HT handleType = HandleGetType(nvPublic->nvIndex);
208
209     switch(handleType)
210     {
211         case TPM_HT_NV_INDEX:
212             nvPublic2->nvPublic2.nvIndex = *nvPublic;
213             break;
214         case TPM_HT_PERMANENT_NV:
215             nvPublic2->nvPublic2.permanentNV = *nvPublic;
216             break;
217 #if EXTERNAL_NV
218         case TPM_HT_EXTERNAL_NV:
219             {
220                 TPMS_NV_PUBLIC_EXP_ATTR* pub = &nvPublic2->nvPublic2.externalNV;
221
222                 pub->attributes = ExpandedAttributesFromLegacy(nvPublic->attributes);
223                 pub->authPolicy = nvPublic->authPolicy;
224                 pub->dataSize = nvPublic->dataSize;
225                 pub->nameAlg = nvPublic->nameAlg;
226                 pub->nvIndex = nvPublic->nvIndex;
227                 break;
228             }
229 #endif

```



```

230         default:
231             return TPM_RCS_HANDLE;
232     }
233
234     nvPublic2->handleType = handleType;
235     return TPM_RC_SUCCESS;
236 }
237
238 /*** NvPublicFromNvPublic2()
239 // This function converts a tagged-union NV public (TPMT_NV_PUBLIC_2) into the
240 // legacy TPMS_NV_PUBLIC representation. This is a lossy conversion: any
241 // bits in the extended area of the attributes are lost, and the Name cannot be
242 // computed based on it.
243 TPM_RC NvPublicFromNvPublic2(
244     TPMT_NV_PUBLIC_2* nvPublic2, // IN: the source T-form NV public area
245     TPMS_NV_PUBLIC*   nvPublic   // OUT: the S-form NV public area to populate
246 )
247 {
248     switch(nvPublic2->handleType)
249     {
250         case TPM_HT_NV_INDEX:
251             *nvPublic = nvPublic2->nvPublic2.nvIndex;
252             break;
253         case TPM_HT_PERMANENT_NV:
254             *nvPublic = nvPublic2->nvPublic2.permanentNV;
255             break;
256 #if EXTERNAL_NV
257         case TPM_HT_EXTERNAL_NV:
258             {
259                 TPMS_NV_PUBLIC_EXP_ATTR* pub = &nvPublic2->nvPublic2.externalNV;
260
261                 nvPublic->attributes = LegacyAttributesFromExpanded(pub->attributes);
262                 nvPublic->authPolicy = pub->authPolicy;
263                 nvPublic->dataSize   = pub->dataSize;
264                 nvPublic->nameAlg    = pub->nameAlg;
265                 break;
266             }
267 #endif
268         default:
269             return TPM_RCS_HANDLE;
270     }
271
272     return TPM_RC_SUCCESS;
273 }
274
275 /*** NvDefineSpace()
276 // This function combines the common functionality of TPM2_NV_DefineSpace and
277 // TPM2_NV_DefineSpace2.
278 TPM_RC NvDefineSpace(TPMI_RH_PROVISION authHandle,
279                     TPM2B_AUTH* auth,
280                     TPMS_NV_PUBLIC* publicInfo,
281                     TPM_RC blameAuthHandle,
282                     TPM_RC blameAuth,
283                     TPM_RC blamePublic)
284 {
285     TPMA_NV attributes = publicInfo->attributes;
286     UINT16 nameSize;
287
288     nameSize = CryptHashGetDigestSize(publicInfo->nameAlg);
289
290     // Input Validation
291
292     // Checks not specific to type
293
294     // If the UndefineSpaceSpecial command is not implemented, then can't have
295     // an index that can only be deleted with policy

```

```

296 #if CC_NV_UndefineSpaceSpecial == NO
297     if(IS_ATTRIBUTE(attributes, TPMA_NV, POLICY_DELETE))
298         return TPM_RCS_ATTRIBUTES + blamePublic;
299 #endif
300
301     // check that the authPolicy consistent with hash algorithm
302
303     if(publicInfo->authPolicy.t.size != 0
304        && publicInfo->authPolicy.t.size != nameSize)
305         return TPM_RCS_SIZE + blamePublic;
306
307     // make sure that the authValue is not too large
308     if(MemoryRemoveTrailingZeros(auth) > CryptHashGetDigestSize(publicInfo->nameAlg))
309         return TPM_RCS_SIZE + blameAuth;
310
311     // If an index is being created by the owner and shEnable is
312     // clear, then we would not reach this point because ownerAuth
313     // can't be given when shEnable is CLEAR. However, if phEnable
314     // is SET but phEnableNV is CLEAR, we have to check here
315     if(authHandle == TPM_RH_PLATFORM && gc.phEnableNV == CLEAR)
316         return TPM_RCS_HIERARCHY + blameAuthHandle;
317
318     // Attribute checks
319     // Eliminate the unsupported types
320     switch(GET_TPM_NT(attributes))
321     {
322     #if CC_NV_Increment == YES
323         case TPM_NT_COUNTER:
324     #endif
325     #if CC_NV_SetBits == YES
326         case TPM_NT_BITS:
327     #endif
328     #if CC_NV_Extend == YES
329         case TPM_NT_EXTEND:
330     #endif
331     #if CC_PolicySecret == YES && defined TPM_NT_PIN_PASS
332         case TPM_NT_PIN_PASS:
333         case TPM_NT_PIN_FAIL:
334     #endif
335         case TPM_NT_ORDINARY:
336             break;
337         default:
338             return TPM_RCS_ATTRIBUTES + blamePublic;
339             break;
340     }
341     // Check that the sizes are OK based on the type
342     switch(GET_TPM_NT(attributes))
343     {
344         case TPM_NT_ORDINARY:
345             // Can't exceed the allowed size for the implementation
346             if(publicInfo->dataSize > MAX_NV_INDEX_SIZE)
347                 return TPM_RCS_SIZE + blamePublic;
348             break;
349         case TPM_NT_EXTEND:
350             if(publicInfo->dataSize != nameSize)
351                 return TPM_RCS_SIZE + blamePublic;
352             break;
353         default:
354             // Everything else needs a size of 8
355             if(publicInfo->dataSize != 8)
356                 return TPM_RCS_SIZE + blamePublic;
357             break;
358     }
359     // Handle other specifics
360     switch(GET_TPM_NT(attributes))
361     {

```

```

362     case TPM_NT_COUNTER:
363         // Counter can't have TPMA_NV_CLEAR_STCLEAR SET (don't clear counters)
364         if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR))
365             return TPM_RCS_ATTRIBUTES + blamePublic;
366         break;
367 #ifdef TPM_NT_PIN_FAIL
368     case TPM_NT_PIN_FAIL:
369         // NV_NO_DA must be SET and AUTHWRITE must be CLEAR
370         // NOTE: As with a PIN_PASS index, the authValue of the index is not
371         // available until the index is written. If AUTHWRITE is the only way to
372         // write then index, it could never be written. Rather than go through
373         // all of the other possible ways to write the Index, it is simply
374         // prohibited to write the index with the authValue. Other checks
375         // below will insure that there seems to be a way to write the index
376         // (i.e., with platform authorization, owner authorization,
377         // or with policyAuth.)
378         // It is not allowed to create a PIN Index that can't be modified.
379         if(!IS_ATTRIBUTE(attributes, TPMA_NV, NO_DA))
380             return TPM_RCS_ATTRIBUTES + blamePublic;
381 #endif
382 #ifdef TPM_NT_PIN_PASS
383     case TPM_NT_PIN_PASS:
384         // AUTHWRITE must be CLEAR (see note above to TPM_NT_PIN_FAIL)
385         if(IS_ATTRIBUTE(attributes, TPMA_NV, AUTHWRITE)
386            || IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK)
387            || IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
388             return TPM_RCS_ATTRIBUTES + blamePublic;
389 #endif // this comes before break because PIN_FAIL falls through
390         break;
391     default:
392         break;
393 }
394
395 // Locks may not be SET and written cannot be SET
396 if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
397    || IS_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED)
398    || IS_ATTRIBUTE(attributes, TPMA_NV, READLOCKED))
399     return TPM_RCS_ATTRIBUTES + blamePublic;
400
401 // There must be a way to read the index.
402 if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERREAD)
403    && !IS_ATTRIBUTE(attributes, TPMA_NV, PPREAD)
404    && !IS_ATTRIBUTE(attributes, TPMA_NV, AUTHREAD)
405    && !IS_ATTRIBUTE(attributes, TPMA_NV, POLICYREAD))
406     return TPM_RCS_ATTRIBUTES + blamePublic;
407
408 // There must be a way to write the index
409 if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERWRITE)
410    && !IS_ATTRIBUTE(attributes, TPMA_NV, PPWRITE)
411    && !IS_ATTRIBUTE(attributes, TPMA_NV, AUTHWRITE)
412    && !IS_ATTRIBUTE(attributes, TPMA_NV, POLICYWRITE))
413     return TPM_RCS_ATTRIBUTES + blamePublic;
414
415 // An index with TPMA_NV_CLEAR_STCLEAR can't have TPMA_NV_WRITEDEFINE SET
416 if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
417    && IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
418     return TPM_RCS_ATTRIBUTES + blamePublic;
419
420 // Make sure that the creator of the index can delete the index
421 if((IS_ATTRIBUTE(attributes, TPMA_NV, PLATFORMCREATE)
422    && authHandle == TPM_RH_OWNER)
423    || (!IS_ATTRIBUTE(attributes, TPMA_NV, PLATFORMCREATE)
424        && authHandle == TPM_RH_PLATFORM))
425     return TPM_RCS_ATTRIBUTES + blameAuthHandle;
426
427 // If TPMA_NV_POLICY_DELETE is SET, then the index must be defined by

```

```

428 // the platform
429 if(IS_ATTRIBUTE(attributes, TPMA_NV, POLICY_DELETE)
430    && TPM_RH_PLATFORM != authHandle)
431     return TPM_RCS_ATTRIBUTES + blamePublic;
432
433 // Make sure that the TPMA_NV_WRITEALL is not set if the index size is larger
434 // than the allowed NV buffer size.
435 if(publicInfo->dataSize > MAX_NV_BUFFER_SIZE
436    && IS_ATTRIBUTE(attributes, TPMA_NV, WRITEALL))
437     return TPM_RCS_SIZE + blamePublic;
438
439 // And finally, see if the index is already defined.
440 if(NvIndexIsDefined(publicInfo->nvIndex))
441     return TPM_RC_NV_DEFINED;
442
443 // Internal Data Update
444 // define the space. A TPM_RC_NV_SPACE error may be returned at this point
445 return NvDefineIndex(publicInfo, auth);
446 }

```

7.97 /tpm/src/command/NVStorage/NV_UndefineSpace.c

```

1  #include "Tpm.h"
2  #include "NV_UndefineSpace_fp.h"
3
4  #if CC_NV_UndefineSpace // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Delete an NV Index
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_ATTRIBUTES TPM2_NV_UndefineSpace is SET in the Index
11 // referenced by 'nvIndex' so this command may
12 // not be used to delete this Index (see
13 // TPM2_NV_UndefineSpaceSpecial())
14 // TPM_RC_NV_AUTHORIZATION attempt to use ownerAuth to delete an index
15 // created by the platform
16 //
17 TPM_RC
18 TPM2_NV_UndefineSpace(NV_UndefineSpace_In* in // IN: input parameter list
19 )
20 {
21     NV_REF locator;
22     NV_INDEX* nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
23
24     // Input Validation
25     // This command can't be used to delete an index with TPMA_NV_POLICY_DELETE SET
26     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, POLICY_DELETE))
27         return TPM_RCS_ATTRIBUTES + RC_NV_UndefineSpace_nvIndex;
28
29     // The owner may only delete an index that was defined with ownerAuth. The
30     // platform may delete an index that was created with either authorization.
31     if(in->authHandle == TPM_RH_OWNER
32        && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
33         return TPM_RC_NV_AUTHORIZATION;
34
35     // Internal Data Update
36
37     // Call implementation dependent internal routine to delete NV index
38     return NvDeleteIndex(nvIndex, locator);
39 }
40
41 #endif // CC_NV_UndefineSpace

```

7.98 /tpm/src/command/NVStorage/NV_UndefineSpaceSpecial.c

```

1  #include "Tpm.h"
2  #include "NV_UndefineSpaceSpecial_fp.h"
3  #include "SessionProcess_fp.h"
4
5  #if CC_NV_UndefineSpaceSpecial // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // Delete a NV index that requires policy to delete.
9  */
10 // Return Type: TPM_RC
11 // TPM_RC_ATTRIBUTES          TPMA_NV_POLICY_DELETE is not SET in the
12 //                             Index referenced by 'nvIndex'
13 TPM_RC
14 TPM2 NV_UndefineSpaceSpecial(
15     NV_UndefineSpaceSpecial_In* in // IN: input parameter list
16 )
17 {
18     TPM_RC    result;
19     NV_REF    locator;
20     NV_INDEX* nvIndex = NvGetIndexInfo(in->nvIndex, &locator);
21     // Input Validation
22     // This operation only applies when the TPMA_NV_POLICY_DELETE attribute is SET
23     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, POLICY_DELETE))
24         return TPM_RCS_ATTRIBUTES + RC_NV_UndefineSpaceSpecial_nvIndex;
25     // Internal Data Update
26     // Call implementation dependent internal routine to delete NV index
27     result = NvDeleteIndex(nvIndex, locator);
28
29     // If we just removed the index providing the authorization, make sure that the
30     // authorization session computation is modified so that it doesn't try to
31     // access the authValue of the just deleted index
32     if(result == TPM_RC_SUCCESS)
33         SessionRemoveAssociationToHandle(in->nvIndex);
34     return result;
35 }
36
37 #endif // CC_NV_UndefineSpaceSpecial

```

7.99 /tpm/src/command/NVStorage/NV_Write.c

```

1  #include "Tpm.h"
2  #include "NV_Write_fp.h"
3
4  #if CC_NV_Write // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Write to a NV index
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_ATTRIBUTES          Index referenced by 'nvIndex' has either
11 //                             TPMA_NV_BITS, TPMA_NV_COUNTER, or
12 //                             TPMA_NV_EVENT attribute SET
13 //                             the authorization was valid but the
14 //                             authorizing entity ('authHandle')
15 //                             is not allowed to write to the Index
16 //                             referenced by 'nvIndex'
17 //                             Index referenced by 'nvIndex' is write
18 //                             locked
19 //                             TPM_RC_NV_RANGE          if TPMA_NV_WRITEALL is SET then the write
20 //                             is not the size of the Index referenced by
21 //                             'nvIndex'; otherwise, the write extends
22 //                             beyond the limits of the Index
23 //

```

```

24 TPM_RC
25 TPM2_NV_Write(NV_Write_In* in // IN: input parameter list
26 )
27 {
28     NV_INDEX* nvIndex    = NvGetIndexInfo(in->nvIndex, NULL);
29     TPMA_NV    attributes = nvIndex->publicArea.attributes;
30     TPM_RC    result;
31
32     // Input Validation
33
34     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
35     // or TPM_RC_NV_LOCKED
36     result = NvWriteAccessChecks(in->authHandle, in->nvIndex, attributes);
37     if(result != TPM_RC_SUCCESS)
38         return result;
39
40     // Bits index, extend index or counter index may not be updated by
41     // TPM2_NV_Write
42     if(IsNvCounterIndex(attributes) || IsNvBitsIndex(attributes)
43        || IsNvExtendIndex(attributes))
44         return TPM_RC_ATTRIBUTES;
45
46     // Make sure that the offset is not too large
47     if(in->offset > nvIndex->publicArea.dataSize)
48         return TPM_RCS_VALUE + RC_NV_Write_offset;
49
50     // Make sure that the selection is within the range of the Index
51     if(in->data.t.size > (nvIndex->publicArea.dataSize - in->offset))
52         return TPM_RC_NV_RANGE;
53
54     // If this index requires a full sized write, make sure that input range is
55     // full sized.
56     // Note: if the requested size is the same as the Index data size, then offset
57     // will have to be zero. Otherwise, the range check above would have failed.
58     if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITEALL)
59        && in->data.t.size < nvIndex->publicArea.dataSize)
60         return TPM_RC_NV_RANGE;
61
62     // Internal Data Update
63
64     // Perform the write. This called routine will SET the TPMA_NV_WRITTEN
65     // attribute if it has not already been SET. If NV isn't available, an error
66     // will be returned.
67     return NvWriteIndexData(nvIndex, in->offset, in->data.t.size, in->data.t.buffer);
68 }
69
70 #endif // CC_NV_Write

```

7.100 /tpm/src/command/NVStorage/NV_WriteLock.c

```

1  #include "Tpm.h"
2  #include "NV_WriteLock_fp.h"
3
4  #if CC_NV_WriteLock // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Set write lock on a NV index
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES          neither TPMA_NV_WRITEDEFINE nor
11 //                                TPMA_NV_WRITE_STCLEAR is SET in Index
12 //                                referenced by 'nvIndex'
13 //     TPM_RC_NV_AUTHORIZATION    the authorization was valid but the
14 //                                authorizing entity ('authHandle')
15 //                                is not allowed to write to the Index

```



```

16 // referenced by 'nvIndex'
17 //
18 TPM_RC
19 TPM2_NV_WriteLock(NV_WriteLock_In* in // IN: input parameter list
20 )
21 {
22     TPM_RC    result;
23     NV_REF    locator;
24     NV_INDEX* nvIndex    = NvGetIndexInfo(in->nvIndex, &locator);
25     TPMA_NV    nvAttributes = nvIndex->publicArea.attributes;
26
27     // Input Validation:
28
29     // Common access checks, NvWriteAccessCheck() may return TPM_RC_NV_AUTHORIZATION
30     // or TPM_RC_NV_LOCKED
31     result = NvWriteAccessChecks(in->authHandle, in->nvIndex, nvAttributes);
32     if(result != TPM_RC_SUCCESS)
33     {
34         if(result == TPM_RC_NV_AUTHORIZATION)
35             return result;
36         // If write access failed because the index is already locked, then it is
37         // no error.
38         return TPM_RC_SUCCESS;
39     }
40     // if neither TPMA_NV_WRITEDEFINE nor TPMA_NV_WRITE_STCLEAR is set, the index
41     // can not be write-locked
42     if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITEDEFINE)
43        && !IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITE_STCLEAR))
44         return TPM_RCS_ATTRIBUTES + RC_NV_WriteLock_nvIndex;
45     // Internal Data Update
46     // Set the WRITELOCK attribute.
47     // Note: if TPMA_NV_WRITELOCKED were already SET, then the write access check
48     // above would have failed and this code isn't executed.
49     SET_ATTRIBUTE(nvAttributes, TPMA_NV, WRITELOCKED);
50
51     // Write index info back
52     return NvWriteIndexAttributes(nvIndex->publicArea.nvIndex, locator, nvAttributes);
53 }
54
55 #endif // CC_NV_WriteLock

```

7.101 /tpm/src/command/Object/ActivateCredential.c

```

1  #include "Tpm.h"
2  #include "ActivateCredential_fp.h"
3
4  #if CC_ActivateCredential // Conditional expansion of this file
5
6  # include "Object_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Activate Credential with an object
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_ATTRIBUTES    'keyHandle' does not reference a decryption key
13 //     TPM_RC_ECC_POINT     'secret' is invalid (when 'keyHandle' is an ECC key)
14 //     TPM_RC_INSUFFICIENT  'secret' is invalid (when 'keyHandle' is an ECC key)
15 //     TPM_RC_INTEGRITY     'credentialBlob' fails integrity test
16 //     TPM_RC_NO_RESULT     'secret' is invalid (when 'keyHandle' is an ECC key)
17 //     TPM_RC_SIZE         'secret' size is invalid or the 'credentialBlob'
18 //                         does not unmarshal correctly
19 //     TPM_RC_TYPE         'keyHandle' does not reference an asymmetric key.
20 //     TPM_RC_VALUE        'secret' is invalid (when 'keyHandle' is an RSA key)
21 TPM_RC
22 TPM2_ActivateCredential(ActivateCredential_In* in, // IN: input parameter list

```

```

23         ActivateCredential_Out* out // OUT: output parameter list
24     )
25     {
26         TPM_RC      result = TPM_RC_SUCCESS;
27         OBJECT*     object; // decrypt key
28         OBJECT*     activateObject; // key associated with credential
29         TPM2B_DATA data; // credential data
30
31         // Input Validation
32
33         // Get decrypt key pointer
34         object = HandleToObject(in->keyHandle);
35
36         // Get certificated object pointer
37         activateObject = HandleToObject(in->activateHandle);
38
39         // input decrypt key must be an asymmetric, restricted decryption key
40         if(!CryptIsAsymAlgorithm(object->publicArea.type)
41            || !IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, decrypt)
42            || !IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, restricted))
43             return TPM_RCS_TYPE + RC_ActivateCredential_keyHandle;
44
45         // Command output
46
47         // Decrypt input credential data via asymmetric decryption. A
48         // TPM_RC_VALUE, TPM_RC_KEY or unmarshal errors may be returned at this
49         // point
50         result = CryptSecretDecrypt(object, NULL, IDENTITY_STRING, &in->secret, &data);
51         if(result != TPM_RC_SUCCESS)
52         {
53             if(result == TPM_RC_KEY)
54                 return TPM_RC_FAILURE;
55             return RcSafeAddToResult(result, RC_ActivateCredential_secret);
56         }
57
58         // Retrieve secret data. A TPM_RC_INTEGRITY error or unmarshal
59         // errors may be returned at this point
60         result = CredentialToSecret(&in->credentialBlob.b,
61                                    &activateObject->name.b,
62                                    &data.b,
63                                    object,
64                                    &out->certInfo);
65         if(result != TPM_RC_SUCCESS)
66             return RcSafeAddToResult(result, RC_ActivateCredential_credentialBlob);
67
68         return TPM_RC_SUCCESS;
69     }
70
71 #endif // CC_ActivateCredential

```

7.102 /tpm/src/command/Object/Create.c

```

1  #include "Tpm.h"
2  #include "Object_spt_fp.h"
3  #include "Create_fp.h"
4
5  #if CC_Create // Conditional expansion of this file
6
7  /*(See part 3 specification)
8  // Create a regular object
9  */
10 // Return Type: TPM_RC
11 // TPM_RC_ATTRIBUTES 'sensitiveDataOrigin' is CLEAR when 'sensitive.data'
12 // is an Empty Buffer, or is SET when 'sensitive.data' is
13 // not empty;

```

```

14 // 'fixedTPM', 'fixedParent', or 'encryptedDuplication'
15 // attributes are inconsistent between themselves or with
16 // those of the parent object;
17 // inconsistent 'restricted', 'decrypt' and 'sign'
18 // attributes;
19 // attempt to inject sensitive data for an asymmetric
20 // key;
21 // TPM_RC_HASH non-duplicable storage key and its parent have
22 // different name algorithm
23 // TPM_RC_KDF incorrect KDF specified for decrypting keyed hash
24 // object
25 // TPM_RC_KEY invalid key size values in an asymmetric key public
26 // area or a provided symmetric key has a value that is
27 // not allowed
28 // TPM_RC_KEY_SIZE key size in public area for symmetric key differs from
29 // the size in the sensitive creation area; may also be
30 // returned if the TPM does not allow the key size to be
31 // used for a Storage Key
32 // TPM_RC_OBJECT_MEMORY a free slot is not available as scratch memory for
33 // object creation
34 // TPM_RC_RANGE the exponent value of an RSA key is not supported.
35 // TPM_RC_SCHEME inconsistent attributes 'decrypt', 'sign', or
36 // 'restricted' and key's scheme ID; or hash algorithm is
37 // inconsistent with the scheme ID for keyed hash object
38 // TPM_RC_SIZE size of public authPolicy or sensitive authValue does
39 // not match digest size of the name algorithm
40 // sensitive data size for the keyed hash object is
41 // larger than is allowed for the scheme
42 // TPM_RC_SYMMETRIC a storage key with no symmetric algorithm specified;
43 // or non-storage key with symmetric algorithm different
44 // from TPM_ALG_NULL
45 // TPM_RC_TYPE unknown object type;
46 // 'parentHandle' does not reference a restricted
47 // decryption key in the storage hierarchy with both
48 // public and sensitive portion loaded
49 // TPM_RC_VALUE exponent is not prime or could not find a prime using
50 // the provided parameters for an RSA key;
51 // unsupported name algorithm for an ECC key
52 // TPM_RC_OBJECT_MEMORY there is no free slot for the object
53 TPM_RC
54 TPM2_Create(Create_In* in, // IN: input parameter list
55             Create_Out* out // OUT: output parameter list
56 )
57 {
58     TPM_RC result = TPM_RC_SUCCESS;
59     OBJECT* parentObject;
60     OBJECT* newObject;
61     TPMT_PUBLIC* publicArea;
62
63     // Input Validation
64     parentObject = HandleToObject(in->parentHandle);
65     pAssert(parentObject != NULL);
66
67     // Does parent have the proper attributes?
68     if(!ObjectIsParent(parentObject))
69         return TPM_RCS_TYPE + RC_Create_parentHandle;
70
71     // Get a slot for the creation
72     newObject = FindEmptyObjectSlot(NULL);
73     if(newObject == NULL)
74         return TPM_RC_OBJECT_MEMORY;
75     // If the TPM2B_PUBLIC was passed as a structure, marshal it into is canonical
76     // form for processing
77
78     // to save typing.
79     publicArea = &newObject->publicArea;

```

```

80
81 // Copy the input structure to the allocated structure
82 *publicArea = in->inPublic.publicArea;
83
84 // Check attributes in input public area. CreateChecks() checks the things that
85 // are unique to creation and then validates the attributes and values that are
86 // common to create and load.
87 result = CreateChecks(parentObject,
88                       /* primaryHierarchy = */ 0,
89                       publicArea,
90                       in->inSensitive.sensitive.data.t.size);
91 if(result != TPM_RC_SUCCESS)
92     return RcSafeAddToResult(result, RC_Create_inPublic);
93 // Clean up the authValue if necessary
94 if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth, publicArea->nameAlg))
95     return TPM_RCS_SIZE + RC_Create_inSensitive;
96
97 // Command Output
98 // Create the object using the default TPM random-number generator
99 result = CryptCreateObject(newObject, &in->inSensitive.sensitive, NULL);
100 if(result != TPM_RC_SUCCESS)
101     return result;
102 // Fill in creation data
103 FillInCreationData(in->parentHandle,
104                   publicArea->nameAlg,
105                   &in->creationPCR,
106                   &in->outsideInfo,
107                   &out->creationData,
108                   &out->creationHash);
109
110 // Compute creation ticket
111 result = TicketComputeCreation(EntityGetHierarchy(in->parentHandle),
112                               &newObject->name,
113                               &out->creationHash,
114                               &out->creationTicket);
115 if(result != TPM_RC_SUCCESS)
116     return result;
117
118 // Prepare output private data from sensitive
119 SensitiveToPrivate(&newObject->sensitive,
120                  &newObject->name,
121                  parentObject,
122                  publicArea->nameAlg,
123                  &out->outPrivate);
124
125 newObject->hierarchy = parentObject->hierarchy;
126
127 // Finish by copying the remaining return values
128 out->outPublic.publicArea = newObject->publicArea;
129
130 return TPM_RC_SUCCESS;
131 }
132
133 #endif // CC_Create

```

7.103 /tpm/src/command/Object/CreateLoaded.c

```

1 #include "Tpm.h"
2 #include "CreateLoaded_fp.h"
3
4 #if CC_CreateLoaded // Conditional expansion of this file
5
6 /*(See part 3 of specification)
7  * Create and load any type of key, including a temporary key.
8  * The input template is a marshaled public area rather than an unmarshaled one as

```

```

9      * used in Create and CreatePrimary. This is so that the label and context that
10     * could be in the template can be processed without changing the formats for the
11     * calls to Create and CreatePrimary.
12     */
13     // Return Type: TPM_RC
14     //     TPM_RC_ATTRIBUTES      'sensitiveDataOrigin' is CLEAR when 'sensitive.data'
15     //                             is an Empty Buffer;
16     //                             'fixedTPM', 'fixedParent', or 'encryptedDuplication'
17     //                             attributes are inconsistent between themselves or with
18     //                             those of the parent object;
19     //                             inconsistent 'restricted', 'decrypt' and 'sign'
20     //                             attributes;
21     //                             attempt to inject sensitive data for an asymmetric
22     //                             key;
23     //                             attempt to create a symmetric cipher key that is not
24     //                             a decryption key
25     //     TPM_RC_FW_LIMITED      The requested hierarchy is FW-limited, but the TPM
26     //                             does not support FW-limited objects or the TPM failed
27     //                             to derive the Firmware Secret.
28     //     TPM_RC_SVN_LIMITED     The requested hierarchy is SVN-limited, but the TPM
29     //                             does not support SVN-limited objects or the TPM failed
30     //                             to derive the Firmware SVN Secret for the requested
31     //                             SVN.
32     //     TPM_RC_KDF             incorrect KDF specified for decrypting keyed hash
33     //                             object
34     //     TPM_RC_KEY             the value of a provided symmetric key is not allowed
35     //     TPM_RC_OBJECT_MEMORY   there is no free slot for the object
36     //     TPM_RC_SCHEME          inconsistent attributes 'decrypt', 'sign',
37     //                             'restricted' and key's scheme ID; or hash algorithm is
38     //                             inconsistent with the scheme ID for keyed hash object
39     //     TPM_RC_SIZE            size of public authorization policy or sensitive
40     //                             authorization value does not match digest size of the
41     //                             name algorithm sensitive data size for the keyed hash
42     //                             object is larger than is allowed for the scheme
43     //     TPM_RC_SYMMETRIC       a storage key with no symmetric algorithm specified;
44     //                             or non-storage key with symmetric algorithm different
45     //                             from TPM_ALG_NULL
46     //     TPM_RC_TYPE            cannot create the object of the indicated type
47     //                             (usually only occurs if trying to derive an RSA key).
48     TPM_RC
49     TPM2_CreateLoaded(CreateLoaded_In* in, // IN: input parameter list
50                      CreateLoaded_Out* out // OUT: output parameter list
51     )
52     {
53         TPM_RC      result = TPM_RC_SUCCESS;
54         OBJECT*      parent = HandleToObject(in->parentHandle);
55         OBJECT*      newObject;
56         BOOL         derivation;
57         TPMT_PUBLIC* publicArea;
58         RAND_STATE    randState;
59         RAND_STATE*   rand = &randState;
60         TPMS_DERIVE   labelContext;
61
62         // Input Validation
63
64         // How the public area is unmarshaled is determined by the parent, so
65         // see if parent is a derivation parent
66         derivation = (parent != NULL && parent->attributes.derivation);
67
68         // If the parent is an object, then make sure that it is either a parent or
69         // derivation parent
70         if(parent != NULL && !parent->attributes.isParent && !derivation)
71             return TPM_RCS_TYPE + RC_CreateLoaded_parentHandle;
72
73         // Get a spot in which to create the newObject
74         newObject = FindEmptyObjectSlot(&out->objectHandle);

```



```

75     if(newObject == NULL)
76         return TPM_RC_OBJECT_MEMORY;
77
78     // Do this to save typing
79     publicArea = &newObject->publicArea;
80
81     // Unmarshal the template into the object space. TPM2_Create() and
82     // TPM2_CreatePrimary() have the publicArea unmarshaled by CommandDispatcher.
83     // This command is different because of an unfortunate property of the
84     // unique field of an ECC key. It is a structure rather than a single TPM2B. If
85     // it had been a TPM2B, then the label and context could be within a TPM2B and
86     // unmarshaled like other public areas. Since it is not, this command needs its
87     // on template that is a TPM2B that is unmarshaled as a BYTE array with a
88     // its own unmarshal function.
89     result = UnmarshalToPublic(publicArea, &in->inPublic, derivation, &labelContext);
90     if(result != TPM_RC_SUCCESS)
91         return result + RC_CreateLoaded_inPublic;
92
93     // Validate that the authorization size is appropriate
94     if(!AdjustAuthSize(&in->inSensitive.sensitive.userAuth, publicArea->nameAlg))
95         return TPM_RC_SIZE + RC_CreateLoaded_inSensitive;
96
97     // Command output
98     if(derivation)
99     {
100         TPMT_KEYEDHASH_SCHEME* scheme;
101         scheme = &parent->publicArea.parameters.keyedHashDetail.scheme;
102
103         // SP800-108 is the only KDF supported by this implementation and there is
104         // no default hash algorithm.
105         pAssert(scheme->details.xor.hashAlg != TPM_ALG_NULL
106                && scheme->details.xor.kdf == TPM_ALG_KDF1_SP800_108);
107         // Don't derive RSA keys
108         if(publicArea->type == TPM_ALG_RSA)
109             return TPM_RC_TYPE + RC_CreateLoaded_inPublic;
110         // sensitiveDataOrigin has to be CLEAR in a derived object. Since this
111         // is specific to a derived object, it is checked here.
112         if(IS_ATTRIBUTE(
113             publicArea->objectAttributes, TPMA_OBJECT, sensitiveDataOrigin))
114             return TPM_RC_ATTRIBUTES;
115         // Check the rest of the attributes
116         result = PublicAttributesValidation(parent, 0, publicArea);
117         if(result != TPM_RC_SUCCESS)
118             return RcSafeAddToResult(result, RC_CreateLoaded_inPublic);
119         // Process the template and sensitive areas to get the actual 'label' and
120         // 'context' values to be used for this derivation.
121         result = SetLabelAndContext(&labelContext, &in->inSensitive.sensitive.data);
122         if(result != TPM_RC_SUCCESS)
123             return result;
124         // Set up the KDF for object generation
125         DRBG_InstantiateSeededKdf((KDF_STATE*)rand,
126                                  scheme->details.xor.hashAlg,
127                                  scheme->details.xor.kdf,
128                                  &parent->sensitive.sensitive.bits.b,
129                                  &labelContext.label.b,
130                                  &labelContext.context.b,
131                                  TPM_MAX_DERIVATION_BITS);
132         // Clear the sensitive size so that the creation functions will not try
133         // to use this value.
134         in->inSensitive.sensitive.data.t.size = 0;
135     }
136     else
137     {
138         // Check attributes in input public area. CreateChecks() checks the things
139         // that are unique to creation and then validates the attributes and values
140         // that are common to create and load.

```



```

141     result = CreateChecks(parent,
142                           (parent == NULL) ? in->parentHandle : 0,
143                           publicArea,
144                           in->inSensitive.sensitive.data.t.size);
145
146     if(result != TPM_RC_SUCCESS)
147         return RcSafeAddToResult(result, RC_CreateLoaded_inPublic);
148     // Creating a primary object
149     if(parent == NULL)
150     {
151         TPM2B_NAME name;
152         TPM2B_SEED primary_seed;
153
154         newObject->attributes.primary = SET;
155         if(HierarchyNormalizeHandle(in->parentHandle) == TPM_RH_ENDORSEMENT)
156             newObject->attributes.epsHierarchy = SET;
157
158         result = HierarchyGetPrimarySeed(in->parentHandle, &primary_seed);
159         if(result != TPM_RC_SUCCESS)
160             return result;
161
162         // If so, use the primary seed and the digest of the template
163         // to seed the DRBG
164         result = DRBG_InstantiateSeeded(
165             (DRBG_STATE*)rand,
166             &primary_seed.b,
167             PRIMARY_OBJECT_CREATION,
168             (TPM2B*)PublicMarshalAndComputeName(publicArea, &name),
169             &in->inSensitive.sensitive.data.b);
170         MemorySet(primary_seed.b.buffer, 0, primary_seed.b.size);
171
172         if(result != TPM_RC_SUCCESS)
173             return result;
174     }
175     else
176     {
177         // This is an ordinary object so use the normal random number generator
178         rand = NULL;
179     }
180 }
181 // Internal data update
182 // Create the object
183 result = CryptCreateObject(newObject, &in->inSensitive.sensitive, rand);
184 DRBG_Uninstantiate((DRBG_STATE*)rand);
185 if(result != TPM_RC_SUCCESS)
186     return result;
187 // if this is not a Primary key and not a derived key, then return the sensitive
188 // area
189 if(parent != NULL && !derivation)
190     // Prepare output private data from sensitive
191     SensitiveToPrivate(&newObject->sensitive,
192                       &newObject->name,
193                       parent,
194                       newObject->publicArea.nameAlg,
195                       &out->outPrivate);
196 else
197     out->outPrivate.t.size = 0;
198 // Set the remaining return values
199 out->outPublic.publicArea = newObject->publicArea;
200 out->name = newObject->name;
201 // Set the remaining attributes for a loaded object
202 ObjectSetLoadedAttributes(newObject, in->parentHandle);
203
204 return result;
205 }
206

```

```
207 #endif // CC_CreateLoaded
```

7.104 /tpm/src/command/Object/Load.c

```
1  #include "Tpm.h"
2  #include "Load_fp.h"
3
4  #if CC_Load // Conditional expansion of this file
5
6  # include "Object_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Load an ordinary or temporary object
10 */
11 // Return Type: TPM_RC
12 //   TPM_RC_ATTRIBUTES      'inPublic' attributes are not allowed with selected
13 //                           parent
14 //   TPM_RC_BINDING         'inPrivate' and 'inPublic' are not
15 //                           cryptographically bound
16 //   TPM_RC_HASH            incorrect hash selection for signing key or
17 //                           the 'nameAlg' for 'inPublic' is not valid
18 //   TPM_RC_INTEGRITY       HMAC on 'inPrivate' was not valid
19 //   TPM_RC_KDF             KDF selection not allowed
20 //   TPM_RC_KEY             the size of the object's 'unique' field is not
21 //                           consistent with the indicated size in the object's
22 //                           parameters
23 //   TPM_RC_OBJECT_MEMORY   no available object slot
24 //   TPM_RC_SCHEME          the signing scheme is not valid for the key
25 //   TPM_RC_SENSITIVE       the 'inPrivate' did not unmarshal correctly
26 //   TPM_RC_SIZE            'inPrivate' missing, or 'authPolicy' size for
27 //                           'inPublic' or is not valid
28 //   TPM_RC_SYMMETRIC       symmetric algorithm not provided when required
29 //   TPM_RC_TYPE            'parentHandle' is not a storage key, or the object
30 //                           to load is a storage key but its parameters do not
31 //                           match the parameters of the parent.
32 //   TPM_RC_VALUE           decryption failure
33 TPM_RC
34 TPM2_Load(Load_In* in, // IN: input parameter list
35          Load_Out* out // OUT: output parameter list
36 )
37 {
38     TPM_RC      result = TPM_RC_SUCCESS;
39     TPMT_SENSITIVE sensitive;
40     OBJECT*     parentObject;
41     OBJECT*     newObject;
42
43     // Input Validation
44     // Don't get invested in loading if there is no place to put it.
45     newObject = FindEmptyObjectSlot(&out->objectHandle);
46     if(newObject == NULL)
47         return TPM_RC_OBJECT_MEMORY;
48
49     if(in->inPrivate.t.size == 0)
50         return TPM_RCS_SIZE + RC_Load_inPrivate;
51
52     parentObject = HandleToObject(in->parentHandle);
53     pAssert(parentObject != NULL);
54     // Is the object that is being used as the parent actually a parent.
55     if(!ObjectIsParent(parentObject))
56         return TPM_RCS_TYPE + RC_Load_parentHandle;
57
58     // Compute the name of object. If there isn't one, it is because the nameAlg is
59     // not valid.
60     PublicMarshalAndComputeName(&in->inPublic.publicArea, &out->name);
61     if(out->name.t.size == 0)
```

```

62     return TPM_RCS_HASH + RC_Load_inPublic;
63
64     // Retrieve sensitive data.
65     result = PrivateToSensitive(&in->inPrivate.b,
66                               &out->name.b,
67                               parentObject,
68                               in->inPublic.publicArea.nameAlg,
69                               &sensitive);
70     if(result != TPM_RC_SUCCESS)
71         return RcSafeAddToResult(result, RC_Load_inPrivate);
72
73     // Internal Data Update
74     // Load and validate object
75     result = ObjectLoad(newObject,
76                        parentObject,
77                        &in->inPublic.publicArea,
78                        &sensitive,
79                        RC_Load_inPublic,
80                        RC_Load_inPrivate,
81                        &out->name);
82     if(result == TPM_RC_SUCCESS)
83     {
84         // Set the common OBJECT attributes for a loaded object.
85         ObjectSetLoadedAttributes(newObject, in->parentHandle);
86     }
87     return result;
88 }
89
90 #endif // CC_Load

```

7.105 /tpm/src/command/Object/LoadExternal.c

```

1  #include "Tpm.h"
2  #include "LoadExternal_fp.h"
3
4  #if CC_LoadExternal // Conditional expansion of this file
5
6  # include "Object_spt_fp.h"
7
8  /*(See part 3 specification)
9  // to load an object that is not a Protected Object into the public portion
10 // of an object into the TPM. The command allows loading of a public area or
11 // both a public and sensitive area
12 */
13 // Return Type: TPM_RC
14 //     TPM_RC_ATTRIBUTES      'fixedParent', 'fixedTPM', and 'restricted' must
15 //                             be CLEAR if sensitive portion of an object is loaded
16 //     TPM_RC_BINDING         the 'inPublic' and 'inPrivate' structures are not
17 //                             cryptographically bound
18 //     TPM_RC_HASH           incorrect hash selection for signing key
19 //     TPM_RC_HIERARCHY      'hierarchy' is turned off, or only NULL hierarchy
20 //                             is allowed when loading public and private parts
21 //                             of an object
22 //     TPM_RC_KDF             incorrect KDF selection for decrypting
23 //                             keyedHash object
24 //     TPM_RC_KEY             the size of the object's 'unique' field is not
25 //                             consistent with the indicated size in the object's
26 //                             parameters
27 //     TPM_RC_OBJECT_MEMORY  if there is no free slot for an object
28 //     TPM_RC_ECC_POINT      for a public-only ECC key, the ECC point is not
29 //                             on the curve
30 //     TPM_RC_SCHEME          the signing scheme is not valid for the key
31 //     TPM_RC_SIZE            'authPolicy' is not zero and is not the size of a
32 //                             digest produced by the object's 'nameAlg'
33 //     TPM_RH_NULL hierarchy

```

```

34 //      TPM_RC_SYMMETRIC      symmetric algorithm not provided when required
35 //      TPM_RC_TYPE           'inPublic' and 'inPrivate' are not the same type
36 TPM_RC
37 TPM2_LoadExternal(LoadExternal_In* in, // IN: input parameter list
38                  LoadExternal_Out* out // OUT: output parameter list
39 )
40 {
41     TPM_RC      result;
42     OBJECT*     object;
43     TPMT_SENSITIVE* sensitive = NULL;
44
45     // Input Validation
46     // Don't get invested in loading if there is no place to put it.
47     object = FindEmptyObjectSlot(&out->objectHandle);
48     if(object == NULL)
49         return TPM_RC_OBJECT_MEMORY;
50
51     // If the hierarchy to be associated with this object is turned off, the object
52     // cannot be loaded.
53     if(!HierarchyIsEnabled(in->hierarchy))
54         return TPM_RCS_HIERARCHY + RC_LoadExternal_hierarchy;
55
56     // For loading an object with both public and sensitive
57     if(in->inPrivate.size != 0)
58     {
59         // An external object with a sensitive area can only be loaded in the
60         // NULL hierarchy
61         if(in->hierarchy != TPM_RH_NULL)
62             return TPM_RCS_HIERARCHY + RC_LoadExternal_hierarchy;
63         // An external object with a sensitive area must have fixedTPM == CLEAR
64         // fixedParent == CLEAR so that it does not appear to be a key created by
65         // this TPM.
66         if(IS_ATTRIBUTE(
67             in->inPublic.publicArea.objectAttributes, TPMA_OBJECT, fixedTPM)
68             || IS_ATTRIBUTE(
69                 in->inPublic.publicArea.objectAttributes, TPMA_OBJECT, fixedParent)
70             || IS_ATTRIBUTE(
71                 in->inPublic.publicArea.objectAttributes, TPMA_OBJECT, restricted))
72             return TPM_RCS_ATTRIBUTES + RC_LoadExternal_inPublic;
73
74         // Have sensitive point to something other than NULL so that object
75         // initialization will load the sensitive part too
76         sensitive = &in->inPrivate.sensitiveArea;
77     }
78
79     // Need the name to initialize the object structure
80     PublicMarshalAndComputeName(&in->inPublic.publicArea, &out->name);
81
82     // Load and validate key
83     result = ObjectLoad(object,
84                        NULL,
85                        &in->inPublic.publicArea,
86                        sensitive,
87                        RC_LoadExternal_inPublic,
88                        RC_LoadExternal_inPrivate,
89                        &out->name);
90     if(result == TPM_RC_SUCCESS)
91     {
92         object->attributes.external = SET;
93         // Set the common OBJECT attributes for a loaded object.
94         ObjectSetLoadedAttributes(object, in->hierarchy);
95     }
96     return result;
97 }
98
99 #endif // CC_LoadExternal

```

7.106 /tpm/src/command/Object/MakeCredential.c

```

1  #include "Tpm.h"
2  #include "MakeCredential_fp.h"
3
4  #if CC_MakeCredential // Conditional expansion of this file
5
6  # include "Object_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Make Credential with an object
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_KEY           'handle' referenced an ECC key that has a unique
13 //                          field that is not a point on the curve of the key
14 //     TPM_RC_SIZE         'credential' is larger than the digest size of
15 //                          Name algorithm of 'handle'
16 //     TPM_RC_TYPE         'handle' does not reference an asymmetric
17 //                          decryption key
18 TPM_RC
19 TPM2_MakeCredential(MakeCredential_In* in, // IN: input parameter list
20                   MakeCredential_Out* out // OUT: output parameter list
21 )
22 {
23     TPM_RC    result = TPM_RC_SUCCESS;
24
25     OBJECT*    object;
26     TPM2B_DATA data;
27
28     // Input Validation
29
30     // Get object pointer
31     object = HandleToObject(in->handle);
32
33     // input key must be an asymmetric, restricted decryption key
34     // NOTE: Needs to be restricted to have a symmetric value.
35     if(!CryptIsAsymAlgorithm(object->publicArea.type)
36        || !IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, decrypt)
37        || !IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, restricted))
38         return TPM_RC_SIZE + RC_MakeCredential_handle;
39
40     // The credential information may not be larger than the digest size used for
41     // the Name of the key associated with handle.
42     if(in->credential.t.size > CryptHashGetDigestSize(object->publicArea.nameAlg))
43         return TPM_RC_SIZE + RC_MakeCredential_credential;
44
45     // Command Output
46
47     // Make encrypt key and its associated secret structure.
48     out->secret.t.size = sizeof(out->secret.t.secret);
49     result = CryptSecretEncrypt(object, IDENTITY_STRING, &data, &out->secret);
50     if(result != TPM_RC_SUCCESS)
51         return result;
52
53     // Prepare output credential data from secret
54     SecretToCredential(
55         &in->credential, &in->objectName.b, &data.b, object, &out->credentialBlob);
56
57     return TPM_RC_SUCCESS;
58 }
59
60 #endif // CC_MakeCredential

```

7.107 /tpm/src/command/Object/ObjectChangeAuth.c

```

1  #include "Tpm.h"
2  #include "ObjectChangeAuth_fp.h"
3
4  #if CC_ObjectChangeAuth // Conditional expansion of this file
5
6  # include "Object_spt_fp.h"
7
8  /*(See part 3 specification)
9  // Create an object
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_SIZE           'newAuth' is larger than the size of the digest
13 //                           of the Name algorithm of 'objectHandle'
14 //     TPM_RC_TYPE           the key referenced by 'parentHandle' is not the
15 //                           parent of the object referenced by 'objectHandle';
16 //                           or 'objectHandle' is a sequence object.
17 TPM_RC
18 TPM2_ObjectChangeAuth(ObjectChangeAuth_In* in, // IN: input parameter list
19                      ObjectChangeAuth_Out* out // OUT: output parameter list
20 )
21 {
22     TPMT_SENSITIVE sensitive;
23
24     OBJECT*      object = HandleToObject(in->objectHandle);
25     TPM2B_NAME    QNCompare;
26
27     // Input Validation
28
29     // Can not change authorization on sequence object
30     if(ObjectIsSequence(object))
31         return TPM_RCS_TYPE + RC_ObjectChangeAuth_objectHandle;
32
33     // Make sure that the authorization value is consistent with the nameAlg
34     if(!AdjustAuthSize(&in->newAuth, object->publicArea.nameAlg))
35         return TPM_RCS_SIZE + RC_ObjectChangeAuth_newAuth;
36
37     // Parent handle should be the parent of object handle. In this
38     // implementation we verify this by checking the QN of object. Other
39     // implementation may choose different method to verify this attribute.
40     ComputeQualifiedName(
41         in->parentHandle, object->publicArea.nameAlg, &object->name, &QNCompare);
42     if(!MemoryEqual2B(&object->qualifiedName.b, &QNCompare.b))
43         return TPM_RCS_TYPE + RC_ObjectChangeAuth_parentHandle;
44
45     // Command Output
46     // Prepare the sensitive area with the new authorization value
47     sensitive      = object->sensitive;
48     sensitive.authValue = in->newAuth;
49
50     // Protect the sensitive area
51     SensitiveToPrivate(&sensitive,
52                      &object->name,
53                      HandleToObject(in->parentHandle),
54                      object->publicArea.nameAlg,
55                      &out->outPrivate);
56     return TPM_RC_SUCCESS;
57 }
58
59 #endif // CC_ObjectChangeAuth

```

7.108 /tpm/src/command/Object/Object_spt.c

```

1  /** Includes

```



```

2  #include "Tpm.h"
3  #include "Object_spt_fp.h"
4  #include "Marshal.h"
5
6  /** Local Functions
7
8  /*** GetIV2BSize()
9  // Get the size of TPM2B_IV in canonical form that will be append to the start of
10 // the sensitive data. It includes both size of size field and size of iv data
11 static UINT16 GetIV2BSize(OBJECT* protector // IN: the protector handle
12 )
13 {
14     TPM_ALG_ID symAlg;
15     UINT16 keyBits;
16
17     // Determine the symmetric algorithm and size of key
18     if(protector == NULL)
19     {
20         // Use the context encryption algorithm and key size
21         symAlg = CONTEXT_ENCRYPT_ALG;
22         keyBits = CONTEXT_ENCRYPT_KEY_BITS;
23     }
24     else
25     {
26         symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
27         keyBits = protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
28     }
29
30     // The IV size is a UINT16 size field plus the block size of the symmetric
31     // algorithm
32     return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
33 }
34
35 /*** ComputeProtectionKeyParms()
36 // This function retrieves the symmetric protection key parameters for
37 // the sensitive data
38 // The parameters retrieved from this function include encryption algorithm,
39 // key size in bit, and a TPM2B_SYM_KEY containing the key material as well as
40 // the key size in bytes
41 // This function is used for any action that requires encrypting or decrypting of
42 // the sensitive area of an object or a credential blob
43 //
44 /* (See part 1 specification)
45     KDF for generating the protection key material:
46     KDFa(hashAlg, seed, "STORAGE", Name, NULL, bits)
47 where
48     hashAlg    for a Primary Object, an algorithm chosen by the TPM vendor
49                for derivations from Primary Seeds. For all other objects,
50                the nameAlg of the object's parent.
51     seed        for a Primary Object in the Platform Hierarchy, the PPS.
52                For Primary Objects in either Storage or Endorsement Hierarchy,
53                the SPS. For Temporary Objects, the context encryption seed.
54                For all other objects, the symmetric seed value in the
55                sensitive area of the object's parent.
56     STORAGE    label to differentiate use of KDFa() (see 4.7)
57     Name        the Name of the object being encrypted
58     bits        the number of bits required for a symmetric key and IV
59 */
60 // Return Type: void
61 static void ComputeProtectionKeyParms(
62     OBJECT* protector, // IN: the protector object
63     TPM_ALG_ID hashAlg, // IN: hash algorithm for KDFa
64     TPM2B* name, // IN: name of the object
65     TPM2B* seedIn, // IN: optional seed for duplication blob.
66                // For non duplication blob, this
67                // parameter should be NULL

```

```

68     TPM_ALG_ID*    symAlg,    // OUT: the symmetric algorithm
69     UINT16*        keyBits,   // OUT: the symmetric key size in bits
70     TPM2B_SYM_KEY* symKey     // OUT: the symmetric key
71 )
72 {
73     const TPM2B* seed = seedIn;
74
75     // Determine the algorithms for the KDF and the encryption/decryption
76     // For TPM_RH_NULL, using context settings
77     if(protector == NULL)
78     {
79         // Use the context encryption algorithm and key size
80         *symAlg      = CONTEXT_ENCRYPT_ALG;
81         symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
82         *keyBits     = CONTEXT_ENCRYPT_KEY_BITS;
83     }
84     else
85     {
86         TPMT_SYM_DEF_OBJECT* symDef;
87         symDef               = &protector->publicArea.parameters.asymDetail.symmetric;
88         *symAlg              = symDef->algorithm;
89         *keyBits             = symDef->keyBits.sym;
90         symKey->t.size = (*keyBits + 7) / 8;
91     }
92     // Get seed for KDF
93     if(seed == NULL)
94         seed = GetSeedForKDF(protector);
95     // KDFa to generate symmetric key and IV value
96     CryptKDFa(hashAlg,
97              seed,
98              STORAGE_KEY,
99              name,
100             NULL,
101             symKey->t.size * 8,
102             symKey->t.buffer,
103             NULL,
104             FALSE);
105     return;
106 }
107
108 /*** ComputeOuterIntegrity()
109 // The sensitive area parameter is a buffer that holds a space for
110 // the integrity value and the marshaled sensitive area. The caller should
111 // skip over the area set aside for the integrity value
112 // and compute the hash of the remainder of the object.
113 // The size field of sensitive is in unmarshaled form and the
114 // sensitive area contents is an array of bytes.
115 /* (See part 1 specification)
116     KDFa(hashAlg, seed, "INTEGRITY", NULL, NULL, bits)    (38)
117 where
118     hashAlg    for a Primary Object, the nameAlg of the object. For all other
119                objects the nameAlg of the object's parent.
120     seed       for a Primary Object in the Platform Hierarchy, the PPS. For
121                Primary Objects in either Storage or Endorsement Hierarchy,
122                the SPS. For a Temporary Object, the context encryption key.
123                For all other objects, the symmetric seed value in the sensitive
124                area of the object's parent.
125     "INTEGRITY" a value used to differentiate the uses of the KDF.
126     bits       the number of bits in the digest produced by hashAlg.
127 Key is then used in the integrity computation.
128     HMACnameAlg(HMACkey, encSensitive || Name )
129 where
130     HMACnameAlg() the HMAC function using nameAlg of the object's parent
131     HMACkey       value derived from the parent symmetric protection value
132     encSensitive  symmetrically encrypted sensitive area
133     Name          the Name of the object being protected

```

```

134  */
135  // Return Type: void
136  static void ComputeOuterIntegrity(
137      TPM2B* name,           // IN: the name of the object
138      OBJECT* protector,    // IN: the object that
139                           // provides protection. For an object,
140                           // it is a parent. For a credential, it
141                           // is the encrypt object. For
142                           // a Temporary Object, it is NULL
143      TPMI_ALG_HASH hashAlg, // IN: algorithm to use for integrity
144      TPM2B* seedIn,        // IN: an external seed may be provided for
145                           // duplication blob. For non duplication
146                           // blob, this parameter should be NULL
147      UINT32 sensitiveSize, // IN: size of the marshaled sensitive data
148      BYTE* sensitiveData,  // IN: sensitive area
149      TPM2B_DIGEST* integrity // OUT: integrity
150  )
151  {
152      HMAC_STATE hmacState;
153      TPM2B_DIGEST hmacKey;
154      const TPM2B* seed = seedIn;
155      //
156      // Get seed for KDF
157      if (seed == NULL)
158          seed = GetSeedForKDF(protector);
159      // Determine the HMAC key bits
160      hmacKey.t.size = CryptHashGetDigestSize(hashAlg);
161
162      // KDFa to generate HMAC key
163      CryptKDFa(hashAlg,
164                seed,
165                INTEGRITY_KEY,
166                NULL,
167                NULL,
168                hmacKey.t.size * 8,
169                hmacKey.t.buffer,
170                NULL,
171                FALSE);
172      // Start HMAC and get the size of the digest which will become the integrity
173      integrity->t.size = CryptHmacStart2B(&hmacState, hashAlg, &hmacKey.b);
174
175      // Adding the marshaled sensitive area to the integrity value
176      CryptDigestUpdate(&hmacState.hashState, sensitiveSize, sensitiveData);
177
178      // Adding name
179      CryptDigestUpdate2B(&hmacState.hashState, name);
180
181      // Compute HMAC
182      CryptHmacEnd2B(&hmacState, &integrity->b);
183
184      return;
185  }
186
187  /*** ComputeInnerIntegrity()
188  // This function computes the integrity of an inner wrap
189  static void ComputeInnerIntegrity(
190      TPM_ALG_ID hashAlg, // IN: hash algorithm for inner wrap
191      TPM2B* name,        // IN: the name of the object
192      UINT16 dataSize,    // IN: the size of sensitive data
193      BYTE* sensitiveData, // IN: sensitive data
194      TPM2B_DIGEST* integrity // OUT: inner integrity
195  )
196  {
197      HASH_STATE hashState;
198      //
199      // Start hash and get the size of the digest which will become the integrity

```

```

200     integrity->t.size = CryptHashStart(&hashState, hashAlg);
201
202     // Adding the marshaled sensitive area to the integrity value
203     CryptDigestUpdate(&hashState, dataSize, sensitiveData);
204
205     // Adding name
206     CryptDigestUpdate2B(&hashState, name);
207
208     // Compute hash
209     CryptHashEnd2B(&hashState, &integrity->b);
210
211     return;
212 }
213
214 /*** ProduceInnerIntegrity()
215 // This function produces an inner integrity for regular private, credential or
216 // duplication blob
217 // It requires the sensitive data being marshaled to the innerBuffer, with the
218 // leading bytes reserved for integrity hash. It assume the sensitive data
219 // starts at address (innerBuffer + integrity size).
220 // This function integrity at the beginning of the inner buffer
221 // It returns the total size of buffer with the inner wrap
222 static UINT16 ProduceInnerIntegrity(
223     TPM2B* name, // IN: the name of the object
224     TPM_ALG_ID hashAlg, // IN: hash algorithm for inner wrap
225     UINT16 dataSize, // IN: the size of sensitive data, excluding the
226                     // leading integrity buffer size
227     BYTE* innerBuffer // IN/OUT: inner buffer with sensitive data in
228                     // it. At input, the leading bytes of this
229                     // buffer is reserved for integrity
230 )
231 {
232     BYTE* sensitiveData; // pointer to the sensitive data
233     TPM2B_DIGEST integrity;
234     UINT16 integritySize;
235     BYTE* buffer; // Auxiliary buffer pointer
236     //
237     // sensitiveData points to the beginning of sensitive data in innerBuffer
238     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
239     sensitiveData = innerBuffer + integritySize;
240
241     ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);
242
243     // Add integrity at the beginning of inner buffer
244     buffer = innerBuffer;
245     TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
246
247     return dataSize + integritySize;
248 }
249
250 /*** CheckInnerIntegrity()
251 // This function check integrity of inner blob
252 // Return Type: TPM_RC
253 // TPM_RC_INTEGRITY if the outer blob integrity is bad
254 // unmarshal errors unmarshal errors while unmarshaling integrity
255 static TPM_RC CheckInnerIntegrity(
256     TPM2B* name, // IN: the name of the object
257     TPM_ALG_ID hashAlg, // IN: hash algorithm for inner wrap
258     UINT16 dataSize, // IN: the size of sensitive data, including the
259                     // leading integrity buffer size
260     BYTE* innerBuffer // IN/OUT: inner buffer with sensitive data in
261                     // it
262 )
263 {
264     TPM_RC result;
265     TPM2B_DIGEST integrity;

```

```

266     TPM2B_DIGEST integrityToCompare;
267     BYTE*          buffer; // Auxiliary buffer pointer
268     INT32          size;
269     //
270     // Unmarshal integrity
271     buffer = innerBuffer;
272     size = (INT32)dataSize;
273     result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
274     if(result == TPM_RC_SUCCESS)
275     {
276         // Compute integrity to compare
277         ComputeInnerIntegrity(
278             hashAlg, name, (UINT16)size, buffer, &integrityToCompare);
279         // Compare outer blob integrity
280         if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
281             result = TPM_RC_INTEGRITY;
282     }
283     return result;
284 }
285
286 /** Public Functions
287
288 **** AdjustAuthSize()
289 // This function will validate that the input authValue is no larger than the
290 // digestSize for the nameAlg. It will then pad with zeros to the size of the
291 // digest.
292 BOOL AdjustAuthSize(TPM2B_AUTH* auth, // IN/OUT: value to adjust
293                     TPMI_ALG_HASH nameAlg // IN:
294 )
295 {
296     UINT16 digestSize;
297     //
298     // If there is no nameAlg, then this is a LoadExternal and the authVale can
299     // be any size up to the maximum allowed by the implementation
300     digestSize = (nameAlg == TPM_ALG_NULL) ? sizeof(TPMU_HA)
301                                           : CryptHashGetDigestSize(nameAlg);
302     if(digestSize < MemoryRemoveTrailingZeros(auth))
303         return FALSE;
304     else if(digestSize > auth->t.size)
305         MemoryPad2B(&auth->b, digestSize);
306     auth->t.size = digestSize;
307
308     return TRUE;
309 }
310
311 **** AreAttributesForParent()
312 // This function is called by create, load, and import functions.
313 //
314 // Note: The 'isParent' attribute is SET when an object is loaded and it has
315 // attributes that are suitable for a parent object.
316 // Return Type: BOOL
317 //     TRUE(1)           properties are those of a parent
318 //     FALSE(0)          properties are not those of a parent
319 BOOL ObjectIsParent(OBJECT* parentObject // IN: parent handle
320 )
321 {
322     return parentObject->attributes.isParent;
323 }
324
325 **** CreateChecks()
326 // Attribute checks that are unique to creation.
327 // If parentObject is not NULL, then this function checks the object's
328 // attributes as an Ordinary or Derived Object with the given parent.
329 // If parentObject is NULL, and primaryHandle is not 0, then this function
330 // checks the object's attributes as a Primary Object in the given hierarchy.
331 // If parentObject is NULL, and primaryHandle is 0, then this function checks

```



```

332 // the object's attributes as an External Object.
333 // Return Type: TPM_RC
334 //     TPM_RC_ATTRIBUTES    sensitiveDataOrigin is not consistent with the
335 //                          object type
336 //     other                returns from PublicAttributesValidation()
337 TPM_RC
338 CreateChecks(OBJECT*          parentObject,
339             TPMI_RH_HIERARCHY primaryHierarchy,
340             TPMT_PUBLIC*      publicArea,
341             UINT16            sensitiveDataSize)
342 {
343     TPMA_OBJECT attributes = publicArea->objectAttributes;
344     TPM_RC      result    = TPM_RC_SUCCESS;
345     //
346     // If the caller indicates that they have provided the data, then make sure that
347     // they have provided some data.
348     if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
349         && (sensitiveDataSize == 0))
350         return TPM_RC_ATTRIBUTES;
351     // For an ordinary object, data can only be provided when sensitiveDataOrigin
352     // is CLEAR
353     if((parentObject != NULL)
354         && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
355         && (sensitiveDataSize != 0))
356         return TPM_RC_ATTRIBUTES;
357     switch(publicArea->type)
358     {
359         case TPM_ALG_KEYEDHASH:
360             // if this is a data object (sign == decrypt == CLEAR) then the
361             // TPM cannot be the data source.
362             if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
363                 && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
364                 && IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
365                 result = TPM_RC_ATTRIBUTES;
366             // comment out the next line in order to prevent a fixedTPM derivation
367             // parent
368             // break;
369         case TPM_ALG_SYMCIPHER:
370             // A restricted key symmetric key (SYMCIPHER and KEYEDHASH)
371             // must have sensitiveDataOrigin SET unless it has fixedParent and
372             // fixedTPM CLEAR.
373             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
374                 if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
375                     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
376                         || IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
377                         result = TPM_RC_ATTRIBUTES;
378             break;
379         default: // Asymmetric keys cannot have the sensitive portion provided
380             if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
381                 result = TPM_RC_ATTRIBUTES;
382             break;
383     }
384     if(TPM_RC_SUCCESS == result)
385     {
386         result =
387             PublicAttributesValidation(parentObject, primaryHierarchy, publicArea);
388     }
389     return result;
390 }
391
392 /*** SchemeChecks
393 // This function is called by TPM2_LoadExternal() and PublicAttributesValidation().
394 // This function validates the schemes in the public area of an object.
395 // Return Type: TPM_RC
396 //     TPM_RC_HASH    non-duplicable storage key and its parent have different
397 //                    name algorithm

```



```

398 //      TPM_RC_KDF          incorrect KDF specified for decrypting keyed hash object
399 //      TPM_RC_KEY          invalid key size values in an asymmetric key public area
400 //      TPM_RCS_SCHEME      inconsistent attributes 'decrypt', 'sign', 'restricted'
401 //                          and key's scheme ID; or hash algorithm is inconsistent
402 //                          with the scheme ID for keyed hash object
403 //      TPM_RC_SYMMETRIC    a storage key with no symmetric algorithm specified; or
404 //                          non-storage key with symmetric algorithm different from
405 //                          TPM_ALG_NULL
406 TPM_RC
407 SchemeChecks(OBJECT*      parentObject, // IN: parent (null if primary seed)
408              TPMT_PUBLIC* publicArea   // IN: public area of the object
409 )
410 {
411     TPMT_SYM_DEF_OBJECT* symAlgs      = NULL;
412     TPM_ALG_ID           scheme       = TPM_ALG_NULL;
413     TPMA_OBJECT          attributes   = publicArea->objectAttributes;
414     TPMU_PUBLIC_PARMS*   parms       = &publicArea->parameters;
415     //
416     switch(publicArea->type)
417     {
418         case TPM_ALG_SYMCIPHER:
419             symAlgs = &parms->symDetail.sym;
420             // If this is a decrypt key, then only the block cipher modes (not
421             // SMAC) are valid. TPM_ALG_NULL is OK too. If this is a 'sign' key,
422             // then any mode that got through the unmarshaling is OK.
423             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
424                 && !CryptSymModeIsValid(symAlgs->mode.sym, TRUE))
425                 return TPM_RCS_SCHEME;
426             break;
427         case TPM_ALG_KEYEDHASH:
428             scheme = parms->keyedHashDetail.scheme.scheme;
429             // if both sign and decrypt
430             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
431                 == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
432             {
433                 // if both sign and decrypt are set or clear, then need
434                 // TPM_ALG_NULL as scheme
435                 if(scheme != TPM_ALG_NULL)
436                     return TPM_RCS_SCHEME;
437             }
438             else if(
439                 IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign) && scheme != TPM_ALG_HMAC)
440                 return TPM_RCS_SCHEME;
441             else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
442             {
443                 if(scheme != TPM_ALG_XOR)
444                     return TPM_RCS_SCHEME;
445                 // If this is a derivation parent, then the KDF needs to be
446                 // SP800-108 for this implementation. This is the only derivation
447                 // supported by this implementation. Other implementations could
448                 // support additional schemes. There is no default.
449                 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
450                 {
451                     if(parms->keyedHashDetail.scheme.details.
452                         xor.kdf != TPM_ALG_KDF1_SP800_108)
453                         return TPM_RCS_SCHEME;
454                     // Must select a digest.
455                     if(CryptHashGetDigestSize(
456                         parms->keyedHashDetail.scheme.details.xor.hashAlg)
457                         == 0)
458                         return TPM_RCS_HASH;
459                 }
460             }
461             break;
462         default: // handling for asymmetric
463             scheme = parms->asymDetail.scheme.scheme;

```

```

464     symAlgs = &parms->asymDetail.symmetric;
465     // if the key is both sign and decrypt, then the scheme must be
466     // TPM_ALG_NULL because there is no way to specify both a sign and a
467     // decrypt scheme in the key.
468     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
469        == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
470     {
471         // scheme must be TPM_ALG_NULL
472         if(scheme != TPM_ALG_NULL)
473             return TPM_RCS_SCHEME;
474     }
475     else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
476     {
477         // If this is a signing key, see if it has a signing scheme
478         if(CryptIsAsymSignScheme(publicArea->type, scheme))
479         {
480             // if proper signing scheme then it needs a proper hash
481             if(parms->asymDetail.scheme.details.anySig.hashAlg
482                == TPM_ALG_NULL)
483                 return TPM_RCS_SCHEME;
484         }
485         else
486         {
487             // signing key that does not have a proper signing scheme.
488             // This is OK if the key is not restricted and its scheme
489             // is TPM_ALG_NULL
490             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
491                || scheme != TPM_ALG_NULL)
492                 return TPM_RCS_SCHEME;
493         }
494     }
495     else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
496     {
497         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
498         {
499             // for a restricted decryption key (a parent), scheme
500             // is required to be TPM_ALG_NULL
501             if(scheme != TPM_ALG_NULL)
502                 return TPM_RCS_SCHEME;
503         }
504         else
505         {
506             // For an unrestricted decryption key, the scheme has to
507             // be a valid scheme or TPM_ALG_NULL
508             if(scheme != TPM_ALG_NULL
509                && !CryptIsAsymDecryptScheme(publicArea->type, scheme))
510                 return TPM_RCS_SCHEME;
511         }
512     }
513     if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
514        || !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
515     {
516         // For an asymmetric key that is not a parent, the symmetric
517         // algorithms must be TPM_ALG_NULL
518         if(symAlgs->algorithm != TPM_ALG_NULL)
519             return TPM_RCS_SYMMETRIC;
520     }
521     // Special checks for an ECC key
522     #if ALG_ECC
523     if(publicArea->type == TPM_ALG_ECC)
524     {
525         TPM_ECC_CURVE        curveID;
526         const TPMT_ECC_SCHEME* curveScheme;
527
528         curveID      = publicArea->parameters.eccDetail.curveID;
529         curveScheme = CryptGetCurveSignScheme(curveID);

```

```

530         // The curveId must be valid or the unmarshaling is busted.
531         pAssert(curveScheme != NULL);
532
533         // If the curveID requires a specific scheme, then the key must
534         // select the same scheme
535         if(curveScheme->scheme != TPM_ALG_NULL)
536         {
537             TPMS_ECC_PARMS* ecc = &publicArea->parameters.eccDetail;
538             if(scheme != curveScheme->scheme)
539                 return TPM_RCS_SCHEME;
540             // The scheme can allow any hash, or not...
541             if(curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
542                && (ecc->scheme.details.anySig.hashAlg
543                   != curveScheme->details.anySig.hashAlg))
544                 return TPM_RCS_SCHEME;
545         }
546         // For now, the KDF must be TPM_ALG_NULL
547         if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
548             return TPM_RCS_KDF;
549     }
550 #endif
551     break;
552 }
553 // If this is a restricted decryption key with symmetric algorithms, then it
554 // is an ordinary parent (not a derivation parent). It needs to specific
555 // symmetric algorithms other than TPM_ALG_NULL
556 if(symAlgs != NULL && IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
557    && IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
558 {
559     if(symAlgs->algorithm == TPM_ALG_NULL)
560         return TPM_RCS_SYMMETRIC;
561 #if 0 ///?
562 // This next check is under investigation. Need to see if it will break Windows
563 // before it is enabled. If it does not, then it should be default because a
564 // the mode used with a parent is always CFB and Part 2 indicates as much.
565 if(symAlgs->mode.sym != TPM_ALG_CFB)
566     return TPM_RCS_MODE;
567 #endif
568 // If this parent is not duplicable, then the symmetric algorithms
569 // (encryption and hash) must match those of its parent
570 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
571    && (parentObject != NULL))
572 {
573     if(publicArea->nameAlg != parentObject->publicArea.nameAlg)
574         return TPM_RCS_HASH;
575     if(!MemoryEqual(symAlgs,
576                    &parentObject->publicArea.parameters,
577                    sizeof(TPMT_SYM_DEF_OBJECT)))
578         return TPM_RCS_SYMMETRIC;
579 }
580 }
581 return TPM_RC_SUCCESS;
582 }
583
584 /*** PublicAttributesValidation()
585 // This function validates the values in the public area of an object.
586 // This function is used in the processing of TPM2_Create, TPM2_CreatePrimary,
587 // TPM2_CreateLoaded(), TPM2_Load(), TPM2_Import(), and TPM2_LoadExternal().
588 // For TPM2_Import() this is only used if the new parent has fixedTPM SET. For
589 // TPM2_LoadExternal(), this is not used for a public-only key
590 // Return Type: TPM_RC
591 //     TPM_RC_ATTRIBUTES 'fixedTPM', 'fixedParent', or 'encryptedDuplication'
592 //     attributes are inconsistent between themselves or with
593 //     those of the parent object;
594 //     inconsistent 'restricted', 'decrypt' and 'sign'
595 //     attributes;

```

```

596 //          attempt to inject sensitive data for an asymmetric key;
597 //          attempt to create a symmetric cipher key that is not
598 //          a decryption key
599 //          TPM_RC_HASH          nameAlg is TPM_ALG_NULL
600 //          TPM_RC_SIZE          'authPolicy' size does not match digest size of the name
601 //          algorithm in 'publicArea'
602 //          other                returns from SchemeChecks()
603 TPM_RC
604 PublicAttributesValidation(
605     // IN: input parent object (if ordinary or derived object; NULL otherwise)
606     OBJECT* parentObject,
607     // IN: hierarchy (if primary object; 0 otherwise)
608     TPMI_RH_HIERARCHY primaryHierarchy,
609     // IN: public area of the object
610     TPMT_PUBLIC* publicArea)
611 {
612     TPMA_OBJECT attributes          = publicArea->objectAttributes;
613     TPMA_OBJECT parentAttributes = TPMA_ZERO_INITIALIZER();
614
615     if(parentObject != NULL)
616         parentAttributes = parentObject->publicArea.objectAttributes;
617     if(publicArea->nameAlg == TPM_ALG_NULL)
618         return TPM_RCS_HASH;
619     // If there is an authPolicy, it needs to be the size of the digest produced
620     // by the nameAlg of the object
621     if((publicArea->authPolicy.t.size != 0
622         && (publicArea->authPolicy.t.size
623             != CryptHashGetDigestSize(publicArea->nameAlg))))
624         return TPM_RCS_SIZE;
625     // If the parent is fixedTPM (including a Primary Object) the object must have
626     // the same value for fixedTPM and fixedParent
627     if(parentObject == NULL || IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
628     {
629         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
630             != IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
631             return TPM_RCS_ATTRIBUTES;
632     }
633     else
634     {
635         // The parent is not fixedTPM so the object can't be fixedTPM
636         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
637             return TPM_RCS_ATTRIBUTES;
638     }
639     // See if sign and decrypt are the same
640     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
641         == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
642     {
643         // a restricted key cannot have both SET or both CLEAR
644         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
645             return TPM_RC_ATTRIBUTES;
646         // only a data object may have both sign and decrypt CLEAR
647         // BTW, since we know that decrypt==sign, no need to check both
648         if(publicArea->type != TPM_ALG_KEYEDHASH
649             && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
650             return TPM_RC_ATTRIBUTES;
651     }
652     // If the object can't be duplicated (directly or indirectly) then there
653     // is no justification for having encryptedDuplication SET
654     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
655         && IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
656         return TPM_RCS_ATTRIBUTES;
657     // If a parent object has fixedTPM CLEAR, the child must have the
658     // same encryptedDuplication value as its parent.
659     // Primary objects are considered to have a fixedTPM parent (the seeds).
660     if(parentObject != NULL && !IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
661     {

```

```

662         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication)
663            != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, encryptedDuplication))
664             return TPM_RCS_ATTRIBUTES;
665     }
666     // firmwareLimited/svnLimited can only be set if fixedTPM is also set.
667     if((IS_ATTRIBUTE(attributes, TPMA_OBJECT, firmwareLimited)
668        || IS_ATTRIBUTE(attributes, TPMA_OBJECT, svnLimited))
669        && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
670     {
671         return TPM_RCS_ATTRIBUTES;
672     }
673
674     // firmwareLimited/svnLimited also impose requirements on the parent key or
675     // primary handle.
676     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, firmwareLimited))
677     {
678         if(parentObject != NULL)
679         {
680             // For an ordinary object, firmwareLimited can only be set if its
681             // parent is also firmwareLimited.
682             if(!IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, firmwareLimited))
683                 return TPM_RCS_ATTRIBUTES;
684         }
685         else if(primaryHierarchy != 0)
686         {
687             // For a primary object, firmwareLimited can only be set if its
688             // hierarchy is a firmware-limited hierarchy.
689             if(!HierarchyIsFirmwareLimited(primaryHierarchy))
690                 return TPM_RCS_ATTRIBUTES;
691         }
692         else
693         {
694             return TPM_RCS_ATTRIBUTES;
695         }
696     }
697     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, svnLimited))
698     {
699         if(parentObject != NULL)
700         {
701             // For an ordinary object, svnLimited can only be set if its
702             // parent is also svnLimited.
703             if(!IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, svnLimited))
704                 return TPM_RCS_ATTRIBUTES;
705         }
706         else if(primaryHierarchy != 0)
707         {
708             // For a primary object, svnLimited can only be set if its
709             // hierarchy is an svn-limited hierarchy.
710             if(!HierarchyIsSvnLimited(primaryHierarchy))
711                 return TPM_RCS_ATTRIBUTES;
712         }
713         else
714         {
715             return TPM_RCS_ATTRIBUTES;
716         }
717     }
718
719     // Special checks for derived objects
720     if((parentObject != NULL) && (parentObject->attributes.derivation == SET))
721     {
722         // A derived object has the same settings for fixedTPM as its parent
723         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
724            != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
725             return TPM_RCS_ATTRIBUTES;
726         // A derived object is required to be fixedParent
727         if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent))

```



```

728         return TPM_RCS_ATTRIBUTES;
729     }
730     return SchemeChecks(parentObject, publicArea);
731 }
732
733 /**** FillInCreationData()
734 // Fill in creation data for an object.
735 // Return Type: void
736 void FillInCreationData(
737     TPMI_DH_OBJECT    parentHandle,    // IN: handle of parent
738     TPMI_ALG_HASH      nameHashAlg,    // IN: name hash algorithm
739     TPML_PCR_SELECTION* creationPCR,    // IN: PCR selection
740     TPM2B_DATA*         outsideData,    // IN: outside data
741     TPM2B_CREATION_DATA* outCreation,    // OUT: creation data for output
742     TPM2B_DIGEST*       creationDigest  // OUT: creation digest
743 )
744 {
745     BYTE    creationBuffer[sizeof(TPMS_CREATION_DATA)];
746     BYTE*    buffer;
747     HASH_STATE hashState;
748     //
749     // Fill in TPMS_CREATION_DATA in outCreation
750
751     // Compute PCR digest
752     PCRComputeCurrentDigest(
753         nameHashAlg, creationPCR, &outCreation->creationData.pcrDigest);
754
755     // Put back PCR selection list
756     outCreation->creationData.pcrSelect = *creationPCR;
757
758     // Get locality
759     outCreation->creationData.locality = LocalityGetAttributes(_plat__LocalityGet());
760     outCreation->creationData.parentNameAlg = TPM_ALG_NULL;
761
762     // If the parent is either a primary seed or TPM_ALG_NULL, then the Name
763     // and QN of the parent are the parent's handle.
764     if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
765     {
766         buffer = &outCreation->creationData.parentName.t.name[0];
767         outCreation->creationData.parentName.t.size =
768             TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
769         // For a primary or temporary object, the parent name (a handle) and the
770         // parent's QN are the same
771         outCreation->creationData.parentQualifiedName =
772             outCreation->creationData.parentName;
773     }
774     else // Regular object
775     {
776         OBJECT* parentObject = HandleToObject(parentHandle);
777         //
778         // Set name algorithm
779         outCreation->creationData.parentNameAlg = parentObject->publicArea.nameAlg;
780
781         // Copy parent name
782         outCreation->creationData.parentName = parentObject->name;
783
784         // Copy parent qualified name
785         outCreation->creationData.parentQualifiedName = parentObject->qualifiedName;
786     }
787     // Copy outside information
788     outCreation->creationData.outsideInfo = *outsideData;
789
790     // Marshal creation data to canonical form
791     buffer = creationBuffer;
792     outCreation->size =
793         TPMS_CREATION_DATA_Marshal(&outCreation->creationData, &buffer, NULL);

```



```

794     // Compute hash for creation field in public template
795     creationDigest->t.size = CryptHashStart(&hashState, nameHashAlg);
796     CryptDigestUpdate(&hashState, outCreation->size, creationBuffer);
797     CryptHashEnd2B(&hashState, &creationDigest->b);
798
799     return;
800 }
801
802 /*** GetSeedForKDF()
803 // Get a seed for KDF. The KDF for encryption and HMAC key use the same seed.
804 const TPM2B* GetSeedForKDF(OBJECT* protector // IN: the protector handle
805 )
806 {
807     // Get seed for encryption key. Use input seed if provided.
808     // Otherwise, using protector object's seedValue. TPM_RH_NULL is the only
809     // exception that we may not have a loaded object as protector. In such a
810     // case, use nullProof as seed.
811     if(protector == NULL)
812         return &gr.nullProof.b;
813     else
814         return &protector->sensitive.seedValue.b;
815 }
816
817 /*** ProduceOuterWrap()
818 // This function produce outer wrap for a buffer containing the sensitive data.
819 // It requires the sensitive data being marshaled to the outerBuffer, with the
820 // leading bytes reserved for integrity hash. If iv is used, iv space should
821 // be reserved at the beginning of the buffer. It assumes the sensitive data
822 // starts at address (outerBuffer + integrity size [+ iv size]).
823 // This function performs:
824 // 1. Add IV before sensitive area if required
825 // 2. encrypt sensitive data, if iv is required, encrypt by iv. otherwise,
826 //    encrypted by a NULL iv
827 // 3. add HMAC integrity at the beginning of the buffer
828 // It returns the total size of blob with outer wrap
829 UINT16
830 ProduceOuterWrap(OBJECT* protector, // IN: The handle of the object that provides
831 // protection. For object, it is parent
832 // handle. For credential, it is the handle
833 // of encrypt object.
834 TPM2B* name, // IN: the name of the object
835 TPM_ALG_ID hashAlg, // IN: hash algorithm for outer wrap
836 TPM2B* seed, // IN: an external seed may be provided for
837 // duplication blob. For non duplication
838 // blob, this parameter should be NULL
839 BOOL useIV, // IN: indicate if an IV is used
840 UINT16 dataSize, // IN: the size of sensitive data, excluding the
841 // leading integrity buffer size or the
842 // optional iv size
843 BYTE* outerBuffer // IN/OUT: outer buffer with sensitive data in
844 // it
845 )
846 {
847     TPM_ALG_ID symAlg;
848     UINT16 keyBits;
849     TPM2B_SYM_KEY symKey;
850     TPM2B_IV ivRNG; // IV from RNG
851     TPM2B_IV* iv = NULL;
852     UINT16 ivSize = 0; // size of iv area, including the size field
853     BYTE* sensitiveData; // pointer to the sensitive data
854     TPM2B_DIGEST integrity;
855     UINT16 integritySize;
856     BYTE* buffer; // Auxiliary buffer pointer
857     //
858     // Compute the beginning of sensitive data. The outer integrity should
859     // always exist if this function is called to make an outer wrap

```

```

860 integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
861 sensitiveData = outerBuffer + integritySize;
862
863 // If iv is used, adjust the pointer of sensitive data and add iv before it
864 if(useIV)
865 {
866     ivSize = GetIV2BSize(protector);
867
868     // Generate IV from RNG. The iv data size should be the total IV area
869     // size minus the size of size field
870     ivRNG.t.size = ivSize - sizeof(UINT16);
871     CryptRandomGenerate(ivRNG.t.size, ivRNG.t.buffer);
872
873     // Marshal IV to buffer
874     buffer = sensitiveData;
875     TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
876
877     // adjust sensitive data starting after IV area
878     sensitiveData += ivSize;
879
880     // Use iv for encryption
881     iv = &ivRNG;
882 }
883 // Compute symmetric key parameters for outer buffer encryption
884 ComputeProtectionKeyParms(
885     protector, hashAlg, name, seed, &symAlg, &keyBits, &symKey);
886 // Encrypt inner buffer in place
887 CryptSymmetricEncrypt(sensitiveData,
888     symAlg,
889     keyBits,
890     symKey.t.buffer,
891     iv,
892     TPM_ALG_CFB,
893     dataSize,
894     sensitiveData);
895 // Compute outer integrity. Integrity computation includes the optional IV
896 // area
897 ComputeOuterIntegrity(name,
898     protector,
899     hashAlg,
900     seed,
901     dataSize + ivSize,
902     outerBuffer + integritySize,
903     &integrity);
904 // Add integrity at the beginning of outer buffer
905 buffer = outerBuffer;
906 TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
907
908 // return the total size in outer wrap
909 return dataSize + integritySize + ivSize;
910 }
911
912 /*** UnwrapOuter()
913 // This function remove the outer wrap of a blob containing sensitive data
914 // This function performs:
915 // 1. check integrity of outer blob
916 // 2. decrypt outer blob
917 //
918 // Return Type: TPM_RC
919 //     TPM_RCS_INSUFFICIENT    error during sensitive data unmarshaling
920 //     TPM_RCS_INTEGRITY       sensitive data integrity is broken
921 //     TPM_RCS_SIZE            error during sensitive data unmarshaling
922 //     TPM_RCS_VALUE           IV size for CFB does not match the encryption
923 //                             algorithm block size
924 TPM_RC
925 UnwrapOuter(OBJECT* protector, // IN: The object that provides

```

```

926                                     // protection. For object, it is parent
927                                     // handle. For credential, it is the
928                                     // encrypt object.
929     TPM2B*      name,                // IN: the name of the object
930     TPM_ALG_ID hashAlg,              // IN: hash algorithm for outer wrap
931     TPM2B*      seed,                // IN: an external seed may be provided for
932                                     // duplication blob. For non duplication
933                                     // blob, this parameter should be NULL.
934     BOOL        useIV,               // IN: indicates if an IV is used
935     UINT16      dataSize,            // IN: size of sensitive data in outerBuffer,
936                                     // including the leading integrity buffer
937                                     // size, and an optional iv area
938     BYTE*       outerBuffer          // IN/OUT: sensitive data
939 )
940 {
941     TPM_RC      result;
942     TPM_ALG_ID  symAlg = TPM_ALG_NULL;
943     TPM2B_SYM_KEY symKey;
944     UINT16      keyBits = 0;
945     TPM2B_IV    ivIn; // input IV retrieved from input buffer
946     TPM2B_IV*   iv = NULL;
947     BYTE*       sensitiveData; // pointer to the sensitive data
948     TPM2B_DIGEST integrityToCompare;
949     TPM2B_DIGEST integrity;
950     INT32       size;
951     //
952     // Unmarshal integrity
953     sensitiveData = outerBuffer;
954     size          = (INT32)dataSize;
955     result        = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
956     if(result == TPM_RC_SUCCESS)
957     {
958         // Compute integrity to compare
959         ComputeOuterIntegrity(name,
960                               protector,
961                               hashAlg,
962                               seed,
963                               (UINT16)size,
964                               sensitiveData,
965                               &integrityToCompare);
966         // Compare outer blob integrity
967         if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
968             return TPM_RC_INTEGRITY;
969         // Get the symmetric algorithm parameters used for encryption
970         ComputeProtectionKeyParms(
971             protector, hashAlg, name, seed, &symAlg, &keyBits, &symKey);
972         // Retrieve IV if it is used
973         if(useIV)
974         {
975             result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
976             if(result == TPM_RC_SUCCESS)
977             {
978                 // The input iv size for CFB must match the encryption algorithm
979                 // block size
980                 if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
981                     result = TPM_RC_VALUE;
982                 else
983                     iv = &ivIn;
984             }
985         }
986     }
987     // If no errors, decrypt private in place. Since this function uses CFB,
988     // CryptSymmetricDecrypt() will not return any errors. It may fail but it will
989     // not return an error.
990     if(result == TPM_RC_SUCCESS)
991         CryptSymmetricDecrypt(sensitiveData,

```

```

992         symAlg,
993         keyBits,
994         symKey.t.buffer,
995         iv,
996         TPM_ALG_CFB,
997         (UINT16)size,
998         sensitiveData);
999     return result;
1000 }
1001
1002 /*** MarshalSensitive()
1003 // This function is used to marshal a sensitive area. Among other things, it
1004 // adjusts the size of the authValue to be no smaller than the digest of
1005 // 'nameAlg'
1006 // Returns the size of the marshaled area.
1007 static UINT16 MarshalSensitive(
1008     OBJECT*      parent,      // IN: the object parent (optional)
1009     BYTE*        buffer,      // OUT: receiving buffer
1010     TPMT_SENSITIVE* sensitive, // IN: the sensitive area to marshal
1011     TPMI_ALG_HASH nameAlg     // IN:
1012 )
1013 {
1014     BYTE* sizeField = buffer; // saved so that size can be
1015                               // marshaled after it is known
1016     UINT16 retVal;
1017     //
1018     // Pad the authValue if needed
1019     MemoryPad2B(&sensitive->authValue.b, CryptHashGetDigestSize(nameAlg));
1020     buffer += 2;
1021
1022     // Marshal the structure
1023 #if ALG_RSA
1024     // If the sensitive size is the special case for a prime in the type
1025     if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) > 0)
1026     {
1027         UINT16 sizeSave = sensitive->sensitive.rsa.t.size;
1028         //
1029         // Turn off the flag that indicates that the sensitive->sensitive contains
1030         // the CRT form of the exponent.
1031         sensitive->sensitive.rsa.t.size &= ~(RSA_prime_flag);
1032         // If the parent isn't fixedTPM, then truncate the sensitive data to be
1033         // the size of the prime. Otherwise, leave it at the current size which
1034         // is the full CRT size.
1035         if(parent == NULL
1036            || !IS_ATTRIBUTE(
1037                parent->publicArea.objectAttributes, TPMA_OBJECT, fixedTPM))
1038             sensitive->sensitive.rsa.t.size /= 5;
1039         retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
1040         // Restore the flag and the size.
1041         sensitive->sensitive.rsa.t.size = sizeSave;
1042     }
1043     else
1044 #endif
1045         retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
1046
1047     // Marshal the size
1048     retVal = (UINT16)(retVal + UINT16_Marshal(&retVal, &sizeField, NULL));
1049
1050     return retVal;
1051 }
1052
1053 /*** SensitiveToPrivate()
1054 // This function prepare the private blob for off the chip storage
1055 // The operations in this function:
1056 // 1. marshal TPM2B_SENSITIVE structure into the buffer of TPM2B_PRIVATE
1057 // 2. apply encryption to the sensitive area.

```

```

1058 // 3. apply outer integrity computation.
1059 void SensitiveToPrivate(
1060     TPMT_SENSITIVE* sensitive, // IN: sensitive structure
1061     TPM2B_NAME* name, // IN: the name of the object
1062     OBJECT* parent, // IN: The parent object
1063     TPM_ALG_ID nameAlg, // IN: hash algorithm in public area. This
1064                         // parameter is used when parentHandle is
1065                         // NULL, in which case the object is
1066                         // temporary.
1067     TPM2B_PRIVATE* outPrivate // OUT: output private structure
1068 )
1069 {
1070     BYTE* sensitiveData; // pointer to the sensitive data
1071     UINT16 dataSize; // data blob size
1072     TPMI_ALG_HASH hashAlg; // hash algorithm for integrity
1073     UINT16 integritySize;
1074     UINT16 ivSize;
1075     //
1076     pAssert(name != NULL && name->t.size != 0);
1077
1078     // Find the hash algorithm for integrity computation
1079     if(parent == NULL)
1080     {
1081         // For Temporary Object, using self name algorithm
1082         hashAlg = nameAlg;
1083     }
1084     else
1085     {
1086         // Otherwise, using parent's name algorithm
1087         hashAlg = parent->publicArea.nameAlg;
1088     }
1089     // Starting of sensitive data without wrappers
1090     sensitiveData = outPrivate->t.buffer;
1091
1092     // Compute the integrity size
1093     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
1094
1095     // Reserve space for integrity
1096     sensitiveData += integritySize;
1097
1098     // Get iv size
1099     ivSize = GetIV2BSize(parent);
1100
1101     // Reserve space for iv
1102     sensitiveData += ivSize;
1103
1104     // Marshal the sensitive area including authValue size adjustments.
1105     dataSize = MarshalSensitive(parent, sensitiveData, sensitive, nameAlg);
1106
1107     //Produce outer wrap, including encryption and HMAC
1108     outPrivate->t.size = ProduceOuterWrap(
1109         parent, &name->b, hashAlg, NULL, TRUE, dataSize, outPrivate->t.buffer);
1110     return;
1111 }
1112
1113 /*** PrivateToSensitive()
1114 // Unwrap an input private area; check the integrity; decrypt and retrieve data
1115 // to a sensitive structure.
1116 // The operations in this function:
1117 // 1. check the integrity HMAC of the input private area
1118 // 2. decrypt the private buffer
1119 // 3. unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE
1120 //
1121 // Return Type: TPM_RC
1122 // TPM_RCS_INTEGRITY if the private area integrity is bad
1123 // TPM_RC_SENSITIVE unmarshal errors while unmarshaling TPMS_ENCRYPT

```

```

1124 //                                     from input private
1125 //     TPM_RCS_SIZE                    error during sensitive data unmarshaling
1126 //     TPM_RCS_VALUE                  outer wrapper does not have an iV of the correct
1127 //                                     size
1128 TPM_RC
1129 PrivateToSensitive(TPM2B*      inPrivate, // IN: input private structure
1130                   TPM2B*      name,      // IN: the name of the object
1131                   OBJECT*      parent,    // IN: parent object
1132                   TPM_ALG_ID   nameAlg,   // IN: hash algorithm in public area. It is
1133                   //            passed separately because we only pass
1134                   //            name, rather than the whole public area
1135                   //            of the object. This parameter is used in
1136                   //            the following two cases: 1. primary
1137                   //            objects. 2. duplication blob with inner
1138                   //            wrap. In other cases, this parameter
1139                   //            will be ignored
1140                   TPMT_SENSITIVE* sensitive // OUT: sensitive structure
1141 )
1142 {
1143     TPM_RC      result;
1144     BYTE*       buffer;
1145     INT32       size;
1146     BYTE*       sensitiveData; // pointer to the sensitive data
1147     UINT16      dataSize;
1148     UINT16      dataSizeInput;
1149     TPMI_ALG_HASH hashAlg; // hash algorithm for integrity
1150     UINT16      integritySize;
1151     UINT16      ivSize;
1152     //
1153     // Make sure that name is provided
1154     pAssert(name != NULL && name->size != 0);
1155
1156     // Find the hash algorithm for integrity computation
1157     // For Temporary Object (parent == NULL) use self name algorithm;
1158     // Otherwise, using parent's name algorithm
1159     hashAlg = (parent == NULL) ? nameAlg : parent->publicArea.nameAlg;
1160
1161     // unwrap outer
1162     result = UnwrapOuter(
1163         parent, name, hashAlg, NULL, TRUE, inPrivate->size, inPrivate->buffer);
1164     if(result != TPM_RC_SUCCESS)
1165         return result;
1166     // Compute the inner integrity size.
1167     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
1168
1169     // Get iv size
1170     ivSize = GetIV2BSize(parent);
1171
1172     // The starting of sensitive data and data size without outer wrapper
1173     sensitiveData = inPrivate->buffer + integritySize + ivSize;
1174     dataSize      = inPrivate->size - integritySize - ivSize;
1175
1176     // Unmarshal input data size
1177     buffer = sensitiveData;
1178     size   = (INT32)dataSize;
1179     result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1180     if(result == TPM_RC_SUCCESS)
1181     {
1182         if((dataSizeInput + sizeof(UINT16)) != dataSize)
1183             result = TPM_RC_SENSITIVE;
1184         else
1185         {
1186             // Unmarshal sensitive buffer to sensitive structure
1187             result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1188             if(result != TPM_RC_SUCCESS || size != 0)
1189                 return result;

```



```

1190         result = TPM_RC_SENSITIVE;
1191     }
1192 }
1193 }
1194 return result;
1195 }
1196
1197 /*** SensitiveToDuplicate()
1198 // This function prepare the duplication blob from the sensitive area.
1199 // The operations in this function:
1200 // 1. marshal TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE
1201 // 2. apply inner wrap to the sensitive area if required
1202 // 3. apply outer wrap if required
1203 void SensitiveToDuplicate(
1204     TPMT_SENSITIVE* sensitive,    // IN: sensitive structure
1205     TPM2B* name,                 // IN: the name of the object
1206     OBJECT* parent,              // IN: The new parent object
1207     TPM_ALG_ID nameAlg,          // IN: hash algorithm in public area. It
1208                                   // is passed separately because we
1209                                   // only pass name, rather than the
1210                                   // whole public area of the object.
1211     TPM2B* seed,                 // IN: the external seed. If external
1212                                   // seed is provided with size of 0,
1213                                   // no outer wrap should be applied
1214                                   // to duplication blob.
1215     TPMT_SYM_DEF_OBJECT* symDef, // IN: Symmetric key definition. If the
1216                                   // symmetric key algorithm is NULL,
1217                                   // no inner wrap should be applied.
1218     TPM2B_DATA* innerSymKey,      // IN/OUT: a symmetric key may be
1219                                   // provided to encrypt the inner
1220                                   // wrap of a duplication blob. May
1221                                   // be generated here if needed.
1222     TPM2B_PRIVATE* outPrivate     // OUT: output private structure
1223 )
1224 {
1225     BYTE* sensitiveData;          // pointer to the sensitive data
1226     TPMI_ALG_HASH outerHash = TPM_ALG_NULL; // The hash algorithm for outer wrap
1227     TPMI_ALG_HASH innerHash = TPM_ALG_NULL; // The hash algorithm for inner wrap
1228     UINT16 dataSize;              // data blob size
1229     BOOL doInnerWrap = FALSE;
1230     BOOL doOuterWrap = FALSE;
1231     //
1232     // Make sure that name is provided
1233     pAssert(name != NULL && name->size != 0);
1234
1235     // Make sure symDef and innerSymKey are not NULL
1236     pAssert(symDef != NULL && innerSymKey != NULL);
1237
1238     // Starting of sensitive data without wrappers
1239     sensitiveData = outPrivate->t.buffer;
1240
1241     // Find out if inner wrap is required
1242     if(symDef->algorithm != TPM_ALG_NULL)
1243     {
1244         doInnerWrap = TRUE;
1245
1246         // Use self nameAlg as inner hash algorithm
1247         innerHash = nameAlg;
1248
1249         // Adjust sensitive data pointer
1250         sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(innerHash);
1251     }
1252     // Find out if outer wrap is required
1253     if(seed->size != 0)
1254     {
1255         doOuterWrap = TRUE;

```

```

1256
1257     // Use parent nameAlg as outer hash algorithm
1258     outerHash = parent->publicArea.nameAlg;
1259
1260     // Adjust sensitive data pointer
1261     sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1262 }
1263 // Marshal sensitive area
1264 dataSize = MarshalSensitive(NULL, sensitiveData, sensitive, nameAlg);
1265
1266 // Apply inner wrap for duplication blob. It includes both integrity and
1267 // encryption
1268 if(doInnerWrap)
1269 {
1270     BYTE* innerBuffer = NULL;
1271     BOOL symKeyInput = TRUE;
1272     innerBuffer = outPrivate->t.buffer;
1273     // Skip outer integrity space
1274     if(doOuterWrap)
1275         innerBuffer += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1276     dataSize = ProduceInnerIntegrity(name, innerHash, dataSize, innerBuffer);
1277     // Generate inner encryption key if needed
1278     if(innerSymKey->t.size == 0)
1279     {
1280         innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
1281         CryptRandomGenerate(innerSymKey->t.size, innerSymKey->t.buffer);
1282
1283         // TPM generates symmetric encryption. Set the flag to FALSE
1284         symKeyInput = FALSE;
1285     }
1286     else
1287     {
1288         // assume the input key size should matches the symmetric definition
1289         pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1290     }
1291
1292     // Encrypt inner buffer in place
1293     CryptSymmetricEncrypt(innerBuffer,
1294                           symDef->algorithm,
1295                           symDef->keyBits.sym,
1296                           innerSymKey->t.buffer,
1297                           NULL,
1298                           TPM_ALG_CFB,
1299                           dataSize,
1300                           innerBuffer);
1301
1302     // If the symmetric encryption key is imported, clear the buffer for
1303     // output
1304     if(symKeyInput)
1305         innerSymKey->t.size = 0;
1306 }
1307 // Apply outer wrap for duplication blob. It includes both integrity and
1308 // encryption
1309 if(doOuterWrap)
1310 {
1311     dataSize = ProduceOuterWrap(
1312         parent, name, outerHash, seed, FALSE, dataSize, outPrivate->t.buffer);
1313 }
1314 // Data size for output
1315 outPrivate->t.size = dataSize;
1316
1317 return;
1318 }
1319
1320 /** DuplicateToSensitive()
1321 // Unwrap a duplication blob. Check the integrity, decrypt and retrieve data

```

```

1322 // to a sensitive structure.
1323 // The operations in this function:
1324 // 1. check the integrity HMAC of the input private area
1325 // 2. decrypt the private buffer
1326 // 3. unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE
1327 //
1328 // Return Type: TPM_RC
1329 //     TPM_RC_INSUFFICIENT    unmarshaling sensitive data from 'inPrivate' failed
1330 //     TPM_RC_INTEGRITY       'inPrivate' data integrity is broken
1331 //     TPM_RC_SIZE            unmarshaling sensitive data from 'inPrivate' failed
1332 TPM_RC
1333 DuplicateToSensitive(
1334     TPM2B*    inPrivate,        // IN: input private structure
1335     TPM2B*    name,            // IN: the name of the object
1336     OBJECT*   parent,          // IN: the parent
1337     TPM_ALG_ID nameAlg,        // IN: hash algorithm in public area.
1338     TPM2B*    seed,            // IN: an external seed may be provided.
1339                                // If external seed is provided with
1340                                // size of 0, no outer wrap is
1341                                // applied
1342     TPMT_SYM_DEF_OBJECT* symDef, // IN: Symmetric key definition. If the
1343                                // symmetric key algorithm is NULL,
1344                                // no inner wrap is applied
1345     TPM2B* innerSymKey,        // IN: a symmetric key may be provided
1346                                // to decrypt the inner wrap of a
1347                                // duplication blob.
1348     TPMT_SENSITIVE* sensitive // OUT: sensitive structure
1349 )
1350 {
1351     TPM_RC result;
1352     BYTE* buffer;
1353     INT32 size;
1354     BYTE* sensitiveData; // pointer to the sensitive data
1355     UINT16 dataSize;
1356     UINT16 dataSizeInput;
1357     //
1358     // Make sure that name is provided
1359     pAssert(name != NULL && name->size != 0);
1360
1361     // Make sure symDef and innerSymKey are not NULL
1362     pAssert(symDef != NULL && innerSymKey != NULL);
1363
1364     // Starting of sensitive data
1365     sensitiveData = inPrivate->buffer;
1366     dataSize      = inPrivate->size;
1367
1368     // Find out if outer wrap is applied
1369     if(seed->size != 0)
1370     {
1371         // Use parent nameAlg as outer hash algorithm
1372         TPMI_ALG_HASH outerHash = parent->publicArea.nameAlg;
1373
1374         result = UnwrapOuter(
1375             parent, name, outerHash, seed, FALSE, dataSize, sensitiveData);
1376         if(result != TPM_RC_SUCCESS)
1377             return result;
1378         // Adjust sensitive data pointer and size
1379         sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1380         dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1381     }
1382     // Find out if inner wrap is applied
1383     if(symDef->algorithm != TPM_ALG_NULL)
1384     {
1385         // assume the input key size matches the symmetric definition
1386         pAssert(innerSymKey->size == (symDef->keyBits.sym + 7) / 8);
1387     }

```

```

1388     // Decrypt inner buffer in place
1389     CryptSymmetricDecrypt(sensitiveData,
1390                          symDef->algorithm,
1391                          symDef->keyBits.sym,
1392                          innerSymKey->buffer,
1393                          NULL,
1394                          TPM_ALG_CFB,
1395                          dataSize,
1396                          sensitiveData);
1397
1398     // Check inner integrity
1399     result = CheckInnerIntegrity(name, nameAlg, dataSize, sensitiveData);
1400     if(result != TPM_RC_SUCCESS)
1401         return result;
1402     // Adjust sensitive data pointer and size
1403     sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1404     dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1405 }
1406 // Unmarshal input data size
1407 buffer = sensitiveData;
1408 size = (INT32)dataSize;
1409 result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1410 if(result == TPM_RC_SUCCESS)
1411 {
1412     if((dataSizeInput + sizeof(UINT16)) != dataSize)
1413         result = TPM_RC_SIZE;
1414     else
1415     {
1416         // Unmarshal sensitive buffer to sensitive structure
1417         result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1418
1419         // if the results is OK make sure that all the data was unmarshaled
1420         if(result == TPM_RC_SUCCESS && size != 0)
1421             result = TPM_RC_SIZE;
1422     }
1423 }
1424 return result;
1425 }
1426
1427 /** SecretToCredential()
1428 // This function prepare the credential blob from a secret (a TPM2B_DIGEST)
1429 // The operations in this function:
1430 // 1. marshal TPM2B_DIGEST structure into the buffer of TPM2B_ID_OBJECT
1431 // 2. encrypt the private buffer, excluding the leading integrity HMAC area
1432 // 3. compute integrity HMAC and append to the beginning of the buffer.
1433 // 4. Set the total size of TPM2B_ID_OBJECT buffer
1434 void SecretToCredential(TPM2B_DIGEST* secret, // IN: secret information
1435                        TPM2B* name, // IN: the name of the object
1436                        TPM2B* seed, // IN: an external seed.
1437                        OBJECT* protector, // IN: the protector
1438                        TPM2B_ID_OBJECT* outIDObject // OUT: output credential
1439 )
1440 {
1441     BYTE* buffer; // Auxiliary buffer pointer
1442     BYTE* sensitiveData; // pointer to the sensitive data
1443     TPMI_ALG_HASH outerHash; // The hash algorithm for outer wrap
1444     UINT16 dataSize; // data blob size
1445     //
1446     pAssert(secret != NULL && outIDObject != NULL);
1447
1448     // use protector's name algorithm as outer hash ???
1449     outerHash = protector->publicArea.nameAlg;
1450
1451     // Marshal secret area to credential buffer, leave space for integrity
1452     sensitiveData = outIDObject->t.credential + sizeof(UINT16)
1453                  + CryptHashGetDigestSize(outerHash);
1454     // Marshal secret area

```

```

1454     buffer    = sensitiveData;
1455     dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1456
1457     // Apply outer wrap
1458     outIDObject->t.size = ProduceOuterWrap(
1459         protector, name, outerHash, seed, FALSE, dataSize, outIDObject->t.credential);
1460     return;
1461 }
1462
1463 /*** CredentialToSecret()
1464 // Unwrap a credential. Check the integrity, decrypt and retrieve data
1465 // to a TPM2B_DIGEST structure.
1466 // The operations in this function:
1467 // 1. check the integrity HMAC of the input credential area
1468 // 2. decrypt the credential buffer
1469 // 3. unmarshal TPM2B_DIGEST structure into the buffer of TPM2B_DIGEST
1470 //
1471 // Return Type: TPM_RC
1472 //     TPM_RC_INSUFFICIENT    error during credential unmarshaling
1473 //     TPM_RC_INTEGRITY       credential integrity is broken
1474 //     TPM_RC_SIZE            error during credential unmarshaling
1475 //     TPM_RC_VALUE           IV size does not match the encryption algorithm
1476 //                             block size
1477 TPM_RC
1478 CredentialToSecret(TPM2B*    inIDObject, // IN: input credential blob
1479                   TPM2B*    name,       // IN: the name of the object
1480                   TPM2B*    seed,       // IN: an external seed.
1481                   OBJECT*    protector,  // IN: the protector
1482                   TPM2B_DIGEST* secret // OUT: secret information
1483 )
1484 {
1485     TPM_RC    result;
1486     BYTE*     buffer;
1487     INT32     size;
1488     TPMI_ALG_HASH outerHash; // The hash algorithm for outer wrap
1489     BYTE*     sensitiveData; // pointer to the sensitive data
1490     UINT16    dataSize;
1491     //
1492     // use protector's name algorithm as outer hash
1493     outerHash = protector->publicArea.nameAlg;
1494
1495     // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
1496     result = UnwrapOuter(protector,
1497                         name,
1498                         outerHash,
1499                         seed,
1500                         FALSE,
1501                         inIDObject->size,
1502                         inIDObject->buffer);
1503     if(result == TPM_RC_SUCCESS)
1504     {
1505         // Compute the beginning of sensitive data
1506         sensitiveData =
1507             inIDObject->buffer + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1508         dataSize =
1509             inIDObject->size - (sizeof(UINT16) + CryptHashGetDigestSize(outerHash));
1510         // Unmarshal secret buffer to TPM2B_DIGEST structure
1511         buffer = sensitiveData;
1512         size = (INT32)dataSize;
1513         result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
1514
1515         // If there were no other unmarshaling errors, make sure that the
1516         // expected amount of data was recovered
1517         if(result == TPM_RC_SUCCESS && size != 0)
1518             return TPM_RC_SIZE;
1519     }

```



```

1520     return result;
1521 }
1522
1523 /*** MemoryRemoveTrailingZeros()
1524 // This function is used to adjust the length of an authorization value.
1525 // It adjusts the size of the TPM2B so that it does not include octets
1526 // at the end of the buffer that contain zero.
1527 // The function returns the number of non-zero octets in the buffer.
1528 UINT16
1529 MemoryRemoveTrailingZeros(TPM2B_AUTH* auth // IN/OUT: value to adjust
1530 )
1531 {
1532     while((auth->t.size > 0) && (auth->t.buffer[auth->t.size - 1] == 0))
1533         auth->t.size--;
1534     return auth->t.size;
1535 }
1536
1537 /*** SetLabelAndContext()
1538 // This function sets the label and context for a derived key. It is possible
1539 // that 'label' or 'context' can end up being an Empty Buffer.
1540 TPM_RC
1541 SetLabelAndContext(TPMS_DERIVE* labelContext, // IN/OUT: the recovered label and
1542                  // context
1543                  TPM2B_SENSITIVE_DATA* sensitive // IN: the sensitive data
1544 )
1545 {
1546     TPMS_DERIVE sensitiveValue;
1547     TPM_RC result;
1548     INT32 size;
1549     BYTE* buff;
1550     //
1551     // Unmarshal a TPMS_DERIVE from the TPM2B_SENSITIVE_DATA buffer
1552     // If there is something to unmarshal...
1553     if(sensitive->t.size != 0)
1554     {
1555         size = sensitive->t.size;
1556         buff = sensitive->t.buffer;
1557         result = TPMS_DERIVE_Unmarshal(&sensitiveValue, &buff, &size);
1558         if(result != TPM_RC_SUCCESS)
1559             return result;
1560         // If there was a label in the public area leave it there, otherwise, copy
1561         // the new value
1562         if(labelContext->label.t.size == 0)
1563             MemoryCopy2B(&labelContext->label.b,
1564                         &sensitiveValue.label.b,
1565                         sizeof(labelContext->label.t.buffer));
1566         // if there was a context string in publicArea, it overrides
1567         if(labelContext->context.t.size == 0)
1568             MemoryCopy2B(&labelContext->context.b,
1569                         &sensitiveValue.context.b,
1570                         sizeof(labelContext->label.t.buffer));
1571     }
1572     return TPM_RC_SUCCESS;
1573 }
1574
1575 /*** UnmarshalToPublic()
1576 // Support function to unmarshal the template. This is used because the
1577 // Input may be a TPMT_TEMPLATE and that structure does not have the same
1578 // size as a TPMT_PUBLIC because of the difference between the 'unique' and
1579 // 'seed' fields.
1580 // If 'derive' is not NULL, then the 'seed' field is assumed to contain
1581 // a 'label' and 'context' that are unmarshaled into 'derive'.
1582 TPM_RC
1583 UnmarshalToPublic(TPMT_PUBLIC* tOut, // OUT: output
1584                  TPM2B_TEMPLATE* tIn, // IN:
1585                  BOOL derivation, // IN: indicates if this is for a derivation

```



```

1586         TPMS_DERIVE* labelContext // OUT: label and context if derivation
1587     )
1588     {
1589         BYTE* buffer = tIn->t.buffer;
1590         INT32 size = tIn->t.size;
1591         TPM_RC result;
1592         //
1593         // make sure that tOut is zeroed so that there are no remnants from previous
1594         // uses
1595         MemorySet(tOut, 0, sizeof(TPMT_PUBLIC));
1596         // Unmarshal the components of the TPMT_PUBLIC up to the unique field
1597         result = TPMI_ALG_PUBLIC_Unmarshal(&tOut->type, &buffer, &size);
1598         if(result != TPM_RC_SUCCESS)
1599             return result;
1600         result = TPMI_ALG_HASH_Unmarshal(&tOut->nameAlg, &buffer, &size, FALSE);
1601         if(result != TPM_RC_SUCCESS)
1602             return result;
1603         result = TPMA_OBJECT_Unmarshal(&tOut->objectAttributes, &buffer, &size);
1604         if(result != TPM_RC_SUCCESS)
1605             return result;
1606         result = TPM2B_DIGEST_Unmarshal(&tOut->authPolicy, &buffer, &size);
1607         if(result != TPM_RC_SUCCESS)
1608             return result;
1609         result =
1610             TPMU_PUBLIC_PARMS_Unmarshal(&tOut->parameters, &buffer, &size, tOut->type);
1611         if(result != TPM_RC_SUCCESS)
1612             return result;
1613         // Now unmarshal a TPMS_DERIVE if this is for derivation
1614         if(derivation)
1615             result = TPMS_DERIVE_Unmarshal(labelContext, &buffer, &size);
1616         else
1617             // otherwise, unmarshal a TPMU_PUBLIC_ID
1618             result = TPMU_PUBLIC_ID_Unmarshal(&tOut->unique, &buffer, &size, tOut->type);
1619         // Make sure the template was used up
1620         if((result == TPM_RC_SUCCESS) && (size != 0))
1621             result = TPM_RC_SIZE;
1622         return result;
1623     }
1624
1625     /*** ObjectSetExternal()
1626     // Set the external attributes for an object.
1627     void ObjectSetExternal(OBJECT* object)
1628     {
1629         object->attributes.external = SET;
1630     }

```

7.109 /tpm/src/command/Object/ReadPublic.c

```

1  #include "Tpm.h"
2  #include "ReadPublic_fp.h"
3
4  #if CC_ReadPublic // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // read public area of a loaded object
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_SEQUENCE can not read the public area of a sequence
11 //     object
12 TPM_RC
13 TPM2_ReadPublic(ReadPublic_In* in, // IN: input parameter list
14                 ReadPublic_Out* out // OUT: output parameter list
15 )
16 {
17     OBJECT* object = HandleToObject(in->objectHandle);

```

```

18
19 // Input Validation
20 // Can not read public area of a sequence object
21 if(ObjectIsSequence(object))
22     return TPM_RC_SEQUENCE;
23
24 // Command Output
25 out->outPublic.publicArea = object->publicArea;
26 out->name                 = object->name;
27 out->qualifiedName        = object->qualifiedName;
28
29 return TPM_RC_SUCCESS;
30 }
31
32 #endif // CC_ReadPublic

```

7.110 /tpm/src/command/Object/Unseal.c

```

1 #include "Tpm.h"
2 #include "Unseal_fp.h"
3
4 #if CC_Unseal // Conditional expansion of this file
5
6 /*(See part 3 specification)
7 // return data in a sealed data blob
8 */
9 // Return Type: TPM_RC
10 // TPM_RC_ATTRIBUTES 'itemHandle' has wrong attributes
11 // TPM_RC_TYPE 'itemHandle' is not a KEYEDHASH data object
12 TPM_RC
13 TPM2_Unseal(Unseal_In* in, Unseal_Out* out)
14 {
15     OBJECT* object;
16     // Input Validation
17     // Get pointer to loaded object
18     object = HandleToObject(in->itemHandle);
19
20     // Input handle must be a data object
21     if(object->publicArea.type != TPM_ALG_KEYEDHASH)
22         return TPM_RCS_TYPE + RC_Unseal_itemHandle;
23     if(IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, decrypt)
24         || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, sign)
25         || IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, restricted))
26         return TPM_RCS_ATTRIBUTES + RC_Unseal_itemHandle;
27     // Command Output
28     // Copy data
29     out->outData = object->sensitive.sensitive.bits;
30     return TPM_RC_SUCCESS;
31 }
32
33 #endif // CC_Unseal

```

7.111 /tpm/src/command/PCR/PCR_Allocate.c

```

1 #include "Tpm.h"
2 #include "PCR_Allocate_fp.h"
3
4 #if CC_PCR_Allocate // Conditional expansion of this file
5
6 /*(See part 3 specification)
7 // Allocate PCR banks
8 */
9 // Return Type: TPM_RC
10 // TPM_RC_PCR the allocation did not have required PCR

```

```

11 //      TPM_RC_NV_UNAVAILABLE    NV is not accessible
12 //      TPM_RC_NV_RATE          NV is in a rate-limiting mode
13 TPM_RC
14 TPM2_PCR_Allocate(PCR_Allocate_In* in, // IN: input parameter list
15                  PCR_Allocate_Out* out // OUT: output parameter list
16 )
17 {
18     TPM_RC result;
19
20     // The command needs NV update. Check if NV is available.
21     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
22     // this point.
23     // Note: These codes are not listed in the return values above because it is
24     // an implementation choice to check in this routine rather than in a common
25     // function that is called before these actions are called. These return values
26     // are described in the Response Code section of Part 3.
27     RETURN_IF_NV_IS_NOT_AVAILABLE;
28
29     // Command Output
30
31     // Call PCR Allocation function.
32     result = PCRAAllocate(
33         &in->pcrAllocation, &out->maxPCR, &out->sizeNeeded, &out->sizeAvailable);
34     if(result == TPM_RC_PCR)
35         return result;
36
37     //
38     out->allocationSuccess = (result == TPM_RC_SUCCESS);
39
40     // if re-configuration succeeds, set the flag to indicate PCR configuration is
41     // going to be changed in next boot
42     if(out->allocationSuccess == YES)
43         g_pcrReConfig = TRUE;
44
45     return TPM_RC_SUCCESS;
46 }
47
48 #endif // CC_PCR_Allocate

```

7.112 /tpm/src/command/PCR/PCR_Event.c

```

1 #include "Tpm.h"
2 #include "PCR_Event_fp.h"
3
4 #if CC_PCR_Event // Conditional expansion of this file
5
6 /*(See part 3 specification)
7 // Update PCR
8 */
9 // Return Type: TPM_RC
10 //      TPM_RC_LOCALITY    current command locality is not allowed to
11 //                          extend the PCR referenced by 'pcrHandle'
12 TPM_RC
13 TPM2_PCR_Event(PCR_Event_In* in, // IN: input parameter list
14               PCR_Event_Out* out // OUT: output parameter list
15 )
16 {
17     HASH_STATE hashState;
18     UINT32      i;
19     UINT16      size;
20
21     // Input Validation
22
23     // If a PCR extend is required
24     if(in->pcrHandle != TPM_RH_NULL)

```

```

25     {
26         // If the PCR is not allow to extend, return error
27         if(!PCRIsExtendAllowed(in->pcrHandle))
28             return TPM_RC_LOCALITY;
29
30         // If PCR is state saved and we need to update orderlyState, check NV
31         // availability
32         if(PCRIsStateSaved(in->pcrHandle))
33             RETURN_IF_ORDERLY;
34     }
35
36     // Internal Data Update
37
38     out->digests.count = HASH_COUNT;
39
40     // Iterate supported PCR bank algorithms to extend
41     for(i = 0; i < HASH_COUNT; i++)
42     {
43         TPM_ALG_ID hash = CryptHashGetAlgByIndex(i);
44         out->digests.digests[i].hashAlg = hash;
45         size = CryptHashStart(&hashState, hash);
46         CryptDigestUpdate2B(&hashState, &in->eventData.b);
47         CryptHashEnd(&hashState, size, (BYTE*)&out->digests.digests[i].digest);
48         if(in->pcrHandle != TPM_RH_NULL)
49             PCRExtend(
50                 in->pcrHandle, hash, size, (BYTE*)&out->digests.digests[i].digest);
51     }
52
53     return TPM_RC_SUCCESS;
54 }
55
56 #endif // CC_PCR_Event

```

7.113 /tpm/src/command/PCR/PCR_Extend.c

```

1  #include "Tpm.h"
2  #include "PCR_Extend_fp.h"
3
4  #if CC_PCR_Extend // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Update PCR
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_LOCALITY          current command locality is not allowed to
11 //                               extend the PCR referenced by 'pcrHandle'
12 TPM_RC
13 TPM2_PCR_Extend(PCR_Extend_In* in // IN: input parameter list
14 )
15 {
16     UINT32 i;
17
18     // Input Validation
19
20     // NOTE: This function assumes that the unmarshaling function for 'digests' will
21     // have validated that all of the indicated hash algorithms are valid. If the
22     // hash algorithms are correct, the unmarshaling code will unmarshal a digest
23     // of the size indicated by the hash algorithm. If the overall size is not
24     // consistent, the unmarshaling code will run out of input data or have input
25     // data left over. In either case, it will cause an unmarshaling error and this
26     // function will not be called.
27
28     // For NULL handle, do nothing and return success
29     if(in->pcrHandle == TPM_RH_NULL)
30         return TPM_RC_SUCCESS;

```

```

31
32 // Check if the extend operation is allowed by the current command locality
33 if(!PCRIsExtendAllowed(in->pcrHandle))
34     return TPM_RC_LOCALITY;
35
36 // If PCR is state saved and we need to update orderlyState, check NV
37 // availability
38 if(PCRIsStateSaved(in->pcrHandle))
39     RETURN_IF_ORDERLY;
40
41 // Internal Data Update
42
43 // Iterate input digest list to extend
44 for(i = 0; i < in->digests.count; i++)
45 {
46     PCRExtend(in->pcrHandle,
47               in->digests.digests[i].hashAlg,
48               CryptHashGetDigestSize(in->digests.digests[i].hashAlg),
49               (BYTE*)&in->digests.digests[i].digest);
50 }
51
52 return TPM_RC_SUCCESS;
53 }
54
55 #endif // CC_PCR_Extend

```

7.114 /tpm/src/command/PCR/PCR_Read.c

```

1  #include "Tpm.h"
2  #include "PCR_Read_fp.h"
3
4  #if CC_PCR_Read // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Read a set of PCR
8  */
9  TPM_RC
10 TPM2_PCR_Read(PCR_Read_In* in, // IN: input parameter list
11               PCR_Read_Out* out // OUT: output parameter list
12 )
13 {
14     // Command Output
15
16     // Call PCR read function. input pcrSelectionIn parameter could be changed
17     // to reflect the actual PCR being returned
18     PCRRead(&in->pcrSelectionIn, &out->pcrValues, &out->pcrUpdateCounter);
19
20     out->pcrSelectionOut = in->pcrSelectionIn;
21
22     return TPM_RC_SUCCESS;
23 }
24
25 #endif // CC_PCR_Read

```

7.115 /tpm/src/command/PCR/PCR_Reset.c

```

1  #include "Tpm.h"
2  #include "PCR_Reset_fp.h"
3
4  #if CC_PCR_Reset // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Reset PCR
8  */

```

```

9  // Return Type: TPM_RC
10 //     TPM_RC_LOCALITY          current command locality is not allowed to
11 //                               reset the PCR referenced by 'pcrHandle'
12 TPM_RC
13 TPM2_PCR_Reset(PCR_Reset_In* in // IN: input parameter list
14 )
15 {
16     // Input Validation
17
18     // Check if the reset operation is allowed by the current command locality
19     if(!PCRIsResetAllowed(in->pcrHandle))
20         return TPM_RC_LOCALITY;
21
22     // If PCR is state saved and we need to update orderlyState, check NV
23     // availability
24     if(PCRIsStateSaved(in->pcrHandle))
25         RETURN_IF_ORDERLY;
26
27     // Internal Data Update
28
29     // Reset selected PCR in all banks to 0
30     PCRSetValue(in->pcrHandle, 0);
31
32     // Indicate that the PCR changed so that pcrCounter will be incremented if
33     // necessary.
34     PCRChanged(in->pcrHandle);
35
36     return TPM_RC_SUCCESS;
37 }
38
39 #endif // CC_PCR_Reset

```

7.116 /tpm/src/command/PCR/PCR_SetAuthPolicy.c

```

1  #include "Tpm.h"
2  #include "PCR_SetAuthPolicy_fp.h"
3
4  #if CC_PCR_SetAuthPolicy // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Set authPolicy to a group of PCR
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_SIZE          size of 'authPolicy' is not the size of a digest
11 //                               produced by 'policyDigest'
12 //     TPM_RC_VALUE        PCR referenced by 'pcrNum' is not a member
13 //                               of a PCR policy group
14 TPM_RC
15 TPM2_PCR_SetAuthPolicy(PCR_SetAuthPolicy_In* in // IN: input parameter list
16 )
17 {
18     UINT32 groupIndex;
19
20     // The command needs NV update. Check if NV is available.
21     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
22     // this point
23     RETURN_IF_NV_IS_NOT_AVAILABLE;
24
25     // Input Validation:
26
27     // Check the authPolicy consistent with hash algorithm
28     if(in->authPolicy.t.size != CryptHashGetDigestSize(in->hashAlg))
29         return TPM_RCS_SIZE + RC_PCR_SetAuthPolicy_authPolicy;
30
31     // If PCR does not belong to a policy group, return TPM_RC_VALUE

```



```

32     if(!PCRBelongsPolicyGroup(in->pcrNum, &groupIndex))
33         return TPM_RC_SUCCESS + RC_PCR_SetAuthPolicy_pcrNum;
34
35     // Internal Data Update
36
37     // Set PCR policy
38     gp.pcrPolicies.hashAlg[groupIndex] = in->hashAlg;
39     gp.pcrPolicies.policy[groupIndex] = in->authPolicy;
40
41     // Save new policy to NV
42     NV_SYNC_PERSISTENT(pcrPolicies);
43
44     return TPM_RC_SUCCESS;
45 }
46
47 #endif // CC_PCR_SetAuthPolicy

```

7.117 /tpm/src/command/PCR/PCR_SetAuthValue.c

```

1  #include "Tpm.h"
2  #include "PCR_SetAuthValue_fp.h"
3
4  #if CC_PCR_SetAuthValue // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Set authValue to a group of PCR
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_VALUE
11 // PCR referenced by 'pcrHandle' is not a member
12 // of a PCR authorization group
13 TPM_RC
14 TPM2_PCR_SetAuthValue(PCR_SetAuthValue_In* in // IN: input parameter list
15 )
16 {
17     UINT32 groupIndex;
18     // Input Validation:
19
20     // If PCR does not belong to an auth group, return TPM_RC_VALUE
21     if(!PCRBelongsAuthGroup(in->pcrHandle, &groupIndex))
22         return TPM_RC_VALUE;
23
24     // The command may cause the orderlyState to be cleared due to the update of
25     // state clear data. If this is the case, Check if NV is available.
26     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
27     // this point
28     RETURN_IF_ORDERLY;
29
30     // Internal Data Update
31
32     // Set PCR authValue
33     MemoryRemoveTrailingZeros(&in->auth);
34     gc.pcrAuthValues.auth[groupIndex] = in->auth;
35
36     return TPM_RC_SUCCESS;
37 }
38 #endif // CC_PCR_SetAuthValue

```

7.118 /tpm/src/command/Random/GetRandom.c

```

1  #include "Tpm.h"
2  #include "GetRandom_fp.h"
3
4  #if CC_GetRandom // Conditional expansion of this file

```

```

5
6  /*(See part 3 specification)
7  // random number generator
8  */
9  TPM_RC
10 TPM2_GetRandom(GetRandom_In* in, // IN: input parameter list
11               GetRandom_Out* out // OUT: output parameter list
12 )
13 {
14     // Command Output
15
16     // if the requested bytes exceed the output buffer size, generates the
17     // maximum bytes that the output buffer allows
18     if(in->bytesRequested > sizeof(TPMU_HA))
19         out->randomBytes.t.size = sizeof(TPMU_HA);
20     else
21         out->randomBytes.t.size = in->bytesRequested;
22
23     CryptRandomGenerate(out->randomBytes.t.size, out->randomBytes.t.buffer);
24
25     return TPM_RC_SUCCESS;
26 }
27
28 #endif // CC_GetRandom

```

7.119 /tpm/src/command/Random/StirRandom.c

```

1  #include "Tpm.h"
2  #include "StirRandom_fp.h"
3
4  #if CC_StirRandom // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // add entropy to the RNG state
8  */
9  TPM_RC
10 TPM2_StirRandom(StirRandom_In* in // IN: input parameter list
11 )
12 {
13     // Internal Data Update
14     CryptRandomStir(in->inData.t.size, in->inData.t.buffer);
15
16     return TPM_RC_SUCCESS;
17 }
18
19 #endif // CC_StirRandom

```

7.120 /tpm/src/command/Session/PolicyRestart.c

```

1  #include "Tpm.h"
2  #include "PolicyRestart_fp.h"
3
4  #if CC_PolicyRestart // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Restore a policy session to its initial state
8  */
9  TPM_RC
10 TPM2_PolicyRestart(PolicyRestart_In* in // IN: input parameter list
11 )
12 {
13     // Initialize policy session data
14     SessionResetPolicyData(SessionGet(in->sessionHandle));
15

```

```

16     return TPM_RC_SUCCESS;
17 }
18
19 #endif // CC_PolicyRestart

```

7.121 /tpm/src/command/Session/StartAuthSession.c

```

1  #include "Tpm.h"
2  #include "StartAuthSession_fp.h"
3
4  #if CC_StartAuthSession // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Start an authorization session
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES      'tpmKey' does not reference a decrypt key
11 //     TPM_RC_CONTEXT_GAP     the difference between the most recently created
12 //                             active context and the oldest active context is at
13 //                             the limits of the TPM
14 //     TPM_RC_HANDLE          input decrypt key handle only has public portion
15 //                             loaded
16 //     TPM_RC_MODE             'symmetric' specifies a block cipher but the mode
17 //                             is not TPM_ALG_CFB.
18 //     TPM_RC_SESSION_HANDLES no session handle is available
19 //     TPM_RC_SESSION_MEMORY  no more slots for loading a session
20 //     TPM_RC_SIZE             nonce less than 16 octets or greater than the size
21 //                             of the digest produced by 'authHash'
22 //     TPM_RC_VALUE            secret size does not match decrypt key type; or the
23 //                             recovered secret is larger than the digest size of
24 //                             the nameAlg of 'tpmKey'; or, for an RSA decrypt key,
25 //                             if 'encryptedSecret' is greater than the
26 //                             public modulus of 'tpmKey'.
27 TPM_RC
28 TPM2_StartAuthSession(StartAuthSession_In* in, // IN: input parameter buffer
29                      StartAuthSession_Out* out // OUT: output parameter buffer
30 )
31 {
32     TPM_RC result = TPM_RC_SUCCESS;
33     OBJECT* tpmKey; // TPM key for decrypt salt
34     TPM2B_DATA salt;
35
36     // Input Validation
37
38     // Check input nonce size. IT should be at least 16 bytes but not larger
39     // than the digest size of session hash.
40     if(in->nonceCaller.t.size < 16
41        || in->nonceCaller.t.size > CryptHashGetDigestSize(in->authHash))
42         return TPM_RCS_SIZE + RC_StartAuthSession_nonceCaller;
43
44     // If an decrypt key is passed in, check its validation
45     if(in->tpmKey != TPM_RH_NULL)
46     {
47         // Get pointer to loaded decrypt key
48         tpmKey = HandleToObject(in->tpmKey);
49
50         // key must be asymmetric with its sensitive area loaded. Since this
51         // command does not require authorization, the presence of the sensitive
52         // area was not already checked as it is with most other commands that
53         // use the sensitive area so check it here
54         if(!CryptIsAsymAlgorithm(tpmKey->publicArea.type))
55             return TPM_RCS_KEY + RC_StartAuthSession_tpmKey;
56         // secret size cannot be 0
57         if(in->encryptedSalt.t.size == 0)
58             return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;

```

```

59     // Decrypting salt requires accessing the private portion of a key.
60     // Therefore, tmpKey can not be a key with only public portion loaded
61     if(tpmKey->attributes.publicOnly)
62         return TPM_RCS_HANDLE + RC_StartAuthSession_tpmKey;
63     // HMAC session input handle check.
64     // tpmKey should be a decryption key
65     if(!IS_ATTRIBUTE(tpmKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
66         return TPM_RCS_ATTRIBUTES + RC_StartAuthSession_tpmKey;
67     // Secret Decryption. A TPM_RC_VALUE, TPM_RC_KEY or Unmarshal errors
68     // may be returned at this point
69     result = CryptSecretDecrypt(
70         tpmKey, &in->nonceCaller, SECRET_KEY, &in->encryptedSalt, &salt);
71     if(result != TPM_RC_SUCCESS)
72         return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
73 }
74 else
75 {
76     // secret size must be 0
77     if(in->encryptedSalt.t.size != 0)
78         return TPM_RCS_VALUE + RC_StartAuthSession_encryptedSalt;
79     salt.t.size = 0;
80 }
81 switch(HandleGetType(in->bind))
82 {
83     case TPM_HT_TRANSIENT:
84     {
85         OBJECT* object = HandleToObject(in->bind);
86         // If the bind handle references a transient object, make sure that we
87         // can get to the authorization value. Also, make sure that the object
88         // has a proper Name (nameAlg != TPM_ALG_NULL). If it doesn't, then
89         // it might be possible to bind to an object where the authValue is
90         // known. This does not create a real issue in that, if you know the
91         // authorization value, you can actually bind to the object. However,
92         // there is a potential
93         if(object->attributes.publicOnly == SET)
94             return TPM_RCS_HANDLE + RC_StartAuthSession_bind;
95         break;
96     }
97     case TPM_HT_NV_INDEX:
98         // a PIN index can't be a bind object
99         {
100             NV_INDEX* nvIndex = NvGetIndexInfo(in->bind, NULL);
101             if(IsNvPinPassIndex(nvIndex->publicArea.attributes)
102                 || IsNvPinFailIndex(nvIndex->publicArea.attributes))
103                 return TPM_RCS_HANDLE + RC_StartAuthSession_bind;
104             break;
105         }
106     default:
107         break;
108 }
109 // If 'symmetric' is a symmetric block cipher (not TPM_ALG_NULL or TPM_ALG_XOR)
110 // then the mode must be CFB.
111 if(in->symmetric.algorithm != TPM_ALG_NULL
112     && in->symmetric.algorithm != TPM_ALG_XOR
113     && in->symmetric.mode.sym != TPM_ALG_CFB)
114     return TPM_RCS_MODE + RC_StartAuthSession_symmetric;
115
116 // Internal Data Update and command output
117
118 // Create internal session structure. TPM_RC_CONTEXT_GAP, TPM_RC_NO_HANDLES
119 // or TPM_RC_SESSION_MEMORY errors may be returned at this point.
120 //
121 // The detailed actions for creating the session context are not shown here
122 // as the details are implementation dependent
123 // SessionCreate sets the output handle and nonceTPM
124 result = SessionCreate(in->sessionType,

```

```

125         in->authHash,
126         &in->nonceCaller,
127         &in->symmetric,
128         in->bind,
129         &salt,
130         &out->sessionHandle,
131         &out->nonceTPM);
132     return result;
133 }
134
135 #endif // CC_StartAuthSession

```

7.122 /tpm/src/command/Signature/Sign.c

```

1  #include "Tpm.h"
2  #include "Sign_fp.h"
3
4  #if CC_Sign // Conditional expansion of this file
5
6  # include "Attest_spt_fp.h"
7
8  /*(See part 3 specification)
9  // sign an externally provided hash using an asymmetric signing key
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_BINDING           The public and private portions of the key are not
13 //                               properly bound.
14 //     TPM_RC_KEY               'signHandle' does not reference a signing key;
15 //     TPM_RC_SCHEME             the scheme is not compatible with sign key type,
16 //                               or input scheme is not compatible with default
17 //                               scheme, or the chosen scheme is not a valid
18 //                               sign scheme
19 //     TPM_RC_TICKET            'validation' is not a valid ticket
20 //     TPM_RC_VALUE             the value to sign is larger than allowed for the
21 //                               type of 'keyHandle'
22
23 TPM_RC
24 TPM2_Sign(Sign_In* in, // IN: input parameter list
25          Sign_Out* out // OUT: output parameter list
26 )
27 {
28     TPM_RC result;
29     TPMT_TK_HASHCHECK ticket;
30     OBJECT* signObject = HandleToObject(in->keyHandle);
31     //
32     // Input Validation
33     if(!IsSigningObject(signObject))
34         return TPM_RCS_KEY + RC_Sign_keyHandle;
35
36     // A key that will be used for x.509 signatures can't be used in TPM2_Sign().
37     if(IS_ATTRIBUTE(signObject->publicArea.objectAttributes, TPMA_OBJECT, x509sign))
38         return TPM_RCS_ATTRIBUTES + RC_Sign_keyHandle;
39
40     // pick a scheme for sign. If the input sign scheme is not compatible with
41     // the default scheme, return an error.
42     if(!CryptSelectSignScheme(signObject, &in->inScheme))
43         return TPM_RCS_SCHEME + RC_Sign_inScheme;
44
45     // If validation is provided, or the key is restricted, check the ticket
46     if(in->validation.digest.t.size != 0
47        || IS_ATTRIBUTE(
48            signObject->publicArea.objectAttributes, TPMA_OBJECT, restricted))
49     {
50         // Compute and compare ticket
51         result = TicketComputeHashCheck(in->validation.hierarchy,

```

```

52         in->inScheme.details.any.hashAlg,
53         &in->digest,
54         &ticket);
55     if(result != TPM_RC_SUCCESS)
56         return result;
57
58     if(!MemoryEqual2B(&in->validation.digest.b, &ticket.digest.b))
59         return TPM_RCS_TICKET + RC_Sign_validation;
60 }
61 else
62     // If we don't have a ticket, at least verify that the provided 'digest'
63     // is the size of the scheme hashAlg digest.
64     // NOTE: this does not guarantee that the 'digest' is actually produced using
65     // the indicated hash algorithm, but at least it might be.
66     {
67         if(in->digest.t.size
68             != CryptHashGetDigestSize(in->inScheme.details.any.hashAlg))
69             return TPM_RCS_SIZE + RC_Sign_digest;
70     }
71
72     // Command Output
73     // Sign the hash. A TPM_RC_VALUE or TPM_RC_SCHEME
74     // error may be returned at this point
75     result = CryptSign(signObject, &in->inScheme, &in->digest, &out->signature);
76
77     return result;
78 }
79
80 #endif // CC_Sign

```

7.123 /tpm/src/command/Signature/VerifySignature.c

```

1  #include "Tpm.h"
2  #include "VerifySignature_fp.h"
3
4  #if CC_VerifySignature // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // This command uses loaded key to validate an asymmetric signature on a message
8  // with the message digest passed to the TPM.
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_ATTRIBUTES          'keyHandle' does not reference a signing key
12 //     TPM_RC_SIGNATURE           signature is not genuine
13 //     TPM_RC_SCHEME              CryptValidateSignature()
14 //     TPM_RC_HANDLE              the input handle is references an HMAC key but
15 //                                the private portion is not loaded
16 TPM_RC
17 TPM2_VerifySignature(VerifySignature_In* in, // IN: input parameter list
18                     VerifySignature_Out* out // OUT: output parameter list
19 )
20 {
21     TPM_RC      result;
22     OBJECT*     signObject = HandleToObject(in->keyHandle);
23     TPMI_RH_HIERARCHY hierarchy;
24
25     // Input Validation
26     // The object to validate the signature must be a signing key.
27     if(!IS_ATTRIBUTE(signObject->publicArea.objectAttributes, TPMA_OBJECT, sign))
28         return TPM_RCS_ATTRIBUTES + RC_VerifySignature_keyHandle;
29
30     // Validate Signature. TPM_RC_SCHEME, TPM_RC_HANDLE or TPM_RC_SIGNATURE
31     // error may be returned by CryptCVerifySignature()
32     result = CryptValidateSignature(in->keyHandle, &in->digest, &in->signature);
33     if(result != TPM_RC_SUCCESS)

```



```

34         return RcSafeAddToResult(result, RC_VerifySignature_signature);
35
36     // Command Output
37
38     hierarchy = GetHierarchy(in->keyHandle);
39     if(hierarchy == TPM_RH_NULL || signObject->publicArea.nameAlg == TPM_ALG_NULL)
40     {
41         // produce empty ticket if hierarchy is TPM_RH_NULL or nameAlg is
42         // TPM_ALG_NULL
43         out->validation.tag = TPM_ST_VERIFIED;
44         out->validation.hierarchy = TPM_RH_NULL;
45         out->validation.digest.t.size = 0;
46     }
47     else
48     {
49         // Compute ticket
50         result = TicketComputeVerified(
51             hierarchy, &in->digest, &signObject->name, &out->validation);
52         if(result != TPM_RC_SUCCESS)
53             return result;
54     }
55
56     return TPM_RC_SUCCESS;
57 }
58
59 #endif // CC_VerifySignature

```

7.124 /tpm/src/command/Startup/Shutdown.c

```

1  #include "Tpm.h"
2  #include "Shutdown_fp.h"
3
4  #if CC_Shutdown // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Shut down TPM for power off
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_TYPE if PCR bank has been re-configured, a
11 // Shutdown(CLEAR) is required
12 TPM_RC
13 TPM2_Shutdown(Shutdown_In* in // IN: input parameter list
14 )
15 {
16     // The command needs NV update. Check if NV is available.
17     // A TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE error may be returned at
18     // this point
19     RETURN_IF_NV_IS_NOT_AVAILABLE;
20
21     // Input Validation
22     // If PCR bank has been reconfigured, a CLEAR state save is required
23     if(g_pcrReConfig && in->shutdownType == TPM_SU_STATE)
24         return TPM_RCS_TYPE + RC_Shutdown_shutdownType;
25     // Internal Data Update
26     gp.orderlyState = in->shutdownType;
27
28     # if USE_DA_USED
29     // CLEAR g_daUsed so that any future DA-protected access will cause the
30     // shutdown to become non-orderly. It is not sufficient to invalidate the
31     // shutdown state after a DA failure because an attacker can inhibit access
32     // to NV and use the fact that an update of failedTries was attempted as an
33     // indication of an authorization failure. By making sure that the orderly state
34     // is CLEAR before any DA attempt, this prevents the possibility of this 'attack.'
35     g_daUsed = FALSE;
36     # endif

```

```

37
38     // PCR private data state save
39     PCRStateSave(in->shutdownType);
40
41 # if ACT_SUPPORT
42     // Save the ACT state
43     ActShutdown(in->shutdownType);
44 # endif
45
46     // Save RAM backed NV index data
47     NvUpdateIndexOrderlyData();
48
49 # if ACCUMULATE_SELF_HEAL_TIMER
50     // Save the current time value
51     go.time = g_time;
52 # endif
53
54     // Save all orderly data
55     NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
56
57     if(in->shutdownType == TPM_SU_STATE)
58     {
59         // Save STATE_RESET and STATE_CLEAR data
60         NvWrite(NV_STATE_CLEAR_DATA, sizeof(STATE_CLEAR_DATA), &gc);
61         NvWrite(NV_STATE_RESET_DATA, sizeof(STATE_RESET_DATA), &gr);
62
63         // Save the startup flags for resume
64         if(g_DrtmPreStartup)
65             gp.orderlyState = TPM_SU_STATE | PRE_STARTUP_FLAG;
66         else if(g_StartupLocality3)
67             gp.orderlyState = TPM_SU_STATE | STARTUP_LOCALITY_3;
68     }
69     // only two shutdown options.
70     else if(in->shutdownType != TPM_SU_CLEAR)
71         return TPM_RCS_VALUE + RC_Shutdown_shutdownType;
72
73     NV_SYNC_PERSISTENT(orderlyState);
74
75     return TPM_RC_SUCCESS;
76 }
77 #endif // CC_Shutdown

```

7.125 /tpm/src/command/Startup/Startup.c

```

1  #include "Tpm.h"
2  #include "Startup_fp.h"
3
4  #if CC_Startup // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Initialize TPM because a system-wide reset
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_LOCALITY      a Startup(STATE) does not have the same H-CRTM
11 //                          state as the previous Startup() or the locality
12 //                          of the startup is not 0 or 3
13 //     TPM_RC_NV_UNINITIALIZED the saved state cannot be recovered and a
14 //                          Startup(CLEAR) is required.
15 //     TPM_RC_VALUE          'startup' type is not compatible with previous
16 //                          shutdown sequence
17
18 TPM_RC
19 TPM2_Startup(Startup_In* in // IN: input parameter list
20 )
21 {

```

```

22     STARTUP_TYPE startup;
23     BYTE          locality = _plat_LocalityGet();
24     BOOL          OK       = TRUE;
25     //
26     // The command needs NV update.
27     RETURN_IF_NV_IS_NOT_AVAILABLE;
28
29     // Get the flags for the current startup locality and the H-CRTM.
30     // Rather than generalizing the locality setting, this code takes advantage
31     // of the fact that the PC Client specification only allows Startup()
32     // from locality 0 and 3. To generalize this probably would require a
33     // redo of the NV space and since this is a feature that is hardly ever used
34     // outside of the PC Client, this code just support the PC Client needs.
35
36     // Input Validation
37     // Check that the locality is a supported value
38     if(locality != 0 && locality != 3)
39         return TPM_RC_LOCALITY;
40     // If there was a H-CRTM, then treat the locality as being 3
41     // regardless of what the Startup() was. This is done to preserve the
42     // H-CRTM PCR so that they don't get overwritten with the normal
43     // PCR startup initialization. This basically means that g_StartupLocality3
44     // and g_DrtmPreStartup can't both be SET at the same time.
45     if(g_DrtmPreStartup)
46         locality = 0;
47     g_StartupLocality3 = (locality == 3);
48
49 # if USE_DA_USED
50     // If there was no orderly shutdown, then there might have been a write to
51     // failedTries that didn't get recorded but only if g_daUsed was SET in the
52     // shutdown state
53     g_daUsed = (gp.orderlyState == SU_DA_USED_VALUE);
54     if(g_daUsed)
55         gp.orderlyState = SU_NONE_VALUE;
56 # endif
57
58     g_prevOrderlyState = gp.orderlyState;
59
60     // If there was a proper shutdown, then the startup modifiers are in the
61     // orderlyState. Turn them off in the copy.
62     if(IS_ORDERLY(g_prevOrderlyState))
63         g_prevOrderlyState &= ~(PRE_STARTUP_FLAG | STARTUP_LOCALITY_3);
64     // If this is a Resume,
65     if(in->startupType == TPM_SU_STATE)
66     {
67         // then there must have been a prior TPM2_ShutdownState(STATE)
68         if(g_prevOrderlyState != TPM_SU_STATE)
69             return TPM_RCS_VALUE + RC_Startup_startupType;
70         // and the part of NV used for state save must have been recovered
71         // correctly.
72         // NOTE: if this fails, then the caller will need to do Startup(CLEAR). The
73         // code for Startup(Clear) cannot fail if the NV can't be read correctly
74         // because that would prevent the TPM from ever getting unstuck.
75         if(g_nvOk == FALSE)
76             return TPM_RC_NV_UNINITIALIZED;
77         // For Resume, the H-CRTM has to be the same as the previous boot
78         if(g_DrtmPreStartup != ((gp.orderlyState & PRE_STARTUP_FLAG) != 0))
79             return TPM_RCS_VALUE + RC_Startup_startupType;
80         if(g_StartupLocality3 != ((gp.orderlyState & STARTUP_LOCALITY_3) != 0))
81             return TPM_RC_LOCALITY;
82     }
83     // Clean up the gp state
84     gp.orderlyState = g_prevOrderlyState;
85
86     // Internal Date Update
87     if((gp.orderlyState == TPM_SU_STATE) && (g_nvOk == TRUE))

```

```

88     {
89         // Always read the data that is only cleared on a Reset because this is not
90         // a reset
91         NvRead(&gr, NV_STATE_RESET_DATA, sizeof(gr));
92         if(in->startupType == TPM_SU_STATE)
93         {
94             // If this is a startup STATE (a Resume) need to read the data
95             // that is cleared on a startup CLEAR because this is not a Reset
96             // or Restart.
97             NvRead(&gc, NV_STATE_CLEAR_DATA, sizeof(gc));
98             startup = SU_RESUME;
99         }
100         else
101             startup = SU_RESTART;
102     }
103     else
104         // Will do a TPM reset if Shutdown(CLEAR) and Startup(CLEAR) or no shutdown
105         // or there was a failure reading the NV data.
106         startup = SU_RESET;
107     // Startup for cryptographic library. Don't do this until after the orderly
108     // state has been read in from NV.
109     OK = OK && CryptStartup(startup);
110
111     // When the cryptographic library has been started, indicate that a TPM2_Startup
112     // command has been received.
113     OK = OK && TPMRegisterStartup();
114
115     # if VENDOR_PERMANENT_AUTH_ENABLED == YES
116     // Read the platform unique value that is used as VENDOR_PERMANENT_AUTH_HANDLE
117     // authorization value
118     g_platformUniqueAuth.t.size = (UINT16) plat_GetUniqueAuth(
119         1, sizeof(g_platformUniqueAuth.t.buffer), g_platformUniqueAuth.t.buffer);
120     # endif
121
122     // Start up subsystems
123     // Start set the safe flag
124     OK = OK && TimeStartup(startup);
125
126     // Start dictionary attack subsystem
127     OK = OK && DASStartup(startup);
128
129     // Enable hierarchies
130     OK = OK && HierarchyStartup(startup);
131
132     // Restore/Initialize PCR
133     OK = OK && PCRStartup(startup, locality);
134
135     // Restore/Initialize command audit information
136     OK = OK && CommandAuditStartup(startup);
137
138     // Restore the ACT
139     # if ACT_SUPPORT
140     OK = OK && ActStartup(startup);
141     # endif
142
143     // The following code was moved from Time.c where it made no sense
144     if(OK)
145     {
146         switch(startup)
147         {
148             case SU_RESUME:
149                 // Resume sequence
150                 gr.restartCount++;
151                 break;
152             case SU_RESTART:
153                 // Hibernate sequence

```

```

154         gr.clearCount++;
155         gr.restartCount++;
156         break;
157     case SU_RESET:
158     default:
159         // Reset object context ID to 0
160         gr.objectContextID = 0;
161         // Reset clearCount to 0
162         gr.clearCount = 0;
163
164         // Reset sequence
165         // Increase resetCount
166         gp.resetCount++;
167
168         // Write resetCount to NV
169         NV_SYNC_PERSISTENT(resetCount);
170
171         gp.totalResetCount++;
172         // We do not expect the total reset counter overflow during the life
173         // time of TPM. if it ever happens, TPM will be put to failure mode
174         // and there is no way to recover it.
175         // The reason that there is no recovery is that we don't increment
176         // the NV totalResetCount when incrementing would make it 0. When the
177         // TPM starts up again, the old value of totalResetCount will be read
178         // and we will get right back to here with the increment failing.
179         if(gp.totalResetCount == 0)
180             FAIL(FATAL_ERROR_INTERNAL);
181
182         // Write total reset counter to NV
183         NV_SYNC_PERSISTENT(totalResetCount);
184
185         // Reset restartCount
186         gr.restartCount = 0;
187
188         break;
189     }
190 }
191 // Initialize session table
192 OK = OK && SessionStartup(startup);
193
194 // Initialize object table
195 OK = OK && ObjectStartup();
196
197 // Initialize index/evict data. This function clears read/write locks
198 // in NV index
199 OK = OK && NvEntityStartup(startup);
200
201 // Initialize the orderly shut down flag for this cycle to SU_NONE_VALUE.
202 gp.orderlyState = SU_NONE_VALUE;
203
204 OK = OK && NV_SYNC_PERSISTENT(orderlyState);
205
206 // This can be reset after the first completion of a TPM2_Startup() after
207 // a power loss. It can probably be reset earlier but this is an OK place.
208 if(OK)
209     g_powerWasLost = FALSE;
210
211 return (OK) ? TPM_RC_SUCCESS : TPM_RC_FAILURE;
212 }
213
214 #endif // CC_Startup

```

7.126 /tpm/src/command/Symmetric/EncryptDecrypt.c

```
1 #include "Tpm.h"
```

```

2  #include "EncryptDecrypt_fp.h"
3  #if CC_EncryptDecrypt2
4  # include "EncryptDecrypt_spt_fp.h"
5  #endif
6
7  #if CC_EncryptDecrypt // Conditional expansion of this file
8
9  /*(See part 3 specification)
10 // symmetric encryption or decryption
11 */
12 // Return Type: TPM_RC
13 //     TPM_RC_KEY           is not a symmetric decryption key with both
14 //                           public and private portions loaded
15 //     TPM_RC_SIZE          'IvIn' size is incompatible with the block cipher mode;
16 //                           or 'inData' size is not an even multiple of the block
17 //                           size for CBC or ECB mode
18 //     TPM_RC_VALUE         'keyHandle' is restricted and the argument 'mode' does
19 //                           not match the key's mode
20 TPM_RC
21 TPM2_EncryptDecrypt(EncryptDecrypt_In* in, // IN: input parameter list
22                    EncryptDecrypt_Out* out // OUT: output parameter list
23 )
24 {
25     # if CC_EncryptDecrypt2
26         return EncryptDecryptShared(
27             in->keyHandle, in->decrypt, in->mode, &in->ivIn, &in->inData, out);
28     # else
29         OBJECT*      symKey;
30         UINT16        keySize;
31         UINT16        blockSize;
32         BYTE*         key;
33         TPM_ALG_ID    alg;
34         TPM_ALG_ID    mode;
35         TPM_RC        result;
36         BOOL          OK;
37         TPMA_OBJECT    attributes;
38
39         // Input Validation
40         symKey      = HandleToObject(in->keyHandle);
41         mode        = symKey->publicArea.parameters.symDetail.sym.mode.sym;
42         attributes  = symKey->publicArea.objectAttributes;
43
44         // The input key should be a symmetric key
45         if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
46             return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
47         // The key must be unrestricted and allow the selected operation
48         OK      = IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted) if (YES == in->decrypt)
49         OK      = OK && IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt);
50         else OK = OK && IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign);
51         if(!OK)
52             return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;
53
54         // If the key mode is not TPM_ALG_NULL...
55         // or TPM_ALG_NULL
56         if(mode != TPM_ALG_NULL)
57         {
58             // then the input mode has to be TPM_ALG_NULL or the same as the key
59             if((in->mode != TPM_ALG_NULL) && (in->mode != mode))
60                 return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
61         }
62         else
63         {
64             // if the key mode is null, then the input can't be null
65             if(in->mode == TPM_ALG_NULL)
66                 return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
67             mode = in->mode;

```



```

68     }
69     // The input iv for ECB mode should be an Empty Buffer. All the other modes
70     // should have an iv size same as encryption block size
71     keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
72     alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
73     blockSize = CryptGetSymmetricBlockSize(alg, keySize);
74
75     // reverify the algorithm. This is mainly to keep static analysis tools happy
76     if(blockSize == 0)
77         return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
78
79     // Note: When an algorithm is not supported by a TPM, the TPM_ALG_XXX for that
80     // algorithm is not defined. However, it is assumed that the TPM_ALG_XXX for
81     // the algorithm is always defined. Both have the same numeric value.
82     // TPM_ALG_XXX is used here so that the code does not get cluttered with
83     // #ifdef's. Having this check does not mean that the algorithm is supported.
84     // If it was not supported the unmarshaling code would have rejected it before
85     // this function were called. This means that, depending on the implementation,
86     // the check could be redundant but it doesn't hurt.
87     if(((mode == TPM_ALG_ECB) && (in->ivIn.t.size != 0))
88        || ((mode != TPM_ALG_ECB) && (in->ivIn.t.size != blockSize)))
89         return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;
90
91     // The input data size of CBC mode or ECB mode must be an even multiple of
92     // the symmetric algorithm's block size
93     if(((mode == TPM_ALG_CBC) || (mode == TPM_ALG_ECB))
94        && ((in->inData.t.size % blockSize) != 0))
95         return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;
96
97     // Copy IV
98     // Note: This is copied here so that the calls to the encrypt/decrypt functions
99     // will modify the output buffer, not the input buffer
100    out->ivOut = in->ivIn;
101
102    // Command Output
103    key = symKey->sensitive.sensitive.sym.t.buffer;
104    // For symmetric encryption, the cipher data size is the same as plain data
105    // size.
106    out->outData.t.size = in->inData.t.size;
107    if(in->decrypt == YES)
108    {
109        // Decrypt data to output
110        result = CryptSymmetricDecrypt(out->outData.t.buffer,
111                                     alg,
112                                     keySize,
113                                     key,
114                                     &(out->ivOut),
115                                     mode,
116                                     in->inData.t.size,
117                                     in->inData.t.buffer);
118    }
119    else
120    {
121        // Encrypt data to output
122        result = CryptSymmetricEncrypt(out->outData.t.buffer,
123                                     alg,
124                                     keySize,
125                                     key,
126                                     &(out->ivOut),
127                                     mode,
128                                     in->inData.t.size,
129                                     in->inData.t.buffer);
130    }
131    return result;
132 # endif // CC_EncryptDecrypt2
133 }

```

```

134
135 #endif // CC_EncryptDecrypt

```

7.127 /tpm/src/command/Symmetric/EncryptDecrypt2.c

```

1  #include "Tpm.h"
2  #include "EncryptDecrypt2_fp.h"
3  #include "EncryptDecrypt_fp.h"
4  #include "EncryptDecrypt_spt_fp.h"
5
6  #if CC_EncryptDecrypt2 // Conditional expansion of this file
7
8  /*(See part 3 specification)
9  // symmetric encryption or decryption using modified parameter list
10 */
11 // Return Type: TPM_RC
12 //     TPM_RC_KEY           is not a symmetric decryption key with both
13 //                           public and private portions loaded
14 //     TPM_RC_SIZE          'ivIn' size is incompatible with the block cipher mode;
15 //                           or 'inData' size is not an even multiple of the block
16 //                           size for CBC or ECB mode
17 //     TPM_RC_VALUE         'keyHandle' is restricted and the argument 'mode' does
18 //                           not match the key's mode
19 TPM_RC
20 TPM2_EncryptDecrypt2(EncryptDecrypt2_In* in, // IN: input parameter list
21                     EncryptDecrypt2_Out* out // OUT: output parameter list
22 )
23 {
24     TPM_RC result;
25     // EncryptDecryptShared() performs the operations as shown in
26     // TPM2_EncryptDecrypt
27     result = EncryptDecryptShared(in->keyHandle,
28                                  in->decrypt,
29                                  in->mode,
30                                  &in->ivIn,
31                                  &in->inData,
32                                  (EncryptDecrypt_Out*)out);
33     // Handle response code swizzle.
34     switch(result)
35     {
36     case TPM_RCS_MODE + RC_EncryptDecrypt_mode:
37         result = TPM_RCS_MODE + RC_EncryptDecrypt2_mode;
38         break;
39     case TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn:
40         result = TPM_RCS_SIZE + RC_EncryptDecrypt2_ivIn;
41         break;
42     case TPM_RCS_SIZE + RC_EncryptDecrypt_inData:
43         result = TPM_RCS_SIZE + RC_EncryptDecrypt2_inData;
44         break;
45     default:
46         break;
47     }
48     return result;
49 }
50
51 #endif // CC_EncryptDecrypt2

```

7.128 /tpm/src/command/Symmetric/EncryptDecrypt_spt.c

```

1  #include "Tpm.h"
2  #include "EncryptDecrypt_fp.h"
3  #include "EncryptDecrypt_spt_fp.h"
4
5  #if CC_EncryptDecrypt2

```

```

6
7  /*(See part 3 specification)
8  // symmetric encryption or decryption
9  */
10 // Return Type: TPM_RC
11 //     TPM_RC_KEY           is not a symmetric decryption key with both
12 //                           public and private portions loaded
13 //     TPM_RC_SIZE         'IvIn' size is incompatible with the block cipher mode;
14 //                           or 'inData' size is not an even multiple of the block
15 //                           size for CBC or ECB mode
16 //     TPM_RC_VALUE        'keyHandle' is restricted and the argument 'mode' does
17 //                           not match the key's mode
18 TPM_RC
19 EncryptDecryptShared(TPMI_DH_OBJECT      keyHandleIn,
20                     TPMI_YES_NO         decryptIn,
21                     TPMI_ALG_SYM_MODE    modeIn,
22                     TPM2B_IV*            ivIn,
23                     TPM2B_MAX_BUFFER*    inData,
24                     EncryptDecrypt_Out*  out)
25 {
26     OBJECT*      symKey;
27     UINT16        keySize;
28     UINT16        blockSize;
29     BYTE*         key;
30     TPM_ALG_ID    alg;
31     TPM_ALG_ID    mode;
32     TPM_RC        result;
33     BOOL          OK;
34     // Input Validation
35     symKey = HandleToObject(keyHandleIn);
36     mode   = symKey->publicArea.parameters.symDetail.sym.mode.sym;
37
38     // The input key should be a symmetric key
39     if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
40         return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
41     // The key must be unrestricted and allow the selected operation
42     OK = !IS_ATTRIBUTE(symKey->publicArea.objectAttributes, TPMA_OBJECT, restricted);
43     if(YES == decryptIn)
44         OK = OK
45             && IS_ATTRIBUTE(
46                 symKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt);
47     else
48         OK = OK
49             && IS_ATTRIBUTE(symKey->publicArea.objectAttributes, TPMA_OBJECT, sign);
50     if(!OK)
51         return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;
52
53     // Make sure that key is an encrypt/decrypt key and not SMAC
54     if(!CryptSymModeIsValid(mode, TRUE))
55         return TPM_RCS_MODE + RC_EncryptDecrypt_keyHandle;
56
57     // If the key mode is not TPM_ALG_NULL...
58     // or TPM_ALG_NULL
59     if(mode != TPM_ALG_NULL)
60     {
61         // then the input mode has to be TPM_ALG_NULL or the same as the key
62         if((modeIn != TPM_ALG_NULL) && (modeIn != mode))
63             return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
64     }
65     else
66     {
67         // if the key mode is null, then the input can't be null
68         if(modeIn == TPM_ALG_NULL)
69             return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
70         mode = modeIn;
71     }

```

```

72 // The input iv for ECB mode should be an Empty Buffer. All the other modes
73 // should have an iv size same as encryption block size
74 keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
75 alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
76 blockSize = CryptGetSymmetricBlockSize(alg, keySize);
77
78 // reverify the algorithm. This is mainly to keep static analysis tools happy
79 if(blockSize == 0)
80     return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
81
82 if(((mode == TPM_ALG_ECB) && (ivIn->t.size != 0))
83 || ((mode != TPM_ALG_ECB) && (ivIn->t.size != blockSize)))
84     return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;
85
86 // The input data size of CBC mode or ECB mode must be an even multiple of
87 // the symmetric algorithm's block size
88 if(((mode == TPM_ALG_CBC) || (mode == TPM_ALG_ECB))
89 && ((inData->t.size % blockSize) != 0))
90     return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;
91
92 // Copy IV
93 // Note: This is copied here so that the calls to the encrypt/decrypt functions
94 // will modify the output buffer, not the input buffer
95 out->ivOut = *ivIn;
96
97 // Command Output
98 key = symKey->sensitive.sensitive.sym.t.buffer;
99 // For symmetric encryption, the cipher data size is the same as plain data
100 // size.
101 out->outData.t.size = inData->t.size;
102 if(decryptIn == YES)
103 {
104     // Decrypt data to output
105     result = CryptSymmetricDecrypt(out->outData.t.buffer,
106                                   alg,
107                                   keySize,
108                                   key,
109                                   &(out->ivOut),
110                                   mode,
111                                   inData->t.size,
112                                   inData->t.buffer);
113 }
114 else
115 {
116     // Encrypt data to output
117     result = CryptSymmetricEncrypt(out->outData.t.buffer,
118                                   alg,
119                                   keySize,
120                                   key,
121                                   &(out->ivOut),
122                                   mode,
123                                   inData->t.size,
124                                   inData->t.buffer);
125 }
126 return result;
127 }
128
129 #endif // CC_EncryptDecrypt

```

7.129 /tpm/src/command/Symmetric/Hash.c

```

1 #include "Tpm.h"
2 #include "Hash_fp.h"
3
4 #if CC_Hash // Conditional expansion of this file

```

```

5
6  /*(See part 3 specification)
7  // Hash a data buffer
8  */
9  TPM_RC
10 TPM2_Hash(Hash_In* in, // IN: input parameter list
11           Hash_Out* out // OUT: output parameter list
12 )
13 {
14     HASH_STATE hashState;
15
16     // Command Output
17
18     // Output hash
19     // Start hash stack
20     out->outHash.t.size = CryptHashStart(&hashState, in->hashAlg);
21     // Adding hash data
22     CryptDigestUpdate2B(&hashState, &in->data.b);
23     // Complete hash
24     CryptHashEnd2B(&hashState, &out->outHash.b);
25
26     // Output ticket
27     out->validation.tag = TPM_ST_HASHCHECK;
28     out->validation.hierarchy = in->hierarchy;
29
30     if(in->hierarchy == TPM_RH_NULL)
31     {
32         // Ticket is not required
33         out->validation.hierarchy = TPM_RH_NULL;
34         out->validation.digest.t.size = 0;
35     }
36     else if(
37         in->data.t.size >= sizeof(TPM_GENERATED_VALUE) && !TicketIsSafe(&in->data.b))
38     {
39         // Ticket is not safe
40         out->validation.hierarchy = TPM_RH_NULL;
41         out->validation.digest.t.size = 0;
42     }
43     else
44     {
45         TPM_RC result;
46         // Compute ticket
47         result = TicketComputeHashCheck(
48             in->hierarchy, in->hashAlg, &out->outHash, &out->validation);
49         if(result != TPM_RC_SUCCESS)
50             return result;
51     }
52
53     return TPM_RC_SUCCESS;
54 }
55
56 #endif // CC_Hash

```

7.130 /tpm/src/command/Symmetric/HMAC.c

```

1  #include "Tpm.h"
2  #include "HMAC_fp.h"
3
4  #if CC_HMAC // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Compute HMAC on a data buffer
8  */
9  // Return Type: TPM_RC
10 // TPM_RC_ATTRIBUTES key referenced by 'handle' is a restricted key

```

```

11 //      TPM_RC_KEY          'handle' does not reference a signing key
12 //      TPM_RC_TYPE        key referenced by 'handle' is not an HMAC key
13 //      TPM_RC_VALUE       'hashAlg' is not compatible with the hash algorithm
14 //                          of the scheme of the object referenced by 'handle'
15 TPM_RC
16 TPM2_HMAC(HMAC_In* in, // IN: input parameter list
17           HMAC_Out* out // OUT: output parameter list
18 )
19 {
20     HMAC_STATE    hmacState;
21     OBJECT*       hmacObject;
22     TPMI_ALG_HASH hashAlg;
23     TPMT_PUBLIC*  publicArea;
24
25     // Input Validation
26
27     // Get HMAC key object and public area pointers
28     hmacObject = HandleToObject(in->handle);
29     publicArea = &hmacObject->publicArea;
30     // Make sure that the key is an HMAC key
31     if(publicArea->type != TPM_ALG_KEYEDHASH)
32         return TPM_RCS_TYPE + RC_HMAC_handle;
33
34     // and that it is unrestricted
35     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
36         return TPM_RCS_ATTRIBUTES + RC_HMAC_handle;
37
38     // and that it is a signing key
39     if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
40         return TPM_RCS_KEY + RC_HMAC_handle;
41
42     // See if the key has a default
43     if(publicArea->parameters.keyedHashDetail.scheme.scheme == TPM_ALG_NULL)
44         // it doesn't so use the input value
45         hashAlg = in->hashAlg;
46     else
47     {
48         // key has a default so use it
49         hashAlg = publicArea->parameters.keyedHashDetail.scheme.details.hmac.hashAlg;
50         // and verify that the input was either the TPM_ALG_NULL or the default
51         if(in->hashAlg != TPM_ALG_NULL && in->hashAlg != hashAlg)
52             hashAlg = TPM_ALG_NULL;
53     }
54     // if we ended up without a hash algorithm then return an error
55     if(hashAlg == TPM_ALG_NULL)
56         return TPM_RCS_VALUE + RC_HMAC_hashAlg;
57
58     // Command Output
59
60     // Start HMAC stack
61     out->outhMAC.t.size = CryptHmacStart2B(
62         &hmacState, hashAlg, &hmacObject->sensitive.sensitive.bits.b);
63     // Adding HMAC data
64     CryptDigestUpdate2B(&hmacState.hashState, &in->buffer.b);
65
66     // Complete HMAC
67     CryptHmacEnd2B(&hmacState, &out->outhMAC.b);
68
69     return TPM_RC_SUCCESS;
70 }
71
72 #endif // CC_HMAC

```


7.131 /tpm/src/command/Symmetric/MAC.c

```

1  #include "Tpm.h"
2  #include "MAC_fp.h"
3
4  #if CC_MAC // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // Compute MAC on a data buffer
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_ATTRIBUTES      key referenced by 'handle' is a restricted key
11 //     TPM_RC_KEY              'handle' does not reference a signing key
12 //     TPM_RC_TYPE             key referenced by 'handle' is not an HMAC key
13 //     TPM_RC_VALUE            'hashAlg' is not compatible with the hash algorithm
14 //                             of the scheme of the object referenced by 'handle'
15 TPM_RC
16 TPM2_MAC(MAC_In* in, // IN: input parameter list
17          MAC_Out* out // OUT: output parameter list
18 )
19 {
20     OBJECT*      keyObject;
21     HMAC_STATE   state;
22     TPMT_PUBLIC* publicArea;
23     TPM_RC       result;
24
25     // Input Validation
26     // Get MAC key object and public area pointers
27     keyObject = HandleToObject(in->handle);
28     publicArea = &keyObject->publicArea;
29
30     // If the key is not able to do a MAC, indicate that the handle selects an
31     // object that can't do a MAC
32     result = CryptSelectMac(publicArea, &in->inScheme);
33     if(result == TPM_RCS_TYPE)
34         return TPM_RCS_TYPE + RC_MAC_handle;
35     // If there is another error type, indicate that the scheme and key are not
36     // compatible
37     if(result != TPM_RC_SUCCESS)
38         return RcSafeAddToResult(result, RC_MAC_inScheme);
39     // Make sure that the key is not restricted
40     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
41         return TPM_RCS_ATTRIBUTES + RC_MAC_handle;
42     // and that it is a signing key
43     if(!IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
44         return TPM_RCS_KEY + RC_MAC_handle;
45     // Command Output
46     out->outMAC.t.size = CryptMacStart(&state,
47                                       &publicArea->parameters,
48                                       in->inScheme,
49                                       &keyObject->sensitive.sensitive.any.b);
50     // If the mac can't start, treat it as a fatal error
51     if(out->outMAC.t.size == 0)
52         return TPM_RC_FAILURE;
53     CryptDigestUpdate2B(&state.hashState, &in->buffer.b);
54     // If the MAC result is not what was expected, it is a fatal error
55     if(CryptHmacEnd2B(&state, &out->outMAC.b) != out->outMAC.t.size)
56         return TPM_RC_FAILURE;
57     return TPM_RC_SUCCESS;
58 }
59
60 #endif // CC_MAC

```

7.132 /tpm/src/command/Testing/GetTestResult.c

```

1  #include "Tpm.h"
2  #include "GetTestResult_fp.h"
3
4  #if CC_GetTestResult // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // returns manufacturer-specific information regarding the results of a self-
8  // test and an indication of the test status.
9  */
10
11 // In the reference implementation, this function is only reachable if the TPM is
12 // not in failure mode meaning that all tests that have been run have completed
13 // successfully. There is not test data and the test result is TPM_RC_SUCCESS.
14 TPM_RC
15 TPM2_GetTestResult(GetTestResult_Out* out // OUT: output parameter list
16 )
17 {
18     // Command Output
19
20     // Call incremental self test function in crypt module
21     out->testResult = CryptGetTestResult(&out->outData);
22
23     return TPM_RC_SUCCESS;
24 }
25
26 #endif // CC_GetTestResult

```

7.133 /tpm/src/command/Testing/IncrementalSelfTest.c

```

1  #include "Tpm.h"
2  #include "IncrementalSelfTest_fp.h"
3
4  #if CC_IncrementalSelfTest // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // perform a test of selected algorithms
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_CANCELED    the command was canceled (some tests may have
11 //                        completed)
12 //     TPM_RC_VALUE       an algorithm in the toTest list is not implemented
13 TPM_RC
14 TPM2_IncrementalSelfTest(IncrementalSelfTest_In* in, // IN: input parameter list
15                          IncrementalSelfTest_Out* out // OUT: output parameter list
16 )
17 {
18     TPM_RC result;
19     // Command Output
20
21     // Call incremental self test function in crypt module. If this function
22     // returns TPM_RC_VALUE, it means that an algorithm on the 'toTest' list is
23     // not implemented.
24     result = CryptIncrementalSelfTest(&in->toTest, &out->toDoList);
25     if(result == TPM_RC_VALUE)
26         return TPM_RCS_VALUE + RC_IncrementalSelfTest_toTest;
27     return result;
28 }
29
30 #endif // CC_IncrementalSelfTest

```

7.134 /tpm/src/command/Testing/SelfTest.c

```

1  #include "Tpm.h"
2  #include "SelfTest_fp.h"
3
4  #if CC_SelfTest // Conditional expansion of this file
5
6  /*(See part 3 specification)
7  // perform a test of TPM capabilities
8  */
9  // Return Type: TPM_RC
10 //     TPM_RC_CANCELED          the command was canceled (some incremental
11 //                               process may have been made)
12 //     TPM_RC_TESTING           self test in process
13 TPM_RC
14 TPM2_SelfTest(SelfTest_In* in // IN: input parameter list
15 )
16 {
17     // Command Output
18
19     // Call self test function in crypt module
20     return CryptSelfTest(in->fullTest);
21 }
22
23 #endif // CC_SelfTest

```

7.135 /tpm/src/command/Vendor/Vendor_TCG_Test.c

```

1  #include "Tpm.h"
2
3  #if CC_Vendor_TCG_Test // Conditional expansion of this file
4  # include "Vendor_TCG_Test_fp.h"
5
6  TPM_RC
7  TPM2_Vendor_TCG_Test(Vendor_TCG_Test_In* in, // IN: input parameter list
8                      Vendor_TCG_Test_Out* out // OUT: output parameter list
9  )
10 {
11     out->outputData = in->inputData;
12     return TPM_RC_SUCCESS;
13 }
14
15 #endif // CC_Vendor_TCG_Test

```

7.136 /tpm/src/crypt/AlgorithmTests.c

```

1  /** Introduction
2  // This file contains the code to perform the various self-test functions.
3  //
4  // NOTE: In this implementation, large local variables are made static to minimize
5  // stack usage, which is critical for stack-constrained platforms.
6
7  /** Includes and Defines
8  #include "Tpm.h"
9
10 #define SELF_TEST_DATA
11
12 #if SELF_TEST
13
14 // These includes pull in the data structures. They contain data definitions for the
15 // various tests.
16 # include "SelfTest.h"
17 # include "SymmetricTest.h"

```

```

18 # include "RsaTestData.h"
19 # include "EccTestData.h"
20 # include "HashTestData.h"
21 # include "KdfTestData.h"
22
23 # define TEST_DEFAULT_TEST_HASH(vector) \
24     if(TEST_BIT(DEFAULT_TEST_HASH, g_toTest)) \
25         TestHash(DEFAULT_TEST_HASH, vector);
26
27 // Make sure that the algorithm has been tested
28 # define CLEAR_BOTH(alg) \
29     { \
30         CLEAR_BIT(alg, *toTest); \
31         if(toTest != &g_toTest) \
32             CLEAR_BIT(alg, g_toTest); \
33     }
34
35 # define SET_BOTH(alg) \
36     { \
37         SET_BIT(alg, *toTest); \
38         if(toTest != &g_toTest) \
39             SET_BIT(alg, g_toTest); \
40     }
41
42 # define TEST_BOTH(alg) \
43     ((toTest != &g_toTest) ? TEST_BIT(alg, *toTest) || TEST_BIT(alg, g_toTest) \
44     : TEST_BIT(alg, *toTest))
45
46 // Can only cancel if doing a list.
47 # define CHECK_CANCELED \
48     if(_plat_IsCanceled() && toTest != &g_toTest) \
49         return TPM_RC_CANCELED;
50
51 /** Hash Tests
52
53 **** Description
54 // The hash test does a known-value HMAC using the specified hash algorithm.
55
56 **** TestHash()
57 // The hash test function.
58 static TPM_RC TestHash(TPM_ALG_ID hashAlg, ALGORITHM_VECTOR* toTest)
59 {
60     static TPM2B_DIGEST computed; // value computed
61     static HMAC_STATE state;
62     UINT16 digestSize;
63     const TPM2B* testDigest = NULL;
64     // TPM2B_TYPE(HMAC_BLOCK, DEFAULT_TEST_HASH_BLOCK_SIZE);
65
66     pAssert(hashAlg != TPM_ALG_NULL);
67 # define HASH_CASE_FOR_TEST(HASH, hash) \
68     case ALG_##HASH##_VALUE: \
69         testDigest = &c_##HASH##_digest.b; \
70         break;
71     switch(hashAlg)
72     {
73         FOR_EACH_HASH(HASH_CASE_FOR_TEST)
74
75         default:
76             FAIL(FATAL_ERROR_INTERNAL);
77     }
78     // Clear the to-test bits
79     CLEAR_BOTH(hashAlg);
80
81     // If there is an algorithm without test vectors, then assume that things are OK.
82     if(testDigest == NULL || testDigest->size == 0)
83         return TPM_RC_SUCCESS;

```

```

84
85 // Set the HMAC key to twice the digest size
86 digestSize = CryptHashGetDigestSize(hashAlg);
87 CryptHmacStart(&state, hashAlg, digestSize * 2, (BYTE*)c_hashTestKey.t.buffer);
88 CryptDigestUpdate(&state.hashState,
89                  2 * CryptHashGetBlockSize(hashAlg),
90                  (BYTE*)c_hashTestData.t.buffer);
91 computed.t.size = digestSize;
92 CryptHmacEnd(&state, digestSize, computed.t.buffer);
93 if((testDigest->size != computed.t.size)
94    || (memcmp(testDigest->buffer, computed.t.buffer, computed.b.size) != 0))
95     SELF_TEST_FAILURE;
96 return TPM_RC_SUCCESS;
97 }
98
99 /** Symmetric Test Functions
100
101 /*** MakeIv()
102 // Internal function to make the appropriate IV depending on the mode.
103 static UINT32 MakeIv(TPM_ALG_ID mode, // IN: symmetric mode
104                     UINT32 size, // IN: block size of the algorithm
105                     BYTE* iv // OUT: IV to fill in
106 )
107 {
108     BYTE i;
109
110     if(mode == TPM_ALG_ECB)
111         return 0;
112     if(mode == TPM_ALG_CTR)
113     {
114         // The test uses an IV that has 0xff in the last byte
115         for(i = 1; i <= size; i++)
116             *iv++ = 0xff - (BYTE)(size - i);
117     }
118     else
119     {
120         for(i = 0; i < size; i++)
121             *iv++ = i;
122     }
123     return size;
124 }
125
126 /*** TestSymmetricAlgorithm()
127 // Function to test a specific algorithm, key size, and mode.
128 static void TestSymmetricAlgorithm(const SYMMETRIC_TEST_VECTOR* test, //
129                                   TPM_ALG_ID mode //
130 )
131 {
132     static BYTE encrypted[MAX_SYM_BLOCK_SIZE * 2];
133     static BYTE decrypted[MAX_SYM_BLOCK_SIZE * 2];
134     static TPM2B_IV iv;
135     //
136     // Get the appropriate IV
137     iv.t.size = (UINT16)MakeIv(mode, test->ivSize, iv.t.buffer);
138
139     // Encrypt known data
140     CryptSymmetricEncrypt(encrypted,
141                          test->alg,
142                          test->keyBits,
143                          test->key,
144                          &iv,
145                          mode,
146                          test->dataInOutSize,
147                          test->dataIn);
148
149     // Check that it matches the expected value
150     if(!MemoryEqual(

```

```

150         encrypted, test->dataOut[mode - TPM_ALG_CTR], test->dataInOutSize))
151     SELF_TEST_FAILURE;
152     // Reinitialize the iv for decryption
153     MakeIv(mode, test->ivSize, iv.t.buffer);
154     CryptSymmetricDecrypt(decrypted,
155                           test->alg,
156                           test->keyBits,
157                           test->key,
158                           &iv,
159                           mode,
160                           test->dataInOutSize,
161                           test->dataOut[mode - TPM_ALG_CTR]);
162     // Make sure that it matches what we started with
163     if(!MemoryEqual(decrypted, test->dataIn, test->dataInOutSize))
164         SELF_TEST_FAILURE;
165 }
166
167 /*** AllSymsAreDone()
168 // Checks if both symmetric algorithms have been tested. This is put here
169 // so that addition of a symmetric algorithm will be relatively easy to handle.
170 //
171 // Return Type: BOOL
172 //     TRUE(1)      all symmetric algorithms tested
173 //     FALSE(0)     not all symmetric algorithms tested
174 static BOOL AllSymsAreDone(ALGORITHM_VECTOR* toTest)
175 {
176     return (!TEST_BOTH(TPM_ALG_AES) && !TEST_BOTH(TPM_ALG_SM4));
177 }
178
179 /*** AllModesAreDone()
180 // Checks if all the modes have been tested.
181 //
182 // Return Type: BOOL
183 //     TRUE(1)      all modes tested
184 //     FALSE(0)     all modes not tested
185 static BOOL AllModesAreDone(ALGORITHM_VECTOR* toTest)
186 {
187     TPM_ALG_ID alg;
188     for(alg = SYM_MODE_FIRST; alg <= SYM_MODE_LAST; alg++)
189         if(TEST_BOTH(alg))
190             return FALSE;
191     return TRUE;
192 }
193
194 /*** TestSymmetric()
195 // If 'alg' is a symmetric block cipher, then all of the modes that are selected are
196 // tested. If 'alg' is a mode, then all algorithms of that mode are tested.
197 static TPM_RC TestSymmetric(TPM_ALG_ID alg, ALGORITHM_VECTOR* toTest)
198 {
199     SYM_INDEX index;
200     TPM_ALG_ID mode;
201     //
202     if(!TEST_BIT(alg, *toTest))
203         return TPM_RC_SUCCESS;
204     if(alg == TPM_ALG_AES || alg == TPM_ALG_SM4 || alg == TPM_ALG_CAMELLIA)
205     {
206         // Will test the algorithm for all modes and key sizes
207         CLEAR_BOTH(alg);
208
209         // A test this algorithm for all modes
210         for(index = 0; index < NUM_SYMS; index++)
211         {
212             if(c_symTestValues[index].alg == alg)
213             {
214                 for(mode = SYM_MODE_FIRST; mode <= SYM_MODE_LAST; mode++)
215                     {

```



```

216         if(TEST_BIT(mode, *toTest))
217             TestSymmetricAlgorithm(&c_symTestValues[index], mode);
218     }
219 }
220 }
221 // if all the symmetric tests are done
222 if(AllSymsAreDone(toTest))
223 {
224     // all symmetric algorithms tested so no modes should be set
225     for(alg = SYM_MODE_FIRST; alg <= SYM_MODE_LAST; alg++)
226         CLEAR_BOTH(alg);
227 }
228 }
229 else if(SYM_MODE_FIRST <= alg && alg <= SYM_MODE_LAST)
230 {
231     // Test this mode for all key sizes and algorithms
232     for(index = 0; index < NUM_SYMS; index++)
233     {
234         // The mode testing only comes into play when doing self tests
235         // by command. When doing self tests by command, the block ciphers are
236         // tested first. That means that all of their modes would have been
237         // tested for all key sizes. If there is no block cipher left to
238         // test, then clear this mode bit.
239         if(!TEST_BIT(TPM_ALG_AES, *toTest) && !TEST_BIT(TPM_ALG_SM4, *toTest))
240         {
241             CLEAR_BOTH(alg);
242         }
243         else
244         {
245             for(index = 0; index < NUM_SYMS; index++)
246             {
247                 if(TEST_BIT(c_symTestValues[index].alg, *toTest))
248                     TestSymmetricAlgorithm(&c_symTestValues[index], alg);
249             }
250             // have tested this mode for all algorithms
251             CLEAR_BOTH(alg);
252         }
253     }
254     if(AllModesAreDone(toTest))
255     {
256         CLEAR_BOTH(TPM_ALG_AES);
257         CLEAR_BOTH(TPM_ALG_SM4);
258     }
259 }
260 else
261     pAssert(alg == 0 && alg != 0);
262 return TPM_RC_SUCCESS;
263 }
264
265 /** RSA Tests
266 # if ALG_RSA
267
268 /** Introduction
269 // The tests are for public key only operations and for private key operations.
270 // Signature verification and encryption are public key operations. They are tested
271 // by using a KVT. For signature verification, this means that a known good
272 // signature is checked by CryptRsaValidateSignature(). If it fails, then the
273 // TPM enters failure mode. For encryption, the TPM encrypts known values using
274 // the selected scheme and checks that the returned value matches the expected
275 // value.
276 //
277 // For private key operations, a full scheme check is used. For a signing key, a
278 // known key is used to sign a known message. Then that signature is verified.
279 // since the signature may involve use of random values, the signature will be
280 // different each time and we can't always check that the signature matches a
281 // known value. The same technique is used for decryption (RSADP/RSAP).

```

```

282 //
283 // When an operation uses the public key and the verification has not been
284 // tested, the TPM will do a KVT.
285 //
286 // The test for the signing algorithm is built into the call for the algorithm
287
288 /*** RsaKeyInitialize()
289 // The test key is defined by a public modulus and a private prime. The TPM's RSA
290 // code computes the second prime and the private exponent.
291 static void RsaKeyInitialize(OBJECT* testObject)
292 {
293     MemoryCopy2B(&testObject->publicArea.unique.rsa.b,
294                 (P2B)&c_rsaPublicModulus,
295                 sizeof(c_rsaPublicModulus));
296     MemoryCopy2B(&testObject->sensitive.sensitive.rsa.b,
297                 (P2B)&c_rsaPrivatePrime,
298                 sizeof(testObject->sensitive.sensitive.rsa.t.buffer));
299     testObject->publicArea.parameters.rsaDetail.keyBits = RSA_TEST_KEY_SIZE * 8;
300     // Use the default exponent
301     testObject->publicArea.parameters.rsaDetail.exponent = 0;
302 }
303
304 /*** TestRsaEncryptDecrypt()
305 // These tests are for a public key encryption that uses a random value.
306 static TPM_RC TestRsaEncryptDecrypt(TPM_ALG_ID scheme, // IN: the scheme
307                                     ALGORITHM_VECTOR* toTest //
308 )
309 {
310     static TPM2B_PUBLIC_KEY_RSA testInput;
311     static TPM2B_PUBLIC_KEY_RSA testOutput;
312     static OBJECT testObject;
313     const TPM2B_RSA_TEST_KEY* kvtValue = NULL;
314     TPM_RC result = TPM_RC_SUCCESS;
315     const TPM2B* testLabel = NULL;
316     TPMT_RSA_DECRYPT rsaScheme;
317     //
318     // Don't need to initialize much of the test object
319     RsaKeyInitialize(&testObject);
320     rsaScheme.scheme = scheme;
321     rsaScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
322     CLEAR_BOTH(scheme);
323     CLEAR_BOTH(TPM_ALG_NULL);
324     if(scheme == TPM_ALG_NULL)
325     {
326         // This is an encryption scheme using the private key without any encoding.
327         memcpy(testInput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue));
328         testInput.t.size = sizeof(c_RsaTestValue);
329         if(TPM_RC_SUCCESS
330            != CryptRsaEncrypt(
331                &testOutput, &testInput.b, &testObject, &rsaScheme, NULL, NULL))
332             SELF_TEST_FAILURE;
333         if(!MemoryEqual(testOutput.t.buffer, c_RsaepKvt.buffer, c_RsaepKvt.size))
334             SELF_TEST_FAILURE;
335         MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
336         if(TPM_RC_SUCCESS
337            != CryptRsaDecrypt(
338                &testOutput.b, &testInput.b, &testObject, &rsaScheme, NULL))
339             SELF_TEST_FAILURE;
340         if(!MemoryEqual(testOutput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue)))
341             SELF_TEST_FAILURE;
342     }
343     else
344     {
345         // TPM_ALG_RSAES:
346         // This is an decryption scheme using padding according to
347         // PKCS#1v2.1, 7.2. This padding uses random bits. To test a public

```

```

348 // key encryption that uses random data, encrypt a value and then
349 // decrypt the value and see that we get the encrypted data back.
350 // The hash is not used by this encryption so it can be TPM_ALG_NULL
351
352 // TPM_ALG_OAEP:
353 // This is also an decryption scheme and it also uses a
354 // pseudo-random
355 // value. However, this also uses a hash algorithm. So, we may need
356 // to test that algorithm before use.
357 if(scheme == TPM_ALG_OAEP)
358 {
359     TEST_DEFAULT_TEST_HASH(toTest);
360     kvtValue = &c_OaepKvt;
361     testLabel = OAEP_TEST_STRING;
362 }
363 else if(scheme == TPM_ALG_RSAES)
364 {
365     kvtValue = &c_RsaesKvt;
366     testLabel = NULL;
367 }
368 else
369     SELF_TEST_FAILURE;
370 // Only use a digest-size portion of the test value
371 memcpy(testInput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
372 testInput.t.size = DEFAULT_TEST_DIGEST_SIZE;
373
374 // See if the encryption works
375 if(TPM_RC_SUCCESS
376    != CryptRsaEncrypt(
377        &testOutput, &testInput.b, &testObject, &rsaScheme, testLabel, NULL))
378     SELF_TEST_FAILURE;
379 MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
380 // see if we can decrypt this value and get the original data back
381 if(TPM_RC_SUCCESS
382    != CryptRsaDecrypt(
383        &testOutput.b, &testInput.b, &testObject, &rsaScheme, testLabel))
384     SELF_TEST_FAILURE;
385 // See if the results compare
386 if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
387    || !MemoryEqual(
388        testOutput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE))
389     SELF_TEST_FAILURE;
390 // Now check that the decryption works on a known value
391 MemoryCopy2B(&testInput.b, (P2B)kvtValue, sizeof(testInput.t.buffer));
392 if(TPM_RC_SUCCESS
393    != CryptRsaDecrypt(
394        &testOutput.b, &testInput.b, &testObject, &rsaScheme, testLabel))
395     SELF_TEST_FAILURE;
396 if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
397    || !MemoryEqual(
398        testOutput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE))
399     SELF_TEST_FAILURE;
400 }
401 return result;
402 }
403
404 /*** TestRsaSignAndVerify()
405 // This function does the testing of the RSA sign and verification functions. This
406 // test does a KVT.
407 static TPM_RC TestRsaSignAndVerify(TPM_ALG_ID scheme, ALGORITHM_VECTOR* toTest)
408 {
409     TPM_RC result = TPM_RC_SUCCESS;
410     static OBJECT testObject;
411     static TPM2B_DIGEST testDigest;
412     static TPMT_SIGNATURE testSig;
413

```

```

414 // Do a sign and signature verification.
415 // RSASSA:
416 // This is a signing scheme according to PKCS#1-v2.1 8.2. It does not
417 // use random data so there is a KVT for the signing operation. On
418 // first use of the scheme for signing, use the TPM's RSA key to
419 // sign a portion of c_RsaTestData and compare the results to c_RsassaKvt. Then
420 // decrypt the data to see that it matches the starting value. This verifies
421 // the signature with a KVT
422
423 // Clear the bits indicating that the function has not been checked. This is to
424 // prevent looping
425 CLEAR_BOTH(scheme);
426 CLEAR_BOTH(TPM_ALG_NULL);
427 CLEAR_BOTH(TPM_ALG_RSA);
428
429 RsaKeyInitialize(&testObject);
430 memcpy(testDigest.t.buffer, (BYTE*)c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
431 testDigest.t.size = DEFAULT_TEST_DIGEST_SIZE;
432 testSig.sigAlg = scheme;
433 testSig.signature.rsapss.hash = DEFAULT_TEST_HASH;
434
435 // RSAPSS:
436 // This is a signing scheme according to PKCS#1-v2.2 8.1 it uses
437 // random data in the signature so there is no KVT for the signing
438 // operation. To test signing, the TPM will use the TPM's RSA key
439 // to sign a portion of c_RsaTestValue and then it will verify the
440 // signature. For verification, c_RsapssKvt is verified before the
441 // user signature blob is verified. The worst case for testing of this
442 // algorithm is two private and one public key operation.
443
444 // The process is to sign known data. If RSASSA is being done, verify that the
445 // signature matches the precomputed value. For both, use the signed value and
446 // see that the verification says that it is a good signature. Then
447 // if testing RSAPSS, do a verify of a known good signature. This ensures that
448 // the validation function works.
449
450 if(TPM_RC_SUCCESS != CryptRsaSign(&testSig, &testObject, &testDigest, NULL))
451     SELF_TEST_FAILURE;
452 // For RSASSA, make sure the results is what we are looking for
453 if(testSig.sigAlg == TPM_ALG_RSASSA)
454 {
455     if(testSig.signature.rsassa.sig.t.size != RSA_TEST_KEY_SIZE
456        || !MemoryEqual(c_RsassaKvt.buffer,
457                        testSig.signature.rsassa.sig.t.buffer,
458                        RSA_TEST_KEY_SIZE))
459         SELF_TEST_FAILURE;
460 }
461 // See if the TPM will validate its own signatures
462 if(TPM_RC_SUCCESS
463    != CryptRsaValidateSignature(&testSig, &testObject, &testDigest))
464     SELF_TEST_FAILURE;
465 // If this is RSAPSS, check the verification with known signature
466 // Have to copy because CryptRsaValidateSignature() eats the signature
467 if(TPM_ALG_RSAPSS == scheme)
468 {
469     MemoryCopy2B(&testSig.signature.rsapss.sig.b,
470                  (P2B)&c_RsapssKvt,
471                  sizeof(testSig.signature.rsapss.sig.t.buffer));
472     if(TPM_RC_SUCCESS
473        != CryptRsaValidateSignature(&testSig, &testObject, &testDigest))
474         SELF_TEST_FAILURE;
475 }
476 return result;
477 }
478
479 /*** TestRSA()

```

```

480 // Function uses the provided vector to indicate which tests to run. It will clear
481 // the vector after each test is run and also clear g_toTest
482 static TPM_RC TestRsa(TPM_ALG_ID alg, ALGORITHM_VECTOR* toTest)
483 {
484     TPM_RC result = TPM_RC_SUCCESS;
485     //
486     switch(alg)
487     {
488         case TPM_ALG_NULL:
489             // This is the RSAEP/RSADP function. If we are processing a list, don't
490             // need to test these now because any other test will validate
491             // RSAEP/RSADP. Can tell this is list of test by checking to see if
492             // 'toTest' is pointing at g_toTest. If so, this is an isolated test
493             // an need to go ahead and do the test;
494             if((toTest == &g_toTest)
495                 || (!TEST_BIT(TPM_ALG_RSASSA, *toTest)
496                     && !TEST_BIT(TPM_ALG_RSAES, *toTest)
497                     && !TEST_BIT(TPM_ALG_RSAPSS, *toTest)
498                     && !TEST_BIT(TPM_ALG_OAEP, *toTest)))
499                 // Not running a list of tests or no other tests on the list
500                 // so run the test now
501                 result = TestRsaEncryptDecrypt(alg, toTest);
502             // if not running the test now, leave the bit on, just in case things
503             // get interrupted
504             break;
505         case TPM_ALG_OAEP:
506         case TPM_ALG_RSAES:
507             result = TestRsaEncryptDecrypt(alg, toTest);
508             break;
509         case TPM_ALG_RSAPSS:
510         case TPM_ALG_RSASSA:
511             result = TestRsaSignAndVerify(alg, toTest);
512             break;
513         default:
514             SELF_TEST_FAILURE;
515     }
516     return result;
517 }
518
519 # endif // ALG_RSA
520
521 /** ECC Tests
522
523 # if ALG_ECC
524
525 /** LoadEccParameter()
526 // This function is mostly for readability and type checking
527 static void LoadEccParameter(TPM2B_ECC_PARAMETER* to, // target
528                             const TPM2B_EC_TEST* from // source
529 )
530 {
531     MemoryCopy2B(&to->b, &from->b, sizeof(to->t.buffer));
532 }
533
534 /** LoadEccPoint()
535 static void LoadEccPoint(TPMS_ECC_POINT* point, // target
536                         const TPM2B_EC_TEST* x, // source
537                         const TPM2B_EC_TEST* y)
538 {
539     MemoryCopy2B(&point->x.b, (TPM2B*)x, sizeof(point->x.t.buffer));
540     MemoryCopy2B(&point->y.b, (TPM2B*)y, sizeof(point->y.t.buffer));
541 }
542
543 /** TestECDH()
544 // This test does a KVT on a point multiply.
545 static TPM_RC TestECDH(TPM_ALG_ID scheme, // IN: for consistency

```



```

546             ALGORITHM_VECTOR* toTest // IN/OUT: modified after test is run
547     )
548     {
549         static TPMS_ECC_POINT      Z;
550         static TPMS_ECC_POINT      Qe;
551         static TPM2B_ECC_PARAMETER ds;
552         TPM_RC                      result = TPM_RC_SUCCESS;
553         //
554         NOT_REFERENCED(scheme);
555         CLEAR_BOTH(TPM_ALG_ECDH);
556         LoadEccParameter(&ds, &c_ecTestKey_ds);
557         LoadEccPoint(&Qe, &c_ecTestKey_QeX, &c_ecTestKey_QeY);
558         if(TPM_RC_SUCCESS != CryptEccPointMultiply(&Z, c_testCurve, &Qe, &ds, NULL, NULL))
559             SELF_TEST_FAILURE;
560         if(!MemoryEqual2B(&c_ecTestEcdh_X.b, &Z.x.b)
561            || !MemoryEqual2B(&c_ecTestEcdh_Y.b, &Z.y.b))
562             SELF_TEST_FAILURE;
563         return result;
564     }
565
566     /*** TestEccSignAndVerify()
567     static TPM_RC TestEccSignAndVerify(TPM_ALG_ID scheme, ALGORITHM_VECTOR* toTest)
568     {
569         static OBJECT      testObject;
570         static TPMT_SIGNATURE testSig;
571         static TPMT_ECC_SCHEME eccScheme;
572
573         testSig.sigAlg      = scheme;
574         testSig.signature.ecdsa.hash = DEFAULT_TEST_HASH;
575
576         eccScheme.scheme    = scheme;
577         eccScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
578
579         CLEAR_BOTH(scheme);
580         CLEAR_BOTH(TPM_ALG_ECDH);
581
582         // ECC signature verification testing uses a KVT.
583         switch(scheme)
584         {
585             case TPM_ALG_ECDSA:
586                 LoadEccParameter(&testSig.signature.ecdsa.signatureR, &c_TestEcDsa_r);
587                 LoadEccParameter(&testSig.signature.ecdsa.signatureS, &c_TestEcDsa_s);
588                 break;
589             case TPM_ALG_ECSCHNORR:
590                 LoadEccParameter(&testSig.signature.ecschnorr.signatureR,
591                                 &c_TestEcSchnorr_r);
592                 LoadEccParameter(&testSig.signature.ecschnorr.signatureS,
593                                 &c_TestEcSchnorr_s);
594                 break;
595             case TPM_ALG_SM2:
596                 // don't have a test for SM2
597                 return TPM_RC_SUCCESS;
598             default:
599                 SELF_TEST_FAILURE;
600                 break;
601         }
602         TEST_DEFAULT_TEST_HASH(toTest);
603
604         // Have to copy the key. This is because the size used in the test vectors
605         // is the size of the ECC parameter for the test key while the size of a point
606         // is TPM dependent
607         MemoryCopy2B(&testObject.sensitive.sensitive.ecc.b,
608                    &c_ecTestKey_ds.b,
609                    sizeof(testObject.sensitive.sensitive.ecc.t.buffer));
610         LoadEccPoint(
611             &testObject.publicArea.unique.ecc, &c_ecTestKey_QsX, &c_ecTestKey_QsY);

```



```

612     testObject.publicArea.parameters.eccDetail.curveID = c_testCurve;
613
614     if(TPM_RC_SUCCESS
615         != CryptEccValidateSignature(
616             &testSig, &testObject, (TPM2B_DIGEST*)&c_ecTestValue.b))
617     {
618         SELF_TEST_FAILURE;
619     }
620     CHECK_CANCELED;
621
622     // Now sign and verify some data
623     if(TPM_RC_SUCCESS
624         != CryptEccSign(
625             &testSig, &testObject, (TPM2B_DIGEST*)&c_ecTestValue, &eccScheme, NULL))
626         SELF_TEST_FAILURE;
627
628     CHECK_CANCELED;
629
630     if(TPM_RC_SUCCESS
631         != CryptEccValidateSignature(
632             &testSig, &testObject, (TPM2B_DIGEST*)&c_ecTestValue))
633         SELF_TEST_FAILURE;
634
635     CHECK_CANCELED;
636
637     return TPM_RC_SUCCESS;
638 }
639
640 /*** TestKDFa()
641 static TPM_RC TestKDFa(ALGORITHM_VECTOR* toTest)
642 {
643     static TPM2B_KDF_TEST_KEY keyOut;
644     UINT32 counter = 0;
645     //
646     CLEAR_BOTH(TPM_ALG_KDF1_SP800_108);
647
648     keyOut.t.size = CryptKDFa(KDF_TEST_ALG,
649                             &c_kdfTestKeyIn.b,
650                             &c_kdfTestLabel.b,
651                             &c_kdfTestContextU.b,
652                             &c_kdfTestContextV.b,
653                             TEST_KDF_KEY_SIZE * 8,
654                             keyOut.t.buffer,
655                             &counter,
656                             FALSE);
657     if(keyOut.t.size != TEST_KDF_KEY_SIZE
658        || !MemoryEqual(keyOut.t.buffer, c_kdfTestKeyOut.t.buffer, TEST_KDF_KEY_SIZE))
659         SELF_TEST_FAILURE;
660
661     return TPM_RC_SUCCESS;
662 }
663
664 /*** TestEcc()
665 static TPM_RC TestEcc(TPM_ALG_ID alg, ALGORITHM_VECTOR* toTest)
666 {
667     TPM_RC result = TPM_RC_SUCCESS;
668     NOT_REFERENCED(toTest);
669     switch(alg)
670     {
671         case TPM_ALG_ECC:
672         case TPM_ALG_ECDH:
673             // If this is in a loop then see if another test is going to deal with
674             // this.
675             // If toTest is not a self-test list
676             if((toTest == &q_toTest)
677                // or this is the only ECC test in the list

```

```

678         || !(TEST_BIT(TPM_ALG_ECDSA, *toTest)
679         || TEST_BIT(ALG_EC Schnorr, *toTest)
680         || TEST_BIT(TPM_ALG_SM2, *toTest)))
681     {
682         result = TestECDH(alg, toTest);
683     }
684     break;
685 case TPM_ALG_ECDSA:
686 case TPM_ALG_EC Schnorr:
687 case TPM_ALG_SM2:
688     result = TestEccSignAndVerify(alg, toTest);
689     break;
690 default:
691     SELF_TEST_FAILURE;
692     break;
693 }
694 return result;
695 }
696
697 # endif // ALG_ECC
698
699 /*** TestAlgorithm()
700 // Dispatches to the correct test function for the algorithm or gets a list of
701 // testable algorithms.
702 //
703 // If 'toTest' is not NULL, then the test decisions are based on the algorithm
704 // selections in 'toTest'. Otherwise, 'g_toTest' is used. When bits are clear in
705 // 'g_toTest' they will also be cleared 'toTest'.
706 //
707 // If there doesn't happen to be a test for the algorithm, its associated bit is
708 // quietly cleared.
709 //
710 // If 'alg' is zero (TPM_ALG_ERROR), then the toTest vector is cleared of any bits
711 // for which there is no test (i.e. no tests are actually run but the vector is
712 // cleared).
713 //
714 // Note: 'toTest' will only ever have bits set for implemented algorithms but 'alg'
715 // can be anything.
716 //
717 // Return Type: TPM_RC
718 // TPM_RC_CANCELED test was canceled
719 LIB_EXPORT
720 TPM_RC
721 TestAlgorithm(TPM_ALG_ID alg, ALGORITHM_VECTOR* toTest)
722 {
723     TPM_ALG_ID first = (alg == TPM_ALG_ERROR) ? TPM_ALG_FIRST : alg;
724     TPM_ALG_ID last = (alg == TPM_ALG_ERROR) ? TPM_ALG_LAST : alg;
725     BOOL doTest = (alg != TPM_ALG_ERROR);
726     TPM_RC result = TPM_RC_SUCCESS;
727
728     if(toTest == NULL)
729         toTest = &g_toTest;
730
731     // This is kind of strange. This function will either run a test of the selected
732     // algorithm or just clear a bit if there is no test for the algorithm. So,
733     // either this loop will be executed once for the selected algorithm or once for
734     // each of the possible algorithms. If it is executed more than once ('alg' ==
735     // ALG_ERROR), then no test will be run but bits will be cleared for
736     // unimplemented algorithms. This was done this way so that there is only one
737     // case statement with all of the algorithms. It was easier to have one case
738     // statement than to have multiple ones to manage whenever an algorithm ID is
739     // added.
740     for(alg = first; (alg <= last); alg++)
741     {
742         // if 'alg' was TPM_ALG_ERROR, then we will be cycling through
743         // values, some of which may not be implemented. If the bit in toTest

```

```

744     // happens to be set, then we could either generated an assert, or just
745     // silently CLEAR it. Decided to just clear.
746     if(!TEST_BIT(alg, g_implementedAlgorithms))
747     {
748         CLEAR_BIT(alg, *toTest);
749         continue;
750     }
751     // Process whatever is left.
752     // NOTE: since this switch will only be called if the algorithm is
753     // implemented, it is not necessary to modify this list except to comment
754     // out the algorithms for which there is no test
755     switch(alg)
756     {
757         // Symmetric block ciphers
758 #   if ALG_AES
759         case TPM_ALG_AES:
760 #   endif // ALG_AES
761 #   if ALG_SM4
762         // if SM4 is implemented, its test is like other block ciphers but
763         // aren't any test vectors for it yet
764         case TPM_ALG_SM4:
765 #   endif // ALG_SM4
766 #   if ALG_CAMELLIA
767         // no test vectors for camellia
768         case TPM_ALG_CAMELLIA:
769 #   endif
770         // Symmetric modes
771 #   if !ALG_CFB
772         #   error CFB is required in all TPM implementations
773 #   endif // !ALG_CFB
774         case TPM_ALG_CFB:
775             if(doTest)
776                 result = TestSymmetric(alg, toTest);
777             break;
778 #   if ALG_CTR
779         case TPM_ALG_CTR:
780 #   endif // ALG_CTR
781 #   if ALG_OFB
782         case TPM_ALG_OFB:
783 #   endif // ALG_OFB
784 #   if ALG_CBC
785         case TPM_ALG_CBC:
786 #   endif // ALG_CBC
787 #   if ALG_ECB
788         case TPM_ALG_ECB:
789 #   endif
790             if(doTest)
791                 result = TestSymmetric(alg, toTest);
792             else
793                 // If doing the initialization of g_toTest vector, only need
794                 // to test one of the modes for the symmetric algorithms. If
795                 // initializing for a SelfTest(FULL_TEST), allow all the modes.
796                 if(toTest == &g_toTest)
797                     CLEAR_BIT(alg, *toTest);
798             break;
799 #   if !ALG_HMAC
800         #   error HMAC is required in all TPM implementations
801 #   endif
802         case TPM_ALG_HMAC:
803             // Clear the bit that indicates that HMAC is required because
804             // HMAC is used as the basic test for all hash algorithms.
805             CLEAR_BOTH(alg);
806             // Testing HMAC means test the default hash
807             if(doTest)
808                 TestHash(DEFAULT_TEST_HASH, toTest);

```

```

809         else
810             // If not testing, then indicate that the hash needs to be
811             // tested because this uses HMAC
812             SET_BOTH(DEFAULT_TEST_HASH);
813         break;
814 // Have to use two arguments for the macro even though only the first is used in the
815 // expansion.
816 # define HASH_CASE_TEST(HASH, hash) case ALG_ ##HASH##_VALUE:
817     FOR_EACH_HASH(HASH_CASE_TEST)
818 # undef HASH_CASE_TEST
819     if(doTest)
820         result = TestHash(alg, toTest);
821     break;
822     // RSA-dependent
823 # if ALG_RSA
824     case TPM_ALG_RSA:
825         CLEAR_BOTH(alg);
826         if(doTest)
827             result = TestRsa(TPM_ALG_NULL, toTest);
828         else
829             SET_BOTH(TPM_ALG_NULL);
830         break;
831     case TPM_ALG_RSASSA:
832     case TPM_ALG_RSAES:
833     case TPM_ALG_RSAPSS:
834     case TPM_ALG_OAEP:
835     case TPM_ALG_NULL: // used or RSADP
836         if(doTest)
837             result = TestRsa(alg, toTest);
838         break;
839 # endif // ALG_RSA
840 # if ALG_KDF1_SP800_108
841     case TPM_ALG_KDF1_SP800_108:
842         if(doTest)
843             result = TestKDFa(toTest);
844         break;
845 # endif // ALG_KDF1_SP800_108
846 # if ALG_ECC
847     // ECC dependent but no tests
848     // case TPM_ALG_ECDAA:
849     // case TPM_ALG_ECMQV:
850     // case TPM_ALG_KDF1_SP800_56a:
851     // case TPM_ALG_KDF2:
852     // case TPM_ALG_MGF1:
853     case TPM_ALG_ECC:
854         CLEAR_BOTH(alg);
855         if(doTest)
856             result = TestEcc(TPM_ALG_ECDH, toTest);
857         else
858             SET_BOTH(TPM_ALG_ECDH);
859         break;
860     case TPM_ALG_ECDSA:
861     case TPM_ALG_ECDH:
862     case TPM_ALG_ECSCHNORR:
863         // case TPM_ALG_SM2:
864         if(doTest)
865             result = TestEcc(alg, toTest);
866         break;
867 # endif // ALG_ECC
868     default:
869         CLEAR_BIT(alg, *toTest);
870         break;
871 }
872 if(result != TPM_RC_SUCCESS)
873     break;
874 }

```

```

875     return result;
876 }
877
878 #endif // SELF_TESTS

```

7.137 /tpm/src/crypt/CryptCmac.c

```

1  /** Introduction
2  //
3  // This file contains the implementation of the message authentication codes based
4  // on a symmetric block cipher. These functions only use the single block
5  // encryption functions of the selected symmetric cryptographic library.
6
7  /** Includes, Defines, and Typedefs
8  #define _CRYPT_HASH_C_
9  #include "Tpm.h"
10 #include "CryptSym.h"
11
12 #if ALG_CMAL
13
14 /** Functions
15
16 /** CryptCmacStart()
17 // This is the function to start the CMAC sequence operation. It initializes the
18 // dispatch functions for the data and end operations for CMAC and initializes the
19 // parameters that are used for the processing of data, including the key, key size
20 // and block cipher algorithm.
21 UINT16
22 CryptCmacStart(
23     SMAC_STATE* state, TPMU_PUBLIC_PARMS* keyParms, TPM_ALG_ID macAlg, TPM2B* key)
24 {
25     tpmCmacState_t* cState = &state->state.cmac;
26     TPMT_SYM_DEF_OBJECT* def = &keyParms->symDetail.sym;
27     //
28     if(macAlg != TPM_ALG_CMAL)
29         return 0;
30     // set up the encryption algorithm and parameters
31     cState->symAlg = def->algorithm;
32     cState->keySizeBits = def->keyBits.sym;
33     cState->iv.t.size = CryptGetSymmetricBlockSize(def->algorithm, def->keyBits.sym);
34     MemoryCopy2B(&cState->symKey.b, key, sizeof(cState->symKey.t.buffer));
35
36     // Set up the dispatch methods for the CMAL
37     state->smacMethods.data = CryptCmacData;
38     state->smacMethods.end = CryptCmacEnd;
39     return cState->iv.t.size;
40 }
41
42 /** CryptCmacData()
43 // This function is used to add data to the CMAL sequence computation. The function
44 // will XOR new data into the IV. If the buffer is full, and there is additional
45 // input data, the data is encrypted into the IV buffer, the new data is then
46 // XOR into the IV. When the data runs out, the function returns without encrypting
47 // even if the buffer is full. The last data block of a sequence will not be
48 // encrypted until the call to CryptCmacEnd(). This is to allow the proper subkey
49 // to be computed and applied before the last block is encrypted.
50 void CryptCmacData(SMAC_STATES* state, UINT32 size, const BYTE* buffer)
51 {
52     tpmCmacState_t* cmacState = &state->cmal;
53     TPM_ALG_ID algorithm = cmacState->symAlg;
54     BYTE* key = cmacState->symKey.t.buffer;
55     UINT16 keySizeInBits = cmacState->keySizeBits;
56     tpmCryptKeySchedule_t keySchedule;
57     TpmCryptSetSymKeyCall_t encrypt;
58     //

```

```

59     // Set up the encryption values based on the algorithm
60     switch(algorithm)
61     {
62         FOR_EACH_SYM(ENCRYPT_CASE)
63         default:
64             FAIL(FATAL_ERROR_INTERNAL);
65     }
66     while(size > 0)
67     {
68         if(cmacState->bcount == cmacState->iv.t.size)
69         {
70             ENCRYPT(&keySchedule, cmacState->iv.t.buffer, cmacState->iv.t.buffer);
71             cmacState->bcount = 0;
72         }
73         for(; (size > 0) && (cmacState->bcount < cmacState->iv.t.size);
74             size--, cmacState->bcount++)
75         {
76             cmacState->iv.t.buffer[cmacState->bcount] ^= *buffer++;
77         }
78     }
79 }
80
81 /*** CryptCmacEnd()
82 // This is the completion function for the CMAC. It does padding, if needed, and
83 // selects the subkey to be applied before the last block is encrypted.
84 UINT16
85 CryptCmacEnd(SMAC_STATES* state, UINT32 outSize, BYTE* outBuffer)
86 {
87     tpmCmacState_t* cState = &state->cmac;
88     // Need to set algorithm, key, and keySizeInBits in the local context so that
89     // the SELECT and ENCRYPT macros will work here
90     TPM_ALG_ID      algorithm      = cState->symAlg;
91     BYTE*           key            = cState->symKey.t.buffer;
92     UINT16          keySizeInBits = cState->keySizeBits;
93     tpmCryptKeySchedule_t keySchedule;
94     TpmCryptSetSymKeyCall_t encrypt;
95     TPM2B_IV        subkey = {{0, {0}}};
96     BOOL            xorVal;
97     UINT16          i;
98
99     subkey.t.size = cState->iv.t.size;
100    // Encrypt a block of zero
101    // Set up the encryption values based on the algorithm
102    switch(algorithm)
103    {
104        FOR_EACH_SYM(ENCRYPT_CASE)
105        default:
106            return 0;
107    }
108    ENCRYPT(&keySchedule, subkey.t.buffer, subkey.t.buffer);
109
110    // shift left by 1 and XOR with 0x0...87 if the MSb was 0
111    xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
112    ShiftLeft(&subkey.b);
113    subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
114    // this is a sanity check to make sure that the algorithm is working properly.
115    // remove this check when debug is done
116    pAssert(cState->bcount <= cState->iv.t.size);
117    // If the buffer is full then no need to compute subkey 2.
118    if(cState->bcount < cState->iv.t.size)
119    {
120        //Pad the data
121        cState->iv.t.buffer[cState->bcount++] ^= 0x80;
122        // The rest of the data is a pad of zero which would simply be XORed
123        // with the iv value so nothing to do...
124        // Now compute K2

```



```

125     xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
126     ShiftLeft(&subkey.b);
127     subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
128 }
129 // XOR the subkey into the IV
130 for(i = 0; i < subkey.t.size; i++)
131     cState->iv.t.buffer[i] ^= subkey.t.buffer[i];
132 ENCRYPT(&keySchedule, cState->iv.t.buffer, cState->iv.t.buffer);
133 i = (UINT16)MIN(cState->iv.t.size, outSize);
134 MemoryCopy(outBuffer, cState->iv.t.buffer, i);
135
136 return i;
137 }
138 #endif

```

7.138 /tpm/src/crypt/CryptEccCrypt.c

```

1  /** Includes and Defines
2  #include "Tpm.h"
3  #include "TpmMath_Util_fp.h"
4  #include "TpmEcc_Util_fp.h"
5
6  #if CC_ECC_Encrypt || CC_ECC_Decrypt
7
8  /** Functions
9
10 /** CryptEccSelectScheme()
11 // This function is used by TPM2_ECC_Decrypt and TPM2_ECC_Encrypt. It sets scheme
12 // either the input scheme or the key scheme. If the key scheme is not TPM_ALG_NULL
13 // then the input scheme must be TPM_ALG_NULL or the same as the key scheme. If
14 // not, then the function returns FALSE.
15 // Return Type: BOOL
16 // TRUE 'scheme' is set
17 // FALSE 'scheme' is not valid (it may have been changed).
18 BOOL CryptEccSelectScheme(OBJECT* key, //IN: key containing default scheme
19 TPMT_KDF_SCHEME* scheme // IN: a decrypt scheme
20 )
21 {
22     TPMT_KDF_SCHEME* keyScheme = &key->publicArea.parameters.eccDetail.kdf;
23
24     // Get sign object pointer
25     if(scheme->scheme == TPM_ALG_NULL)
26         *scheme = *keyScheme;
27     if(keyScheme->scheme == TPM_ALG_NULL)
28         keyScheme = scheme;
29     return (
30         scheme->scheme != TPM_ALG_NULL
31         && (keyScheme->scheme == scheme->scheme
32             && keyScheme->details.anyKdf.hashAlg == scheme->details.anyKdf.hashAlg));
33 }
34
35 /** CryptEccEncrypt()
36 //This function performs ECC-based data obfuscation. The only scheme that is currently
37 // supported is MGF1 based. See Part 1, Annex D for details.
38 // Return Type: TPM_RC
39 // TPM_RC_CURVE unsupported curve
40 // TPM_RC_HASH hash not allowed
41 // TPM_RC_SCHEME 'scheme' is not supported
42 // TPM_RC_NO_RESULT internal error in big number processing
43 LIB_EXPORT TPM_RC CryptEccEncrypt(
44     OBJECT* key, // IN: public key of recipient
45     TPMT_KDF_SCHEME* scheme, // IN: scheme to use.
46     TPM2B_MAX_BUFFER* plainText, // IN: the text to obfuscate
47     TPMS_ECC_POINT* c1, // OUT: public ephemeral key
48     TPM2B_MAX_BUFFER* c2, // OUT: obfuscated text

```

```

49     TPM2B_DIGEST*      c3          // OUT: digest of ephemeral key
50                               //      and plainText
51 )
52 {
53     CRYPT_CURVE_INITIALIZED(E, key->publicArea.parameters.eccDetail.curveID);
54     CRYPT_POINT_INITIALIZED(PB, &key->publicArea.unique.ecc);
55     CRYPT_POINT_VAR(Px);
56     TPMS_ECC_POINT p2;
57     CRYPT_ECC_NUM(D);
58     TPM2B_TYPE(2ECC, MAX_ECC_KEY_BYTES * 2);
59     TPM2B_2ECC z;
60     int i;
61     HASH_STATE hashState;
62     TPM_RC retVal = TPM_RC_SUCCESS;
63     //
64 #   if defined DEBUG_ECC_ENCRYPT && DEBUG_ECC_ENCRYPT == YES
65     RND_DEBUG dbg;
66     // This value is one less than the value from the reference so that it
67     // will become the correct value after having one added
68     TPM2B_ECC_PARAMETER k = {24, {0x38, 0x4F, 0x30, 0x35, 0x30, 0x73, 0xAE, 0xEC,
69                                   0xE7, 0xA1, 0x65, 0x43, 0x30, 0xA9, 0x62, 0x04,
70                                   0xD3, 0x79, 0x82, 0xA3, 0xE1, 0x5B, 0x2C, 0xB4}};
71     RND_DEBUG_Instantiate(&dbg, &k.b);
72 #   define RANDOM (RAND_STATE*)&dbg
73
74 #   else
75 #   define RANDOM NULL
76 #   endif
77     if(E == NULL)
78         ERROR_EXIT(TPM_RC_CURVE);
79     if(TPM_ALG_KDF2 != scheme->scheme)
80         ERROR_EXIT(TPM_RC_SCHEME);
81     // generate an ephemeral key from a random k
82     if(!TpmEcc_GenerateKeyPair(D, Px, E, RANDOM)
83         // C1 is the public part of the ephemeral key
84         || !TpmEcc_PointTo2B(c1, Px, E)
85         // Compute P2
86         || (TpmEcc_PointMult(Px, PB, D, NULL, NULL, E) != TPM_RC_SUCCESS)
87         || !TpmEcc_PointTo2B(&p2, Px, E))
88         ERROR_EXIT(TPM_RC_NO_RESULT);
89
90     //Compute the C3 value hash(x2 || M || y2)
91     if(0 == CryptHashStart(&hashState, scheme->details.mgf1.hashAlg))
92         ERROR_EXIT(TPM_RC_HASH);
93     CryptDigestUpdate2B(&hashState, &p2.x.b);
94     CryptDigestUpdate2B(&hashState, &plainText->b);
95     CryptDigestUpdate2B(&hashState, &p2.y.b);
96     c3->t.size = CryptHashEnd(&hashState, sizeof(c3->t.buffer), c3->t.buffer);
97
98     MemoryCopy2B(&z.b, &p2.x.b, sizeof(z.t.buffer));
99     MemoryConcat2B(&z.b, &p2.y.b, sizeof(z.t.buffer));
100    // Generate the mask value from MGF1 and put it in the return buffer
101    c2->t.size = CryptMGF_KDF(plainText->t.size,
102                             c2->t.buffer,
103                             scheme->details.mgf1.hashAlg,
104                             z.t.size,
105                             z.t.buffer,
106                             1);
107    // XOR the plainText into the generated mask to create the obfuscated data
108    for(i = 0; i < plainText->t.size; i++)
109        c2->t.buffer[i] ^= plainText->t.buffer[i];
110    Exit:
111        CRYPT_CURVE_FREE(E);
112        return retVal;
113 }
114

```

```

115  /*** CryptEccDecrypt()
116  // This function performs ECC decryption and integrity check of the input data.
117  // Return Type: TPM_RC
118  //     TPM_RC_CURVE      unsupported curve
119  //     TPM_RC_HASH       hash not allowed
120  //     TPM_RC_SCHEME     'scheme' is not supported
121  //     TPM_RC_NO_RESULT  internal error in big number processing
122  //     TPM_RC_VALUE      C3 did not match hash of recovered data
123  LIB_EXPORT TPM_RC CryptEccDecrypt(
124      OBJECT*      key,          // IN: key used for data recovery
125      TPMT_KDF_SCHEME* scheme,   // IN: scheme to use.
126      TPM2B_MAX_BUFFER* plainText, // OUT: the recovered text
127      TPMS_ECC_POINT* c1,        // IN: public ephemeral key
128      TPM2B_MAX_BUFFER* c2,      // IN: obfuscated text
129      TPM2B_DIGEST* c3           // IN: digest of ephemeral key
130                                // and plainText
131  )
132  {
133      CRYPT_CURVE_INITIALIZED(E, key->publicArea.parameters.eccDetail.curveID);
134      CRYPT_ECC_INITIALIZED(D, &key->sensitive.sensitive.ecc.b);
135      CRYPT_POINT_INITIALIZED(C1, c1);
136      TPMS_ECC_POINT p2;
137      TPM2B_TYPE(2ECC, MAX_ECC_KEY_BYTES * 2);
138      TPM2B_DIGEST check;
139      TPM2B_2ECC z;
140      int i;
141      HASH_STATE hashState;
142      TPM_RC retVal = TPM_RC_SUCCESS;
143      //
144      if(E == NULL)
145          ERROR_EXIT(TPM_RC_CURVE);
146      if(TPM_ALG_KDF2 != scheme->scheme)
147          ERROR_EXIT(TPM_RC_SCHEME);
148      // Generate the Z value
149      TpmEcc_PointMult(C1, C1, D, NULL, NULL, E);
150      TpmEcc_PointTo2B(&p2, C1, E);
151
152      // Start the hash to check the algorithm
153      if(0 == CryptHashStart(&hashState, scheme->details.mgf1.hashAlg))
154          ERROR_EXIT(TPM_RC_HASH);
155      CryptDigestUpdate2B(&hashState, &p2.x.b);
156
157      MemoryCopy2B(&z.b, &p2.x.b, sizeof(z.t.buffer));
158      MemoryConcat2B(&z.b, &p2.y.b, sizeof(z.t.buffer));
159
160      // Generate the mask
161      plainText->t.size = CryptMGF_KDF(c2->t.size,
162                                     plainText->t.buffer,
163                                     scheme->details.mgf1.hashAlg,
164                                     z.t.size,
165                                     z.t.buffer,
166                                     1);
167      // XOR the obfuscated data into the generated mask to create the plainText data
168      for(i = 0; i < plainText->t.size; i++)
169          plainText->t.buffer[i] ^= c2->t.buffer[i];
170
171      // Complete the hash and verify the data
172      CryptDigestUpdate2B(&hashState, &plainText->b);
173      CryptDigestUpdate2B(&hashState, &p2.y.b);
174      check.t.size = CryptHashEnd(&hashState, sizeof(check.t.buffer), check.t.buffer);
175      if(!MemoryEqual2B(&check.b, &c3->b))
176          ERROR_EXIT(TPM_RC_VALUE);
177  Exit:
178      CRYPT_CURVE_FREE(E);
179      return retVal;
180  }

```

```

181
182 #endif // CC_ECC_Encrypt || CC_ECC_Encrypt

```

7.139 /tpm/src/crypt/CryptEccData.c

```

1  /*(Auto-generated)
2  *   Created by TpmStructures; Version 4.4 Mar 26, 2019
3  *   Date: Aug 30, 2019   Time: 02:11:52PM
4  */
5
6  #include "Tpm.h"
7  #include "OIDs.h"
8
9  #if ALG_ECC
10
11  // This file contains the TPM Specific ECC curve metadata and pointers to the ecc-lib
12  // specific
13  // constant structure.
14  // The CURVE_NAME macro is used to remove the name string from normal builds, but
15  // leaves the
16  // string available in the initialization lists for potential use during debugging by
17  // changing this
18  // macro (and the structure declaration)
19  # define CURVE_NAME(N)
20
21  # define comma
22  const TPM_ECC_CURVE_METADATA eccCurves[] = {
23  # if ECC_NIST_P192
24      comma{TPM_ECC_NIST_P192,
25          192,
26          {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA256}}},
27          {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
28          OID_ECC_NIST_P192 CURVE_NAME("NIST_P192")}
29  #   undef comma
30  #   define comma ,
31  # endif // ECC_NIST_P192
32  # if ECC_NIST_P224
33      comma{TPM_ECC_NIST_P224,
34          224,
35          {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA256}}},
36          {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
37          OID_ECC_NIST_P224 CURVE_NAME("NIST_P224")}
38  #   undef comma
39  #   define comma ,
40  # endif // ECC_NIST_P224
41  # if ECC_NIST_P256
42      comma{TPM_ECC_NIST_P256,
43          256,
44          {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA256}}},
45          {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
46          OID_ECC_NIST_P256 CURVE_NAME("NIST_P256")}
47  #   undef comma
48  #   define comma ,
49  # endif // ECC_NIST_P256
50  # if ECC_NIST_P384
51      comma{TPM_ECC_NIST_P384,
52          384,
53          {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA384}}},
54          {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
55          OID_ECC_NIST_P384 CURVE_NAME("NIST_P384")}
56  #   undef comma
57  #   define comma ,
58  # endif // ECC_NIST_P384
59  # if ECC_NIST_P521
60      comma{TPM_ECC_NIST_P521,

```

```

58         521,
59         {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA512}}},
60         {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
61         OID_ECC_NIST_P521 CURVE_NAME("NIST_P521")}
62 #     undef comma
63 #     define comma ,
64 # endif // ECC_NIST_P521
65 # if ECC_BN_P256
66     comma{TPM_ECC_BN_P256,
67         256,
68         {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
69         {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
70         OID_ECC_BN_P256 CURVE_NAME("BN_P256")}
71 #     undef comma
72 #     define comma ,
73 # endif // ECC_BN_P256
74 # if ECC_BN_P638
75     comma{TPM_ECC_BN_P638,
76         638,
77         {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
78         {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
79         OID_ECC_BN_P638 CURVE_NAME("BN_P638")}
80 #     undef comma
81 #     define comma ,
82 # endif // ECC_BN_P638
83 # if ECC_SM2_P256
84     comma{TPM_ECC_SM2_P256,
85         256,
86         {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SM3_256}}},
87         {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
88         OID_ECC_SM2_P256 CURVE_NAME("SM2_P256")}
89 #     undef comma
90 #     define comma ,
91 # endif // ECC_SM2_P256
92 };
93
94 #endif // TPM_ALG_ECC

```

7.140 /tpm/src/crypt/CryptEccKeyExchange.c

```

1  /** Introduction
2  // This file contains the functions that are used for the two-phase, ECC,
3  // key-exchange protocols
4
5  #include "Tpm.h"
6  #include "TpmMath_Util_fp.h"
7  #include "TpmEcc_Util_fp.h"
8
9  #if CC_ZGen_2Phase == YES
10
11  /** Functions
12
13  # if ALG_ECMQV
14
15  /** avf1()
16  // This function does the associated value computation required by MQV key
17  // exchange.
18  // Process:
19  // 1. Convert 'xQ' to an integer 'xqi' using the convention specified in Appendix C.3.
20  // 2. Calculate
21  //     xqm = xqi mod 2^ceil(f/2) (where f = ceil(log2(n))).
22  // 3. Calculate the associate value function
23  //     avf(Q) = xqm + 2ceil(f / 2)
24  // Always returns TRUE(1).
25  static BOOL avf1(Crypt_Int* bnX, // IN/OUT: the reduced value

```

```

26         Crypt_Int* bnN    // IN: the order of the curve
27     )
28     {
29         // compute  $f = 2^{(\text{ceil}(\log_2(n)) / 2)}$ 
30         int f = (ExtMath_SizeInBits(bnN) + 1) / 2;
31         //  $x' = 2^f + (x \bmod 2^f)$ 
32         ExtMath_MaskBits(bnX, f); // This is mod  $2 \cdot 2^f$  but it doesn't matter because
33         // the next operation will SET the extra bit anyway
34         if(!ExtMath_SetBit(bnX, f))
35         {
36             FAIL(FATAL_ERROR_CRYPTO);
37         }
38         return TRUE;
39     }
40
41     /*** C_2_2_MQV()
42     // This function performs the key exchange defined in SP800-56A
43     // 6.1.1.4 Full MQV, C(2, 2, ECC MQV).
44     //
45     // CAUTION: Implementation of this function may require use of essential claims in
46     // patents not owned by TCG members.
47     //
48     // Points 'QsB' and 'QeB' are required to be on the curve of 'inQsA'. The function
49     // will fail, possibly catastrophically, if this is not the case.
50     // Return Type: TPM_RC
51     //     TPM_RC_NO_RESULT    the value for dsA does not give a valid point on the
52     //                          curve
53     static TPM_RC C_2_2_MQV(TPMS_ECC_POINT* outZ, // OUT: the computed point
54                             TPM_ECC_CURVE curveId, // IN: the curve for the computations
55                             TPM2B_ECC_PARAMETER* dsA, // IN: static private TPM key
56                             TPM2B_ECC_PARAMETER* deA, // IN: ephemeral private TPM key
57                             TPMS_ECC_POINT* QsB, // IN: static public party B key
58                             TPMS_ECC_POINT* QeB // IN: ephemeral public party B key
59     )
60     {
61         CRYPT_CURVE_INITIALIZED(E, curveId);
62         CRYPT_POINT_VAR(pQeA);
63         CRYPT_POINT_INITIALIZED(pQeB, QeB);
64         CRYPT_POINT_INITIALIZED(pQsB, QsB);
65         CRYPT_ECC_NUM(bnTa);
66         CRYPT_ECC_INITIALIZED(bnDeA, deA);
67         CRYPT_ECC_INITIALIZED(bnDsA, dsA);
68         CRYPT_ECC_NUM(bnN);
69         CRYPT_ECC_NUM(bnXeB);
70         TPM_RC retVal;
71         //
72         // Parameter checks
73         if(E == NULL)
74             ERROR_EXIT(TPM_RC_VALUE);
75         pAssert(
76             outZ != NULL && pQeB != NULL && pQsB != NULL && deA != NULL && dsA != NULL);
77         // Process:
78         // 1.  $\text{implicitsigA} = (deA + \text{avf}(QeA)dsA) \bmod n$ .
79         // 2.  $P = h(\text{implicitsigA})(QeB + \text{avf}(QeB)QsB)$ .
80         // 3. If  $P = O$ , output an error indicator.
81         // 4.  $Z = xP$ , where  $xP$  is the x-coordinate of  $P$ .
82
83         // Compute the public ephemeral key pQeA = [deA]G
84         if((retVal =
85             TpmEcc_PointMult(pQeA, ExtEcc_CurveGetG(curveId), bnDeA, NULL, NULL, E))
86             != TPM_RC_SUCCESS)
87             goto Exit;
88
89         // 1.  $\text{implicitsigA} = (deA + \text{avf}(QeA)dsA) \bmod n$ .
90         //  $tA := (dsA + deA \text{ avf}(XeA)) \bmod n$  (3)
91         // Compute 'tA' = ('deA' + 'dsA' avf('XeA')) mod n

```



```

92     // Ta = avf(XeA);
93     ExtMath_Copy(bnTa, ExtEcc_PointX(pQeA));
94     avf1(bnTa, bnN);
95     // do Ta = dsA * Ta mod n = dsA * avf(XeA) mod n
96     ExtMath_ModMult(bnTa, bnDsA, bnTa, bnN);
97     // now Ta = deA + Ta mod n = deA + dsA * avf(XeA) mod n
98     ExtMath_Add(bnTa, bnTa, bnDeA);
99     ExtMath_Mod(bnTa, bnN);
100
101     // 2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).
102     // Put this in because almost every case of h is == 1 so skip the call when
103     // not necessary.
104     if(!ExtMath_IsEqualWord(ExtEcc_CurveGetCofactor(curveId), 1))
105         // Cofactor is not 1 so compute Ta := Ta * h mod n
106         ExtMath_ModMult(bnTa,
107                         bnTa,
108                         ExtEcc_CurveGetCofactor(curveId),
109                         ExtEcc_CurveGetOrder(curveId));
110
111     // Now that 'tA' is (h * 'tA' mod n)
112     // 'outZ' = (tA)(Qe,B + avf(Qe,B)Qs,B).
113
114     // first, compute XeB = avf(XeB)
115     avf1(bnXeB, bnN);
116
117     // QsB := [XeB]QsB
118     TpmEcc_PointMult(pQsB, pQsB, bnXeB, NULL, NULL, E);
119     ExtEcc_PointAdd(pQeB, pQeB, pQsB, E);
120
121     // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
122     // If the result is not the point at infinity, return QeB
123     TpmEcc_PointMult(pQeB, pQeB, bnTa, NULL, NULL, E);
124     if(ExtEcc_IsInfinityPoint(pQeB))
125         ERROR_EXIT(TPM_RC_NO_RESULT);
126     // Convert Crypt_Int* E to TPM2B E
127     TpmEcc_PointTo2B(outZ, pQeB, E);
128
129 Exit:
130     CRYPT_CURVE_FREE(E);
131     return retVal;
132 }
133
134 # endif // ALG_ECMQV
135
136 /*** C_2_2_ECDH()
137 // This function performs the two phase key exchange defined in SP800-56A,
138 // 6.1.1.2 Full Unified Model, C(2, 2, ECC CDH).
139 //
140 static TPM_RC C_2_2_ECDH(TPMS_ECC_POINT* outZs, // OUT: Zs
141                          TPMS_ECC_POINT* outZe, // OUT: Ze
142                          TPM_ECC_CURVE curveId, // IN: the curve for the computations
143                          TPM2B_ECC_PARAMETER* dsA, // IN: static private TPM key
144                          TPM2B_ECC_PARAMETER* deA, // IN: ephemeral private TPM key
145                          TPMS_ECC_POINT* QsB, // IN: static public party B key
146                          TPMS_ECC_POINT* QeB // IN: ephemeral public party B key
147 )
148 {
149     CRYPT_CURVE_INITIALIZED(E, curveId);
150     CRYPT_ECC_INITIALIZED(bnAs, dsA);
151     CRYPT_ECC_INITIALIZED(bnAe, deA);
152     CRYPT_POINT_INITIALIZED(ecBs, QsB);
153     CRYPT_POINT_INITIALIZED(ecBe, QeB);
154     CRYPT_POINT_VAR(ecZ);
155     TPM_RC retVal;
156     //
157     // Parameter checks

```

```

158     if(E == NULL)
159         ERROR_EXIT(TPM_RC_CURVE);
160     pAssert(
161         outZs != NULL && dsA != NULL && deA != NULL && QsB != NULL && QeB != NULL);
162
163     // Do the point multiply for the Zs value ([dsA]QsB)
164     retVal = TpmEcc_PointMult(ecZ, ecBs, bnAs, NULL, NULL, E);
165     if(retVal == TPM_RC_SUCCESS)
166     {
167         // Convert the Zs value.
168         TpmEcc_PointTo2B(outZs, ecZ, E);
169         // Do the point multiply for the Ze value ([deA]QeB)
170         retVal = TpmEcc_PointMult(ecZ, ecBe, bnAe, NULL, NULL, E);
171         if(retVal == TPM_RC_SUCCESS)
172             TpmEcc_PointTo2B(outZe, ecZ, E);
173     }
174 Exit:
175     CRYPT_CURVE_FREE(E);
176     return retVal;
177 }
178
179 /*** CryptEcc2PhaseKeyExchange()
180 // This function is the dispatch routine for the EC key exchange functions that use
181 // two ephemeral and two static keys.
182 // Return Type: TPM_RC
183 // TPM_RC_SCHEME scheme is not defined
184 LIB_EXPORT TPM_RC CryptEcc2PhaseKeyExchange(
185     TPMS_ECC_POINT* outZ1, // OUT: a computed point
186     TPMS_ECC_POINT* outZ2, // OUT: and optional second point
187     TPM_ECC_CURVE curveId, // IN: the curve for the computations
188     TPM_ALG_ID scheme, // IN: the key exchange scheme
189     TPM2B_ECC_PARAMETER* dsA, // IN: static private TPM key
190     TPM2B_ECC_PARAMETER* deA, // IN: ephemeral private TPM key
191     TPMS_ECC_POINT* QsB, // IN: static public party B key
192     TPMS_ECC_POINT* QeB, // IN: ephemeral public party B key
193 )
194 {
195     pAssert(
196         outZ1 != NULL && dsA != NULL && deA != NULL && QsB != NULL && QeB != NULL);
197
198     // Initialize the output points so that they are empty until one of the
199     // functions decides otherwise
200     outZ1->x.b.size = 0;
201     outZ1->y.b.size = 0;
202     if(outZ2 != NULL)
203     {
204         outZ2->x.b.size = 0;
205         outZ2->y.b.size = 0;
206     }
207     switch(scheme)
208     {
209         case TPM_ALG_ECDH:
210             return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
211             break;
212     # if ALG_ECMQV
213         case TPM_ALG_ECMQV:
214             return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
215             break;
216     # endif
217     # if ALG_SM2
218         case TPM_ALG_SM2:
219             return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
220             break;
221     # endif
222     default:
223         return TPM_RC_SCHEME;

```

```

224     }
225 }
226
227 # if ALG_SM2
228
229 /*** ComputeWForSM2()
230 // Compute the value for w used by SM2
231 static UINT32 ComputeWForSM2(TPM_ECC_CURVE curveId)
232 {
233     // w := ceil(ceil(log2(n)) / 2) - 1
234     return (ExtMath_MostSigBitNum(ExtEcc_CurveGetOrder(curveId)) / 2 - 1);
235 }
236
237 /*** avfSm2()
238 // This function does the associated value computation required by SM2 key
239 // exchange. This is different from the avf() in the international standards
240 // because it returns a value that is half the size of the value returned by the
241 // standard avf(). For example, if 'n' is 15, 'Ws' ('w' in the standard) is 2 but
242 // the 'W' here is 1. This means that an input value of 14 (1110b) would return a
243 // value of 110b with the standard but 10b with the scheme in SM2.
244 static Crypt_Int* avfSm2(Crypt_Int* bn, // IN/OUT: the reduced value
245                          UINT32 w // IN: the value of w
246 )
247 {
248     // a) set w := ceil(ceil(log2(n)) / 2) - 1
249     // b) set x' := 2^w + (x & (2^w - 1))
250     // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
251
252     ExtMath_MaskBits(bn, w); // as with avf1, this is too big by a factor of 2 but
253                             // it doesn't matter because we SET the extra bit
254                             // anyway
255     if(!ExtMath_SetBit(bn, w))
256     {
257         FAIL(FATAL_ERROR_CRYPTO);
258     }
259     return bn;
260 }
261
262 /*** SM2KeyExchange()
263 // This function performs the key exchange defined in SM2.
264 // The first step is to compute
265 // 'tA' = ('dsA' + 'deA' avf(Xe,A)) mod 'n'
266 // Then, compute the 'Z' value from
267 // 'outZ' = ('h' 'tA' mod 'n') ('QsA' + [avf('QeB.x')]('QeB')).
268 // The function will compute the ephemeral public key from the ephemeral
269 // private key.
270 // All points are required to be on the curve of 'inQsA'. The function will fail
271 // catastrophically if this is not the case
272 // Return Type: TPM_RC
273 // TPM_RC_NO_RESULT the value for dsA does not give a valid point on the
274 // curve
275 LIB_EXPORT TPM_RC SM2KeyExchange(
276     TPMS_ECC_POINT* outZ, // OUT: the computed point
277     TPM_ECC_CURVE curveId, // IN: the curve for the computations
278     TPM2B_ECC_PARAMETER* dsAIn, // IN: static private TPM key
279     TPM2B_ECC_PARAMETER* deAIn, // IN: ephemeral private TPM key
280     TPMS_ECC_POINT* QsBIn, // IN: static public party B key
281     TPMS_ECC_POINT* QeBIn // IN: ephemeral public party B key
282 )
283 {
284     CRYPT_CURVE_INITIALIZED(E, curveId);
285     CRYPT_ECC_INITIALIZED(dsA, dsAIn);
286     CRYPT_ECC_INITIALIZED(deA, deAIn);
287     CRYPT_POINT_INITIALIZED(QsB, QsBIn);
288     CRYPT_POINT_INITIALIZED(QeB, QeBIn);
289     CRYPT_INT_WORD_INITIALIZED(One, 1);

```

```

290     CRYPT_POINT_VAR(QeA);
291     CRYPT_ECC_NUM(XeB);
292     CRYPT_POINT_VAR(Z);
293     CRYPT_ECC_NUM(Ta);
294     CRYPT_ECC_NUM(QeA_X);
295     UINT32 w;
296     TPM_RC retVal = TPM_RC_NO_RESULT;
297     //
298     // Parameter checks
299     if(E == NULL)
300         ERROR_EXIT(TPM_RC_CURVE);
301     pAssert(outZ != NULL && dsA != NULL && deA != NULL && QsB != NULL && QeB != NULL);
302
303     // Compute the value for w
304     w = ComputeWForSM2(curveId);
305
306     // Compute the public ephemeral key pQeA = [de,A]G
307     if(!ExtEcc_PointMultiply(QeA, ExtEcc_CurveGetG(curveId), deA, E))
308         goto Exit;
309
310     // tA := (ds,A + de,A avf(Xe,A)) mod n (3)
311     // Compute 'tA' = ('dsA' + 'deA' avf('XeA')) mod n
312     // Ta = avf(XeA);
313     // do Ta = de,A * Ta = deA * avf(XeA)
314     ExtMath_Copy(QeA_X, ExtEcc_PointX(QeA)); // create mutable copy
315     ExtMath_Multiply(Ta, deA, avfSm2(QeA_X, w));
316     // now Ta = dsA + Ta = dsA + deA * avf(XeA)
317     ExtMath_Add(Ta, dsA, Ta);
318     ExtMath_Mod(Ta, ExtEcc_CurveGetOrder(curveId));
319
320     // outZ = [h tA mod n] (Qs,B + [avf(Xe,B)](Qe,B)) (4)
321     // Put this in because almost every case of h is == 1 so skip the call when
322     // not necessary.
323     if(!ExtMath_IsEqualWord(ExtEcc_CurveGetCofactor(curveId), 1))
324         // Cofactor is not 1 so compute Ta := Ta * h mod n
325         ExtMath_ModMult(
326             Ta, Ta, ExtEcc_CurveGetCofactor(curveId), ExtEcc_CurveGetOrder(curveId));
327     // Now that 'tA' is (h * 'tA' mod n)
328     // 'outZ' = ['tA'] (QsB + [avf(QeB.x)](QeB)).
329     ExtMath_Copy(XeB, ExtEcc_PointX(QeB));
330     if(!ExtEcc_PointMultiplyAndAdd(Z, QsB, One, QeB, avfSm2(XeB, w), E))
331         goto Exit;
332     // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
333     if(!ExtEcc_PointMultiply(Z, Z, Ta, E))
334         goto Exit;
335     // Convert Crypt_Int* E to TPM2B E
336     TpmEcc_PointTo2B(outZ, Z, E);
337     retVal = TPM_RC_SUCCESS;
338 Exit:
339     CRYPT_CURVE_FREE(E);
340     return retVal;
341 }
342 # endif
343
344 #endif // CC_ZGen_2Phase

```

7.141 /tpm/src/crypt/CryptEccMain.c

```

1  /** Includes and Defines
2  #include "Tpm.h"
3  #include "TpmMath_Util_fp.h"
4  #include "TpmEcc_Util_fp.h"
5  #include "TpmEcc_Signature_ECDSA_fp.h" // required for pairwise test in key
generation
6  #if ALG_ECC

```

```

7  /** Functions
8
9  # if SIMULATION
10 void EccSimulationEnd(void)
11 {
12 # if SIMULATION
13 // put things to be printed at the end of the simulation here
14 # endif
15 }
16 # endif // SIMULATION
17
18 /** CryptEccInit()
19 // This function is called at _TPM_Init
20 BOOL CryptEccInit(void)
21 {
22     return TRUE;
23 }
24
25 /** CryptEccStartup()
26 // This function is called at TPM2_Startup().
27 BOOL CryptEccStartup(void)
28 {
29     return TRUE;
30 }
31
32 /** ClearPoint2B(generic)
33 // Initialize the size values of a TPMS_ECC_POINT structure.
34 void ClearPoint2B(TPMS_ECC_POINT* p // IN: the point
35 )
36 {
37     if(p != NULL)
38     {
39         p->x.t.size = 0;
40         p->y.t.size = 0;
41     }
42 }
43
44 /** CryptEccGetParametersByCurveId()
45 // This function returns a pointer to the curve data that is associated with
46 // the indicated curveId.
47 // If there is no curve with the indicated ID, the function returns NULL. This
48 // function is in this module so that it can be called by GetCurve data.
49 // Return Type: const TPM_ECC_CURVE_METADATA*
50 // NULL curve with the indicated TPM_ECC_CURVE is not implemented
51 // != NULL pointer to the curve data
52 LIB_EXPORT const TPM_ECC_CURVE_METADATA* CryptEccGetParametersByCurveId(
53     TPM_ECC_CURVE curveId // IN: the curveID
54 )
55 {
56     int i;
57     for(i = 0; i < ECC_CURVE_COUNT; i++)
58     {
59         if(eccCurves[i].curveId == curveId)
60             return &eccCurves[i];
61     }
62     return NULL;
63 }
64
65 /** CryptEccGetKeySizeForCurve()
66 // This function returns the key size in bits of the indicated curve.
67 LIB_EXPORT UINT16 CryptEccGetKeySizeForCurve(TPM_ECC_CURVE curveId // IN: the curve
68 )
69 {
70     const TPM_ECC_CURVE_METADATA* curve = CryptEccGetParametersByCurveId(curveId);
71     UINT16 keySizeInBits;
72     //

```

```

73     keySizeInBits = (curve != NULL) ? curve->keySizeBits : 0;
74     return keySizeInBits;
75 }
76
77 /***CryptEccGetOID()
78 const BYTE* CryptEccGetOID(TPM_ECC_CURVE curveId)
79 {
80     const TPM_ECC_CURVE METADATA* curve = CryptEccGetParametersByCurveId(curveId);
81     return (curve != NULL) ? curve->OID : NULL;
82 }
83
84 /*** CryptEccGetCurveByIndex()
85 // This function returns the number of the 'i'-th implemented curve. The normal
86 // use would be to call this function with 'i' starting at 0. When the 'i' is greater
87 // than or equal to the number of implemented curves, TPM_ECC_NONE is returned.
88 LIB_EXPORT TPM_ECC_CURVE CryptEccGetCurveByIndex(UINT16 i)
89 {
90     if(i >= ECC_CURVE_COUNT)
91         return TPM_ECC_NONE;
92     return eccCurves[i].curveId;
93 }
94
95 /*** CryptCapGetECCCurve()
96 // This function returns the list of implemented ECC curves.
97 // Return Type: TPML_YES_NO
98 //     YES         if no more ECC curve is available
99 //     NO          if there are more ECC curves not reported
100 TPML_YES_NO
101 CryptCapGetECCCurve(TPM_ECC_CURVE curveID, // IN: the starting ECC curve
102                    UINT32 maxCount, // IN: count of returned curves
103                    TPML_ECC_CURVE* curveList // OUT: ECC curve list
104 )
105 {
106     TPML_YES_NO more = NO;
107     UINT16 i;
108     UINT32 count = ECC_CURVE_COUNT;
109     TPM_ECC_CURVE curve;
110
111     // Initialize output property list
112     curveList->count = 0;
113
114     // The maximum count of curves we may return is MAX_ECC_CURVES
115     if(maxCount > MAX_ECC_CURVES)
116         maxCount = MAX_ECC_CURVES;
117
118     // Scan the eccCurveValues array
119     for(i = 0; i < count; i++)
120     {
121         curve = CryptEccGetCurveByIndex(i);
122         // If curveID is less than the starting curveID, skip it
123         if(curve < curveID)
124             continue;
125         if(curveList->count < maxCount)
126         {
127             // If we have not filled up the return list, add more curves to
128             // it
129             curveList->eccCurves[curveList->count] = curve;
130             curveList->count++;
131         }
132         else
133         {
134             // If the return list is full but we still have curves
135             // available, report this and stop iterating
136             more = YES;
137             break;
138         }
139     }

```



```

139     }
140     return more;
141 }
142
143 /*** CryptCapGetOneECCCurve()
144 // This function returns whether the ECC curve is implemented.
145 BOOL CryptCapGetOneECCCurve(TPM_ECC_CURVE curveID // IN: the ECC curve
146 )
147 {
148     UINT16 i;
149
150     // Scan the eccCurveValues array
151     for(i = 0; i < ECC_CURVE_COUNT; i++)
152     {
153         if(CryptEccGetCurveByIndex(i) == curveID)
154         {
155             return TRUE;
156         }
157     }
158     return FALSE;
159 }
160
161 /*** CryptGetCurveSignScheme()
162 // This function will return a pointer to the scheme of the curve.
163 const TPMT_ECC_SCHEME* CryptGetCurveSignScheme(
164     TPM_ECC_CURVE curveId // IN: The curve selector
165 )
166 {
167     const TPM_ECC_CURVE_METADATA* curve = CryptEccGetParametersByCurveId(curveId);
168
169     if(curve != NULL)
170         return &(curve->sign);
171     else
172         return NULL;
173 }
174
175 /*** CryptGenerateR()
176 // This function computes the commit random value for a split signing scheme.
177 //
178 // If 'c' is NULL, it indicates that 'r' is being generated
179 // for TPM2 Commit.
180 // If 'c' is not NULL, the TPM will validate that the 'gr.commitArray'
181 // bit associated with the input value of 'c' is SET. If not, the TPM
182 // returns FALSE and no 'r' value is generated.
183 // Return Type: BOOL
184 //     TRUE(1)         r value computed
185 //     FALSE(0)        no r value computed
186 BOOL CryptGenerateR(TPM2B_ECC_PARAMETER* r, // OUT: the generated random value
187     UINT16* c, // IN/OUT: count value.
188     TPMI_ECC_CURVE curveID, // IN: the curve for the value
189     TPM2B_NAME* name // IN: optional name of a key to
190                     // associate with 'r'
191 )
192 {
193     // This holds the marshaled g_commitCounter.
194     TPM2B_TYPE(8B, 8);
195     TPM2B_8B cntnr = {{8, {0}}};
196     UINT32 iterations;
197     TPM2B_ECC_PARAMETER n;
198     UINT64 currentCount = gr.commitCounter;
199     UINT16 t1;
200     //
201     if(!TpmMath_IntTo2B(ExtEcc_CurveGetOrder(curveID), (TPM2B*)&n, 0))
202         return FALSE;
203
204     // If this is the commit phase, use the current value of the commit counter

```

```

205     if(c != NULL)
206     {
207         // if the array bit is not set, can't use the value.
208         if(!TEST_BIT((*c & COMMIT_INDEX_MASK), gr.commitArray))
209             return FALSE;
210
211         // If it is the sign phase, figure out what the counter value was
212         // when the commitment was made.
213         //
214         // When gr.commitArray has less than 64K bits, the extra
215         // bits of 'c' are used as a check to make sure that the
216         // signing operation is not using an out of range count value
217         t1 = (UINT16)currentCount;
218
219         // If the lower bits of c are greater or equal to the lower bits of t1
220         // then the upper bits of t1 must be one more than the upper bits
221         // of c
222         if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
223             // Since the counter is behind, reduce the current count
224             currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
225
226         t1 = (UINT16)currentCount;
227         if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
228             return FALSE;
229         // set the counter to the value that was
230         // present when the commitment was made
231         currentCount = (currentCount & 0xfffffffffff0000) | *c;
232     }
233     // Marshal the count value to a TPM2B buffer for the KDF
234     cntr.t.size = sizeof(currentCount);
235     UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
236
237     // Now can do the KDF to create the random value for the signing operation
238     // During the creation process, we may generate an r that does not meet the
239     // requirements of the random value.
240     // want to generate a new r.
241     r->t.size = n.t.size;
242
243     for(iterations = 1; iterations < 1000000;)
244     {
245         int i;
246         CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG,
247                 &gr.commitNonce.b,
248                 COMMIT_STRING,
249                 &name->b,
250                 &cntr.b,
251                 n.t.size * 8,
252                 r->t.buffer,
253                 &iterations,
254                 FALSE);
255
256         // "random" value must be less than the prime
257         if(UnsignedCompareB(r->b.size, r->b.buffer, n.t.size, n.t.buffer) >= 0)
258             continue;
259
260         // in this implementation it is required that at least bit
261         // in the upper half of the number be set
262         for(i = n.t.size / 2; i >= 0; i--)
263             if(r->b.buffer[i] != 0)
264                 return TRUE;
265     }
266     return FALSE;
267 }
268
269 /*** CryptCommit()
270 // This function is called when the count value is committed. The 'gr.commitArray'

```

```

271 // value associated with the current count value is SET and g_commitCounter is
272 // incremented. The low-order 16 bits of old value of the counter is returned.
273 UINT16
274 CryptCommit(void)
275 {
276     UINT16 oldCount = (UINT16)gr.commitCounter;
277     gr.commitCounter++;
278     SET_BIT(oldCount & COMMIT_INDEX_MASK, gr.commitArray);
279     return oldCount;
280 }
281
282 /*** CryptEndCommit()
283 // This function is called when the signing operation using the committed value
284 // is completed. It clears the gr.commitArray bit associated with the count
285 // value so that it can't be used again.
286 void CryptEndCommit(UINT16 c // IN: the counter value of the commitment
287 )
288 {
289     ClearBit((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
290 }
291
292 /*** CryptEccGetParameters()
293 // This function returns the ECC parameter details of the given curve.
294 // Return Type: BOOL
295 //     TRUE(1)          success
296 //     FALSE(0)         unsupported ECC curve ID
297 BOOL CryptEccGetParameters(
298     TPM_ECC_CURVE curveId, // IN: ECC curve ID
299     TPMS_ALGORITHM_DETAIL_ECC* parameters // OUT: ECC parameters
300 )
301 {
302     const TPM_ECC_CURVE_METADATA* curve = CryptEccGetParametersByCurveId(curveId);
303     BOOL found = curve != NULL;
304
305     if(found)
306     {
307         parameters->curveID = curve->curveId;
308         parameters->keySize = curve->keySizeBits;
309         parameters->kdf = curve->kdf;
310         parameters->sign = curve->sign;
311         // BnTo2B(data->prime, &parameters->p.b, 0);
312         found = found
313             && TpmMath_IntTo2B(ExtEcc_CurveGetPrime(curveId),
314                               &parameters->p.b,
315                               parameters->p.t.size);
316         found = found
317             && TpmMath_IntTo2B(ExtEcc_CurveGet_a(curveId), &parameters->a.b, 0);
318         found = found
319             && TpmMath_IntTo2B(ExtEcc_CurveGet_b(curveId), &parameters->b.b, 0);
320         found = found
321             && TpmMath_IntTo2B(ExtEcc_CurveGetGx(curveId),
322                               &parameters->gX.b,
323                               parameters->p.t.size);
324         found = found
325             && TpmMath_IntTo2B(ExtEcc_CurveGetGy(curveId),
326                               &parameters->gY.b,
327                               parameters->p.t.size);
328         // BnTo2B(data->base.x, &parameters->gX.b, 0);
329         // BnTo2B(data->base.y, &parameters->gY.b, 0);
330         found =
331             found
332             && TpmMath_IntTo2B(ExtEcc_CurveGetOrder(curveId), &parameters->n.b, 0);
333         found =
334             found
335             && TpmMath_IntTo2B(ExtEcc_CurveGetCofactor(curveId), &parameters->h.b, 0);
336         // if we got into this IF but failed to get a parameter from the external

```

```

337     // library, our crypto systems are broken; enter failure mode.
338     if(!found)
339     {
340         FAIL(FATAL_ERROR_MATHLIBRARY);
341     }
342 }
343 return found;
344 }
345
346 /*** TpmEcc_IsValidPrivateEcc()
347 // Checks that 0 < 'x' < 'q'
348 BOOL TpmEcc_IsValidPrivateEcc(const Crypt_Int* x, // IN: private key to check
349                               const Crypt_EccCurve* E // IN: the curve to check
350 )
351 {
352     BOOL retVal;
353     retVal =
354         (!ExtMath_IsZero(x)
355          && (ExtMath_UnsignedCmp(x, ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E)))
356             < 0));
357     return retVal;
358 }
359
360 LIB_EXPORT BOOL CryptEccIsValidPrivateKey(TPM2B_ECC_PARAMETER* d,
361                                           TPM_ECC_CURVE curveId)
362 {
363     CRYPT_INT_INITIALIZED(bnD, MAX_ECC_PARAMETER_BYTES * 8, d);
364     return !ExtMath_IsZero(bnD)
365         && (ExtMath_UnsignedCmp(bnD, ExtEcc_CurveGetOrder(curveId)) < 0);
366 }
367
368 /*** TpmEcc_PointMult()
369 // This function does a point multiply of the form 'R' = ['d']'S' + ['u']'Q' where the
370 // parameters are Crypt_Int* values. If 'S' is NULL and d is not NULL, then it
371 // computes
372 // 'R' = ['d']'G' + ['u']'Q' or just 'R' = ['d']'G' if 'u' and 'Q' are NULL.
373 // If 'skipChecks' is TRUE, then the function will not verify that the inputs are
374 // correct for the domain. This would be the case when the values were created by the
375 // CryptoEngine code.
376 // It will return TPM_RC_NO_RESULT if the resulting point is the point at infinity.
377 // Return Type: TPM_RC
378 // TPM_RC_NO_RESULT result of multiplication is a point at infinity
379 // TPM_RC_ECC_POINT 'S' or 'Q' is not on the curve
380 // TPM_RC_VALUE 'd' or 'u' is not < n
381 TPM_RC
382 TpmEcc_PointMult(Crypt_Point* R, // OUT: computed point
383                  const Crypt_Point* S, // IN: optional point to multiply by 'd'
384                  const Crypt_Int* d, // IN: scalar for [d]S or [d]G
385                  const Crypt_Point* Q, // IN: optional second point
386                  const Crypt_Int* u, // IN: optional second scalar
387                  const Crypt_EccCurve* E // IN: curve parameters
388 )
389 {
390     BOOL OK;
391     //
392     TEST(TPM_ALG_ECDH);
393
394     // Need one scalar
395     OK = (d != NULL || u != NULL);
396
397     // If S is present, then d has to be present. If S is not
398     // present, then d may or may not be present
399     OK = OK && (((S == NULL) == (d == NULL)) || (d != NULL));
400
401     // either both u and Q have to be provided or neither can be provided (don't
402     // know what to do if only one is provided.

```

```

402     OK = OK && ((u == NULL) == (Q == NULL));
403
404     OK = OK && (E != NULL);
405     if(!OK)
406         return TPM_RC_VALUE;
407
408     OK = (S == NULL) || ExtEcc_IsPointOnCurve(S, E);
409     OK = OK && ((Q == NULL) || ExtEcc_IsPointOnCurve(Q, E));
410     if(!OK)
411         return TPM_RC_ECC_POINT;
412
413     if((d != NULL) && (S == NULL))
414         S = ExtEcc_CurveGetG(ExtEcc_CurveGetCurveId(E));
415     // If only one scalar, don't need Shamir's trick
416     if((d == NULL) || (u == NULL))
417     {
418         if(d == NULL)
419             OK = ExtEcc_PointMultiply(R, Q, u, E);
420         else
421             OK = ExtEcc_PointMultiply(R, S, d, E);
422     }
423     else
424     {
425         OK = ExtEcc_PointMultiplyAndAdd(R, S, d, Q, u, E);
426     }
427     return (OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT);
428 }
429
430 /**TpmEcc_GenPrivateScalar()
431 // This function gets random values that are the size of the key plus 64 bits. The
432 // value is reduced (mod ('q' - 1)) and incremented by 1 ('q' is the order of the
433 // curve. This produces a value ('d') such that 1 <= 'd' < 'q'. This is the method
434 // of FIPS 186-4 Section B.4.1 "Key Pair Generation Using Extra Random Bits".
435 // Return Type: BOOL
436 //     TRUE(1)      success
437 //     FALSE(0)     failure generating private key
438 BOOL TpmEcc_GenPrivateScalar(
439     Crypt_Int*      dOut, // OUT: the qualified random value
440     const Crypt_EccCurve* E, // IN: curve for which the private key
441                             // needs to be appropriate
442     RAND_STATE* rand // IN: state for DRBG
443 )
444 {
445     TPM_ECC_CURVE    curveId = ExtEcc_CurveGetCurveId(E);
446     const Crypt_Int* order    = ExtEcc_CurveGetOrder(curveId);
447     BOOL             OK;
448     UINT32           orderBits = ExtMath_SizeInBits(order);
449     UINT32           orderBytes = BITS_TO_BYTES(orderBits);
450     CRYPT_INT_VAR(bnExtraBits, MAX_ECC_KEY_BITS + 64);
451     CRYPT_INT_VAR(nMinus1, MAX_ECC_KEY_BITS);
452     //
453     OK = TpmMath_GetRandomInteger(bnExtraBits, (orderBytes * 8) + 64, rand);
454     OK = OK && ExtMath_SubtractWord(nMinus1, order, 1);
455     OK = OK && ExtMath_Mod(bnExtraBits, nMinus1);
456     OK = OK && ExtMath_AddWord(dOut, bnExtraBits, 1);
457     return OK && !g_inFailureMode;
458 }
459
460 /**TpmEcc_GenerateKeyPair()
461 // This function gets a private scalar from the source of random bits and does
462 // the point multiply to get the public key.
463 BOOL TpmEcc_GenerateKeyPair(Crypt_Int*      bnD, // OUT: private scalar
464                             Crypt_Point*    ecQ, // OUT: public point
465                             const Crypt_EccCurve* E, // IN: curve for the point
466                             RAND_STATE* rand // IN: DRBG state to use
467 )

```

```

468 {
469     BOOL OK = FALSE;
470     // Get a private scalar
471     OK = TpmEcc_GenPrivateScalar(bnD, E, rand);
472
473     // Do a point multiply
474     OK = OK && ExtEcc_PointMultiply(ecQ, NULL, bnD, E);
475     return OK;
476 }
477
478 /***CryptEccNewKeyPair(***)
479 // This function creates an ephemeral ECC. It is ephemeral in that
480 // is expected that the private part of the key will be discarded
481 LIB_EXPORT TPM_RC CryptEccNewKeyPair(
482     TPMS_ECC_POINT*    Qout,    // OUT: the public point
483     TPM2B_ECC_PARAMETER* dOut,  // OUT: the private scalar
484     TPM_ECC_CURVE      curveId  // IN: the curve for the key
485 )
486 {
487     CRYPT_CURVE_INITIALIZED(E, curveId);
488     CRYPT_POINT_VAR(ecQ);
489     CRYPT_ECC_NUM(bnD);
490     BOOL OK;
491
492     if(E == NULL)
493         return TPM_RC_CURVE;
494
495     TEST(TPM_ALG_ECDH);
496     OK = TpmEcc_GenerateKeyPair(bnD, ecQ, E, NULL);
497     if(OK)
498     {
499         TpmEcc_PointTo2B(Qout, ecQ, E);
500         TpmMath_IntTo2B(bnD, &dOut->b, Qout->x.t.size);
501     }
502     else
503     {
504         Qout->x.t.size = Qout->y.t.size = dOut->t.size = 0;
505     }
506     CRYPT_CURVE_FREE(E);
507     return OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
508 }
509
510 /*** CryptEccPointMultiply()
511 // This function computes 'R' := ['dIn']'G' + ['uIn']'QIn'. Where 'dIn' and
512 // 'uIn' are scalars, 'G' and 'QIn' are points on the specified curve and 'G' is the
513 // default generator of the curve.
514 //
515 // The 'xOut' and 'yOut' parameters are optional and may be set to NULL if not
516 // used.
517 //
518 // It is not necessary to provide 'uIn' if 'QIn' is specified but one of 'uIn' and
519 // 'dIn' must be provided. If 'dIn' and 'QIn' are specified but 'uIn' is not
520 // provided, then 'R' = ['dIn']'QIn'.
521 //
522 // If the multiply produces the point at infinity, the TPM_RC_NO_RESULT is returned.
523 //
524 // The sizes of 'xOut' and 'yOut' will be set to be the size of the degree of
525 // the curve
526 //
527 // It is a fatal error if 'dIn' and 'uIn' are both unspecified (NULL) or if 'Qin'
528 // or 'Rout' is unspecified.
529 //
530 // Return Type: TPM_RC
531 //     TPM_RC_ECC_POINT          the point 'Pin' or 'Qin' is not on the curve
532 //     TPM_RC_NO_RESULT          the product point is at infinity
533 //     TPM_RC_CURVE              bad curve

```



```

534 //      TPM_RC_VALUE          'dIn' or 'uIn' out of range
535 //
536 LIB_EXPORT TPM_RC CryptEccPointMultiply(
537     TPMS_ECC_POINT*      Rout,      // OUT: the product point R
538     TPM_ECC_CURVE        curveId,   // IN: the curve to use
539     TPMS_ECC_POINT*      Pin,       // IN: first point (can be null)
540     TPM2B_ECC_PARAMETER* dIn,       // IN: scalar value for [dIn]Qin
541                                     // the Pin
542     TPMS_ECC_POINT*      Qin,       // IN: point Q
543     TPM2B_ECC_PARAMETER* uIn        // IN: scalar value for the multiplier
544                                     // of Q
545 )
546 {
547     CRYPT_CURVE_INITIALIZED(E, curveId);
548     CRYPT_POINT_INITIALIZED(ecP, Pin);
549     CRYPT_ECC_INITIALIZED(bnD, dIn); // If dIn is null, then bnD is null
550     CRYPT_ECC_INITIALIZED(bnU, uIn);
551     CRYPT_POINT_INITIALIZED(ecQ, Qin);
552     CRYPT_POINT_VAR(ecR);
553     TPM_RC retVal;
554     //
555     retVal = TpmEcc_PointMult(ecR, ecP, bnD, ecQ, bnU, E);
556
557     if(retVal == TPM_RC_SUCCESS)
558         TpmEcc_PointTo2B(Rout, ecR, E);
559     else
560         ClearPoint2B(Rout);
561     CRYPT_CURVE_FREE(E);
562     return retVal;
563 }
564
565 /*** CryptEccIsPointOnCurve()
566 // This function is used to test if a point is on a defined curve. It does this
567 // by checking that 'y'^2 mod 'p' = 'x'^3 + 'a'*'x' + 'b' mod 'p'.
568 //
569 // It is a fatal error if 'Q' is not specified (is NULL).
570 // Return Type: BOOL
571 //      TRUE(1)      point is on curve
572 //      FALSE(0)     point is not on curve or curve is not supported
573 LIB_EXPORT BOOL CryptEccIsPointOnCurve(
574     TPM_ECC_CURVE        curveId,   // IN: the curve selector
575     TPMS_ECC_POINT*      Qin        // IN: the point.
576 )
577 {
578     CRYPT_CURVE_INITIALIZED(E, curveId);
579     CRYPT_POINT_INITIALIZED(ecQ, Qin);
580     BOOL OK;
581     //
582     pAssert(Qin != NULL);
583     OK = (E != NULL && (ExtEcc_IsPointOnCurve(ecQ, E)));
584     return OK;
585 }
586
587 /*** CryptEccGenerateKey()
588 // This function generates an ECC key pair based on the input parameters.
589 // This routine uses KDFa to produce candidate numbers. The method is according
590 // to FIPS 186-3, section B.1.2 "Key Pair Generation by Testing Candidates."
591 // According to the method in FIPS 186-3, the resulting private value 'd' should be
592 // 1 <= 'd' < 'n' where 'n' is the order of the base point.
593 //
594 // It is a fatal error if 'Qout', 'dOut', is not provided (is NULL).
595 //
596 // If the curve is not supported
597 // If 'seed' is not provided, then a random number will be used for the key
598 // Return Type: TPM_RC
599 //      TPM_RC_CURVE          curve is not supported

```

```

600 //      TPM_RC_NO_RESULT      could not verify key with signature (FIPS only)
601 LIB_EXPORT TPM_RC CryptEccGenerateKey(
602     TPMT_PUBLIC* publicArea,    // IN/OUT: The public area template for
603                                 //      the new key. The public key
604                                 //      area will be replaced computed
605                                 //      ECC public key
606     TPMT_SENSITIVE* sensitive,  // OUT: the sensitive area will be
607                                 //      updated to contain the private
608                                 //      ECC key and the symmetric
609                                 //      encryption key
610     RAND_STATE* rand            // IN: if not NULL, the deterministic
611                                 //      RNG state
612 )
613 {
614     CRYPT_CURVE_INITIALIZED(E, publicArea->parameters.eccDetail.curveID);
615     CRYPT_ECC_NUM(bnD);
616     CRYPT_POINT_VAR(ecQ);
617     BOOL OK;
618     TPM_RC retVal;
619     //
620     TEST(TPM_ALG_ECDSA); // ECDSA is used to verify each key
621
622     // Validate parameters
623     if(E == NULL)
624         ERROR_EXIT(TPM_RC_CURVE);
625
626     publicArea->unique.ecc.x.t.size = 0;
627     publicArea->unique.ecc.y.t.size = 0;
628     sensitive->sensitive.ecc.t.size = 0;
629
630     OK = TpmEcc_GenerateKeyPair(bnD, ecQ, E, rand);
631     if(OK)
632     {
633         TpmEcc_PointTo2B(&publicArea->unique.ecc, ecQ, E);
634         TpmMath_IntTo2B(
635             bnD, &sensitive->sensitive.ecc.b, publicArea->unique.ecc.x.t.size);
636     }
637 # if FIPS_COMPLIANT
638     // See if PWCT is required
639     if(OK && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
640     {
641         CRYPT_ECC_NUM(bnT);
642         CRYPT_ECC_NUM(bnS);
643         TPM2B_DIGEST digest;
644         //
645         TEST(TPM_ALG_ECDSA);
646         digest.t.size = MIN(sensitive->sensitive.ecc.t.size, sizeof(digest.t.buffer));
647         // Get a random value to sign using the built in DRBG state
648         DRBG_Generate(NULL, digest.t.buffer, digest.t.size);
649         if(g_inFailureMode)
650             return TPM_RC_FAILURE;
651         TpmEcc_SignEcdsa(bnT, bnS, E, bnD, &digest, NULL);
652         // and make sure that we can validate the signature
653         OK = TpmEcc_ValidateSignatureEcdsa(bnT, bnS, E, ecQ, &digest)
654             == TPM_RC_SUCCESS;
655     }
656 # endif
657     retVal = (OK) ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
658 Exit:
659     CRYPT_CURVE_FREE(E);
660     return retVal;
661 }
662
663 #endif // ALG_ECC

```

7.142 /tpm/src/crypt/CryptEccSignature.c

```

1  /** Includes and Defines
2  #include "Tpm.h"
3  #include "TpmEcc_Signature_ECDSA_fp.h"
4  #include "TpmEcc_Signature_ECDAAs_fp.h"
5  #include "TpmEcc_Signature_Schnorr_fp.h"
6  #include "TpmEcc_Signature_SM2_fp.h"
7  #include "TpmEcc_Util_fp.h"
8  #include "TpmMath_Util_fp.h"
9  #include "CryptEccSignature_fp.h"
10
11 #if ALG_ECC
12
13 /** Utility Functions
14
15 /** Signing Functions
16
17 /** CryptEccSign()
18 // This function is the dispatch function for the various ECC-based
19 // signing schemes.
20 // There is a bit of ugliness to the parameter passing. In order to test this,
21 // we sometime would like to use a deterministic RNG so that we can get the same
22 // signatures during testing. The easiest way to do this for most schemes is to
23 // pass in a deterministic RNG and let it return canned values during testing.
24 // There is a competing need for a canned parameter to use in ECDAAs. To accommodate
25 // both needs with minimal fuss, a special type of RAND_STATE is defined to carry
26 // the address of the commit value. The setup and handling of this is not very
27 // different for the caller than what was in previous versions of the code.
28 // Return Type: TPM_RC
29 //      TPM_RC_SCHEME          'scheme' is not supported
30 LIB_EXPORT TPM_RC CryptEccSign(TPMT_SIGNATURE* signature, // OUT: signature
31                                OBJECT* signKey, // IN: ECC key to sign the hash
32                                const TPM2B_DIGEST* digest, // IN: digest to sign
33                                TPMT_ECC_SCHEME* scheme, // IN: signing scheme
34                                RAND_STATE* rand)
35 {
36     CRYPT_CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
37     CRYPT_ECC_INITIALIZED(bnD, &signKey->sensitive.sensitive.ecc.b);
38     CRYPT_ECC_NUM(bnR);
39     CRYPT_ECC_NUM(bnS);
40     TPM_RC retVal = TPM_RC_SCHEME;
41     //
42     NOT_REFERENCED(scheme);
43     if(E == NULL)
44         ERROR_EXIT(TPM_RC_VALUE);
45     signature->signature.ecdaa.signatureR.t.size =
46         sizeof(signature->signature.ecdaa.signatureR.t.buffer);
47     signature->signature.ecdaa.signatureS.t.size =
48         sizeof(signature->signature.ecdaa.signatureS.t.buffer);
49     TEST(signature->sigAlg);
50     switch(signature->sigAlg)
51     {
52     case TPM_ALG_ECDSA:
53         retVal = TpmEcc_SignEcDSA(bnR, bnS, E, bnD, digest, rand);
54         break;
55 # if ALG_ECDAAs
56     case TPM_ALG_ECDAAs:
57         retVal = TpmEcc_SignEcdaa(&signature->signature.ecdaa.signatureR,
58                                   bnS,
59                                   E,
60                                   bnD,
61                                   digest,
62                                   scheme,
63                                   signKey,
64                                   rand);

```

```

65         bnR      = NULL;
66         break;
67 # endif
68 # if ALG_ECSCHNORR
69     case TPM_ALG_ECSCHNORR:
70         retVal = TpmEcc_SignEcSchnorr(
71             bnR, bnS, E, bnD, digest, signature->signature.ecschnorr.hash, rand);
72         break;
73 # endif
74 # if ALG_SM2
75     case TPM_ALG_SM2:
76         retVal = TpmEcc_SignEcSm2(bnR, bnS, E, bnD, digest, rand);
77         break;
78 # endif
79     default:
80         break;
81 }
82 // If signature generation worked, convert the results.
83 if(retVal == TPM_RC_SUCCESS)
84 {
85     NUMBYTES orderBytes = (NUMBYTES)BITS_TO_BYTES(
86         ExtMath_SizeInBits(ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E))));
87     if(bnR != NULL)
88         TpmMath_IntTo2B(
89             bnR, &signature->signature.ecdaa.signatureR.b, orderBytes);
90     if(bnS != NULL)
91         TpmMath_IntTo2B(
92             bnS, &signature->signature.ecdaa.signatureS.b, orderBytes);
93 }
94 Exit:
95     CRYPT_CURVE_FREE(E);
96     return retVal;
97 }
98
99 //***** Signature Validation *****
100
101 /*** CryptEccValidateSignature()
102 // This function validates an EcDsa or EcSchnorr signature.
103 // The point 'Qin' needs to have been validated to be on the curve of 'curveId'.
104 // Return Type: TPM_RC
105 // TPM_RC_SIGNATURE not a valid signature
106 LIB_EXPORT TPM_RC CryptEccValidateSignature(
107     TPMT_SIGNATURE* signature, // IN: signature to be verified
108     OBJECT* signKey, // IN: ECC key signed the hash
109     const TPM2B_DIGEST* digest // IN: digest that was signed
110 )
111 {
112     CRYPT_CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveId);
113     CRYPT_ECC_NUM(bnR);
114     CRYPT_ECC_NUM(bnS);
115     CRYPT_POINT_INITIALIZED(ecQ, &signKey->publicArea.unique.ecc);
116     const Crypt_Int* order;
117     TPM_RC retVal;
118
119     if(E == NULL)
120         ERROR_EXIT(TPM_RC_VALUE);
121
122     order = ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E));
123
124     // // Make sure that the scheme is valid
125     switch(signature->sigAlg)
126     {
127         case TPM_ALG_ECDSA:
128 # if ALG_ECSCHNORR
129             case TPM_ALG_ECSCHNORR:
130 # endif

```

```

131 # if ALG_SM2
132     case TPM_ALG_SM2:
133 # endif
134     break;
135     default:
136         ERROR_EXIT(TPM_RC_SCHEME);
137         break;
138 }
139 // Can convert r and s after determining that the scheme is an ECC scheme. If
140 // this conversion doesn't work, it means that the unmarshaling code for
141 // an ECC signature is broken.
142 TpmMath_IntFrom2B(bnR, &signature->signature.ecdsa.signatureR.b);
143 TpmMath_IntFrom2B(bnS, &signature->signature.ecdsa.signatureS.b);
144
145 // r and s have to be greater than 0 but less than the curve order
146 if(ExtMath_IsZero(bnR) || ExtMath_IsZero(bnS))
147     ERROR_EXIT(TPM_RC_SIGNATURE);
148 if((ExtMath_UnsignedCmp(bnS, order) >= 0)
149    || (ExtMath_UnsignedCmp(bnR, order) >= 0))
150     ERROR_EXIT(TPM_RC_SIGNATURE);
151
152 switch(signature->sigAlg)
153 {
154     case TPM_ALG_ECDSA:
155         retVal = TpmEcc_ValidateSignatureEcDSA(bnR, bnS, E, ecQ, digest);
156         break;
157
158 # if ALG_EC Schnorr
159     case TPM_ALG_EC Schnorr:
160         retVal = TpmEcc_ValidateSignatureEcSchnorr(
161             bnR, bnS, signature->signature.any.hashAlg, E, ecQ, digest);
162         break;
163 # endif
164 # if ALG_SM2
165     case TPM_ALG_SM2:
166         retVal = TpmEcc_ValidateSignatureEcSm2(bnR, bnS, E, ecQ, digest);
167         break;
168 # endif
169     default:
170         FAIL(FATAL_ERROR_INTERNAL);
171 }
172 Exit:
173     CRYPT_CURVE_FREE(E);
174     return retVal;
175 }
176
177 /***CryptEccCommitCompute()
178 // This function performs the point multiply operations required by TPM2_Commit.
179 //
180 // If 'B' or 'M' is provided, they must be on the curve defined by 'curveId'. This
181 // routine does not check that they are on the curve and results are unpredictable
182 // if they are not.
183 //
184 // It is a fatal error if 'r' is NULL. If 'B' is not NULL, then it is a
185 // fatal error if 'd' is NULL or if 'K' and 'L' are both NULL.
186 // If 'M' is not NULL, then it is a fatal error if 'E' is NULL.
187 //
188 // Return Type: TPM_RC
189 //     TPM_RC_NO_RESULT      if 'K', 'L' or 'E' was computed to be the point
190 //                           at infinity
191 //     TPM_RC_CANCELED       a cancel indication was asserted during this
192 //                           function
193 LIB_EXPORT TPM_RC CryptEccCommitCompute(
194     TPMS_ECC_POINT*      K,      // OUT: [d]B or [r]Q
195     TPMS_ECC_POINT*      L,      // OUT: [x]B
196     TPMS_ECC_POINT*      E,      // OUT: [x]M

```

```

197     TPM_ECC_CURVE      curveId,    // IN: the curve for the computations
198     TPMS_ECC_POINT*    M,          // IN: M (optional)
199     TPMS_ECC_POINT*    B,          // IN: B (optional)
200     TPM2B_ECC_PARAMETER* d,        // IN: d (optional)
201     TPM2B_ECC_PARAMETER* r          // IN: the computed r value (required)
202 )
203 {
204     // Normally initialize E as the curve, but
205     // E means something else in this function
206     CRYPT_CURVE_INITIALIZED(curve, curveId);
207     CRYPT_ECC_INITIALIZED(bnR, r);
208     TPM_RC retVal = TPM_RC_SUCCESS;
209     //
210     // Validate that the required parameters are provided.
211     // Note: E has to be provided if computing E := [r]Q or E := [r]M. Will do
212     // E := [r]Q if both M and B are NULL.
213     pAssert(r != NULL && E != NULL);
214
215     // Initialize the output points in case they are not computed
216     ClearPoint2B(K);
217     ClearPoint2B(L);
218     ClearPoint2B(E);
219
220     // Sizes of the r parameter may not be zero
221     pAssert(r->t.size > 0);
222
223     // If B is provided, compute K=[d]B and L=[r]B
224     if(B != NULL)
225     {
226         CRYPT_ECC_INITIALIZED(bnD, d);
227         CRYPT_POINT_INITIALIZED(pB, B);
228         CRYPT_POINT_VAR(pK);
229         CRYPT_POINT_VAR(pL);
230         //
231         pAssert(d != NULL && K != NULL && L != NULL);
232
233         if(!ExtEcc_IsPointOnCurve(pB, curve))
234             ERROR_EXIT(TPM_RC_VALUE);
235         // do the math for K = [d]B
236         if((retVal = TpmEcc_PointMult(pK, pB, bnD, NULL, NULL, curve))
237             != TPM_RC_SUCCESS)
238             goto Exit;
239         // Convert BN K to TPM2B K
240         TpmEcc_PointTo2B(K, pK, curve);
241         // compute L= [r]B after checking for cancel
242         if(_plat_IsCanceled())
243             ERROR_EXIT(TPM_RC_CANCELED);
244         // compute L = [r]B
245         if(!TpmEcc_IsValidPrivateEcc(bnR, curve))
246             ERROR_EXIT(TPM_RC_VALUE);
247         if((retVal = TpmEcc_PointMult(pL, pB, bnR, NULL, NULL, curve))
248             != TPM_RC_SUCCESS)
249             goto Exit;
250         // Convert BN L to TPM2B L
251         TpmEcc_PointTo2B(L, pL, curve);
252     }
253     if((M != NULL) || (B == NULL))
254     {
255         CRYPT_POINT_INITIALIZED(pM, M);
256         CRYPT_POINT_VAR(pE);
257         //
258         // Make sure that a place was provided for the result
259         pAssert(E != NULL);
260
261         // if this is the third point multiply, check for cancel first
262         if((B != NULL) && _plat_IsCanceled())

```



```

263         ERROR_EXIT(TPM_RC_CANCELED);
264
265         // If M provided, then pM will not be NULL and will compute E = [r]M.
266         // However, if M was not provided, then pM will be NULL and E = [r]G
267         // will be computed
268         if((retVal = TpmEcc_PointMult(pE, pM, bnR, NULL, NULL, curve))
269             != TPM_RC_SUCCESS)
270             goto Exit;
271         // Convert E to 2B format
272         TpmEcc_PointTo2B(E, pE, curve);
273     }
274 Exit:
275     CRYPT_CURVE_FREE(curve);
276     return retVal;
277 }
278
279 #endif // ALG_ECC

```

7.143 /tpm/src/crypt/CryptHash.c

```

1  /** Description
2  //
3  // This file contains implementation of cryptographic functions for hashing.
4  //
5  /** Includes, Defines, and Types
6
7  #define _CRYPT_HASH_C_
8  #include "Tpm.h"
9  #include "CryptHash_fp.h"
10 #include "CryptHash.h"
11 #include "OIDs.h"
12
13 // Instance each of the hash descriptors based on the implemented algorithms
14 FOR_EACH_HASH(HASH_DEF_TEMPLATE)
15 // Instance a 'null' def.
16 HASH_DEF NULL_Def = {{0}};
17
18 // Create a table of pointers to the defined hash definitions
19 #define HASH_DEF_ENTRY(HASH, Hash) &Hash##_Def,
20 PHASH_DEF HashDefArray[] = {
21     // for each implemented HASH, expands to: &HASH_Def,
22     FOR_EACH_HASH(HASH_DEF_ENTRY) & NULL_Def};
23 #undef HASH_DEF_ENTRY
24
25 /** Obligatory Initialization Functions
26
27 /** CryptHashInit()
28 // This function is called by _TPM_Init do perform the initialization operations for
29 // the library.
30 BOOL CryptHashInit(void)
31 {
32     LibHashInit();
33     return TRUE;
34 }
35
36 /** CryptHashStartup()
37 // This function is called by TPM2_Startup(). It checks that the size of the
38 // HashDefArray is consistent with the HASH_COUNT.
39 BOOL CryptHashStartup(void)
40 {
41     int i = sizeof(HashDefArray) / sizeof(PHASH_DEF) - 1;
42     return (i == HASH_COUNT);
43 }
44
45 /** Hash Information Access Functions

```

```

46  /*** Introduction
47  // These functions provide access to the hash algorithm description information.
48
49  /*** CryptGetHashDef()
50  // This function accesses the hash descriptor associated with a hash a
51  // algorithm. The function returns a pointer to a 'null' descriptor if hashAlg is
52  // TPM_ALG_NULL or not a defined algorithm.
53  PHASH_DEF
54  CryptGetHashDef(TPM_ALG_ID hashAlg)
55  {
56  #define GET_DEF(HASH, Hash) \
57      case ALG_##HASH##_VALUE: \
58          return &Hash##_Def;
59      switch(hashAlg)
60      {
61          FOR_EACH_HASH(GET_DEF)
62          default:
63              return &NULL_Def;
64      }
65  #undef GET_DEF
66  }
67
68  /*** CryptHashIsValidAlg()
69  // This function tests to see if an algorithm ID is a valid hash algorithm. If
70  // flag is true, then TPM_ALG_NULL is a valid hash.
71  // Return Type: BOOL
72  //     TRUE(1)      hashAlg is a valid, implemented hash on this TPM
73  //     FALSE(0)     hashAlg is not valid for this TPM
74  BOOL CryptHashIsValidAlg(TPM_ALG_ID hashAlg, // IN: the algorithm to check
75                          BOOL flag // IN: TRUE if TPM_ALG_NULL is to be treated
76                                // as a valid hash
77  )
78  {
79      if(hashAlg == TPM_ALG_NULL)
80          return flag;
81      return CryptGetHashDef(hashAlg) != &NULL_Def;
82  }
83
84  /*** CryptHashGetAlgByIndex()
85  // This function is used to iterate through the hashes. TPM_ALG_NULL
86  // is returned for all indexes that are not valid hashes.
87  // If the TPM implements 3 hashes, then an 'index' value of 0 will
88  // return the first implemented hash and an 'index' of 2 will return the
89  // last. All other index values will return TPM_ALG_NULL.
90  //
91  // Return Type: TPM_ALG_ID
92  // TPM_ALG_***      a hash algorithm
93  // TPM_ALG_NULL     this can be used as a stop value
94  LIB_EXPORT TPM_ALG_ID CryptHashGetAlgByIndex(UINT32 index // IN: the index
95  )
96  {
97      TPM_ALG_ID hashAlg;
98      if(index >= HASH_COUNT)
99          hashAlg = TPM_ALG_NULL;
100      else
101          hashAlg = HashDefArray[index]->hashAlg;
102      return hashAlg;
103  }
104
105  /*** CryptHashGetDigestSize()
106  // Returns the size of the digest produced by the hash. If 'hashAlg' is not a hash
107  // algorithm, the TPM will FAIL.
108  // Return Type: UINT16
109  //     0      TPM_ALG_NULL
110  //     > 0    the digest size
111  //

```

```

112 LIB_EXPORT UINT16 CryptHashGetDigestSize(
113     TPM_ALG_ID hashAlg // IN: hash algorithm to look up
114 )
115 {
116     return CryptGetHashDef(hashAlg)->digestSize;
117 }
118
119 /*** CryptHashGetBlockSize()
120 // Returns the size of the block used by the hash. If 'hashAlg' is not a hash
121 // algorithm, the TPM will FAIL.
122 // Return Type: UINT16
123 // 0          TPM_ALG_NULL
124 // > 0       the digest size
125 //
126 LIB_EXPORT UINT16 CryptHashGetBlockSize(
127     TPM_ALG_ID hashAlg // IN: hash algorithm to look up
128 )
129 {
130     return CryptGetHashDef(hashAlg)->blockSize;
131 }
132
133 /*** CryptHashGetOid()
134 // This function returns a pointer to DER-encoded OID for a hash algorithm. All OIDs
135 // are full OID values including the Tag (0x06) and length byte.
136 LIB_EXPORT const BYTE* CryptHashGetOid(TPM_ALG_ID hashAlg)
137 {
138     return CryptGetHashDef(hashAlg)->OID;
139 }
140
141 /*** CryptHashGetContextAlg()
142 // This function returns the hash algorithm associated with a hash context.
143 TPM_ALG_ID
144 CryptHashGetContextAlg(PHASH_STATE state // IN: the context to check
145 )
146 {
147     return state->hashAlg;
148 }
149
150 /*** State Import and Export
151
152 /*** CryptHashCopyState
153 // This function is used to clone a HASH_STATE.
154 LIB_EXPORT void CryptHashCopyState(HASH_STATE* out, // OUT: destination of the state
155     const HASH_STATE* in // IN: source of the state
156 )
157 {
158     pAssert(out->type == in->type);
159     out->hashAlg = in->hashAlg;
160     out->def = in->def;
161     if(in->hashAlg != TPM_ALG_NULL)
162     {
163         HASH_STATE_COPY(out, in);
164     }
165     if(in->type == HASH_STATE_HMAC)
166     {
167         const HMAC_STATE* hIn = (HMAC_STATE*)in;
168         HMAC_STATE* hOut = (HMAC_STATE*)out;
169         hOut->hmacKey = hIn->hmacKey;
170     }
171     return;
172 }
173
174 /*** CryptHashExportState()
175 // This function is used to export a hash or HMAC hash state. This function
176 // would be called when preparing to context save a sequence object.
177 void CryptHashExportState(

```

```

178     PCHASH_STATE internalFmt,          // IN: the hash state formatted for use by
179                                         // library
180     PEXPORT_HASH_STATE externalFmt    // OUT: the exported hash state
181 )
182 {
183     BYTE* outBuf = (BYTE*)externalFmt;
184     //
185     MUST_BE(sizeof(HASH_STATE) <= sizeof(EXPORT_HASH_STATE));
186     // the following #define is used to move data from an aligned internal data
187     // structure to a byte buffer (external format data.
188 #define CopyToOffset(value) \
189     memcpy(&outBuf[offsetof(HASH_STATE, value)], \
190           &internalFmt->value, \
191           sizeof(internalFmt->value))
192     // Copy the hashAlg
193     CopyToOffset(hashAlg);
194     CopyToOffset(type);
195 #ifdef HASH_STATE_SMAC
196     if(internalFmt->type == HASH_STATE_SMAC)
197     {
198         memcpy(outBuf, internalFmt, sizeof(HASH_STATE));
199         return;
200     }
201 #endif
202     if(internalFmt->type == HASH_STATE_HMAC)
203     {
204         HMAC_STATE* from = (HMAC_STATE*)internalFmt;
205         memcpy(&outBuf[offsetof(HMAC_STATE, hmacKey)],
206               &from->hmacKey,
207               sizeof(from->hmacKey));
208     }
209     if(internalFmt->hashAlg != TPM_ALG_NULL)
210         HASH_STATE_EXPORT(externalFmt, internalFmt);
211 }
212
213 /*** CryptHashImportState()
214 // This function is used to import the hash state. This function
215 // would be called to import a hash state when the context of a sequence object
216 // was being loaded.
217 void CryptHashImportState(
218     PHASH_STATE internalFmt,          // OUT: the hash state formatted for use by
219                                         // the library
220     PCEXPORT_HASH_STATE externalFmt  // IN: the exported hash state
221 )
222 {
223     BYTE* inBuf = (BYTE*)externalFmt;
224     //
225 #define CopyFromOffset(value) \
226     memcpy(&internalFmt->value, \
227           &inBuf[offsetof(HASH_STATE, value)], \
228           sizeof(internalFmt->value))
229
230     // Copy the hashAlg of the byte-aligned input structure to the structure-aligned
231     // internal structure.
232     CopyFromOffset(hashAlg);
233     CopyFromOffset(type);
234     if(internalFmt->hashAlg != TPM_ALG_NULL)
235     {
236 #ifdef HASH_STATE_SMAC
237         if(internalFmt->type == HASH_STATE_SMAC)
238         {
239             memcpy(internalFmt, inBuf, sizeof(HASH_STATE));
240             return;
241         }
242 #endif
243         internalFmt->def = CryptGetHashDef(internalFmt->hashAlg);

```

```

244     HASH_STATE_IMPORT(internalFmt, inBuf);
245     if(internalFmt->type == HASH_STATE_HMAC)
246     {
247         HMAC_STATE* to = (HMAC_STATE*)internalFmt;
248         memcpy(&to->hmacKey,
249             &inBuf[offsetof(HMAC_STATE, hmacKey)],
250             sizeof(to->hmacKey));
251     }
252 }
253 }
254
255 /** State Modification Functions
256
257 ****HashEnd()
258 // Local function to complete a hash that uses the hashDef instead of an algorithm
259 // ID. This function is used to complete the hash and only return a partial digest.
260 // The return value is the size of the data copied.
261 static UINT16 HashEnd(PHASH_STATE hashState, // IN: the hash state
262                     UINT32 dOutSize, // IN: the size of receive buffer
263                     PBYTE dOut // OUT: the receive buffer
264 )
265 {
266     BYTE temp[MAX_DIGEST_SIZE];
267     if((hashState->hashAlg == TPM_ALG_NULL) || (hashState->type != HASH_STATE_HASH))
268         dOutSize = 0;
269     if(dOutSize > 0)
270     {
271         hashState->def = CryptGetHashDef(hashState->hashAlg);
272         // Set the final size
273         dOutSize = MIN(dOutSize, hashState->def->digestSize);
274         // Complete into the temp buffer and then copy
275         HASH_END(hashState, temp);
276         // Don't want any other functions calling the HASH_END method
277         // directly.
278 #undef HASH_END
279         memcpy(dOut, &temp, dOutSize);
280     }
281     hashState->type = HASH_STATE_EMPTY;
282     return (UINT16)dOutSize;
283 }
284
285 **** CryptHashStart()
286 // Functions starts a hash stack
287 // Start a hash stack and returns the digest size. As a side effect, the
288 // value of 'stateSize' in hashState is updated to indicate the number of bytes
289 // of state that were saved. This function calls GetHashServer() and that function
290 // will put the TPM into failure mode if the hash algorithm is not supported.
291 //
292 // This function does not use the sequence parameter. If it is necessary to import
293 // or export context, this will start the sequence in a local state
294 // and export the state to the input buffer. Will need to add a flag to the state
295 // structure to indicate that it needs to be imported before it can be used.
296 // (BLEH).
297 // Return Type: UINT16
298 // 0 hash is TPM_ALG_NULL
299 // >0 digest size
300 LIB_EXPORT UINT16 CryptHashStart(
301     PHASH_STATE hashState, // OUT: the running hash state
302     TPM_ALG_ID hashAlg // IN: hash algorithm
303 )
304 {
305     UINT16 retVal;
306
307     TEST(hashAlg);
308
309     hashState->hashAlg = hashAlg;

```

```

310     if(hashAlg == TPM_ALG_NULL)
311     {
312         retVal = 0;
313     }
314     else
315     {
316         hashState->def = CryptGetHashDef(hashAlg);
317         HASH_START(hashState);
318         retVal = hashState->def->digestSize;
319     }
320 #undef HASH_START
321     hashState->type = HASH_STATE_HASH;
322     return retVal;
323 }
324
325 /*** CryptDigestUpdate()
326 // Add data to a hash or HMAC, SMAC stack.
327 //
328 void CryptDigestUpdate(PHASH_STATE hashState, // IN: the hash context information
329                        UINT32      dataSize,   // IN: the size of data to be added
330                        const BYTE* data        // IN: data to be hashed
331 )
332 {
333     if(hashState->hashAlg != TPM_ALG_NULL)
334     {
335         if((hashState->type == HASH_STATE_HASH)
336            || (hashState->type == HASH_STATE_HMAC))
337             HASH_DATA(hashState, dataSize, (BYTE*)data);
338 #if SMAC_IMPLEMENTED
339         else if(hashState->type == HASH_STATE_SMAC)
340             (hashState->state.smac.smacMethods.data)(
341                 &hashState->state.smac.state, dataSize, data);
342 #endif // SMAC_IMPLEMENTED
343         else
344             FAIL(FATAL_ERROR_INTERNAL);
345     }
346     return;
347 }
348
349 /*** CryptHashEnd()
350 // Complete a hash or HMAC computation. This function will place the smaller of
351 // 'digestSize' or the size of the digest in 'dOut'. The number of bytes in the
352 // placed in the buffer is returned. If there is a failure, the returned value
353 // is <= 0.
354 // Return Type: UINT16
355 //      0      no data returned
356 //      > 0    the number of bytes in the digest or dOutSize, whichever is smaller
357 LIB_EXPORT UINT16 CryptHashEnd(PHASH_STATE hashState, // IN: the state of hash stack
358                               UINT32      dOutSize,   // IN: size of digest buffer
359                               BYTE*      dOut        // OUT: hash digest
360 )
361 {
362     pAssert(hashState->type == HASH_STATE_HASH);
363     return HashEnd(hashState, dOutSize, dOut);
364 }
365
366 /*** CryptHashBlock()
367 // Start a hash, hash a single block, update 'digest' and return the size of
368 // the results.
369 //
370 // The 'digestSize' parameter can be smaller than the digest. If so, only the more
371 // significant bytes are returned.
372 // Return Type: UINT16
373 //      >= 0    number of bytes placed in 'dOut'
374 LIB_EXPORT UINT16 CryptHashBlock(TPM_ALG_ID hashAlg, // IN: The hash algorithm
375                                  UINT32      dataSize, // IN: size of buffer to hash

```



```

376         const BYTE* data,           // IN: the buffer to hash
377         UINT32 dOutSize,           // IN: size of the digest buffer
378         BYTE* dOut                // OUT: digest buffer
379     )
380     {
381         HASH_STATE state;
382         CryptHashStart(&state, hashAlg);
383         CryptDigestUpdate(&state, dataSize, data);
384         return HashEnd(&state, dOutSize, dOut);
385     }
386
387     /*** CryptDigestUpdate2B()
388     // This function updates a digest (hash or HMAC) with a TPM2B.
389     //
390     // This function can be used for both HMAC and hash functions so the
391     // 'digestState' is void so that either state type can be passed.
392     LIB_EXPORT void CryptDigestUpdate2B(PHASH_STATE state, // IN: the digest state
393         const TPM2B* bIn // IN: 2B containing the data
394     )
395     {
396         // Only compute the digest if a pointer to the 2B is provided.
397         // In CryptDigestUpdate(), if size is zero or buffer is NULL, then no change
398         // to the digest occurs. This function should not provide a buffer if bIn is
399         // not provided.
400         pAssert(bIn != NULL);
401         CryptDigestUpdate(state, bIn->size, bIn->buffer);
402         return;
403     }
404
405     /*** CryptHashEnd2B()
406     // This function is the same as CryptCompleteHash() but the digest is
407     // placed in a TPM2B. This is the most common use and this is provided
408     // for specification clarity. 'digest.size' should be set to indicate the number of
409     // bytes to place in the buffer
410     // Return Type: UINT16
411     // >=0 the number of bytes placed in 'digest.buffer'
412     LIB_EXPORT UINT16 CryptHashEnd2B(
413         PHASH_STATE state, // IN: the hash state
414         P2B digest // IN: the size of the buffer Out: requested
415         // number of bytes
416     )
417     {
418         return CryptHashEnd(state, digest->size, digest->buffer);
419     }
420
421     /*** CryptDigestUpdateInt()
422     // This function is used to include an integer value to a hash stack. The function
423     // marshals the integer into its canonical form before calling CryptDigestUpdate().
424     LIB_EXPORT void CryptDigestUpdateInt(
425         void* state, // IN: the state of hash stack
426         UINT32 intSize, // IN: the size of 'intValue' in bytes
427         UINT64 intValue // IN: integer value to be hashed
428     )
429     {
430         #if LITTLE_ENDIAN_TPM
431             intValue = REVERSE_ENDIAN_64(intValue);
432         #endif
433         CryptDigestUpdate(state, intSize, &((BYTE*)&intValue)[8 - intSize]);
434     }
435
436     /*** HMAC Functions
437
438     /*** CryptHmacStart()
439     // This function is used to start an HMAC using a temp
440     // hash context. The function does the initialization
441     // of the hash with the HMAC key XOR iPad and updates the

```

```

442 // HMAC key XOR oPad.
443 //
444 // The function returns the number of bytes in a digest produced by 'hashAlg'.
445 // Return Type: UINT16
446 // >= 0      number of bytes in digest produced by 'hashAlg' (may be zero)
447 //
448 LIB_EXPORT UINT16 CryptHmacStart(PHMAC_STATE state, // IN/OUT: the state buffer
449                                TPM_ALG_ID hashAlg, // IN: the algorithm to use
450                                UINT16 keySize, // IN: the size of the HMAC key
451                                const BYTE* key // IN: the HMAC key
452 )
453 {
454     PHASH_DEF hashDef;
455     BYTE* pb;
456     UINT32 i;
457     //
458     hashDef = CryptGetHashDef(hashAlg);
459     if(hashDef->digestSize != 0)
460     {
461         // If the HMAC key is larger than the hash block size, it has to be reduced
462         // to fit. The reduction is a digest of the hashKey.
463         if(keySize > hashDef->blockSize)
464         {
465             // if the key is too big, reduce it to a digest of itself
466             state->hmacKey.t.size = CryptHashBlock(
467                 hashAlg, keySize, key, hashDef->digestSize, state->hmacKey.t.buffer);
468         }
469         else
470         {
471             memcpy(state->hmacKey.t.buffer, key, keySize);
472             state->hmacKey.t.size = keySize;
473         }
474         // XOR the key with iPad (0x36)
475         pb = state->hmacKey.t.buffer;
476         for(i = state->hmacKey.t.size; i > 0; i--)
477             *pb++ ^= 0x36;
478
479         // if the keySize is smaller than a block, fill the rest with 0x36
480         for(i = hashDef->blockSize - state->hmacKey.t.size; i > 0; i--)
481             *pb++ = 0x36;
482
483         // Increase the oPadSize to a full block
484         state->hmacKey.t.size = hashDef->blockSize;
485
486         // Start a new hash with the HMAC key
487         // This will go in the caller's state structure and may be a sequence or not
488         CryptHashStart((PHASH_STATE)state, hashAlg);
489         CryptDigestUpdate(
490             (PHASH_STATE)state, state->hmacKey.t.size, state->hmacKey.t.buffer);
491         // XOR the key block with 0x5c ^ 0x36
492         for(pb = state->hmacKey.t.buffer, i = hashDef->blockSize; i > 0; i--)
493             *pb++ ^= (0x5c ^ 0x36);
494     }
495     // Set the hash algorithm
496     state->hashState.hashAlg = hashAlg;
497     // Set the hash state type
498     state->hashState.type = HASH_STATE_HMAC;
499
500     return hashDef->digestSize;
501 }
502
503 /*** CryptHmacEnd()
504 // This function is called to complete an HMAC. It will finish the current
505 // digest, and start a new digest. It will then add the oPadKey and the
506 // completed digest and return the results in dOut. It will not return more
507 // than dOutSize bytes.

```

```

508 // Return Type: UINT16
509 // >= 0      number of bytes in 'dOut' (may be zero)
510 LIB_EXPORT UINT16 CryptHmacEnd(PHMAC_STATE state, // IN: the hash state buffer
511                               UINT32 dOutSize, // IN: size of digest buffer
512                               BYTE* dOut // OUT: hash digest
513 )
514 {
515     BYTE temp[MAX_DIGEST_SIZE];
516     PHASH_STATE hState = (PHASH_STATE)&state->hashState;
517
518     #if SMAC_IMPLEMENTED
519     if(hState->type == HASH_STATE_SMAC)
520         return (state->hashState.state.smac.smacMethods.end) (
521             &state->hashState.state.smac.state, dOutSize, dOut);
522     #endif
523     pAssert(hState->type == HASH_STATE_HMAC);
524     hState->def = CryptGetHashDef(hState->hashAlg);
525     // Change the state type for completion processing
526     hState->type = HASH_STATE_HASH;
527     if(hState->hashAlg == TPM_ALG_NULL)
528         dOutSize = 0;
529     else
530     {
531         // Complete the current hash
532         HashEnd(hState, hState->def->digestSize, temp);
533         // Do another hash starting with the oPad
534         CryptHashStart(hState, hState->hashAlg);
535         CryptDigestUpdate(hState, state->hmacKey.t.size, state->hmacKey.t.buffer);
536         CryptDigestUpdate(hState, hState->def->digestSize, temp);
537     }
538     return HashEnd(hState, dOutSize, dOut);
539 }
540
541 /*** CryptHmacStart2B()
542 // This function starts an HMAC and returns the size of the digest
543 // that will be produced.
544 //
545 // This function is provided to support the most common use of starting an HMAC
546 // with a TPM2B key.
547 //
548 // The caller must provide a block of memory in which the hash sequence state
549 // is kept. The caller should not alter the contents of this buffer until the
550 // hash sequence is completed or abandoned.
551 //
552 // Return Type: UINT16
553 // > 0      the digest size of the algorithm
554 // = 0      the hashAlg was TPM_ALG_NULL
555 LIB_EXPORT UINT16 CryptHmacStart2B(
556     PHMAC_STATE hmacState, // OUT: the state of HMAC stack. It will be used
557                           // in HMAC update and completion
558     TPMI_ALG_HASH hashAlg, // IN: hash algorithm
559     P2B key // IN: HMAC key
560 )
561 {
562     return CryptHmacStart(hmacState, hashAlg, key->size, key->buffer);
563 }
564
565 /*** CryptHmacEnd2B()
566 // This function is the same as CryptHmacEnd() but the HMAC result
567 // is returned in a TPM2B which is the most common use.
568 // Return Type: UINT16
569 // >=0      the number of bytes placed in 'digest'
570 LIB_EXPORT UINT16 CryptHmacEnd2B(
571     PHMAC_STATE hmacState, // IN: the state of HMAC stack
572     P2B digest // OUT: HMAC
573 )

```

```

574 {
575     return CryptHmacEnd(hmacState, digest->size, digest->buffer);
576 }
577
578 /** Mask and Key Generation Functions
579 **** CryptMGF_KDF()
580 // This function performs MGF1/KDF1 or KDF2 using the selected hash. KDF1 and KDF2 are
581 // T('n') = T('n'-1) || H('seed' || 'counter') with the difference being that, with
582 // KDF1, 'counter' starts at 0 but with KDF2, 'counter' starts at 1. The caller
583 // determines which version by setting the initial value of counter to either 0 or 1.
584 // Note: Any value that is not 0 is considered to be 1.
585 //
586 // This function returns the length of the mask produced which
587 // could be zero if the digest algorithm is not supported
588 // Return Type: UINT16
589 //      0      hash algorithm was TPM_ALG_NULL
590 //      > 0     should be the same as 'mSize'
591 LIB_EXPORT UINT16 CryptMGF_KDF(UINT32 mSize, // IN: length of the mask to be produced
592                                BYTE* mask, // OUT: buffer to receive the mask
593                                TPM_ALG_ID hashAlg, // IN: hash to use
594                                UINT32 seedSize, // IN: size of the seed
595                                BYTE* seed, // IN: seed size
596                                UINT32 counter // IN: counter initial value
597 )
598 {
599     HASH_STATE hashState;
600     PHASH_DEF hDef = CryptGetHashDef(hashAlg);
601     UINT32 hLen;
602     UINT32 bytes;
603     //
604     // If there is no digest to compute return
605     if((hDef->digestSize == 0) || (mSize == 0))
606         return 0;
607     if(counter != 0)
608         counter = 1;
609     hLen = hDef->digestSize;
610     for(bytes = 0; bytes < mSize; bytes += hLen)
611     {
612         // Start the hash and include the seed and counter
613         CryptHashStart(&hashState, hashAlg);
614         CryptDigestUpdate(&hashState, seedSize, seed);
615         CryptDigestUpdateInt(&hashState, 4, counter);
616         // Get as much as will fit.
617         CryptHashEnd(&hashState, MIN((mSize - bytes), hLen), &mask[bytes]);
618         counter++;
619     }
620     return (UINT16)mSize;
621 }
622
623 /*** CryptKDFa()
624 // This function performs the key generation according to Part 1 of the
625 // TPM specification.
626 //
627 // This function returns the number of bytes generated which may be zero.
628 //
629 // The 'key' and 'keyStream' pointers are not allowed to be NULL. The other
630 // pointer values may be NULL. The value of 'sizeInBits' must be no larger
631 // than (2^18)-1 = 256K bits (32385 bytes).
632 //
633 // The 'once' parameter is set to allow incremental generation of a large
634 // value. If this flag is TRUE, 'sizeInBits' will be used in the HMAC computation
635 // but only one iteration of the KDF is performed. This would be used for
636 // XOR obfuscation so that the mask value can be generated in digest-sized
637 // chunks rather than having to be generated all at once in an arbitrarily
638 // large buffer and then XORed into the result. If 'once' is TRUE, then
639 // 'sizeInBits' must be a multiple of 8.

```

```

640 //
641 // Any error in the processing of this command is considered fatal.
642 // Return Type: UINT16
643 // 0 hash algorithm is not supported or is TPM_ALG_NULL
644 // > 0 the number of bytes in the 'keyStream' buffer
645 LIB_EXPORT UINT16 CryptKDFa(
646     TPM_ALG_ID hashAlg, // IN: hash algorithm used in HMAC
647     const TPM2B* key, // IN: HMAC key
648     const TPM2B* label, // IN: a label for the KDF
649     const TPM2B* contextU, // IN: context U
650     const TPM2B* contextV, // IN: context V
651     UINT32 sizeInBits, // IN: size of generated key in bits
652     BYTE* keyStream, // OUT: key buffer
653     UINT32* counterInOut, // IN/OUT: caller may provide the iteration
654     // counter for incremental operations to
655     // avoid large intermediate buffers.
656     UINT16 blocks, // IN: If non-zero, this is the maximum number
657     // of blocks to be returned, regardless
658     // of sizeInBits
659 )
660 {
661     UINT32 counter = 0; // counter value
662     INT16 bytes; // number of bytes to produce
663     UINT16 generated; // number of bytes generated
664     BYTE* stream = keyStream;
665     HMAC_STATE hState;
666     UINT16 digestSize = CryptHashGetDigestSize(hashAlg);
667
668     pAssert(key != NULL && keyStream != NULL);
669
670     TEST(TPM_ALG_KDF1_SP800_108);
671
672     if(digestSize == 0)
673         return 0;
674
675     if(counterInOut != NULL)
676         counter = *counterInOut;
677
678     // If the size of the request is larger than the numbers will handle,
679     // it is a fatal error.
680     pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);
681
682     // The number of bytes to be generated is the smaller of the sizeInBits bytes or
683     // the number of requested blocks. The number of blocks is the smaller of the
684     // number requested or the number allowed by sizeInBits. A partial block is
685     // a full block.
686     bytes = (blocks > 0) ? blocks * digestSize : (UINT16)BITS_TO_BYTES(sizeInBits);
687     generated = bytes;
688
689     // Generate required bytes
690     for(; bytes > 0; bytes -= digestSize)
691     {
692         counter++;
693         // Start HMAC
694         if(CryptHmacStart(&hState, hashAlg, key->size, key->buffer) == 0)
695             return 0;
696         // Adding counter
697         CryptDigestUpdateInt(&hState.hashState, 4, counter);
698
699         // Adding label
700         if(label != NULL)
701             HASH_DATA(&hState.hashState, label->size, (BYTE*)label->buffer);
702         // Add a null. SP108 is not very clear about when the 0 is needed but to
703         // make this like the previous version that did not add an 0x00 after
704         // a null-terminated string, this version will only add a null byte
705         // if the label parameter did not end in a null byte, or if no label

```



```

706     // is present.
707     if((label == NULL) || (label->size == 0)
708        || (label->buffer[label->size - 1] != 0))
709         CryptDigestUpdateInt(&hState.hashState, 1, 0);
710     // Adding contextU
711     if(contextU != NULL)
712         HASH_DATA(&hState.hashState, contextU->size, contextU->buffer);
713     // Adding contextV
714     if(contextV != NULL)
715         HASH_DATA(&hState.hashState, contextV->size, contextV->buffer);
716     // Adding size in bits
717     CryptDigestUpdateInt(&hState.hashState, 4, sizeInBits);
718
719     // Complete and put the data in the buffer
720     CryptHmacEnd(&hState, bytes, stream);
721     stream = &stream[digestSize];
722 }
723 // Masking in the KDF is disabled. If the calling function wants something
724 // less than even number of bytes, then the caller should do the masking
725 // because there is no universal way to do it here
726 if(counterInOut != NULL)
727     *counterInOut = counter;
728 return generated;
729 }
730
731 /*** CryptKDFe()
732 // This function implements KDFe() as defined in TPM specification part 1.
733 //
734 // This function returns the number of bytes generated which may be zero.
735 //
736 // The 'Z' and 'keyStream' pointers are not allowed to be NULL. The other
737 // pointer values may be NULL. The value of 'sizeInBits' must be no larger
738 // than (2^18)-1 = 256K bits (32385 bytes).
739 // Any error in the processing of this command is considered fatal.
740 // Return Type: UINT16
741 //      0      hash algorithm is not supported or is TPM_ALG_NULL
742 //      > 0    the number of bytes in the 'keyStream' buffer
743 //
744 LIB_EXPORT UINT16 CryptKDFe(TPM_ALG_ID hashAlg, // IN: hash algorithm used in HMAC
745                             TPM2B* Z, // IN: Z
746                             const TPM2B* label, // IN: a label value for the KDF
747                             TPM2B* partyUInfo, // IN: PartyUInfo
748                             TPM2B* partyVInfo, // IN: PartyVInfo
749                             UINT32 sizeInBits, // IN: size of generated key in bits
750                             BYTE* keyStream // OUT: key buffer
751 )
752 {
753     HASH_STATE hashState;
754     PHASH_DEF hashDef = CryptGetHashDef(hashAlg);
755
756     UINT32 counter = 0; // counter value
757     UINT16 hLen;
758     BYTE* stream = keyStream;
759     INT16 bytes; // number of bytes to generate
760
761     pAssert(keyStream != NULL && Z != NULL && ((sizeInBits + 7) / 8) < INT16_MAX);
762     //
763     hLen = hashDef->digestSize;
764     bytes = (INT16)((sizeInBits + 7) / 8);
765     if(hashAlg == TPM_ALG_NULL || bytes == 0)
766         return 0;
767
768     // Generate required bytes
769     //The inner loop of that KDF uses:
770     // Hash[i] := H(counter | Z | OtherInfo) (5)
771     // Where:

```



```

772 // Hash[i]          the hash generated on the i-th iteration of the loop.
773 // H()              an approved hash function
774 // counter          a 32-bit counter that is initialized to 1 and incremented
775 //                  on each iteration
776 // Z                the X coordinate of the product of a public ECC key and a
777 //                  different private ECC key.
778 // OtherInfo        a collection of qualifying data for the KDF defined below.
779 // In this specification, OtherInfo will be constructed by:
780 //   OtherInfo := Use | PartyUInfo | PartyVInfo
781 for(; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
782 {
783     if(bytes < hLen)
784         hLen = bytes;
785     counter++;
786     // Do the hash
787     CryptHashStart(&hashState, hashAlg);
788     // Add counter
789     CryptDigestUpdateInt(&hashState, 4, counter);
790
791     // Add Z
792     if(Z != NULL)
793         CryptDigestUpdate2B(&hashState, Z);
794     // Add label
795     if(label != NULL)
796         CryptDigestUpdate2B(&hashState, label);
797
798     // NIST.SP.800-56Cr2.pdf section 4.1 states that no NULL
799     // character is required here.
800     // Note, this is different from KDFa which is specified in
801     // NIST.SP.800-108r1.pdf section 4 (a NULL character is required
802     // for that case).
803
804     // Add PartyUInfo
805     if(partyUInfo != NULL)
806         CryptDigestUpdate2B(&hashState, partyUInfo);
807
808     // Add PartyVInfo
809     if(partyVInfo != NULL)
810         CryptDigestUpdate2B(&hashState, partyVInfo);
811
812     // Compute Hash. hLen was changed to be the smaller of bytes or hLen
813     // at the start of each iteration.
814     CryptHashEnd(&hashState, hLen, stream);
815 }
816
817 // Mask off bits if the required bits is not a multiple of byte size
818 if((sizeInBits % 8) != 0)
819     keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
820
821 return (UINT16)((sizeInBits + 7) / 8);
822 }

```

7.144 /tpm/src/crypt/CryptPrime.c

```

1  /** Introduction
2  // This file contains the code for prime validation.
3
4  #include "Tpm.h"
5  #include "CryptPrime_fp.h"
6  #include "TpmMath_Util_fp.h"
7
8  //#define CPRI_PRIME
9  //#include "PrimeTable.h"
10
11 #include "CryptPrimeSieve_fp.h"

```

```

12
13 extern const uint32_t      s_LastPrimeInTable;
14 extern const uint32_t      s_PrimeTableSize;
15 extern const uint32_t      s_PrimesInTable;
16 extern const unsigned char s_PrimeTable[];
17 extern const Crypt_Int*    s_CompositeOfSmallPrimes;
18
19 /** Functions
20
21 /** Root2()
22 // This finds ceil(sqrt(n)) to use as a stopping point for searching the prime
23 // table.
24 static uint32_t Root2(uint32_t n)
25 {
26     int32_t last = (int32_t)(n >> 2);
27     int32_t next = (int32_t)(n >> 1);
28     int32_t diff;
29     int32_t stop = 10;
30     //
31     // get a starting point
32     for(; next != 0; last >=> 1, next >=> 2)
33     ;
34     last++;
35     do
36     {
37         next = (last + (n / last)) >> 1;
38         diff = next - last;
39         last = next;
40         if(stop-- == 0)
41             FAIL(FATAL_ERROR_INTERNAL);
42     } while(diff < -1 || diff > 1);
43     if((n / next) > (unsigned)next)
44         next++;
45     pAssert(next != 0);
46     pAssert(((n / next) <= (unsigned)next) && (n / (next + 1) < (unsigned)next));
47     return next;
48 }
49
50 /** IsPrimeInt()
51 // This will do a test of a word of up to 32-bits in size.
52 BOOL IsPrimeInt(uint32_t n)
53 {
54     uint32_t i;
55     uint32_t stop;
56     if(n < 3 || ((n & 1) == 0))
57         return (n == 2);
58     if(n <= s_LastPrimeInTable)
59     {
60         n >=> 1;
61         return ((s_PrimeTable[n >> 3] >> (n & 7)) & 1);
62     }
63     // Need to search
64     stop = Root2(n) >> 1;
65     // starting at 1 is equivalent to staring at (1 << 1) + 1 = 3
66     for(i = 1; i < stop; i++)
67     {
68         if((s_PrimeTable[i >> 3] >> (i & 7)) & 1)
69             // see if this prime evenly divides the number
70             if((n % ((i << 1) + 1)) == 0)
71                 return FALSE;
72     }
73     return TRUE;
74 }
75
76 /** TpmMath_IsProbablyPrime()
77 // This function is used when the key sieve is not implemented. This function

```

```

78 // Will try to eliminate some of the obvious things before going on
79 // to perform MillerRabin as a final verification of primeness.
80 BOOL TpmMath_IsProbablyPrime(Crypt_Int* prime, // IN:
81                             RAND_STATE* rand // IN: the random state just
82                             // in case Miller-Rabin is required
83 )
84 {
85     uint32_t leastSignificant32 = ExtMath_GetLeastSignificant32bits(prime);
86     // is even?
87     if((leastSignificant32 & 0x1) == 0)
88         return FALSE;
89
90     if(ExtMath_SizeInBits(prime) <= 32)
91         return IsPrimeInt(leastSignificant32);
92
93     // this s_LastPrimeInTable check guarantees that the full prime table check
94     // is incorporated in IsPrimeInt. If this fails then something like this
95     // old code needs to be added back.
96     // if(ExtMath_UnsignedCmpWord(prime, s_LastPrimeInTable) <= 0)
97     // {
98     //     // check fast prime table before doing slower checks
99     //     crypt_uword_t temp = prime->d[0] >> 1;
100     //     return ((s_PrimeTable[temp >> 3] >> (temp & 7)) & 1);
101     // }
102     MUST_BE(sizeof(s_LastPrimeInTable) <= 4);
103
104     // check using GCD before doing a full Miller Rabin.
105     {
106         CRYPT_INT_VAR(gcd, LARGEST_NUMBER_BITS);
107         ExtMath_GCD(gcd, prime, s_CompositeOfSmallPrimes);
108         if(!ExtMath_IsEqualWord(gcd, 1))
109             return FALSE;
110     }
111     return MillerRabin(prime, rand);
112 }
113
114 /*** MillerRabinRounds()
115 // Function returns the number of Miller-Rabin rounds necessary to give an
116 // error probability equal to the security strength of the prime. These values
117 // are from FIPS 186-3.
118 UINT32
119 MillerRabinRounds(UINT32 bits // IN: Number of bits in the RSA prime
120 )
121 {
122     if(bits < 511)
123         return 8; // don't really expect this
124     if(bits < 1536)
125         return 5; // for 512 and 1K primes
126     return 4; // for 3K public modulus and greater
127 }
128
129 /*** MillerRabin()
130 // This function performs a Miller-Rabin test from FIPS 186-3. It does
131 // 'iterations' trials on the number. In all likelihood, if the number
132 // is not prime, the first test fails.
133 // Return Type: BOOL
134 // TRUE(1) probably prime
135 // FALSE(0) composite
136 BOOL MillerRabin(Crypt_Int* bnW, RAND_STATE* rand)
137 {
138     CRYPT_INT_MAX(bnWm1);
139     CRYPT_PRIME_VAR(bnM);
140     CRYPT_PRIME_VAR(bnB);
141     CRYPT_PRIME_VAR(bnZ);
142     BOOL ret = FALSE; // Assumed composite for easy exit
143     unsigned int a;

```

```

144     unsigned int j;
145     int         wLen;
146     int         i;
147     int         iterations = MillerRabinRounds(ExtMath_SizeInBits(bnW));
148     //
149     INSTRUMENT_INC(MillerRabinTrials[PrimeIndex]);
150
151     pAssert(bnW->size > 1);
152     // Let a be the largest integer such that 2^a divides w1.
153     ExtMath_SubtractWord(bnWm1, bnW, 1);
154     pAssert(bnWm1->size != 0);
155
156     // Since w is odd (w-1) is even so start at bit number 1 rather than 0
157     // Get the number of bits in bnWm1 so that it doesn't have to be recomputed
158     // on each iteration.
159     i = (int)(bnWm1->size * RADIX_BITS);
160     // Now find the largest power of 2 that divides w1
161     for(a = 1; (a < (bnWm1->size * RADIX_BITS)) && (ExtMath_TestBit(bnWm1, a) == 0);
162         a++)
163     {
164     }
165     // 2. m = (w1) / 2^a
166     ExtMath_ShiftRight(bnM, bnWm1, a);
167     // 3. wlen = len (w).
168     wLen = ExtMath_SizeInBits(bnW);
169     // 4. For i = 1 to iterations do
170     for(i = 0; i < iterations; i++)
171     {
172         // 4.1 Obtain a string b of wlen bits from an RBG.
173         // Ensure that 1 < b < w1.
174         // 4.2 If ((b <= 1) or (b >= w1)), then go to step 4.1.
175         while(TpmMath_GetRandomInteger(bnB, wLen, rand)
176             && ((ExtMath_UnsignedCmpWord(bnB, 1) <= 0)
177                || (ExtMath_UnsignedCmp(bnB, bnWm1) >= 0)))
178         ;
179         if(g_inFailureMode)
180             return FALSE;
181
182         // 4.3 z = b^m mod w.
183         // if ModExp fails, then say this is not
184         // prime and bail out.
185         ExtMath_ModExp(bnZ, bnB, bnM, bnW);
186
187         // 4.4 If ((z == 1) or (z = w == 1)), then go to step 4.7.
188         if((ExtMath_UnsignedCmpWord(bnZ, 1) == 0)
189            || (ExtMath_UnsignedCmp(bnZ, bnWm1) == 0))
190             goto step4point7;
191         // 4.5 For j = 1 to a - 1 do.
192         for(j = 1; j < a; j++)
193         {
194             // 4.5.1 z = z^2 mod w.
195             ExtMath_ModMult(bnZ, bnZ, bnZ, bnW);
196             // 4.5.2 If (z = w1), then go to step 4.7.
197             if(ExtMath_UnsignedCmp(bnZ, bnWm1) == 0)
198                 goto step4point7;
199             // 4.5.3 If (z = 1), then go to step 4.6.
200             if(ExtMath_IsEqualWord(bnZ, 1))
201                 goto step4point6;
202         }
203         // 4.6 Return COMPOSITE.
204     step4point6:
205         INSTRUMENT_INC(failedAtIteration[i]);
206         goto end;
207         // 4.7 Continue. Comment: Increment i for the do-loop in step 4.
208     step4point7:
209         continue;

```

```

210     }
211     // 5. Return PROBABLY PRIME
212     ret = TRUE;
213 end:
214     return ret;
215 }
216
217 #if ALG_RSA
218
219 /*** RsaCheckPrime()
220 // This will check to see if a number is prime and appropriate for an
221 // RSA prime.
222 //
223 // This has different functionality based on whether we are using key
224 // sieving or not. If not, the number checked to see if it is divisible by
225 // the public exponent, then the number is adjusted either up or down
226 // in order to make it a better candidate. It is then checked for being
227 // probably prime.
228 //
229 // If sieving is used, the number is used to root a sieving process.
230 //
231 TPM_RC
232 RsaCheckPrime(Crypt_Int* prime, UINT32 exponent, RAND_STATE* rand)
233 {
234     # if !RSA_KEY_SIEVE
235         TPM_RC retVal = TPM_RC_SUCCESS;
236         UINT32 modE = ExtMath_ModWord(prime, exponent);
237
238         NOT_REFERENCED(rand);
239
240         if(modE == 0)
241             // evenly divisible so add two keeping the number odd
242             ExtMath_AddWord(prime, prime, 2);
243         // want 0 != (p - 1) mod e
244         // which is 1 != p mod e
245         else if(modE == 1)
246             // subtract 2 keeping number odd and insuring that
247             // 0 != (p - 1) mod e
248             ExtMath_SubtractWord(prime, prime, 2);
249
250         if(TpmMath_IsProbablyPrime(prime, rand) == 0)
251             ERROR_EXIT(g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_VALUE);
252     Exit:
253         return retVal;
254     # else
255         return PrimeSelectWithSieve(prime, exponent, rand);
256     # endif
257 }
258
259 /*** RsaAdjustPrimeCandidate()
260 //
261 // For this math, we assume that the RSA numbers are fixed-point numbers with
262 // the decimal point to the "left" of the most significant bit. This approach helps
263 // make it clear what is happening with the MSb of the values.
264 // The two RSA primes have to be large enough so that their product will be a number
265 // with the necessary number of significant bits. For example, we want to be able
266 // to multiply two 1024-bit numbers to produce a number with 2028 significant bits. If
267 // we accept any 1024-bit prime that has its MSb set, then it is possible to produce a
268 // product that does not have the MSb SET. For example, if we use tiny keys of 16 bits
269 // and have two 8-bit 'primes' of 0x80, then the public key would be 0x4000 which is
270 // only 15-bits. So, what we need to do is made sure that each of the primes is large
271 // enough so that the product of the primes is twice as large as each prime. A little
272 // arithmetic will show that the only way to do this is to make sure that each of the
273 // primes is no less than root(2)/2. That's what this functions does.
274 // This function adjusts the candidate prime so that it is odd and >= root(2)/2.
275 // This allows the product of these two numbers to be .5, which, in fixed point

```

```

276 // notation means that the most significant bit is 1.
277 // For this routine, the root(2)/2 (0.7071067811865475) approximated with 0xB505
278 // which is, in fixed point, 0.7071075439453125 or an error of 0.000108%. Just setting
279 // the upper two bits would give a value > 0.75 which is an error of > 6%. Given the
280 // amount of time all the other computations take, reducing the error is not much of
281 // a cost, but it isn't totally required either.
282 //
283 // This function can be replaced with a function that just sets the two most
284 // significant bits of each prime candidate without introducing any computational
285 // issues.
286 //
287 static void RsaAdjustPrimeCandidate(BYTE* bigNumberBuffer, size_t bufSize)
288 {
289     // first, ensure the last byte is odd, making the entire value odd
290     bigNumberBuffer[bufSize - 1] |= 1;
291
292     // second, get the most significant 32 bits.
293     uint32_t msw = (bigNumberBuffer[0] << 24) | (bigNumberBuffer[1] << 16)
294         | (bigNumberBuffer[2] << 8) | (bigNumberBuffer[3] << 0);
295
296     // Multiplying 0xff...f by 0x4AFB gives 0xff...f - 0xB5050...0
297     uint32_t adjusted = (msw >> 16) * 0x4AFB;
298     adjusted += ((msw & 0xFFFF) * 0x4AFB) >> 16;
299     adjusted += 0xB5050000UL;
300
301     // put the value back
302     bigNumberBuffer[0] = (uint8_t)(adjusted >> 24);
303     bigNumberBuffer[1] = (uint8_t)(adjusted >> 16);
304     bigNumberBuffer[2] = (uint8_t)(adjusted >> 8);
305     bigNumberBuffer[3] = (uint8_t)(adjusted >> 0);
306 }
307
308 /***TpmRsa_GeneratePrimeForRSA()
309 // Function to generate a prime of the desired size with the proper attributes
310 // for an RSA prime.
311 // succeeds, or enters failure mode.
312 TPM_RC TpmRsa_GeneratePrimeForRSA(
313     Crypt_Int* prime, // IN/OUT: points to the BN that will get the
314                       // random value
315     UINT32 bits, // IN: number of bits to get
316     UINT32 exponent, // IN: the exponent
317     RAND_STATE* rand // IN: the random state
318 )
319 {
320     // Only try to handle specific sizes of keys.
321     // this is necessary so the RsaAdjustPrimeCandidate function works correctly.
322     pAssert((bits % 32) == 0);
323
324     // create buffer large enough for the largest key
325     TPM2B_TYPE(LARGEST, LARGEST_NUMBER);
326     TPM2B_LARGEST large;
327
328     NUMBYTES bytes = (NUMBYTES)BITS_TO_BYTES(bits);
329     BOOL OK = (bytes <= sizeof(large.t.buffer));
330     BOOL found = FALSE;
331     while(OK && !found)
332     {
333         OK = TpmMath_GetRandomBits(large.t.buffer, bits, rand); // new
334         large.t.size = bytes;
335         RsaAdjustPrimeCandidate(large.t.buffer, bytes);
336         // convert from 2B to Integer for prime checks
337         OK = OK
338             && (ExtMath_IntFromBytes(prime, large.t.buffer, large.t.size) != NULL);
339         found = OK && (RsaCheckPrime(prime, exponent, rand) == TPM_RC_SUCCESS);
340     }
341 }

```



```

342     if(!OK)
343     {
344         FAIL(FATAL_ERROR_CRYPTO);
345     }
346
347     return (OK && found) ? TPM_RC_SUCCESS : TPM_RC_FAILURE;
348 }
349
350 #endif // ALG_RSA

```

7.145 /tpm/src/crypt/CryptPrimeSieve.c

```

1  /** Includes and defines
2
3  #include "Tpm.h"
4
5  #if RSA_KEY_SIEVE
6
7  # include "CryptPrimeSieve_fp.h"
8
9  // This determines the number of bits in the largest sieve field.
10 # define MAX_FIELD_SIZE 2048
11
12 extern const uint32_t      s_LastPrimeInTable;
13 extern const uint32_t      s_PrimeTableSize;
14 extern const uint32_t      s_PrimesInTable;
15 extern const unsigned char s_PrimeTable[];
16
17 // This table is set of prime markers. Each entry is the prime value
18 // for the ((n + 1) * 1024) prime. That is, the entry in s_PrimeMarkers[1]
19 // is the value for the 2,048th prime. This is used in the PrimeSieve
20 // to adjust the limit for the prime search. When processing smaller
21 // prime candidates, fewer primes are checked directly before going to
22 // Miller-Rabin. As the prime grows, it is worth spending more time eliminating
23 // primes as, a) the density is lower, and b) the cost of Miller-Rabin is
24 // higher.
25 const uint32_t s_PrimeMarkersCount = 6;
26 const uint32_t s_PrimeMarkers[]    = {8167, 17881, 28183, 38891, 49871, 60961};
27 uint32_t      primeLimit;
28
29 /** Functions
30
31 /*** RsaAdjustPrimeLimit()
32 // This used during the sieve process. The iterator for getting the
33 // next prime (RsaNextPrime()) will return primes until it hits the
34 // limit (primeLimit) set up by this function. This causes the sieve
35 // process to stop when an appropriate number of primes have been
36 // sieved.
37 LIB_EXPORT void RsaAdjustPrimeLimit(uint32_t requestedPrimes)
38 {
39     if(requestedPrimes == 0 || requestedPrimes > s_PrimesInTable)
40         requestedPrimes = s_PrimesInTable;
41     requestedPrimes = (requestedPrimes - 1) / 1024;
42     if(requestedPrimes < s_PrimeMarkersCount)
43         primeLimit = s_PrimeMarkers[requestedPrimes];
44     else
45         primeLimit = s_LastPrimeInTable;
46     primeLimit >>= 1;
47 }
48
49 /*** RsaNextPrime()
50 // This the iterator used during the sieve process. The input is the
51 // last prime returned (or any starting point) and the output is the
52 // next higher prime. The function returns 0 when the primeLimit is
53 // reached.

```

```

54 LIB_EXPORT uint32_t RsaNextPrime(uint32_t lastPrime)
55 {
56     if(lastPrime == 0)
57         return 0;
58     lastPrime >>= 1;
59     for(lastPrime += 1; lastPrime <= primeLimit; lastPrime++)
60     {
61         if(((s_PrimeTable[lastPrime >> 3] >> (lastPrime & 0x7)) & 1) == 1)
62             return ((lastPrime << 1) + 1);
63     }
64     return 0;
65 }
66
67 // This table contains a previously sieved table. It has
68 // the bits for 3, 5, and 7 removed. Because of the
69 // factors, it needs to be aligned to 105 and has
70 // a repeat of 105.
71 const BYTE seedValues[] = {0x16, 0x29, 0xcb, 0xa4, 0x65, 0xda, 0x30, 0x6c, 0x99, 0x96,
72                             0x4c, 0x53, 0xa2, 0x2d, 0x52, 0x96, 0x49, 0xcb, 0xb4, 0x61,
73                             0xd8, 0x32, 0x2d, 0x99, 0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93,
74                             0x96, 0x69, 0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
75                             0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb, 0xb4, 0x64,
76                             0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c, 0x5a, 0xa6, 0x0d, 0xc3,
77                             0x96, 0x69, 0xc9, 0x34, 0x25, 0xda, 0x22, 0x65, 0x99, 0xb4,
78                             0x4c, 0x1b, 0x86, 0x2d, 0xd3, 0x92, 0x69, 0x4a, 0xb4, 0x45,
79                             0xca, 0x32, 0x69, 0x99, 0x36, 0x0c, 0x5b, 0xa6, 0x25, 0xd3,
80                             0x94, 0x68, 0x8b, 0x94, 0x65, 0xd2, 0x32, 0x6d, 0x18, 0xb6,
81                             0x4c, 0x4b, 0xa6, 0x29, 0xd1};
82
83 # define USE_NIBBLE
84
85 # ifndef USE_NIBBLE
86 static const BYTE bitsInByte[256] =
87     {0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03, 0x01, 0x02, 0x02, 0x03, 0x02,
88     0x03, 0x03, 0x04, 0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04, 0x02, 0x03,
89     0x03, 0x04, 0x03, 0x04, 0x04, 0x05, 0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03,
90     0x04, 0x02, 0x03, 0x03, 0x04, 0x04, 0x03, 0x04, 0x04, 0x05, 0x02, 0x03, 0x03, 0x04,
91     0x03, 0x04, 0x04, 0x05, 0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06, 0x01,
92     0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04, 0x02, 0x03, 0x03, 0x04, 0x03, 0x04,
93     0x04, 0x05, 0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05, 0x03, 0x04, 0x04,
94     0x05, 0x04, 0x05, 0x05, 0x06, 0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
95     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06, 0x03, 0x04, 0x04, 0x05, 0x04,
96     0x05, 0x05, 0x06, 0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07, 0x01, 0x02,
97     0x02, 0x03, 0x02, 0x03, 0x03, 0x04, 0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04,
98     0x05, 0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05, 0x03, 0x04, 0x04, 0x05,
99     0x04, 0x05, 0x05, 0x06, 0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05, 0x03,
100    0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06, 0x03, 0x04, 0x04, 0x05, 0x04, 0x05,
101    0x05, 0x06, 0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07, 0x02, 0x03, 0x03,
102    0x04, 0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x04, 0x04, 0x05, 0x05, 0x06, 0x05,
103    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x04, 0x05, 0x06, 0x04, 0x05, 0x05, 0x06,
104    0x06, 0x06, 0x07, 0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06, 0x04, 0x05,
105    0x05, 0x06, 0x05, 0x06, 0x06, 0x07, 0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06,
106    0x07, 0x05, 0x06, 0x06, 0x07, 0x06, 0x07, 0x07, 0x08};
107 # define BitsInByte(x) bitsInByte[(unsigned char)x]
108 # else
109 const BYTE bitsInNibble[16] = {0x00,
110                                0x01,
111                                0x01,
112                                0x02,
113                                0x01,
114                                0x02,
115                                0x02,
116                                0x03,
117                                0x01,
118                                0x02,
119                                0x02,

```

```

120                                     0x03,
121                                     0x02,
122                                     0x03,
123                                     0x03,
124                                     0x04};
125 #   define BitsInByte(x)              \
126     (bitsInNibble[(unsigned char)(x)&0xf] \
127      + bitsInNibble[((unsigned char)(x) >> 4) & 0xf])
128 #   endif
129
130 /*** BitsInArray()
131 // This function counts the number of bits set in an array of bytes.
132 static int BitsInArray(const unsigned char* a, // IN: A pointer to an array of bytes
133                       unsigned int      aSize // IN: the number of bytes to sum
134 )
135 {
136     int j = 0;
137     for(; aSize; a++, aSize--)
138         j += BitsInByte(*a);
139     return j;
140 }
141
142 /*** FindNthSetBit()
143 // This function finds the nth SET bit in a bit array. The 'n' parameter is
144 // between 1 and the number of bits in the array (always a multiple of 8).
145 // If called when the array does not have n bits set, it will return -1
146 // Return Type: unsigned int
147 //     <0      no bit is set or no bit with the requested number is set
148 //     >=0     the number of the bit in the array that is the nth set
149 LIB_EXPORT int FindNthSetBit(
150     const UINT16 aSize, // IN: the size of the array to check
151     const BYTE*  a,     // IN: the array to check
152     const UINT32 n      // IN: the number of the SET bit
153 )
154 {
155     UINT16 i;
156     int    retValue;
157     UINT32 sum = 0;
158     BYTE   sel;
159
160     //find the bit
161     for(i = 0; (i < (int)aSize) && (sum < n); i++)
162         sum += BitsInByte(a[i]);
163     i--;
164     // The chosen bit is in the byte that was just accessed
165     // Compute the offset to the start of that byte
166     retValue = i * 8 - 1;
167     sel      = a[i];
168     // Subtract the bits in the last byte added.
169     sum -= BitsInByte(sel);
170     // Now process the byte, one bit at a time.
171     for(; (sel != 0) && (sum != n); retValue++, sel = sel >> 1)
172         sum += (sel & 1) != 0;
173     return (sum == n) ? retValue : -1;
174 }
175
176 typedef struct
177 {
178     UINT32 prime;
179     UINT16 count;
180 } SIEVE_MARKS;
181
182 // clang-format off
183 const SIEVE_MARKS sieveMarks[6] = {{31, 7},
184                                     {73, 5},
185                                     {241, 4},

```

```

186         {1621, 3},
187         {UINT16_MAX, 2},
188         {UINT32_MAX, 1}};
189
190 const size_t MAX_SIEVE_MARKS = (sizeof(sieveMarks) / sizeof(sieveMarks[0]));
191 // clang-format on
192
193 /*** PrimeSieve()
194 // This function does a prime sieve over the input 'field' which has as its
195 // starting address the value in bnN. Since this initializes the Sieve
196 // using a precomputed field with the bits associated with 3, 5 and 7 already
197 // turned off, the value of pnN may need to be adjusted by a few counts to allow
198 // the precomputed field to be used without modification.
199 //
200 // To get better performance, one could address the issue of developing the
201 // composite numbers. When the size of the prime gets large, the time for doing
202 // the divisions goes up, noticeably. It could be better to develop larger composite
203 // numbers even if they need to be Crypt_Int*'s themselves. The object would be to
204 // reduce the number of times that the large prime is divided into a few large
205 // divides and then use smaller divides to get to the final 16 bit (or smaller)
206 // remainders.
207 LIB_EXPORT UINT32 PrimeSieve(Crypt_Int* bnN, // IN/OUT: number to sieve
208                             UINT32 fieldSize, // IN: size of the field area in bytes
209                             BYTE* field // IN: field
210 )
211 {
212     UINT32 i;
213     UINT32 j;
214     UINT32 fieldBits = fieldSize * 8;
215     UINT32 r;
216     BYTE* pField;
217     INT32 iter;
218     UINT32 adjust;
219     UINT32 mark = 0;
220     UINT32 count = sieveMarks[0].count;
221     UINT32 stop = sieveMarks[0].prime;
222     UINT32 composite;
223     UINT32 pList[8];
224     UINT32 next;
225
226     pAssert(field != NULL && bnN != NULL);
227
228     // If the remainder is odd, then subtracting the value will give an even number,
229     // but we want an odd number, so subtract the 105+rem. Otherwise, just subtract
230     // the even remainder.
231     adjust = (UINT32)ExtMath_ModWord(bnN, 105);
232     if(adjust & 1)
233         adjust += 105;
234
235     // Adjust the input number so that it points to the first number in a
236     // aligned field.
237     ExtMath_SubtractWord(bnN, bnN, adjust);
238     // pAssert(ExtMath_ModWord(bnN, 105) == 0);
239     pField = field;
240     for(i = fieldSize; i >= sizeof(seedValues);
241         pField += sizeof(seedValues), i -= sizeof(seedValues))
242     {
243         memcpy(pField, seedValues, sizeof(seedValues));
244     }
245     if(i != 0)
246         memcpy(pField, seedValues, i);
247
248     // Cycle through the primes, clearing bits
249     // Have already done 3, 5, and 7
250     iter = 7;
251

```

```

252 # define NEXT_PRIME(iter) (iter = RsaNextPrime(iter))
253 // Get the next N primes where N is determined by the mark in the sieveMarks
254 while((composite = NEXT_PRIME(iter)) != 0)
255 {
256     next      = 0;
257     i         = count;
258     pList[i--] = composite;
259     for(; i > 0; i--)
260     {
261         next      = NEXT_PRIME(iter);
262         pList[i] = next;
263         if(next != 0)
264             composite *= next;
265     }
266     // Get the remainder when dividing the base field address
267     // by the composite
268     composite = (UINT32)ExtMath_ModWord(bnN, composite);
269     // 'composite' is divisible by the composite components. for each of the
270     // composite components, divide 'composite'. That remainder (r) is used to
271     // pick a starting point for clearing the array. The stride is equal to the
272     // composite component. Note, the field only contains odd numbers. If the
273     // field were expanded to contain all numbers, then half of the bits would
274     // have already been cleared. We can save the trouble of clearing them a
275     // second time by having a stride of 2*next. Or we can take all of the even
276     // numbers out of the field and use a stride of 'next'
277     for(i = count; i > 0; i--)
278     {
279         next = pList[i];
280         if(next == 0)
281             goto done;
282         r = composite % next;
283         // these computations deal with the fact that we have picked a field-sized
284         // range that is aligned to a 105 count boundary. The problem is, this
285         // only contains odd numbers. If we take our prime guess and walk through
286         // the numbers using that prime as the 'stride', then every other 'stride'
287         // is going to be an even number. So, we are actually counting by 2 * the
288         // stride
289         // We want the count to start on an odd number at the start of our field.
290         // That is, we want to assume that we have counted up to the edge of the field
291         // by the 'stride' and now we are going to start flipping bits in the field
292         // as we continue to count up by 'stride'. If we take the base of our field and
293         // divide by the stride, we find out how much we find out how short the
294         // last quotient of count was from reaching the edge of the bit field. Say we get a
295         // of 3 and remainder of 1. This means that after 3 strides, we are 1 short
296         // of the start of the field and the next stride will either land within the
297         // field or step completely over it. The confounding factor is that our
298         // going only contains odd numbers and our stride is actually 2 * stride. If the
299         // quotient is even, then that means that when we add 2 * stride, we are
300         // to hit another even number. So, we have to know if we need to back off
301         // by 1 stride before we start counting by 2 * stride.
302         // We can tell from the remainder whether we are on an even or odd
303         // stride when we hit the beginning of the table. If we are on an odd
304         // stride
305         // (r & 1), we would start half a stride in (next - r)/2. If we are on an

```

```

304         // even stride, we need 0.5 strides (next - r/2) because the table only
has
305         // odd numbers. If the remainder happens to be zero, then the start of the
306         // table is on stride so no adjustment is necessary.
307         if(r & 1)
308             j = (next - r) / 2;
309         else if(r == 0)
310             j = 0;
311         else
312             j = next - (r / 2);
313         for(; j < fieldBits; j += next)
314             ClearBit(j, field, fieldSize);
315     }
316     if(next >= stop)
317     {
318         mark++;
319         if(mark >= MAX_SIEVE_MARKS)
320         {
321             // prime iteration should have broken out of the loop before this.
322             FAIL_EXIT(FATAL_ERROR_INTERNAL, i, 0);
323         }
324         count = sieveMarks[mark].count;
325         stop = sieveMarks[mark].prime;
326     }
327 }
328 done:
329     i = BitsInArray(field, fieldSize);
330
331 Exit:
332     INSTRUMENT_INC(totalFieldsSieved[PrimeIndex]);
333     INSTRUMENT_ADD(bitsInFieldAfterSieve[PrimeIndex], i);
334     INSTRUMENT_ADD(emptyFieldsSieved[PrimeIndex], (i == 0));
335     return i;
336 }
337
338 # ifdef SIEVE_DEBUG
339 static uint32_t fieldSize = 210;
340
341 /**SetFieldSize()
342 // Function to set the field size used for prime generation. Used for tuning.
343 LIB_EXPORT uint32_t SetFieldSize(uint32_t newFieldSize)
344 {
345     if(newFieldSize == 0 || newFieldSize > MAX_FIELD_SIZE)
346         fieldSize = MAX_FIELD_SIZE;
347     else
348         fieldSize = newFieldSize;
349     return fieldSize;
350 }
351 # endif // SIEVE_DEBUG
352
353 /**PrimeSelectWithSieve()
354 // This function will sieve the field around the input prime candidate. If the
355 // sieve field is not empty, one of the one bits in the field is chosen for testing
356 // with Miller-Rabin. If the value is prime, 'pnP' is updated with this value
357 // and the function returns success. If this value is not prime, another
358 // pseudo-random candidate is chosen and tested. This process repeats until
359 // all values in the field have been checked. If all bits in the field have
360 // been checked and none is prime, the function returns FALSE and a new random
361 // value needs to be chosen.
362 // Return Type: TPM_RC
363 //     TPM_RC_FAILURE    TPM in failure mode, probably due to entropy source
364 //     TPM_RC_SUCCESS    candidate is probably prime
365 //     TPM_RC_NO_RESULT  candidate is not prime and couldn't find and alternative
366 //                       in the field
367 LIB_EXPORT TPM_RC PrimeSelectWithSieve(
368     Crypt_Int* candidate, // IN/OUT: The candidate to filter

```



```

369     UINT32     e,           // IN: the exponent
370     RAND_STATE* rand        // IN: the random number generator state
371 )
372 {
373     BYTE    field[MAX_FIELD_SIZE];
374     UINT32  ones;
375     INT32   chosen;
376     CRYPT_PRIME_VAR(test);
377     UINT32  modE;
378     # ifnndef SIEVE_DEBUG
379         UINT32 fieldSize = MAX_FIELD_SIZE;
380     # endif
381     UINT32 primeSize;
382     //
383     // Adjust the field size and prime table list to fit the size of the prime
384     // being tested. This is done to try to optimize the trade-off between the
385     // dividing done for sieving and the time for Miller-Rabin. When the size
386     // of the prime is large, the cost of Miller-Rabin is fairly high, as is the
387     // cost of the sieving. However, the time for Miller-Rabin goes up considerably
388     // faster than the cost of dividing by a number of primes.
389     primeSize = ExtMath_SizeInBits(candidate);
390
391     if(primeSize <= 512)
392     {
393         RsaAdjustPrimeLimit(1024); // Use just the first 1024 primes
394     }
395     else if(primeSize <= 1024)
396     {
397         RsaAdjustPrimeLimit(4096); // Use just the first 4K primes
398     }
399     else
400     {
401         RsaAdjustPrimeLimit(0); // Use all available
402     }
403
404     // Save the low-order word to use as a search generator and make sure that
405     // it has some interesting range to it
406     uint32_t first = ExtMath_GetLeastSignificant32bits(candidate);
407     first |= 0x80000000;
408
409     // Sieve the field
410     ones = PrimeSieve(candidate, fieldSize, field);
411
412     // PrimeSieve shouldn't fail, but does call functions that may.
413     if(!g_inFailureMode)
414     {
415         pAssert(ones > 0 && ones < (fieldSize * 8));
416         for(; ones > 0; ones--)
417         {
418             // Decide which bit to look at and find its offset
419             chosen = FindNthSetBit((UINT16)fieldSize, field, ((first % ones) + 1));
420
421             if((chosen < 0) || (chosen >= (INT32)(fieldSize * 8)))
422                 FAIL(FATAL_ERROR_INTERNAL);
423
424             // Set this as the trial prime
425             ExtMath_AddWord(test, candidate, (crypt_uword_t)(chosen * 2));
426
427             // The exponent might not have been one of the tested primes so
428             // make sure that it isn't divisible and make sure that 0 != (p-1) mod e
429             // Note: This is the same as 1 != p mod e
430             modE = (UINT32)ExtMath_ModWord(test, e);
431             if((modE != 0) && (modE != 1) && MillerRabin(test, rand))
432             {
433                 ExtMath_Copy(candidate, test);
434                 return TPM_RC_SUCCESS;

```

```

435     }
436     // Clear the bit just tested
437     ClearBit(chosen, field, fieldSize);
438 }
439 // Ran out of bits and couldn't find a prime in this field
440 INSTRUMENT_INC(noPrimeFields[PrimeIndex]);
441 }
442 return (g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_NO_RESULT);
443 }
444
445 # if RSA_INSTRUMENT
446 static char a[256];
447
448 /*** PrintTuple()
449 char* PrintTuple(UINT32* i)
450 {
451     sprintf(a, "%d, %d, %d", i[0], i[1], i[2]);
452     return a;
453 }
454
455 # define CLEAR_VALUE(x) memset(x, 0, sizeof(x))
456
457 /*** RsaSimulationEnd()
458 void RsaSimulationEnd(void)
459 {
460     int i;
461     UINT32 averages[3];
462     UINT32 nonFirst = 0;
463     if((PrimeCounts[0] + PrimeCounts[1] + PrimeCounts[2]) != 0)
464     {
465         printf("Primes generated = %s\n", PrintTuple(PrimeCounts));
466         printf("Fields sieved = %s\n", PrintTuple(totalFieldsSieved));
467         printf("Fields with no primes = %s\n", PrintTuple(noPrimeFields));
468         printf("Primes checked with Miller-Rabin = %s\n",
469             PrintTuple(MillerRabinTrials));
470         for(i = 0; i < 3; i++)
471             averages[i] = (totalFieldsSieved[i] != 0
472                 ? bitsInFieldAfterSieve[i] / totalFieldsSieved[i]
473                 : 0);
474         printf("Average candidates in field %s\n", PrintTuple(averages));
475         for(i = 1; i < (sizeof(failedAtIteration) / sizeof(failedAtIteration[0]));
476             i++)
477             nonFirst += failedAtIteration[i];
478         printf("Miller-Rabin failures not in first round = %d\n", nonFirst);
479     }
480     CLEAR_VALUE(PrimeCounts);
481     CLEAR_VALUE(totalFieldsSieved);
482     CLEAR_VALUE(noPrimeFields);
483     CLEAR_VALUE(MillerRabinTrials);
484     CLEAR_VALUE(bitsInFieldAfterSieve);
485 }
486
487 /*** GetSieveStats()
488 LIB_EXPORT void GetSieveStats(
489     uint32_t* trials, uint32_t* emptyFields, uint32_t* averageBits)
490 {
491     uint32_t totalBits;
492     uint32_t fields;
493     *trials = MillerRabinTrials[0] + MillerRabinTrials[1] + MillerRabinTrials[2];
494     *emptyFields = noPrimeFields[0] + noPrimeFields[1] + noPrimeFields[2];
495     fields = totalFieldsSieved[0] + totalFieldsSieved[1] + totalFieldsSieved[2];
496     totalBits = bitsInFieldAfterSieve[0] + bitsInFieldAfterSieve[1]
497         + bitsInFieldAfterSieve[2];
498     if(fields != 0)
499         *averageBits = totalBits / fields;
500     else

```

```

501     *averageBits = 0;
502     CLEAR_VALUE(PrimeCounts);
503     CLEAR_VALUE(totalFieldsSieved);
504     CLEAR_VALUE(noPrimeFields);
505     CLEAR_VALUE(MillerRabinTrials);
506     CLEAR_VALUE(bitsInFieldAfterSieve);
507 }
508 # endif
509
510 #endif // RSA_KEY_SIEVE
511
512 #if !RSA_INSTRUMENT
513
514 /** RsaSimulationEnd()
515  * Stub for call when not doing instrumentation.
516  */
517 void RsaSimulationEnd(void)
518 {
519     return;
520 }
521 #endif

```

7.146 /tpm/src/crypt/CryptRand.c

```

1  /** Introduction
2  // This file implements a DRBG with a behavior according to SP800-90A using
3  // a block cypher. This is also compliant to ISO/IEC 18031:2011(E) C.3.2.
4  //
5  // A state structure is created for use by TPM.lib and functions
6  // within the CryptoEngine may use their own state structures when they need to have
7  // deterministic values.
8  //
9  // A debug mode is available that allows the random numbers generated for TPM.lib
10 // to be repeated during runs of the simulator. The switch for it is in
11 // TpmBuildSwitches.h. It is USE_DEBUG_RNG.
12 //
13 //
14 // This is the implementation layer of CTR DRBG mechanism as defined in SP800-90A
15 // and the functions are organized as closely as practical to the organization in
16 // SP800-90A. It is intended to be compiled as a separate module that is linked
17 // with a secure application so that both reside inside the same boundary
18 // [SP 800-90A 8.5]. The secure application in particular manages the accesses
19 // protected storage for the state of the DRBG instantiations, and supplies the
20 // implementation functions here with a valid pointer to the working state of the
21 // given instantiations (as a DRBG_STATE structure).
22 //
23 // This DRBG mechanism implementation does not support prediction resistance. Thus
24 // 'prediction_resistance_flag' is omitted from Instantiate_function(),
25 // Reseed_function(), Generate_function() argument lists [SP 800-90A 9.1, 9.2,
26 // 9.3], as well as from the working state data structure DRBG_STATE [SP 800-90A
27 // 9.1].
28 //
29 // This DRBG mechanism implementation always uses the highest security strength of
30 // available in the block ciphers. Thus 'requested_security_strength' parameter is
31 // omitted from Instantiate_function() and Generate_function() argument lists
32 // [SP 800-90A 9.1, 9.2, 9.3], as well as from the working state data structure
33 // DRBG_STATE [SP 800-90A 9.1].
34 //
35 // Internal functions (ones without Crypt prefix) expect validated arguments and
36 // therefore use assertions instead of runtime parameter checks and mostly return
37 // void instead of a status value.
38
39 #include "Tpm.h"
40
41 // Pull in the test vector definitions and define the space
42 #include "PRNG_TestVectors.h"

```

```

43
44 const BYTE DRBG_NistTestVector_Entropy[] = {DRBG_TEST_INITIATE_ENTROPY};
45 const BYTE DRBG_NistTestVector_GeneratedInterm[] = {DRBG_TEST_GENERATED_INTERM};
46
47 const BYTE DRBG_NistTestVector_EntropyReseed[] = {DRBG_TEST_RESEED_ENTROPY};
48 const BYTE DRBG_NistTestVector_Generated[] = {DRBG_TEST_GENERATED};
49
50 /** Derivation Functions
51 /** Description
52 // The functions in this section are used to reduce the personalization input values
53 // to make them usable as input for reseeding and instantiation. The overall
54 // behavior is intended to produce the same results as described in SP800-90A,
55 // section 10.4.2 "Derivation Function Using a Block Cipher Algorithm
56 // (Block_Cipher_df)." The code is broken into several subroutines to deal with the
57 // fact that the data used for personalization may come in several separate blocks
58 // such as a Template hash and a proof value and a primary seed.
59
60 /** Derivation Function Defines and Structures
61
62 #define DF_COUNT (DRBG_KEY_SIZE_WORDS / DRBG_IV_SIZE_WORDS + 1)
63 #if DRBG_KEY_SIZE_BITS != 128 && DRBG_KEY_SIZE_BITS != 256
64 # error "CryptRand.c only written for AES with 128- or 256-bit keys."
65 #endif
66
67 typedef tpmKeyScheduleAES DRBG_KEY_SCHEDULE;
68
69 typedef struct
70 {
71     DRBG_KEY_SCHEDULE keySchedule;
72     DRBG_IV iv[DF_COUNT];
73     DRBG_IV out1;
74     DRBG_IV buf;
75     int contents;
76 } DF_STATE, *PDF_STATE;
77
78 /** DfCompute()
79 // This function does the incremental update of the derivation function state. It
80 // encrypts the 'iv' value and XOR's the results into each of the blocks of the
81 // output. This is equivalent to processing all of input data for each output block.
82 static void DfCompute(PDF_STATE dfState)
83 {
84     int i;
85     int iv;
86     crypt_ushort_t* pIv;
87     crypt_ushort_t temp[DRBG_IV_SIZE_WORDS] = {0};
88     //
89     for(iv = 0; iv < DF_COUNT; iv++)
90     {
91         pIv = (crypt_ushort_t*)&dfState->iv[iv].words[0];
92         for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
93         {
94             temp[i] ^= pIv[i] ^ dfState->buf.words[i];
95         }
96         DRBG_ENCRYPT(&dfState->keySchedule, &temp, pIv);
97     }
98     for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
99         dfState->buf.words[i] = 0;
100     dfState->contents = 0;
101 }
102
103 /** DfStart()
104 // This initializes the output blocks with an encrypted counter value and
105 // initializes the key schedule.
106 static void DfStart(PDF_STATE dfState, uint32_t inputLength)
107 {
108     BYTE init[8];

```

```

109     int         i;
110     UINT32      drbgSeedSize = sizeof(DRBG_SEED);
111
112     const BYTE dfKey[DRBG_KEY_SIZE_BYTES] =
113     { 0x00,
114       0x01,
115       0x02,
116       0x03,
117       0x04,
118       0x05,
119       0x06,
120       0x07,
121       0x08,
122       0x09,
123       0x0a,
124       0x0b,
125       0x0c,
126       0x0d,
127       0x0e,
128       0x0f
129 #if DRBG_KEY_SIZE_BYTES > 16
130     ,
131     0x10,
132     0x11,
133     0x12,
134     0x13,
135     0x14,
136     0x15,
137     0x16,
138     0x17,
139     0x18,
140     0x19,
141     0x1a,
142     0x1b,
143     0x1c,
144     0x1d,
145     0x1e,
146     0x1f
147 #endif
148     };
149     memset(dfState, 0, sizeof(DF_STATE));
150     DRBG_ENCRYPT_SETUP(&dfKey[0], DRBG_KEY_SIZE_BITS, &dfState->keySchedule);
151     // Create the first chaining values
152     for(i = 0; i < DF_COUNT; i++)
153         ((BYTE*)&dfState->iv[i])[3] = (BYTE)i;
154     DfCompute(dfState);
155     // initialize the first 64 bits of the IV in a way that doesn't depend
156     // on the size of the words used.
157     UINT32_TO_BYTE_ARRAY(inputLength, init);
158     UINT32_TO_BYTE_ARRAY(drbgSeedSize, &init[4]);
159     memcpy(&dfState->iv[0], init, 8);
160     dfState->contents = 4;
161 }
162
163 /*** DfUpdate()
164  * This updates the state with the input data. A byte at a time is moved into the
165  * state buffer until it is full and then that block is encrypted by DfCompute().
166  */
167 static void DfUpdate(PDF_STATE dfState, int size, const BYTE* data)
168 {
169     while(size > 0)
170     {
171         int toFill = DRBG_IV_SIZE_BYTES - dfState->contents;
172         if(size < toFill)
173             toFill = size;
174         // Copy as many bytes as there are or until the state buffer is full
175         memcpy(&dfState->buf.bytes[dfState->contents], data, toFill);

```

```

175         // Reduce the size left by the amount copied
176         size -= toFill;
177         // Advance the data pointer by the amount copied
178         data += toFill;
179         // increase the buffer contents count by the amount copied
180         dfState->contents += toFill;
181         pAssert(dfState->contents <= DRBG_IV_SIZE_BYTES);
182         // If we have a full buffer, do a computation pass.
183         if(dfState->contents == DRBG_IV_SIZE_BYTES)
184             DfCompute(dfState);
185     }
186 }
187
188 /*** DfEnd()
189 // This function is called to get the result of the derivation function computation.
190 // If the buffer is not full, it is padded with zeros. The output buffer is
191 // structured to be the same as a DRBG_SEED value so that the function can return
192 // a pointer to the DRBG_SEED value in the DF_STATE structure.
193 static DRBG_SEED* DfEnd(PDF_STATE dfState)
194 {
195     // Since DfCompute is always called when a buffer is full, there is always
196     // space in the buffer for the terminator
197     dfState->buf.bytes[dfState->contents++] = 0x80;
198     // If the buffer is not full, pad with zeros
199     while(dfState->contents < DRBG_IV_SIZE_BYTES)
200         dfState->buf.bytes[dfState->contents++] = 0;
201     // Do a final state update
202     DfCompute(dfState);
203     return (DRBG_SEED*)&dfState->iv;
204 }
205
206 /*** DfBuffer()
207 // Function to take an input buffer and do the derivation function to produce a
208 // DRBG_SEED value that can be used in DRBG_Reseed();
209 static DRBG_SEED* DfBuffer(DRBG_SEED* output, // OUT: receives the result
210                             int size, // IN: size of the buffer to add
211                             BYTE* buf // IN: address of the buffer
212 )
213 {
214     DF_STATE dfState;
215     if(size == 0 || buf == NULL)
216         return NULL;
217     // Initialize the derivation function
218     DfStart(&dfState, size);
219     DfUpdate(&dfState, size, buf);
220     DfEnd(&dfState);
221     memcpy(output, &dfState.iv[0], sizeof(DRBG_SEED));
222     return output;
223 }
224
225 /*** DRBG_GetEntropy()
226 // Even though this implementation never fails, it may get blocked
227 // indefinitely long in the call to get entropy from the platform
228 // (DRBG_GetEntropy32()).
229 // This function is only used during instantiation of the DRBG for
230 // manufacturing and on each start-up after a non-orderly shutdown.
231 //
232 // Return Type: BOOL
233 //     TRUE(1)           requested entropy returned
234 //     FALSE(0)          entropy Failure
235 BOOL DRBG_GetEntropy(UINT32 requiredEntropy, // IN: requested number of bytes of full
236                                     // entropy
237                                     BYTE* entropy // OUT: buffer to return collected entropy
238 )
239 {
240     #if !USE_DEBUG_RNG

```



```

241
242     UINT32 obtainedEntropy;
243     INT32  returnedEntropy;
244
245     // If in debug mode, always use the self-test values for initialization
246     if(IsSelfTest())
247     {
248 #endif
249         // If doing simulated DRBG, then check to see if the
250         // entropyFailure condition is being tested
251         if(!IsEntropyBad())
252         {
253             // In self-test, the caller should be asking for exactly the seed
254             // size of entropy.
255             pAssert(requiredEntropy == sizeof(DRBG_NistTestVector_Entropy));
256             memcpy(entropy,
257                 DRBG_NistTestVector_Entropy,
258                 sizeof(DRBG_NistTestVector_Entropy));
259         }
260 #if !USE_DEBUG_RNG
261     }
262     else if(!IsEntropyBad())
263     {
264         // Collect entropy
265         // Note: In debug mode, the only "entropy" value ever returned
266         // is the value of the self-test vector.
267         for(returnedEntropy = 1, obtainedEntropy = 0;
268             obtainedEntropy < requiredEntropy && !IsEntropyBad();
269             obtainedEntropy += returnedEntropy)
270         {
271             returnedEntropy = _plat_GetEntropy(&entropy[obtainedEntropy],
272                 requiredEntropy - obtainedEntropy);
273             if(returnedEntropy <= 0)
274                 SetEntropyBad();
275         }
276     }
277 #endif
278     return !IsEntropyBad();
279 }
280
281 /*** IncrementIv()
282 // This function increments the IV value by 1. It is used by EncryptDRBG().
283 void IncrementIv(DRBG_IV* iv)
284 {
285     BYTE* ivP = ((BYTE*)iv) + DRBG_IV_SIZE_BYTES;
286     while(--ivP >= (BYTE*)iv) && ((*ivP = (*ivP + 1) & 0xFF) == 0))
287     ;
288 }
289
290 /*** EncryptDRBG()
291 // This does the encryption operation for the DRBG. It will encrypt
292 // the input state counter (IV) using the state key. Into the output
293 // buffer for as many times as it takes to generate the required
294 // number of bytes.
295 static BOOL EncryptDRBG(BYTE*          dOut,
296                        UINT32         dOutBytes,
297                        DRBG_KEY_SCHEDULE* keySchedule,
298                        DRBG_IV*       iv,
299                        UINT32* lastValue // Points to the last output value
300 )
301 {
302 #if FIPS_COMPLIANT
303     // For FIPS compliance, the DRBG has to do a continuous self-test to make sure
304     // that
305     // no two consecutive values are the same. This overhead is not incurred if the
306     TPM

```

```

305     // is not required to be FIPS compliant
306     //
307     UINT32 temp[DRBG_IV_SIZE_BYTES / sizeof(UINT32)];
308     int i;
309     BYTE* p;
310
311     for(; dOutBytes > 0;)
312     {
313         // Increment the IV before each encryption (this is what makes this
314         // different from normal counter-mode encryption
315         IncrementIv(iv);
316         DRBG_ENCRYPT(keySchedule, iv, temp);
317     // Expect a 16 byte block
318     # if DRBG_IV_SIZE_BITS != 128
319     #   error "Unsupported IV size in DRBG"
320     # endif
321         if((lastValue[0] == temp[0]) && (lastValue[1] == temp[1])
322           && (lastValue[2] == temp[2]) && (lastValue[3] == temp[3]))
323         {
324             FAIL_BOOL(FATAL_ERROR_ENTROPY);
325         }
326         lastValue[0] = temp[0];
327         lastValue[1] = temp[1];
328         lastValue[2] = temp[2];
329         lastValue[3] = temp[3];
330         i = MIN(dOutBytes, DRBG_IV_SIZE_BYTES);
331         dOutBytes -= i;
332         for(p = (BYTE*)temp; i > 0; i--)
333             *dOut++ = *p++;
334     }
335     #else // version without continuous self-test
336     NOT_REFERENCED(lastValue);
337     for(; dOutBytes >= DRBG_IV_SIZE_BYTES;
338          dOut = &dOut[DRBG_IV_SIZE_BYTES], dOutBytes -= DRBG_IV_SIZE_BYTES)
339     {
340         // Increment the IV
341         IncrementIv(iv);
342         DRBG_ENCRYPT(keySchedule, iv, dOut);
343     }
344     // If there is a partial, generate into a block-sized
345     // temp buffer and copy to the output.
346     if(dOutBytes != 0)
347     {
348         BYTE temp[DRBG_IV_SIZE_BYTES];
349         // Increment the IV
350         IncrementIv(iv);
351         DRBG_ENCRYPT(keySchedule, iv, temp);
352         memcpy(dOut, temp, dOutBytes);
353     }
354     #endif
355     return TRUE;
356 }
357
358 /*** DRBG_Update()
359 // This function performs the state update function.
360 // According to SP800-90A, a temp value is created by doing CTR mode
361 // encryption of 'providedData' and replacing the key and IV with
362 // these values. The one difference is that, with counter mode, the
363 // IV is incremented after each block is encrypted and in this
364 // operation, the counter is incremented before each block is
365 // encrypted. This function implements an 'optimized' version
366 // of the algorithm in that it does the update of the drbgState->seed
367 // in place and then 'providedData' is XORed into drbgState->seed
368 // to complete the encryption of 'providedData'. This works because
369 // the IV is the last thing that gets encrypted.
370 //

```

```

371 static BOOL DRBG_Update(
372     DRBG_STATE* drbgState, // IN:OUT state to update
373     DRBG_KEY_SCHEDULE* keySchedule, // IN: the key schedule (optional)
374     DRBG_SEED* providedData // IN: additional data
375 )
376 {
377     UINT32 i;
378     BYTE* temp = (BYTE*)&drbgState->seed;
379     DRBG_KEY* key = pDRBG_KEY(&drbgState->seed);
380     DRBG_IV* iv = pDRBG_IV(&drbgState->seed);
381     DRBG_KEY_SCHEDULE localKeySchedule;
382     //
383     pAssert(drbgState->magic == DRBG_MAGIC);
384
385     // If an key schedule was not provided, make one
386     if(keySchedule == NULL)
387     {
388         if(DRBG_ENCRYPT_SETUP((BYTE*)key, DRBG_KEY_SIZE_BITS, &localKeySchedule) != 0)
389         {
390             FAIL_BOOL(FATAL_ERROR_INTERNAL);
391         }
392         keySchedule = &localKeySchedule;
393     }
394     // Encrypt the temp value
395
396     EncryptDRBG(temp, sizeof(DRBG_SEED), keySchedule, iv, drbgState->lastValue);
397     if(providedData != NULL)
398     {
399         BYTE* pP = (BYTE*)providedData;
400         for(i = DRBG_SEED_SIZE_BYTES; i != 0; i--)
401             *temp++ ^= *pP++;
402     }
403     // Since temp points to the input key and IV, we are done and
404     // don't need to copy the resulting 'temp' to drbgState->seed
405     return TRUE;
406 }
407
408 /*** DRBG_Reseed()
409 // This function is used when reseeding of the DRBG is required. If
410 // entropy is provided, it is used in lieu of using hardware entropy.
411 // Note: the provided entropy must be the required size.
412 //
413 // Return Type: BOOL
414 //     TRUE(1)      reseed succeeded
415 //     FALSE(0)     reseed failed, probably due to the entropy generation
416 BOOL DRBG_Reseed(DRBG_STATE* drbgState, // IN: the state to update
417     DRBG_SEED* providedEntropy, // IN: entropy
418     DRBG_SEED* additionalData // IN:
419 )
420 {
421     DRBG_SEED seed;
422
423     pAssert((drbgState != NULL) && (drbgState->magic == DRBG_MAGIC));
424
425     if(providedEntropy == NULL)
426     {
427         providedEntropy = &seed;
428         if(!DRBG_GetEntropy(sizeof(DRBG_SEED), (BYTE*)providedEntropy))
429             return FALSE;
430     }
431     if(additionalData != NULL)
432     {
433         unsigned int i;
434
435         // XOR the provided data into the provided entropy
436         for(i = 0; i < sizeof(DRBG_SEED); i++)

```

```

437         ((BYTE*)providedEntropy)[i] ^= ((BYTE*)additionalData)[i];
438     }
439     DRBG_Update(drbgState, NULL, providedEntropy);
440
441     drbgState->reseedCounter = 1;
442
443     return TRUE;
444 }
445
446 /*** DRBG_SelfTest()
447 // This is run when the DRBG is instantiated and at startup.
448 //
449 // Return Type: BOOL
450 //     TRUE(1)      test OK
451 //     FALSE(0)     test failed
452 BOOL DRBG_SelfTest(void)
453 {
454     BYTE        buf[sizeof(DRBG_NistTestVector_Generated)];
455     DRBG_SEED    seed;
456     UINT32       i;
457     BYTE*        p;
458     DRBG_STATE   testState;
459     //
460     pAssert(!IsSelfTest());
461
462     SetSelfTest();
463     SetDrbgTested();
464     // Do an instantiate
465     if(!DRBG_Instantiate(&testState, 0, NULL))
466         return FALSE;
467 #if DRBG_DEBUG_PRINT
468     dbgDumpMemBlock(
469         pDRBG_KEY(&testState), DRBG_KEY_SIZE_BYTES, "Key after Instantiate");
470     dbgDumpMemBlock(
471         pDRBG_IV(&testState), DRBG_IV_SIZE_BYTES, "Value after Instantiate");
472 #endif
473     if(DRBG_Generate((RAND_STATE*)&testState, buf, sizeof(buf)) == 0)
474         return FALSE;
475 #if DRBG_DEBUG_PRINT
476     dbgDumpMemBlock(
477         pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES, "Key after 1st Generate");
478     dbgDumpMemBlock(
479         pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES, "Value after 1st Generate");
480 #endif
481     if(memcmp(buf, DRBG_NistTestVector_GeneratedInterm, sizeof(buf)) != 0)
482         return FALSE;
483     memcpy(seed.bytes, DRBG_NistTestVector_EntropyReseed, sizeof(seed));
484     DRBG_Reseed(&testState, &seed, NULL);
485 #if DRBG_DEBUG_PRINT
486     dbgDumpMemBlock((BYTE*)pDRBG_KEY(&testState.seed),
487                     DRBG_KEY_SIZE_BYTES,
488                     "Key after 2nd Generate");
489     dbgDumpMemBlock((BYTE*)pDRBG_IV(&testState.seed),
490                     DRBG_IV_SIZE_BYTES,
491                     "Value after 2nd Generate");
492     dbgDumpMemBlock(buf, sizeof(buf), "2nd Generated");
493 #endif
494     if(DRBG_Generate((RAND_STATE*)&testState, buf, sizeof(buf)) == 0)
495         return FALSE;
496     if(memcmp(buf, DRBG_NistTestVector_Generated, sizeof(buf)) != 0)
497         return FALSE;
498     ClearSelfTest();
499
500     DRBG_Uninstantiate(&testState);
501     for(p = (BYTE*)&testState, i = 0; i < sizeof(DRBG_STATE); i++)
502     {

```

```

503         if(*p++)
504             return FALSE;
505     }
506     // Simulate hardware failure to make sure that we get an error when
507     // trying to instantiate
508     SetEntropyBad();
509     if(DRBG_Instantiate(&testState, 0, NULL))
510         return FALSE;
511     ClearEntropyBad();
512
513     return TRUE;
514 }
515
516 /** Public Interface
517 **** Description
518 // The functions in this section are the interface to the RNG. These
519 // are the functions that are used by TPM.lib.
520
521 **** CryptRandomStir()
522 // This function is used to cause a reseed. A DRBG_SEED amount of entropy is
523 // collected from the hardware and then additional data is added.
524 //
525 // Return Type: TPM_RC
526 //          TPM_RC_NO_RESULT      failure of the entropy generator
527 LIB_EXPORT TPM_RC CryptRandomStir(UINT16 additionalDataSize, BYTE* additionalData)
528 {
529 #if !USE_DEBUG_RNG
530     DRBG_SEED tmpBuf;
531     DRBG_SEED dfResult;
532     //
533     // All reseed with outside data starts with a buffer full of entropy
534     if(!DRBG_GetEntropy(sizeof(tmpBuf), (BYTE*)&tmpBuf))
535         return TPM_RC_NO_RESULT;
536
537     DRBG_Reseed(&drbgDefault,
538                &tmpBuf,
539                DfBuffer(&dfResult, additionalDataSize, additionalData));
540     drbgDefault.reseedCounter = 1;
541
542     return TPM_RC_SUCCESS;
543 #else
544 // If doing debug, use the input data as the initial setting for the RNG state
545 // so that the test can be reset at any time.
546 // Note: If this is called with a data size of 0 or less, nothing happens. The
547 // presumption is that, in a debug environment, the caller will have specific
548 // values for initialization, so this check is just a simple way to prevent
549 // inadvertent programming errors from screwing things up. This doesn't use an
550 // pAssert() because the non-debug version of this function will accept these
551 // parameters as meaning that there is no additionalData and only hardware
552 // entropy is used.
553 if((additionalDataSize > 0) && (additionalData != NULL))
554 {
555     memset(drbgDefault.seed.bytes, 0, sizeof(drbgDefault.seed.bytes));
556     memcpy(drbgDefault.seed.bytes,
557            additionalData,
558            MIN(additionalDataSize, sizeof(drbgDefault.seed.bytes)));
559 }
560 drbgDefault.reseedCounter = 1;
561
562     return TPM_RC_SUCCESS;
563 #endif
564 }
565
566
567 /** CryptRandomGenerate()
568 // Generate a 'randomSize' number of random bytes.

```

```

569 LIB_EXPORT UINT16 CryptRandomGenerate(UINT16 randomSize, BYTE* buffer)
570 {
571     return DRBG_Generate((RAND_STATE*)&drbgDefault, buffer, randomSize);
572 }
573
574 /*** DRBG_InstantiateSeededKdf()
575 // This function is used to instantiate a KDF-based RNG. This is used for derivations.
576 // This function always returns TRUE.
577 LIB_EXPORT BOOL DRBG_InstantiateSeededKdf(
578     KDF_STATE* state, // OUT: buffer to hold the state
579     TPM_ALG_ID hashAlg, // IN: hash algorithm
580     TPM_ALG_ID kdf, // IN: the KDF to use
581     TPM2B* seed, // IN: the seed to use
582     const TPM2B* label, // IN: a label for the generation process.
583     TPM2B* context, // IN: the context value
584     UINT32 limit // IN: Maximum number of bits from the KDF
585 )
586 {
587     state->magic = KDF_MAGIC;
588     state->limit = limit;
589     state->seed = seed;
590     state->hash = hashAlg;
591     state->kdf = kdf;
592     state->label = label;
593     state->context = context;
594     state->digestSize = CryptHashGetDigestSize(hashAlg);
595     state->counter = 0;
596     state->residual.t.size = 0;
597     return TRUE;
598 }
599
600 /*** DRBG_AdditionalData()
601 // Function to reseed the DRBG with additional entropy. This is normally called
602 // before computing the protection value of a primary key in the Endorsement
603 // hierarchy.
604 LIB_EXPORT void DRBG_AdditionalData(DRBG_STATE* drbgState, // IN:OUT state to update
605     TPM2B* additionalData // IN: value to incorporate
606 )
607 {
608     DRBG_SEED dfResult;
609     if(drbgState->magic == DRBG_MAGIC)
610     {
611         DfBuffer(&dfResult, additionalData->size, additionalData->buffer);
612         DRBG_Reseed(drbgState, &dfResult, NULL);
613     }
614 }
615
616 /*** DRBG_InstantiateSeeded()
617 // This function is used to instantiate a random number generator from seed values.
618 // The nominal use of this generator is to create sequences of pseudo-random
619 // numbers from a seed value.
620 //
621 // Return Type: TPM_RC
622 // TPM_RC_FAILURE DRBG self-test failure
623 LIB_EXPORT TPM_RC DRBG_InstantiateSeeded(
624     DRBG_STATE* drbgState, // IN/OUT: buffer to hold the state
625     const TPM2B* seed, // IN: the seed to use
626     const TPM2B* purpose, // IN: a label for the generation process.
627     const TPM2B* name, // IN: name of the object
628     const TPM2B* additional // IN: additional data
629 )
630 {
631     DF_STATE dfState;
632     int totalInputSize;
633     // DRBG should have been tested, but...
634     if(!IsDrbgTested() && !DRBG_SelfTest())

```



```

635     {
636         FAIL_RC(FATAL_ERROR_SELF_TEST);
637     }
638     // Initialize the DRBG state
639     memset(&drbgState, 0, sizeof(DRBG_STATE));
640     drbgState->magic = DRBG_MAGIC;
641
642     // Size all of the values
643     totalInputSize = (seed != NULL) ? seed->size : 0;
644     totalInputSize += (purpose != NULL) ? purpose->size : 0;
645     totalInputSize += (name != NULL) ? name->size : 0;
646     totalInputSize += (additional != NULL) ? additional->size : 0;
647
648     // Initialize the derivation
649     DfStart(&dfState, totalInputSize);
650
651     // Run all the input strings through the derivation function
652     if(seed != NULL)
653         DfUpdate(&dfState, seed->size, seed->buffer);
654     if(purpose != NULL)
655         DfUpdate(&dfState, purpose->size, purpose->buffer);
656     if(name != NULL)
657         DfUpdate(&dfState, name->size, name->buffer);
658     if(additional != NULL)
659         DfUpdate(&dfState, additional->size, additional->buffer);
660
661     // Used the derivation function output as the "entropy" input. This is not
662     // how it is described in SP800-90A but this is the equivalent function
663     DRBG_Reseed((DRBG_STATE*)drbgState, DfEnd(&dfState), NULL);
664
665     return TPM_RC_SUCCESS;
666 }
667
668 /*** CryptRandStartup()
669 // This function is called when TPM_Startup is executed. This function always returns
670 // TRUE.
671 LIB_EXPORT BOOL CryptRandStartup(void)
672 {
673 #if !DRBG_STATE_SAVE
674     // If not saved in NV, re-instantiate on each startup
675     return DRBG_Instantiate(&drbgDefault, 0, NULL);
676 #else
677     // If the running state is saved in NV, NV has to be loaded before it can
678     // be updated
679     if(go.drbgState.magic == DRBG_MAGIC)
680         return DRBG_Reseed(&go.drbgState, NULL, NULL);
681     else
682         return DRBG_Instantiate(&go.drbgState, 0, NULL);
683 #endif
684 }
685
686 /*** CryptRandInit()
687 // This function is called when _TPM_Init is being processed.
688 //
689 // Return Type: BOOL
690 //      TRUE(1)      success
691 //      FALSE(0)     failure
692 LIB_EXPORT BOOL CryptRandInit(void)
693 {
694 #if !USE_DEBUG_RNG
695     _plat__GetEntropy(NULL, 0);
696 #endif
697     return DRBG_SelfTest();
698 }
699
700 /*** DRBG_Generate()

```

```

701 // This function generates a random sequence according SP800-90A.
702 // If 'random' is not NULL, then 'randomSize' bytes of random values are generated.
703 // If 'random' is NULL or 'randomSize' is zero, then the function returns
704 // zero without generating any bits or updating the reseed counter.
705 // This function returns the number of bytes produced which could be less than the
706 // number requested if the request is too large ("too large" is implementation
707 // dependent.)
708 LIB_EXPORT UINT16 DRBG_Generate(
709     RAND_STATE* state,
710     BYTE* random,          // OUT: buffer to receive the random values
711     UINT16 randomSize      // IN: the number of bytes to generate
712 )
713 {
714     if(state == NULL)
715         state = (RAND_STATE*)&drbgDefault;
716     if(random == NULL)
717         return 0;
718
719     // If the caller used a KDF state, generate a sequence from the KDF not to
720     // exceed the limit.
721     if(state->kdf.magic == KDF_MAGIC)
722     {
723         KDF_STATE* kdf      = (KDF_STATE*)state;
724         UINT32 counter      = (UINT32)kdf->counter;
725         INT32 bytesLeft     = randomSize;
726         //
727         // If the number of bytes to be returned would put the generator
728         // over the limit, then return 0
729         if(((kdf->counter * kdf->digestSize) + randomSize) * 8) > kdf->limit)
730             return 0;
731         // Process partial and full blocks until all requested bytes provided
732         while(bytesLeft > 0)
733         {
734             // If there is any residual data in the buffer, copy it to the output
735             // buffer
736             if(kdf->residual.t.size > 0)
737             {
738                 INT32 size;
739                 //
740                 // Don't use more of the residual than will fit or more than are
741                 // available
742                 size = MIN(kdf->residual.t.size, bytesLeft);
743
744                 // Copy some or all of the residual to the output. The residual is
745                 // at the end of the buffer. The residual might be a full buffer.
746                 MemoryCopy(
747                     random,
748                     &kdf->residual.t.buffer[kdf->digestSize - kdf->residual.t.size],
749                     size);
750
751                 // Advance the buffer pointer
752                 random += size;
753
754                 // Reduce the number of bytes left to get
755                 bytesLeft -= size;
756
757                 // And reduce the residual size appropriately
758                 kdf->residual.t.size -= (UINT16)size;
759             }
760             else
761             {
762                 UINT16 blocks = (UINT16)(bytesLeft / kdf->digestSize);
763                 //
764                 // Get the number of required full blocks
765                 if(blocks > 0)
766                 {

```

```

767         UINT16 size = blocks * kdf->digestSize;
768         // Get some number of full blocks and put them in the return
buffer
769         CryptKDFa(kdf->hash,
770                 kdf->seed,
771                 kdf->label,
772                 kdf->context,
773                 NULL,
774                 kdf->limit,
775                 random,
776                 &counter,
777                 blocks);
778
779         // reduce the size remaining to be moved and advance the pointer
780         bytesLeft -= size;
781         random += size;
782     }
783     else
784     {
785         // Fill the residual buffer with a full block and then loop to
786         // top to get part of it copied to the output.
787         kdf->residual.t.size = CryptKDFa(kdf->hash,
788                                         kdf->seed,
789                                         kdf->label,
790                                         kdf->context,
791                                         NULL,
792                                         kdf->limit,
793                                         kdf->residual.t.buffer,
794                                         &counter,
795                                         1);
796     }
797 }
798 }
799 kdf->counter = counter;
800 return randomSize;
801 }
802 else if(state->drbg.magic == DRBG_MAGIC)
803 {
804     DRBG_STATE* drbgState = (DRBG_STATE*)state;
805     DRBG_KEY_SCHEDULE keySchedule;
806     DRBG_SEED* seed = &drbgState->seed;
807
808     if(drbgState->reseedCounter >= CTR_DRBG_MAX_REQUESTS_PER_RESEED)
809     {
810         if(drbgState == &drbgDefault)
811         {
812             DRBG_Reseed(drbgState, NULL, NULL);
813             if(IsEntropyBad() && !IsSelfTest())
814                 return 0;
815         }
816         else
817         {
818             // If this is a PRNG then the only way to get
819             // here is if the SW has run away.
820             FAIL_IMMEDIATE(FATAL_ERROR_INTERNAL, 0);
821         }
822     }
823     // if the allowed number of bytes in a request is larger than the
824     // less than the number of bytes that can be requested, then check
825     #if UINT16_MAX >= CTR_DRBG_MAX_BYTES_PER_REQUEST
826     if(randomSize > CTR_DRBG_MAX_BYTES_PER_REQUEST)
827         randomSize = CTR_DRBG_MAX_BYTES_PER_REQUEST;
828     #endif
829     // Create encryption schedule
830     if(DRBG_ENCRYPT_SETUP(
831         (BYTE*)pDRBG_KEY(seed), DRBG_KEY_SIZE_BITS, &keySchedule)

```

```

832         != 0)
833     {
834         FAIL_IMMEDIATE(FATAL_ERROR_INTERNAL, 0);
835     }
836     // Generate the random data
837     EncryptDRBG(
838         random, randomSize, &keySchedule, pDRBG_IV(seed), drbgState->lastValue);
839     // Do a key update
840     DRBG_Update(drbgState, &keySchedule, NULL);
841
842     // Increment the reseed counter
843     drbgState->reseedCounter += 1;
844 }
845 else
846 {
847     // invalid DRBG state structure
848     FAIL_IMMEDIATE(FATAL_ERROR_INTERNAL, 0);
849 }
850 return randomSize;
851 }
852
853 /*** DRBG_Instantiate()
854 // This is CTR_DRBG_Instantiate_algorithm() from [SP 800-90A 10.2.1.3.1].
855 // This is called when a the TPM DRBG is to be instantiated. This is
856 // called to instantiate a DRBG used by the TPM for normal
857 // operations.
858 //
859 // Return Type: BOOL
860 //     TRUE(1)           instantiation succeeded
861 //     FALSE(0)          instantiation failed
862 LIB_EXPORT BOOL DRBG_Instantiate(
863     DRBG_STATE* drbgState,      // OUT: the instantiated value
864     UINT16      pSize,          // IN: Size of personalization string
865     BYTE*        personalization // IN: The personalization string
866 )
867 {
868     DRBG_SEED seed;
869     DRBG_SEED dfResult;
870     //
871     pAssert((pSize == 0) || (pSize <= sizeof(seed)) || (personalization != NULL));
872     // If the DRBG has not been tested, test when doing an instantiation. Since
873     // Instantiation is called during self test, make sure we don't get stuck in a
874     // loop.
875     if(!IsDrbgTested() && !IsSelfTest() && !DRBG_SelfTest())
876         return FALSE;
877     // If doing a self test, DRBG_GetEntropy will return the NIST
878     // test vector value.
879     if(!DRBG_GetEntropy(sizeof(seed), (BYTE*)&seed))
880         return FALSE;
881     // set everything to zero
882     memset(drbgState, 0, sizeof(DRBG_STATE));
883     drbgState->magic = DRBG_MAGIC;
884
885     // Steps 1, 2, 3, 6, 7 of SP 800-90A 10.2.1.3.1 are exactly what
886     // reseeding does. So, do a reduction on the personalization value (if any)
887     // and do a reseed.
888     DRBG_Reseed(drbgState, &seed, DfBuffer(&dfResult, pSize, personalization));
889
890     return TRUE;
891 }
892
893 /*** DRBG_Uninstantiate()
894 // This is Uninstantiate_function() from [SP 800-90A 9.4].
895 //
896 // Return Type: TPM_RC
897 //     TPM_RC_VALUE      not a valid state

```

```

898 LIB_EXPORT TPM_RC DRBG_Uninstantiate(
899     DRBG_STATE* drbgState // IN/OUT: working state to erase
900 )
901 {
902     if((drbgState == NULL) || (drbgState->magic != DRBG_MAGIC))
903         return TPM_RC_VALUE;
904     memset(drbgState, 0, sizeof(DRBG_STATE));
905     return TPM_RC_SUCCESS;
906 }

```

7.147 /tpm/src/crypt/CryptRsa.c

```

1  /** Introduction
2  //
3  // This file contains implementation of cryptographic primitives for RSA.
4  // Vendors may replace the implementation in this file with their own library
5  // functions.
6
7  /** Includes
8  // Need this define to get the 'private' defines for this function
9  #define CRYPT_RSA_C
10 #include "Tpm.h"
11 #include "TpmMath_Util_fp.h"
12
13 #if ALG_RSA
14
15 /** Obligatory Initialization Functions
16
17 /** CryptRsaInit()
18 // Function called at _TPM_Init().
19 BOOL CryptRsaInit(void)
20 {
21     return TRUE;
22 }
23
24 /** CryptRsaStartup()
25 // Function called at TPM2_Startup()
26 BOOL CryptRsaStartup(void)
27 {
28     return TRUE;
29 }
30
31 /** Internal Functions
32
33 /** RsaInitializeExponent()
34 // This function initializes the bignum data structure that holds the private
35 // exponent. This function returns the pointer to the private exponent value so that
36 // it can be used in an initializer for a data declaration.
37
38 static privateExponent* RsaInitializeExponent(privateExponent* Z)
39 {
40     // verify privateExponent packing matches the usage of the bn pointer as an
41     // array in below function
42     MUST_BE(offsetof(privateExponent, Q) == sizeof_MEMBER(privateExponent, P));
43
44     Crypt_Int** bn = (Crypt_Int**) &Z->P;
45     int i;
46     //
47     for(i = 0; i < 5; i++)
48     {
49         bn[i] = (Crypt_Int*) &(Z->entries[i]);
50         ExtMath_Initialize_Int(bn[i], MAX_RSA_KEY_BITS / 2);
51     }
52     return Z;
53 }

```

```

54
55 /** MakePgreaterThanQ()
56 // This function swaps the pointers for P and Q if Q happens to be larger than Q.
57 static void MakePgreaterThanQ(privateExponent* Z)
58 {
59     if(ExtMath_UnsignedCmp(Z->P, Z->Q) < 0)
60     {
61         Crypt_Int* bnT = Z->P;
62         Z->P           = Z->Q;
63         Z->Q           = bnT;
64     }
65 }
66
67 /** PackExponent()
68 // This function takes the bignum private exponent and converts it into TPM2B form.
69 // In this form, the size field contains the overall size of the packed data. The
70 // buffer contains 5, equal sized values in P, Q, dP, dQ, qInv order. For example, if
71 // a key has a 2Kb public key, then the packed private key will contain 5, 1Kb values.
72 // This form makes it relatively easy to load and save the values without changing
73 // the normal unmarshaling to do anything more than allow a larger TPM2B for the
74 // private key. Also, when exporting the value, all that is needed is to change the
75 // size field of the private key in order to save just the P value.
76 // Return Type: BOOL
77 //     TRUE(1)      success
78 //     FALSE(0)     failure           // The data is too big to fit
79 static BOOL PackExponent(TPM2B_PRIVATE_KEY_RSA* packed, privateExponent* Z)
80 {
81     int i;
82     UINT16 primeSize = (UINT16)BITS_TO_BYTES(ExtMath_MostSigBitNum(Z->P));
83     UINT16 pS        = primeSize;
84     //
85     pAssert((primeSize * 5) <= sizeof(packed->t.buffer));
86     packed->t.size = (primeSize * 5) + RSA_prime_flag;
87     for(i = 0; i < 5; i++)
88         if(!ExtMath_IntToBytes(
89             (Crypt_Int*)&Z->entries[i], &packed->t.buffer[primeSize * i], &pS))
90             return FALSE;
91     if(pS != primeSize)
92         return FALSE;
93     return TRUE;
94 }
95
96 /** UnpackExponent()
97 // This function unpacks the private exponent from its TPM2B form into its bignum
98 // form.
99 // Return Type: BOOL
100 //     TRUE(1)      success
101 //     FALSE(0)     TPM2B is not the correct size
102 static BOOL UnpackExponent(TPM2B_PRIVATE_KEY_RSA* b, privateExponent* Z)
103 {
104     UINT16 primeSize = b->t.size & ~RSA_prime_flag;
105     int i;
106     Crypt_Int** bn = &Z->P;
107     //
108     GOTO_ERROR_UNLESS(b->t.size & RSA_prime_flag);
109     RsaInitializeExponent(Z);
110     GOTO_ERROR_UNLESS((primeSize % 5) == 0);
111     primeSize /= 5;
112     for(i = 0; i < 5; i++)
113         GOTO_ERROR_UNLESS(
114             ExtMath_IntFromBytes(bn[i], &b->t.buffer[primeSize * i], primeSize)
115             != NULL);
116     MakePgreaterThanQ(Z);
117     return TRUE;
118 Error:
119     return FALSE;

```



```

120 }
121
122 /*** ComputePrivateExponent()
123 // This function computes the private exponent from the primes.
124 // Return Type: BOOL
125 //     TRUE(1)          success
126 //     FALSE(0)         failure
127 static BOOL ComputePrivateExponent(
128     Crypt_Int*      pubExp, // IN: the public exponent
129     privateExponent* Z      // IN/OUT: on input, has primes P and Q. On
130                             // output, has P, Q, dP, dQ, and pInv
131 )
132 {
133     BOOL pOK;
134     BOOL qOK;
135     CRYPT_PRIME_VAR(pT);
136     //
137     // make p the larger value so that m2 is always less than p
138     MakePgreaterThanQ(Z);
139
140     //dP = (1/e) mod (p-1)
141     pOK = ExtMath_SubtractWord(pT, Z->P, 1);
142     pOK = pOK && ExtMath_ModInverse(Z->dP, pubExp, pT);
143     //dQ = (1/e) mod (q-1)
144     qOK = ExtMath_SubtractWord(pT, Z->Q, 1);
145     qOK = qOK && ExtMath_ModInverse(Z->dQ, pubExp, pT);
146     // qInv = (1/q) mod p
147     if(pOK && qOK)
148         pOK = qOK = ExtMath_ModInverse(Z->qInv, Z->Q, Z->P);
149     if(!pOK)
150         ExtMath_SetWord(Z->P, 0);
151     if(!qOK)
152         ExtMath_SetWord(Z->Q, 0);
153     return pOK && qOK;
154 }
155
156 /*** RsaPrivateKeyOp()
157 // This function is called to do the exponentiation with the private key. Compile
158 // options allow use of the simple (but slow) private exponent, or the more complex
159 // but faster CRT method.
160 // Return Type: BOOL
161 //     TRUE(1)          success
162 //     FALSE(0)         failure
163 static BOOL RsaPrivateKeyOp(Crypt_Int* inOut, // IN/OUT: number to be exponentiated
164                             privateExponent* Z)
165 {
166     CRYPT_RSA_VAR(M1);
167     CRYPT_RSA_VAR(M2);
168     CRYPT_RSA_VAR(M);
169     CRYPT_RSA_VAR(H);
170     //
171     MakePgreaterThanQ(Z);
172     // m1 = cdP mod p
173     GOTO_ERROR_UNLESS(ExtMath_ModExp(M1, inOut, Z->dP, Z->P));
174     // m2 = cdQ mod q
175     GOTO_ERROR_UNLESS(ExtMath_ModExp(M2, inOut, Z->dQ, Z->Q));
176     // h = qInv * (m1 - m2) mod p = qInv * (m1 + P - m2) mod P because Q < P
177     // so m2 < P
178     GOTO_ERROR_UNLESS(ExtMath_Subtract(H, Z->P, M2));
179     GOTO_ERROR_UNLESS(ExtMath_Add(H, H, M1));
180     GOTO_ERROR_UNLESS(ExtMath_ModMult(H, H, Z->qInv, Z->P));
181     // m = m2 + h * q
182     GOTO_ERROR_UNLESS(ExtMath_Multiply(M, H, Z->Q));
183     GOTO_ERROR_UNLESS(ExtMath_Add(inOut, M2, M));
184     return TRUE;
185 Error:

```

```

186     return FALSE;
187 }
188
189 /*** RSAEP()
190 // This function performs the RSAEP operation defined in PKCS#1v2.1. It is
191 // an exponentiation of a value ('m') with the public exponent ('e'), modulo
192 // the public ('n').
193 //
194 // Return Type: TPM_RC
195 //     TPM_RC_VALUE      number to exponentiate is larger than the modulus
196 //
197 static TPM_RC RSAEP(TPM2B* dInOut, // IN: size of the encrypted block and the size of
198                        // the encrypted value. It must be the size of
199                        // the modulus.
200                        // OUT: the encrypted data. Will receive the
201                        // decrypted value
202                        OBJECT* key // IN: the key to use
203 )
204 {
205     TPM2B_TYPE(4BYTES, 4);
206     TPM2B_4BYTES e2B;
207     UINT32 e = key->publicArea.parameters.rsaDetail.exponent;
208     //
209     if(e == 0)
210         e = RSA_DEFAULT_PUBLIC_EXPONENT;
211     UINT32_TO_BYTE_ARRAY(e, e2B.t.buffer);
212     e2B.t.size = 4;
213     return ModExpB(dInOut->size,
214                   dInOut->buffer,
215                   dInOut->size,
216                   dInOut->buffer,
217                   e2B.t.size,
218                   e2B.t.buffer,
219                   key->publicArea.unique.rsa.t.size,
220                   key->publicArea.unique.rsa.t.buffer);
221 }
222
223 /*** RSADP()
224 // This function performs the RSADP operation defined in PKCS#1v2.1. It is
225 // an exponentiation of a value ('c') with the private exponent ('d'), modulo
226 // the public modulus ('n'). The decryption is in place.
227 //
228 // This function also checks the size of the private key. If the size indicates
229 // that only a prime value is present, the key is converted to being a private
230 // exponent.
231 //
232 // Return Type: TPM_RC
233 //     TPM_RC_SIZE      the value to decrypt is larger than the modulus
234 //
235 static TPM_RC RSADP(TPM2B* inOut, // IN/OUT: the value to encrypt
236                     OBJECT* key // IN: the key
237 )
238 {
239     CRYPT_RSA_INITIALIZED(bnM, inOut);
240     NEW_PRIVATE_EXPONENT(Z);
241     if(UnsignedCompareB(inOut->size,
242                       inOut->buffer,
243                       key->publicArea.unique.rsa.t.size,
244                       key->publicArea.unique.rsa.t.buffer)
245         >= 0)
246         return TPM_RC_SIZE;
247     // private key operation requires that private exponent be loaded
248     // During self-test, this might not be the case so load it up if it hasn't
249     // already done
250     // been done
251     if((key->sensitive.sensitive.rsa.t.size & RSA_prime_flag) == 0)

```

```

252     {
253         if(CryptRsaLoadPrivateExponent(&key->publicArea, &key->sensitive)
254             != TPM_RC_SUCCESS)
255             return TPM_RC_BINDING;
256     }
257     GOTO_ERROR_UNLESS(UnpackExponent(&key->sensitive.sensitive.rsa, Z));
258     GOTO_ERROR_UNLESS(RsaPrivateKeyOp(bnM, Z));
259     GOTO_ERROR_UNLESS(TpmMath_IntTo2B(bnM, inOut, inOut->size));
260     return TPM_RC_SUCCESS;
261 Error:
262     return TPM_RC_FAILURE;
263 }
264
265 /*** OaepEncode()
266 // This function performs OAEP padding. The size of the buffer to receive the
267 // OAEP padded data must equal the size of the modulus
268 //
269 // Return Type: TPM_RC
270 //     TPM_RC_VALUE      'hashAlg' is not valid or message size is too large
271 //
272 static TPM_RC OaepEncode(
273     TPM2B* padded,      // OUT: the pad data
274     TPM_ALG_ID hashAlg, // IN: algorithm to use for padding
275     const TPM2B* label, // IN: null-terminated string (may be NULL)
276     TPM2B* message,     // IN: the message being padded
277     RAND_STATE* rand     // IN: the random number generator to use
278 )
279 {
280     INT32 padLen;
281     INT32 dbSize;
282     INT32 i;
283     BYTE mySeed[MAX_DIGEST_SIZE];
284     BYTE* seed = mySeed;
285     UINT16 hLen = CryptHashGetDigestSize(hashAlg);
286     BYTE mask[MAX_RSA_KEY_BYTES];
287     BYTE* pp;
288     BYTE* pm;
289     TPM_RC retVal = TPM_RC_SUCCESS;
290
291     pAssert(padded != NULL && message != NULL);
292
293     // A value of zero is not allowed because the KDF can't produce a result
294     // if the digest size is zero.
295     if(hLen == 0)
296         return TPM_RC_VALUE;
297
298     // Basic size checks
299     // make sure digest isn't too big for key size
300     if(padded->size < (2 * hLen) + 2)
301         ERROR_EXIT(TPM_RC_HASH);
302
303     // and that message will fit messageSize <= k - 2hLen - 2
304     if(message->size > (padded->size - (2 * hLen) - 2))
305         ERROR_EXIT(TPM_RC_VALUE);
306
307     // Hash L even if it is null
308     // Offset into padded leaving room for masked seed and byte of zero
309     pp = &padded->buffer[hLen + 1];
310     if(CryptHashBlock(hashAlg, label->size, (BYTE*)label->buffer, hLen, pp) != hLen)
311         ERROR_EXIT(TPM_RC_FAILURE);
312
313     // concatenate PS of k mLen 2hLen 2
314     padLen = padded->size - message->size - (2 * hLen) - 2;
315     MemorySet(&pp[hLen], 0, padLen);
316     pp[hLen + padLen] = 0x01;
317     padLen += 1;

```

```

318     memcpy(&pp[hLen + padLen], message->buffer, message->size);
319
320     // The total size of db = hLen + pad + mSize;
321     dbSize = hLen + padLen + message->size;
322
323     DRBG_Generate(rand, mySeed, (UINT16)hLen);
324     if(g_inFailureMode)
325         ERROR_EXIT(TPM_RC_FAILURE);
326     // mask = MGF1 (seed, nSize hLen 1)
327     CryptMGF_KDF(dbSize, mask, hashAlg, hLen, seed, 0);
328
329     // Create the masked db
330     pm = mask;
331     for(i = dbSize; i > 0; i--)
332         *pp++ ^= *pm++;
333     pp = &padded->buffer[hLen + 1];
334
335     // Run the masked data through MGF1
336     if(CryptMGF_KDF(hLen, &padded->buffer[1], hashAlg, dbSize, pp, 0)
337        != (unsigned)hLen)
338         ERROR_EXIT(TPM_RC_VALUE);
339     // Now XOR the seed to create masked seed
340     pp = &padded->buffer[1];
341     pm = seed;
342     for(i = hLen; i > 0; i--)
343         *pp++ ^= *pm++;
344     // Set the first byte to zero
345     padded->buffer[0] = 0x00;
346 Exit:
347     return retVal;
348 }
349
350 /*** OaepDecode()
351 // This function performs OAEP padding checking. The size of the buffer to receive
352 // the recovered data. If the padding is not valid, the 'dSize' size is set to zero
353 // and the function returns TPM_RC_VALUE.
354 //
355 // The 'dSize' parameter is used as an input to indicate the size available in the
356 // buffer.
357
358 // If insufficient space is available, the size is not changed and the return code
359 // is TPM_RC_VALUE.
360 //
361 // Return Type: TPM_RC
362 //      TPM_RC_VALUE      the value to decode was larger than the modulus, or
363 //                        the padding is wrong or the buffer to receive the
364 //                        results is too small
365 //
366 //
367 static TPM_RC OaepDecode(
368     TPM2B*      dataOut, // OUT: the recovered data
369     TPM_ALG_ID  hashAlg, // IN: algorithm to use for padding
370     const TPM2B* label,   // IN: null-terminated string (may be NULL)
371     TPM2B*      padded    // IN: the padded data
372 )
373 {
374     UINT32 i;
375     BYTE  seedMask[MAX_DIGEST_SIZE];
376     UINT32 hLen = CryptHashGetDigestSize(hashAlg);
377
378     BYTE  mask[MAX_RSA_KEY_BYTES];
379     BYTE* pp;
380     BYTE* pm;
381     TPM_RC retVal = TPM_RC_SUCCESS;
382
383     // Strange size (anything smaller can't be an OAEP padded block)

```

```

384 // Also check for no leading 0
385 if((padded->size < (unsigned)((2 * hLen) + 2)) || (padded->buffer[0] != 0))
386     ERROR_EXIT(TPM_RC_VALUE);
387 // Use the hash size to determine what to put through MGF1 in order
388 // to recover the seedMask
389 CryptMGF_KDF(hLen,
390              seedMask,
391              hashAlg,
392              padded->size - hLen - 1,
393              &padded->buffer[hLen + 1],
394              0);
395
396 // Recover the seed into seedMask
397 pAssert(hLen <= sizeof(seedMask));
398 pp = &padded->buffer[1];
399 pm = seedMask;
400 for(i = hLen; i > 0; i--)
401     *pm++ ^= *pp++;
402
403 // Use the seed to generate the data mask
404 CryptMGF_KDF(padded->size - hLen - 1, mask, hashAlg, hLen, seedMask, 0);
405
406 // Use the mask generated from seed to recover the padded data
407 pp = &padded->buffer[hLen + 1];
408 pm = mask;
409 for(i = (padded->size - hLen - 1); i > 0; i--)
410     *pm++ ^= *pp++;
411
412 // Make sure that the recovered data has the hash of the label
413 // Put trial value in the seed mask
414 if((CryptHashBlock(hashAlg, label->size, (BYTE*)label->buffer, hLen, seedMask))
415    != hLen)
416     FAIL(FATAL_ERROR_INTERNAL);
417 if(memcmp(seedMask, mask, hLen) != 0)
418     ERROR_EXIT(TPM_RC_VALUE);
419
420 // find the start of the data
421 pm = &mask[hLen];
422 for(i = (UINT32)padded->size - (2 * hLen) - 1; i > 0; i--)
423 {
424     if(*pm++ != 0)
425         break;
426 }
427 // If we ran out of data or didn't end with 0x01, then return an error
428 if(i == 0 || pm[-1] != 0x01)
429     ERROR_EXIT(TPM_RC_VALUE);
430
431 // pm should be pointing at the first part of the data
432 // and i is one greater than the number of bytes to move
433 i--;
434 if(i > dataOut->size)
435     // Special exit to preserve the size of the output buffer
436     return TPM_RC_VALUE;
437 memcpy(dataOut->buffer, pm, i);
438 dataOut->size = (UINT16)i;
439 Exit:
440 if(retVal != TPM_RC_SUCCESS)
441     dataOut->size = 0;
442 return retVal;
443 }
444
445 /*** PKCS1v1_5Encode()
446 // This function performs the encoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in
447 // PKCS#1V2.1
448 // Return Type: TPM_RC
449 // TPM_RC_VALUE message size is too large

```

```

450 //
451 static TPM_RC RSAES_PKCS1v1_5Encode(TPM2B* padded, // OUT: the pad data
452                                     TPM2B* message, // IN: the message being padded
453                                     RAND_STATE* rand)
454 {
455     UINT32 ps = padded->size - message->size - 3;
456     //
457     if(message->size > padded->size - 11)
458         return TPM_RC_VALUE;
459     // move the message to the end of the buffer
460     memcpy(&padded->buffer[padded->size - message->size],
461           message->buffer,
462           message->size);
463     // Set the first byte to 0x00 and the second to 0x02
464     padded->buffer[0] = 0;
465     padded->buffer[1] = 2;
466
467     // Fill with random bytes
468     DRBG_Generate(rand, &padded->buffer[2], (UINT16)ps);
469     if(g_inFailureMode)
470         return TPM_RC_FAILURE;
471
472     // Set the delimiter for the random field to 0
473     padded->buffer[2 + ps] = 0;
474
475     // Now, the only messy part. Make sure that all the 'ps' bytes are non-zero
476     // In this implementation, use the value of the current index
477     for(ps++; ps > 1; ps--)
478     {
479         if(padded->buffer[ps] == 0)
480             padded->buffer[ps] = 0x55; // In the < 0.5% of the cases that the
481                                         // random value is 0, just pick a value to
482                                         // put into the spot.
483     }
484     return TPM_RC_SUCCESS;
485 }
486
487 /*** RSAES_Decode()
488 // This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in
489 // PKCS#1V2.1
490 //
491 // Return Type: TPM_RC
492 // TPM_RC_FAIL decoding error or results would no fit into provided buffer
493 //
494 static TPM_RC RSAES_Decode(TPM2B* message, // OUT: the recovered message
495                           TPM2B* coded, // IN: the encoded message
496 )
497 {
498     BOOL fail = FALSE;
499     UINT16 pSize;
500
501     fail = (coded->size < 11);
502     fail = (coded->buffer[0] != 0x00) | fail;
503     fail = (coded->buffer[1] != 0x02) | fail;
504     for(pSize = 2; pSize < coded->size; pSize++)
505     {
506         if(coded->buffer[pSize] == 0)
507             break;
508     }
509     pSize++;
510
511     // Make sure that pSize has not gone over the end and that there are at least 8
512     // bytes of pad data.
513     fail = (pSize > coded->size) | fail;
514     fail = ((pSize - 2) <= 8) | fail;
515     if((message->size < (UINT16)(coded->size - pSize)) || fail)

```



```

516         return TPM_RC_VALUE;
517     message->size = coded->size - pSize;
518     memcpy(message->buffer, &coded->buffer[pSize], coded->size - pSize);
519     return TPM_RC_SUCCESS;
520 }
521
522 /*** CryptRsaPssSaltSize()
523 // This function computes the salt size used in PSS. It is broken out so that
524 // the X509 code can get the same value that is used by the encoding function in this
525 // module.
526 INT16
527 CryptRsaPssSaltSize(INT16 hashSize, INT16 outSize)
528 {
529     INT16 saltSize;
530     //
531     // (Mask Length) = (outSize - hashSize - 1);
532     // Max saltSize is (Mask Length) - 1
533     saltSize = (outSize - hashSize - 1) - 1;
534     // Use the maximum salt size allowed by FIPS 186-4
535     if(saltSize > hashSize)
536         saltSize = hashSize;
537     else if(saltSize < 0)
538         saltSize = 0;
539     return saltSize;
540 }
541
542 /*** PssEncode()
543 // This function creates an encoded block of data that is the size of modulus.
544 // The function uses the maximum salt size that will fit in the encoded block.
545 //
546 // Returns TPM_RC_SUCCESS or goes into failure mode.
547 static TPM_RC PssEncode(TPM2B* out, // OUT: the encoded buffer
548                         TPM_ALG_ID hashAlg, // IN: hash algorithm for the encoding
549                         TPM2B* digest, // IN: the digest
550                         RAND_STATE* rand // IN: random number source
551 )
552 {
553     UINT32 hLen = CryptHashGetDigestSize(hashAlg);
554     BYTE salt[MAX_RSA_KEY_BYTES - 1];
555     UINT16 saltSize;
556     BYTE* ps = salt;
557     BYTE* pOut;
558     UINT16 mLen;
559     HASH_STATE hashState;
560
561     // These are fatal errors indicating bad TPM firmware
562     pAssert(out != NULL && hLen > 0 && digest != NULL);
563
564     // Get the size of the mask
565     mLen = (UINT16)(out->size - hLen - 1);
566
567     // Set the salt size
568     saltSize = CryptRsaPssSaltSize((INT16)hLen, (INT16)out->size);
569
570     //using eOut for scratch space
571     // Set the first 8 bytes to zero
572     pOut = out->buffer;
573     memset(pOut, 0, 8);
574
575     // Get set the salt
576     DRBG_Generate(rand, salt, saltSize);
577     if(g_inFailureMode)
578         return TPM_RC_FAILURE;
579
580     // Create the hash of the pad || input hash || salt
581     CryptHashStart(&hashState, hashAlg);

```

```

582     CryptDigestUpdate(&hashState, 8, pOut);
583     CryptDigestUpdate2B(&hashState, digest);
584     CryptDigestUpdate(&hashState, saltSize, salt);
585     CryptHashEnd(&hashState, hLen, &pOut[out->size - hLen - 1]);
586
587     // Create a mask
588     if(CryptMGF_KDF(mLen, pOut, hashAlg, hLen, &pOut[mLen], 0) != mLen)
589         FAIL(FATAL_ERROR_INTERNAL);
590
591     // Since this implementation uses key sizes that are all even multiples of
592     // 8, just need to make sure that the most significant bit is CLEAR
593     *pOut &= 0x7f;
594
595     // Before we mess up the pOut value, set the last byte to 0xbc
596     pOut[out->size - 1] = 0xbc;
597
598     // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
599     pOut = &pOut[mLen - saltSize - 1];
600     *pOut++ ^= 0x01;
601
602     // XOR the salt data into the buffer
603     for(; saltSize > 0; saltSize--)
604         *pOut++ ^= *ps++;
605
606     // and we are done
607     return TPM_RC_SUCCESS;
608 }
609
610 /*** PssDecode()
611 // This function checks that the PSS encoded block was built from the
612 // provided digest. If the check is successful, TPM_RC_SUCCESS is returned.
613 // Any other value indicates an error.
614 //
615 // This implementation of PSS decoding is intended for the reference TPM
616 // implementation and is not at all generalized. It is used to check
617 // signatures over hashes and assumptions are made about the sizes of values.
618 // Those assumptions are enforced by this implementation.
619 // This implementation does allow for a variable size salt value to have been
620 // used by the creator of the signature.
621 //
622 // Return Type: TPM_RC
623 //     TPM_RC_SCHEME      'hashAlg' is not a supported hash algorithm
624 //     TPM_RC_VALUE       decode operation failed
625 //
626 static TPM_RC PssDecode(
627     TPM_ALG_ID hashAlg, // IN: hash algorithm to use for the encoding
628     TPM2B* dIn,         // In: the digest to compare
629     TPM2B* eIn          // IN: the encoded data
630 )
631 {
632     UINT32 hLen = CryptHashGetDigestSize(hashAlg);
633     BYTE mask[MAX_RSA_KEY_BYTES];
634     BYTE* pm = mask;
635     BYTE* pe;
636     BYTE pad[8] = {0};
637     UINT32 i;
638     UINT32 mLen;
639     BYTE fail;
640     TPM_RC retVal = TPM_RC_SUCCESS;
641     HASH_STATE hashState;
642
643     // These errors are indicative of failures due to programmer error
644     pAssert(dIn != NULL && eIn != NULL);
645     pe = eIn->buffer;
646
647     // check the hash scheme

```

```

648     if(hLen == 0)
649         ERROR_EXIT(TPM_RC_SCHEME);
650
651     // most significant bit must be zero
652     fail = pe[0] & 0x80;
653
654     // last byte must be 0xbc
655     fail |= pe[eIn->size - 1] ^ 0xbc;
656
657     // Use the hLen bytes at the end of the buffer to generate a mask
658     // Doesn't start at the end which is a flag byte
659     mLen = eIn->size - hLen - 1;
660     CryptMGF_KDF(mLen, mask, hashAlg, hLen, &pe[mLen], 0);
661
662     // Clear the MSO of the mask to make it consistent with the encoding.
663     mask[0] &= 0x7F;
664
665     pAssert(mLen <= sizeof(mask));
666     // XOR the data into the mask to recover the salt. This sequence
667     // advances eIn so that it will end up pointing to the seed data
668     // which is the hash of the signature data
669     for(i = mLen; i > 0; i--)
670         *pm++ ^= *pe++;
671
672     // Find the first byte of 0x01 after a string of all 0x00
673     for(pm = mask, i = mLen; i > 0; i--)
674     {
675         if(*pm == 0x01)
676             break;
677         else
678             fail |= *pm++;
679     }
680     // i should not be zero
681     fail |= (i == 0);
682
683     // if we have failed, will continue using the entire mask as the salt value so
684     // that the timing attacks will not disclose anything (I don't think that this
685     // is a problem for TPM applications but, usually, we don't fail so this
686     // doesn't cost anything).
687     if(fail)
688     {
689         i = mLen;
690         pm = mask;
691     }
692     else
693     {
694         pm++;
695         i--;
696     }
697     // i contains the salt size and pm points to the salt. Going to use the input
698     // hash and the seed to recreate the hash in the lower portion of eIn.
699     CryptHashStart(&hashState, hashAlg);
700
701     // add the pad of 8 zeros
702     CryptDigestUpdate(&hashState, 8, pad);
703
704     // add the provided digest value
705     CryptDigestUpdate(&hashState, dIn->size, dIn->buffer);
706
707     // and the salt
708     CryptDigestUpdate(&hashState, i, pm);
709
710     // get the result
711     fail |= (CryptHashEnd(&hashState, hLen, mask) != hLen);
712
713     // Compare all bytes

```

```

714     for(pm = mask; hLen > 0; hLen--)
715         // don't use fail = because that could skip the increment and compare
716         // operations after the first failure and that gives away timing
717         // information.
718         fail |= *pm++ ^ *pe++;
719
720     retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
721 Exit:
722     return retVal;
723 }
724
725 /*** MakeDerTag()
726 // Construct the DER value that is used in RSASSA
727 // Return Type: INT16
728 // > 0      size of value
729 // <= 0      no hash exists
730 INT16
731 MakeDerTag(TPM_ALG_ID hashAlg, INT16 sizeOfBuffer, BYTE* buffer)
732 {
733     // 0x30, 0x31, // SEQUENCE (2 elements) 1st
734     // 0x30, 0x0D, // SEQUENCE (2 elements)
735     // 0x06, 0x09, // HASH OID
736     // 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01,
737     // 0x05, 0x00, // NULL
738     // 0x04, 0x20 // OCTET STRING
739     HASH_DEF* info = CryptGetHashDef(hashAlg);
740     INT16 oidSize;
741     // If no OID, can't do encode
742     GOTO_ERROR_UNLESS(info != NULL);
743     oidSize = 2 + (info->OID)[1];
744     // make sure this fits in the buffer
745     GOTO_ERROR_UNLESS(sizeOfBuffer >= (oidSize + 8));
746     *buffer++ = 0x30; // 1st SEQUENCE
747     // Size of the 1st SEQUENCE is 6 bytes + size of the hash OID + size of the
748     // digest size
749     *buffer++ = (BYTE)(6 + oidSize + info->digestSize); //
750     *buffer++ = 0x30; // 2nd SEQUENCE
751     // size is 4 bytes of overhead plus the side of the OID
752     *buffer++ = (BYTE)(2 + oidSize);
753     MemoryCopy(buffer, info->OID, oidSize);
754     buffer += oidSize;
755     *buffer++ = 0x05; // Add a NULL
756     *buffer++ = 0x00;
757
758     *buffer++ = 0x04;
759     *buffer++ = (BYTE)(info->digestSize);
760     return oidSize + 8;
761 Error:
762     return 0;
763 }
764
765 /*** RSASSA_Encode()
766 // Encode a message using PKCS1v1.5 method.
767 //
768 // Return Type: TPM_RC
769 // TPM_RC_SCHEME 'hashAlg' is not a supported hash algorithm
770 // TPM_RC_SIZE 'eOutSize' is not large enough
771 // TPM_RC_VALUE 'hInSize' does not match the digest size of hashAlg
772 static TPM_RC RSASSA_Encode(TPM2B* pOut, // IN:OUT on in, the size of the public key
773                             // on out, the encoded area
774                             TPM_ALG_ID hashAlg, // IN: hash algorithm for PKCS1v1_5
775                             TPM2B* hIn // IN: digest value to encode
776 )
777 {
778     BYTE DER[20];
779     BYTE* der = DER;

```

```

780     INT32  derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
781     BYTE*  eOut;
782     INT32  fillSize;
783     TPM_RC retVal = TPM_RC_SUCCESS;
784
785     // Can't use this scheme if the algorithm doesn't have a DER string defined.
786     if(derSize == 0)
787         ERROR_EXIT(TPM_RC_SCHEME);
788
789     // If the digest size of 'hashAlg' doesn't match the input digest size, then
790     // the DER will misidentify the digest so return an error
791     if(CryptHashGetDigestSize(hashAlg) != hIn->size)
792         ERROR_EXIT(TPM_RC_VALUE);
793     fillSize = pOut->size - derSize - hIn->size - 3;
794     eOut      = pOut->buffer;
795
796     // Make sure that this combination will fit in the provided space
797     if(fillSize < 8)
798         ERROR_EXIT(TPM_RC_SIZE);
799
800     // Start filling
801     *eOut++ = 0; // initial byte of zero
802     *eOut++ = 1; // byte of 0x01
803     for(; fillSize > 0; fillSize--)
804         *eOut++ = 0xff; // bunch of 0xff
805     *eOut++ = 0; // another 0
806     for(; derSize > 0; derSize--)
807         *eOut++ = *der++; // copy the DER
808     der = hIn->buffer;
809     for(fillSize = hIn->size; fillSize > 0; fillSize--)
810         *eOut++ = *der++; // copy the hash
811 Exit:
812     return retVal;
813 }
814
815 /** RSASSA_Decode()
816  * This function performs the RSASSA decoding of a signature.
817  *
818  * Return Type: TPM_RC
819  *   TPM_RC_VALUE      decode unsuccessful
820  *   TPM_RC_SCHEME     'hashAlg' is not supported
821  */
822 static TPM_RC RSASSA_Decode(
823     TPM_ALG_ID hashAlg, // IN: hash algorithm to use for the encoding
824     TPM2B*      hIn,    // IN: the digest to compare
825     TPM2B*      eIn     // IN: the encoded data
826 )
827 {
828     BYTE  fail;
829     BYTE  DER[20];
830     BYTE* der = DER;
831     INT32 derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
832     BYTE* pe;
833     INT32 hashSize = CryptHashGetDigestSize(hashAlg);
834     INT32 fillSize;
835     TPM_RC retVal;
836     BYTE*  digest;
837     UINT16 digestSize;
838
839     pAssert(hIn != NULL && eIn != NULL);
840     pe = eIn->buffer;
841
842     // Can't use this scheme if the algorithm doesn't have a DER string
843     // defined or if the provided hash isn't the right size
844     if(derSize == 0 || (unsigned)hashSize != hIn->size)
845         ERROR_EXIT(TPM_RC_SCHEME);

```

```

846
847 // Make sure that this combination will fit in the provided space
848 // Since no data movement takes place, can just walk through this
849 // and accept nearly random values. This can only be called from
850 // CryptValidateSignature() so eInSize is known to be in range.
851 fillSize = eIn->size - derSize - hashSize - 3;
852
853 // Start checking (fail will become non-zero if any of the bytes do not have
854 // the expected value.
855 fail = *pe++; // initial byte of zero
856 fail |= *pe++ ^ 1; // byte of 0x01
857 for(; fillSize > 0; fillSize--)
858     fail |= *pe++ ^ 0xff; // bunch of 0xff
859 fail |= *pe++; // another 0
860 for(; derSize > 0; derSize--)
861     fail |= *pe++ ^ *der++; // match the DER
862 digestSize = hIn->size;
863 digest = hIn->buffer;
864 for(; digestSize > 0; digestSize--)
865     fail |= *pe++ ^ *digest++; // match the hash
866 retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
867 Exit:
868     return retVal;
869 }
870
871 /** Externally Accessible Functions
872
873 **** CryptRsaSelectScheme()
874 // This function is used by TPM2_RSA_Decrypt and TPM2_RSA_Encrypt. It sets up
875 // the rules to select a scheme between input and object default.
876 // This function assumes the RSA object is loaded.
877 // If a default scheme is defined in object, the default scheme should be chosen,
878 // otherwise, the input scheme should be chosen.
879 // In the case that both the object and 'scheme' are not TPM_ALG_NULL, then
880 // if the schemes are the same, the input scheme will be chosen.
881 // if the scheme are not compatible, a NULL pointer will be returned.
882 //
883 // The return pointer may point to a TPM_ALG_NULL scheme.
884 TPMT_RSA_DECRYPT* CryptRsaSelectScheme(
885     TPMI_DH_OBJECT    rsaHandle, // IN: handle of an RSA key
886     TPMT_RSA_DECRYPT*  scheme     // IN: a sign or decrypt scheme
887 )
888 {
889     OBJECT*          rsaObject;
890     TPMT_ASYM_SCHEME* keyScheme;
891     TPMT_RSA_DECRYPT* retVal = NULL;
892
893     // Get sign object pointer
894     rsaObject = HandleToObject(rsaHandle);
895     keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
896
897     // if the default scheme of the object is TPM_ALG_NULL, then select the
898     // input scheme
899     if(keyScheme->scheme == TPM_ALG_NULL)
900     {
901         retVal = scheme;
902     }
903     // if the object scheme is not TPM_ALG_NULL and the input scheme is
904     // TPM_ALG_NULL, then select the default scheme of the object.
905     else if(scheme->scheme == TPM_ALG_NULL)
906     {
907         // if input scheme is NULL
908         retVal = (TPMT_RSA_DECRYPT*)keyScheme;
909     }
910     // get here if both the object scheme and the input scheme are
911     // not TPM_ALG_NULL. Need to insure that they are the same.

```



```

912 // IMPLEMENTATION NOTE: This could cause problems if future versions have
913 // schemes that have more values than just a hash algorithm. A new function
914 // (IsSchemeSame()) might be needed then.
915 else if(keyScheme->scheme == scheme->scheme
916        && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
917 {
918     retVal = scheme;
919 }
920 // two different, incompatible schemes specified will return NULL
921 return retVal;
922 }
923
924 /*** CryptRsaLoadPrivateExponent()
925 // This function is called to generate the private exponent of an RSA key.
926 // Return Type: TPM_RC
927 // TPM_RC_BINDING      public and private parts of 'rsaKey' are not matched
928 TPM_RC
929 CryptRsaLoadPrivateExponent(TPMT_PUBLIC* publicArea, TPMT_SENSITIVE* sensitive)
930 {
931     //
932     if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) == 0)
933     {
934         if((sensitive->sensitive.rsa.t.size * 2) == publicArea->unique.rsa.t.size)
935         {
936             NEW_PRIVATE_EXPONENT(Z);
937             CRYPT_RSA_INITIALIZED(bnN, &publicArea->unique.rsa);
938             CRYPT_RSA_VAR(bnQr);
939             CRYPT_INT_VAR(bnE, RADIX_BITS);
940
941             TEST(TPM_ALG_NULL);
942
943             GOTO_ERROR_UNLESS((sensitive->sensitive.rsa.t.size * 2)
944                             == publicArea->unique.rsa.t.size);
945             // Initialize the exponent
946             ExtMath_SetWord(bnE, publicArea->parameters.rsaDetail.exponent);
947             if(ExtMath_IsZero(bnE))
948                 ExtMath_SetWord(bnE, RSA_DEFAULT_PUBLIC_EXPONENT);
949             // Convert first prime to 2B
950             GOTO_ERROR_UNLESS(
951                 TpmMath_IntFrom2B(Z->P, &sensitive->sensitive.rsa.b) != NULL);
952
953             // Find the second prime by division. This uses 'bQ' rather than Z->Q
954             // because the division could make the quotient larger than a prime during
955             // some intermediate step.
956             GOTO_ERROR_UNLESS(ExtMath_Divide(Z->Q, bnQr, bnN, Z->P));
957             GOTO_ERROR_UNLESS(ExtMath_IsZero(bnQr));
958             // Compute the private exponent and return it if found
959             GOTO_ERROR_UNLESS(ComputePrivateExponent(bnE, Z));
960             GOTO_ERROR_UNLESS(PackExponent(&sensitive->sensitive.rsa, Z));
961         }
962         else
963             GOTO_ERROR_UNLESS(((sensitive->sensitive.rsa.t.size / 5) * 2)
964                             == publicArea->unique.rsa.t.size);
965         sensitive->sensitive.rsa.t.size |= RSA_prime_flag;
966     }
967     return TPM_RC_SUCCESS;
968 Error:
969     return TPM_RC_BINDING;
970 }
971
972 /*** CryptRsaEncrypt()
973 // This is the entry point for encryption using RSA. Encryption is
974 // use of the public exponent. The padding parameter determines what
975 // padding will be used.
976 //
977 // The 'cOutSize' parameter must be at least as large as the size of the key.

```

```

978 //
979 // If the padding is RSA_PAD_NONE, 'dIn' is treated as a number. It must be
980 // lower in value than the key modulus.
981 // NOTE: If dIn has fewer bytes than cOut, then we don't add low-order zeros to
982 // dIn to make it the size of the RSA key for the call to RSAEP. This is
983 // because the high order bytes of dIn might have a numeric value that is
984 // greater than the value of the key modulus. If this had low-order zeros
985 // added, it would have a numeric value larger than the modulus even though
986 // it started out with a lower numeric value.
987 //
988 // Return Type: TPM_RC
989 // TPM_RC_VALUE 'cOutSize' is too small (must be the size
990 // of the modulus)
991 // TPM_RC_SCHEME 'padType' is not a supported scheme
992 //
993 LIB_EXPORT TPM_RC CryptRsaEncrypt(
994     TPM2B_PUBLIC_KEY_RSA* cOut, // OUT: the encrypted data
995     TPM2B* dIn, // IN: the data to encrypt
996     OBJECT* key, // IN: the key used for encryption
997     TPMT_RSA_DECRYPT* scheme, // IN: the type of padding and hash
998     // if needed
999     const TPM2B* label, // IN: in case it is needed
1000     RAND_STATE* rand // IN: random number generator
1001     // state (mostly for testing)
1002 )
1003 {
1004     TPM_RC retVal = TPM_RC_SUCCESS;
1005     TPM2B_PUBLIC_KEY_RSA dataIn;
1006     //
1007     // if the input and output buffers are the same, copy the input to a scratch
1008     // buffer so that things don't get messed up.
1009     if(dIn == &cOut->b)
1010     {
1011         MemoryCopy2B(&dataIn.b, dIn, sizeof(dataIn.t.buffer));
1012         dIn = &dataIn.b;
1013     }
1014     // All encryption schemes return the same size of data
1015     cOut->t.size = key->publicArea.unique.rsa.t.size;
1016     TEST(scheme->scheme);
1017     switch(scheme->scheme)
1018     {
1019     case TPM_ALG_NULL: // 'raw' encryption
1020     {
1021         INT32 i;
1022         INT32 dSize = dIn->size;
1023         // dIn can have more bytes than cOut as long as the extra bytes
1024         // are zero. Note: the more significant bytes of a number in a byte
1025         // buffer are the bytes at the start of the array.
1026         for(i = 0; (i < dSize) && (dIn->buffer[i] == 0); i++)
1027             ;
1028         dSize -= i;
1029         if(dSize > cOut->t.size)
1030             ERROR_EXIT(TPM_RC_VALUE);
1031         // Pad cOut with zeros if dIn is smaller
1032         memset(cOut->t.buffer, 0, cOut->t.size - dSize);
1033         // And copy the rest of the value
1034         memcpy(&cOut->t.buffer[cOut->t.size - dSize], &dIn->buffer[i], dSize);
1035
1036         // If the size of dIn is the same as cOut dIn could be larger than
1037         // the modulus. If it is, then RSAEP() will catch it.
1038     }
1039     break;
1040     case TPM_ALG_RSAES:
1041         retVal = RSAES_PKCS1v1_5Encode(&cOut->b, dIn, rand);
1042         break;
1043 
```

```

1044     case TPM_ALG_OAEP:
1045         retVal =
1046             OaepEncode(&cOut->b, scheme->details.oaep.hashAlg, label, dIn, rand);
1047         break;
1048     default:
1049         ERROR_EXIT(TPM_RC_SCHEME);
1050         break;
1051 }
1052 // All the schemes that do padding will come here for the encryption step
1053 // Check that the Encoding worked
1054 if(retVal == TPM_RC_SUCCESS)
1055     // Padding OK so do the encryption
1056     retVal = RSAEP(&cOut->b, key);
1057 Exit:
1058     return retVal;
1059 }
1060
1061 /*** CryptRsaDecrypt()
1062 // This is the entry point for decryption using RSA. Decryption is
1063 // use of the private exponent. The 'padType' parameter determines what
1064 // padding was used.
1065 //
1066 // Return Type: TPM_RC
1067 //     TPM_RC_SIZE          'cInSize' is not the same as the size of the public
1068 //                          modulus of 'key'; or numeric value of the encrypted
1069 //                          data is greater than the modulus
1070 //     TPM_RC_VALUE         'dOutSize' is not large enough for the result
1071 //     TPM_RC_SCHEME        'padType' is not supported
1072 //
1073 LIB_EXPORT TPM_RC CryptRsaDecrypt(
1074     TPM2B*      dOut,    // OUT: the decrypted data
1075     TPM2B*      cIn,     // IN: the data to decrypt
1076     OBJECT*     key,     // IN: the key to use for decryption
1077     TPMT_RSA_DECRYPT* scheme, // IN: the padding scheme
1078     const TPM2B* label    // IN: in case it is needed for the scheme
1079 )
1080 {
1081     TPM_RC retVal;
1082
1083     // Make sure that the necessary parameters are provided
1084     pAssert(cIn != NULL && dOut != NULL && key != NULL);
1085
1086     // Size is checked to make sure that the encrypted value is the right size
1087     if(cIn->size != key->publicArea.unique.rsa.t.size)
1088         ERROR_EXIT(TPM_RC_SIZE);
1089
1090     TEST(scheme->scheme);
1091
1092     // For others that do padding, do the decryption in place and then
1093     // go handle the decoding.
1094     retVal = RSADP(cIn, key);
1095     if(retVal == TPM_RC_SUCCESS)
1096     {
1097         // Remove padding
1098         switch(scheme->scheme)
1099         {
1100             case TPM_ALG_NULL:
1101                 if(dOut->size < cIn->size)
1102                     return TPM_RC_VALUE;
1103                 MemoryCopy2B(dOut, cIn, dOut->size);
1104                 break;
1105             case TPM_ALG_RSAES:
1106                 retVal = RSAES_Decode(dOut, cIn);
1107                 break;
1108             case TPM_ALG_OAEP:
1109                 retVal = OaepDecode(dOut, scheme->details.oaep.hashAlg, label, cIn);

```

```

1110         break;
1111     default:
1112         retVal = TPM_RC_SCHEME;
1113         break;
1114     }
1115 }
1116 Exit:
1117     return retVal;
1118 }
1119
1120 /*** CryptRsaSign()
1121 // This function is used to generate an RSA signature of the type indicated in
1122 // 'scheme'.
1123 //
1124 // Return Type: TPM_RC
1125 //     TPM_RC_SCHEME      'scheme' or 'hashAlg' are not supported
1126 //     TPM_RC_VALUE       'hInSize' does not match 'hashAlg' (for RSASSA)
1127 //
1128 LIB_EXPORT TPM_RC CryptRsaSign(TPMT_SIGNATURE* sigOut,
1129                                OBJECT* key, // IN: key to use
1130                                TPM2B_DIGEST* hIn, // IN: the digest to sign
1131                                RAND_STATE* rand // IN: the random number generator
1132                                                // to use (mostly for testing)
1133 )
1134 {
1135     TPM_RC retVal = TPM_RC_SUCCESS;
1136     UINT16 modSize;
1137
1138     // parameter checks
1139     pAssert(sigOut != NULL && key != NULL && hIn != NULL);
1140
1141     modSize = key->publicArea.unique.rsa.t.size;
1142
1143     // for all non-null signatures, the size is the size of the key modulus
1144     sigOut->signature.rsapss.sig.t.size = modSize;
1145
1146     TEST(sigOut->sigAlg);
1147
1148     switch(sigOut->sigAlg)
1149     {
1150     case TPM_ALG_NULL:
1151         sigOut->signature.rsapss.sig.t.size = 0;
1152         return TPM_RC_SUCCESS;
1153     case TPM_ALG_RSAPSS:
1154         retVal = PssEncode(&sigOut->signature.rsapss.sig.b,
1155                            sigOut->signature.rsapss.hash,
1156                            &hIn->b,
1157                            rand);
1158         break;
1159     case TPM_ALG_RSASSA:
1160         retVal = RSASSA_Encode(&sigOut->signature.rsassa.sig.b,
1161                                sigOut->signature.rsassa.hash,
1162                                &hIn->b);
1163         break;
1164     default:
1165         retVal = TPM_RC_SCHEME;
1166     }
1167     if(retVal == TPM_RC_SUCCESS)
1168     {
1169         // Do the encryption using the private key
1170         retVal = RSADP(&sigOut->signature.rsapss.sig.b, key);
1171     }
1172     return retVal;
1173 }
1174
1175 /*** CryptRsaValidateSignature()

```

```

1176 // This function is used to validate an RSA signature. If the signature is valid
1177 // TPM_RC_SUCCESS is returned. If the signature is not valid, TPM_RC_SIGNATURE is
1178 // returned. Other return codes indicate either parameter problems or fatal errors.
1179 //
1180 // Return Type: TPM_RC
1181 //     TPM_RC_SIGNATURE    the signature does not check
1182 //     TPM_RC_SCHEME       unsupported scheme or hash algorithm
1183 //
1184 LIB_EXPORT TPM_RC CryptRsaValidateSignature(
1185     TPMT_SIGNATURE* sig,    // IN: signature
1186     OBJECT* key,           // IN: public modulus
1187     TPM2B_DIGEST* digest // IN: The digest being validated
1188 )
1189 {
1190     TPM_RC retVal;
1191     //
1192     // Fatal programming errors
1193     pAssert(key != NULL && sig != NULL && digest != NULL);
1194     switch(sig->sigAlg)
1195     {
1196         case TPM_ALG_RSAPSS:
1197         case TPM_ALG_RSASSA:
1198             break;
1199         default:
1200             return TPM_RC_SCHEME;
1201     }
1202
1203     // Errors that might be caused by calling parameters
1204     if(sig->signature.rsassa.sig.t.size != key->publicArea.unique.rsa.t.size)
1205         ERROR_EXIT(TPM_RC_SIGNATURE);
1206
1207     TEST(sig->sigAlg);
1208
1209     // Decrypt the block
1210     retVal = RSAEP(&sig->signature.rsassa.sig.b, key);
1211     if(retVal == TPM_RC_SUCCESS)
1212     {
1213         switch(sig->sigAlg)
1214         {
1215             case TPM_ALG_RSAPSS:
1216                 retVal = PssDecode(sig->signature.any.hashAlg,
1217                                     &digest->b,
1218                                     &sig->signature.rsassa.sig.b);
1219                 break;
1220             case TPM_ALG_RSASSA:
1221                 retVal = RSASSA_Decode(sig->signature.any.hashAlg,
1222                                         &digest->b,
1223                                         &sig->signature.rsassa.sig.b);
1224                 break;
1225             default:
1226                 return TPM_RC_SCHEME;
1227         }
1228     }
1229     Exit:
1230     return (retVal != TPM_RC_SUCCESS) ? TPM_RC_SIGNATURE : TPM_RC_SUCCESS;
1231 }
1232
1233 # if SIMULATION && USE_RSA_KEY_CACHE
1234 extern int s_rsaKeyCacheEnabled;
1235 int
1236     GetCachedRsaKey(
1237         TPMT_PUBLIC* publicArea, TPMT_SENSITIVE* sensitive, RAND_STATE* rand);
1238 # define GET_CACHED_KEY(publicArea, sensitive, rand) \
1239     (s_rsaKeyCacheEnabled && GetCachedRsaKey(publicArea, sensitive, rand))
1240 # else
1241 # define GET_CACHED_KEY(key, rand)
1242 # endif

```

```

1242
1243 /*** CryptRsaGenerateKey()
1244 // Generate an RSA key from a provided seed
1245 /*(See part 1 specification)
1246 // The formulation is:
1247 //     KDFa(hash, seed, label, Name, Counter, bits)
1248 // Where:
1249 //     hash         the nameAlg from the public template
1250 //     seed         a seed (will be a primary seed for a primary key)
1251 //     label        a distinguishing label including vendor ID and
1252 //                 vendor-assigned part number for the TPM.
1253 //     Name         the nameAlg from the template and the hash of the template
1254 //                 using nameAlg.
1255 //     Counter      a 32-bit integer that is incremented each time the KDF is
1256 //                 called in order to produce a specific key. This value
1257 //                 can be a 32-bit integer in host format and does not need
1258 //                 to be put in canonical form.
1259 //     bits         the number of bits needed for the key.
1260 // The following process is implemented to find a RSA key pair:
1261 // 1. pick a random number with enough bits from KDFa as a prime candidate
1262 // 2. set the first two significant bits and the least significant bit of the
1263 //    prime candidate
1264 // 3. check if the number is a prime. if not, pick another random number
1265 // 4. Make sure the difference between the two primes are more than 2^104.
1266 //    Otherwise, restart the process for the second prime
1267 // 5. If the counter has reached its maximum but we still can not find a valid
1268 //    RSA key pair, return an internal error. This is an artificial bound.
1269 //    Other implementation may choose a smaller number to indicate how many
1270 //    times they are willing to try.
1271 */
1272 // Return Type: TPM_RC
1273 //     TPM_RC_CANCELED    operation was canceled
1274 //     TPM_RC_RANGE       public exponent is not supported
1275 //     TPM_RC_VALUE       could not find a prime using the provided parameters
1276 LIB_EXPORT TPM_RC CryptRsaGenerateKey(
1277     TPMT_PUBLIC*    publicArea,
1278     TPMT_SENSITIVE* sensitive,
1279     RAND_STATE*     rand // IN: if not NULL, the deterministic
1280                        //     RNG state
1281 )
1282 {
1283     UINT32 i;
1284     CRYPT_RSA_VAR(bnD);
1285     CRYPT_RSA_VAR(bnN);
1286     CRYPT_INT_WORD(bnPubExp);
1287     UINT32 e = publicArea->parameters.rsaDetail.exponent;
1288     int    keySizeInBits;
1289     TPM_RC retVal = TPM_RC_NO_RESULT;
1290     NEW_PRIVATE_EXPONENT(Z);
1291     //
1292     // Need to make sure that the caller did not specify an exponent that is
1293     // not supported
1294     e = publicArea->parameters.rsaDetail.exponent;
1295     if(e == 0)
1296         e = RSA_DEFAULT_PUBLIC_EXPONENT;
1297     else
1298     {
1299         if(e < 65537)
1300             ERROR_EXIT(TPM_RC_RANGE);
1301         // Check that e is prime
1302         if(!IsPrimeInt(e))
1303             ERROR_EXIT(TPM_RC_RANGE);
1304     }
1305     ExtMath_SetWord(bnPubExp, e);
1306
1307

```



```

1308 // check for supported key size.
1309 keySizeInBits = publicArea->parameters.rsaDetail.keyBits;
1310 if(((keySizeInBits % 1024) != 0)
1311    || (keySizeInBits > MAX_RSA_KEY_BITS) // this might be redundant, but...
1312    || (keySizeInBits == 0))
1313     ERROR_EXIT(TPM_RC_VALUE);
1314
1315 // Set the prime size for instrumentation purposes
1316 INSTRUMENT_SET(PrimeIndex, PRIME_INDEX(keySizeInBits / 2));
1317
1318 # if SIMULATION && USE_RSA_KEY_CACHE
1319     if(GET_CACHED_KEY(publicArea, sensitive, rand))
1320         return TPM_RC_SUCCESS;
1321 # endif
1322
1323 // Make sure that key generation has been tested
1324 TEST(TPM_ALG_NULL);
1325
1326 // The prime is computed in P. When a new prime is found, Q is checked to
1327 // see if it is zero. If so, P is copied to Q and a new P is found.
1328 // When both P and Q are non-zero, the modulus and
1329 // private exponent are computed and a trial encryption/decryption is
1330 // performed. If the encrypt/decrypt fails, assume that at least one of the
1331 // primes is composite. Since we don't know which one, set Q to zero and start
1332 // over and find a new pair of primes.
1333
1334 for(i = 1; (retVal == TPM_RC_NO_RESULT) && (i != 100); i++)
1335 {
1336     if(_plat_IsCanceled())
1337         ERROR_EXIT(TPM_RC_CANCELED);
1338
1339     if(TpmRsa_GeneratePrimeForRSA(Z->P, keySizeInBits / 2, e, rand)
1340        == TPM_RC_FAILURE)
1341     {
1342         retVal = TPM_RC_FAILURE;
1343         goto Exit;
1344     }
1345
1346     INSTRUMENT_INC(PrimeCounts[PrimeIndex]);
1347
1348     // If this is the second prime, make sure that it differs from the
1349     // first prime by at least 2^100
1350     if(ExtMath_IsZero(Z->Q))
1351     {
1352         // copy p to q and compute another prime in p
1353         ExtMath_Copy(Z->Q, Z->P);
1354         continue;
1355     }
1356     // Make sure that the difference is at least 100 bits. Need to do it this
1357     // way because the big numbers are only positive values
1358     if(ExtMath_UnsignedCmp(Z->P, Z->Q) < 0)
1359         ExtMath_Subtract(bnD, Z->Q, Z->P);
1360     else
1361         ExtMath_Subtract(bnD, Z->P, Z->Q);
1362     if(ExtMath_MostSigBitNum(bnD) < 100)
1363         continue;
1364
1365     //Form the public modulus and set the unique value
1366     ExtMath_Multiply(bnN, Z->P, Z->Q);
1367     TpmMath_IntTo2B(
1368         bnN, &publicArea->unique.rsa.b, (NUMBYTES)BITS_TO_BYTES(keySizeInBits));
1369     // Make sure everything came out right. The MSb of the values must be one
1370     if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1371        || (publicArea->unique.rsa.t.size
1372            != (NUMBYTES)BITS_TO_BYTES(keySizeInBits)))
1373         FAIL(FATAL_ERROR_INTERNAL);

```

```

1374
1375 // Make sure that we can form the private exponent values
1376 if(ComputePrivateExponent(bnPubExp, Z) != TRUE)
1377 {
1378     // If ComputePrivateExponent could not find an inverse for
1379     // Q, then copy P and recompute P. This might
1380     // cause both to be recomputed if P is also zero
1381     if(ExtMath_IsZero(Z->Q))
1382         ExtMath_Copy(Z->Q, Z->P);
1383     continue;
1384 }
1385
1386 // Pack the private exponent into the sensitive area
1387 PackExponent(&sensitive->sensitive.rsa, Z);
1388 // Make sure everything came out right. The MSb of the values must be one
1389 if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1390    || ((sensitive->sensitive.rsa.t.buffer[0] & 0x80) == 0))
1391     FAIL(FATAL_ERROR_INTERNAL);
1392
1393 retVal = TPM_RC_SUCCESS;
1394 // Do a trial encryption decryption if this is a signing key
1395 if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
1396 {
1397     CRYPT_RSA_VAR(temp1);
1398     CRYPT_RSA_VAR(temp2);
1399     TpmMath_GetRandomInRange(temp1, bnN, rand);
1400
1401     // Encrypt with public exponent...
1402     ExtMath_ModExp(temp2, temp1, bnPubExp, bnN);
1403     // ... then decrypt with private exponent
1404     RsaPrivateKeyOp(temp2, Z);
1405
1406     // If the starting and ending values are not the same,
1407     // start over )-;
1408     if(ExtMath_UnsignedCmp(temp2, temp1) != 0)
1409     {
1410         ExtMath_SetWord(Z->Q, 0);
1411         retVal = TPM_RC_NO_RESULT;
1412     }
1413 }
1414 }
1415 Exit:
1416 return retVal;
1417 }
1418
1419 #endif // ALG_RSA

```

7.148 /tpm/src/crypt/CryptSelfTest.c

```

1  /** Introduction
2  // The functions in this file are designed to support self-test of cryptographic
3  // functions in the TPM. The TPM allows the user to decide whether to run self-test
4  // on a demand basis or to run all the self-tests before proceeding.
5  //
6  // The self-tests are controlled by a set of bit vectors. The
7  // 'g_untestedDecryptionAlgorithms' vector has a bit for each decryption algorithm
8  // that needs to be tested and 'g_untestedEncryptionAlgorithms' has a bit for
9  // each encryption algorithm that needs to be tested. Before an algorithm
10 // is used, the appropriate vector is checked (indexed using the algorithm ID).
11 // If the bit is 1, then the test function should be called.
12 //
13 // For more information, see TpmSelfTests.txt
14
15 #include "Tpm.h"
16

```

```

17  /*** Functions
18
19  /*** RunSelfTest()
20  // Local function to run self-test
21  static TPM_RC CryptRunSelfTests(
22      ALGORITHM_VECTOR* toTest // IN: the vector of the algorithms to test
23  )
24  {
25      TPM_ALG_ID alg;
26
27      // For each of the algorithms that are in the toTestVecor, need to run a
28      // test
29      for(alg = TPM_ALG_FIRST; alg <= TPM_ALG_LAST; alg++)
30      {
31          if(TEST_BIT(alg, *toTest))
32          {
33              TPM_RC result = CryptTestAlgorithm(alg, toTest);
34              if(result != TPM_RC_SUCCESS)
35                  return result;
36          }
37      }
38      return TPM_RC_SUCCESS;
39  }
40
41  /*** CryptSelfTest()
42  // This function is called to start/complete a full self-test.
43  // If 'fullTest' is NO, then only the untested algorithms will be run. If
44  // 'fullTest' is YES, then 'g_untestedDecryptionAlgorithms' is reinitialized and then
45  // all tests are run.
46  // This implementation of the reference design does not support processing outside
47  // the framework of a TPM command. As a consequence, this command does not
48  // complete until all tests are done. Since this can take a long time, the TPM
49  // will check after each test to see if the command is canceled. If so, then the
50  // TPM will returned TPM_RC_CANCELED. To continue with the self-tests, call
51  // TPM2_SelfTest(fullTest == No) and the TPM will complete the testing.
52  // Return Type: TPM_RC
53  //      TPM_RC_CANCELED      if the command is canceled
54  LIB_EXPORT
55  TPM_RC
56  CryptSelfTest(TPMI_YES_NO fullTest // IN: if full test is required
57  )
58  {
59      #if ALLOW_FORCE_FAILURE_MODE
60          if(g_forceFailureMode)
61              FAIL(FATAL_ERROR_FORCED);
62      #endif
63
64      // If the caller requested a full test, then reset the to test vector so that
65      // all the tests will be run
66      if(fullTest == YES)
67      {
68          MemoryCopy(g_toTest, g_implementedAlgorithms, sizeof(g_toTest));
69      }
70      return CryptRunSelfTests(&g_toTest);
71  }
72
73  /*** CryptIncrementalSelfTest()
74  // This function is used to perform an incremental self-test. This implementation
75  // will perform the toTest values before returning. That is, it assumes that the
76  // TPM cannot perform background tasks between commands.
77  //
78  // This command may be canceled. If it is, then there is no return result.
79  // However, this command can be run again and the incremental progress will not
80  // be lost.
81  // Return Type: TPM_RC
82  //      TPM_RC_CANCELED      processing of this command was canceled

```

```

83 //      TPM_RC_TESTING      if toTest list is not empty
84 //      TPM_RC_VALUE        an algorithm in the toTest list is not implemented
85 TPM_RC
86 CryptIncrementalSelfTest(TPML_ALG* toTest,    // IN: list of algorithms to be tested
87                         TPML_ALG* toDoList    // OUT: list of algorithms needing test
88 )
89 {
90     ALGORITHM_VECTOR toTestVector = {0};
91     TPM_ALG_ID       alg;
92     UINT32           i;
93
94     pAssert(toTest != NULL && toDoList != NULL);
95     if(toTest->count > 0)
96     {
97         // Transcribe the toTest list into the toTestVector
98         for(i = 0; i < toTest->count; i++)
99         {
100             alg = toTest->algorithms[i];
101
102             // make sure that the algorithm value is not out of range
103             if((alg > TPM_ALG_LAST) || !TEST_BIT(alg, g_implementedAlgorithms))
104                 return TPM_RC_VALUE;
105             SET_BIT(alg, toTestVector);
106         }
107         // Run the test
108         if(CryptRunSelfTests(&toTestVector) == TPM_RC_CANCELED)
109             return TPM_RC_CANCELED;
110     }
111     // Fill in the toDoList with the algorithms that are still untested
112     toDoList->count = 0;
113
114     for(alg = TPM_ALG_FIRST;
115         toDoList->count < MAX_ALG_LIST_SIZE && alg <= TPM_ALG_LAST;
116         alg++)
117     {
118         if(TEST_BIT(alg, g_toTest))
119             toDoList->algorithms[toDoList->count++] = alg;
120     }
121     return TPM_RC_SUCCESS;
122 }
123
124 /*** CryptInitializeToTest()
125 // This function will initialize the data structures for testing all the
126 // algorithms. This should not be called unless CryptAlgsSetImplemented() has
127 // been called
128 void CryptInitializeToTest(void)
129 {
130     // Indicate that nothing has been tested
131     memset(&g_cryptoSelfTestState, 0, sizeof(g_cryptoSelfTestState));
132
133     // Copy the implemented algorithm vector
134     MemoryCopy(g_toTest, g_implementedAlgorithms, sizeof(g_toTest));
135
136     // Setting the algorithm to null causes the test function to just clear
137     // out any algorithms for which there is no test.
138     CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);
139
140     return;
141 }
142
143 /*** CryptTestAlgorithm()
144 // Only point of contact with the actual self tests. If a self-test fails, there
145 // is no return and the TPM goes into failure mode.
146 // The call to TestAlgorithm uses an algorithm selector and a bit vector. When the
147 // test is run, the corresponding bit in 'toTest' and in 'g_toTest' is CLEAR. If
148 // 'toTest' is NULL, then only the bit in 'g_toTest' is CLEAR.

```

```

149 // There is a special case for the call to TestAlgorithm(). When 'alg' is
150 // ALG_ERROR, TestAlgorithm() will CLEAR any bit in 'toTest' for which it has
151 // no test. This allows the knowledge about which algorithms have test to be
152 // accessed through the interface that provides the test.
153 // Return Type: TPM_RC
154 //     TPM_RC_CANCELED      test was canceled
155 LIB_EXPORT
156 TPM_RC
157 CryptTestAlgorithm(TPM_ALG_ID alg, ALGORITHM_VECTOR* toTest)
158 {
159     TPM_RC result;
160     #if SELF_TEST
161         result = TestAlgorithm(alg, toTest);
162     #else
163         // If this is an attempt to determine the algorithms for which there is a
164         // self test, pretend that all of them do. We do that by not clearing any
165         // of the algorithm bits. When/if this function is called to run tests, it
166         // will over report. This can be changed so that any call to check on which
167         // algorithms have tests, 'toTest' can be cleared.
168         if(alg != TPM_ALG_ERROR)
169         {
170             CLEAR_BIT(alg, g_toTest);
171             if(toTest != NULL)
172                 CLEAR_BIT(alg, *toTest);
173         }
174         result = TPM_RC_SUCCESS;
175     #endif
176     return result;
177 }

```

7.149 /tpm/src/crypt/CryptSmac.c

```

1  /** Introduction
2  //
3  // This file contains the implementation of the message authentication codes based
4  // on a symmetric block cipher. These functions only use the single block
5  // encryption functions of the selected symmetric cryptographic library.
6
7  /** Includes, Defines, and Typedefs
8  #define _CRYPT_HASH_C_
9  #include "Tpm.h"
10
11 #if SMAC_IMPLEMENTED
12
13 /** CryptSmacStart()
14 // Function to start an SMAC.
15 UINT16
16 CryptSmacStart(HASH_STATE* state,
17                TPMU_PUBLIC_PARMS* keyParameters,
18                TPM_ALG_ID macAlg, // IN: the type of MAC
19                TPM2B* key)
20 {
21     UINT16 retVal = 0;
22     //
23     // Make sure that the key size is correct. This should have been checked
24     // at key load, but...
25     if(BITS_TO_BYTES(keyParameters->symDetail.sym.keyBits.sym) == key->size)
26     {
27         switch(macAlg)
28         {
29             # if ALG_CMAC
30                 case TPM_ALG_CMAC:
31                     retVal =
32                         CryptCmacStart(&state->state.smac, keyParameters, macAlg, key);
33                     break;

```

```

34  # endif
35      default:
36          break;
37      }
38  }
39  state->type = (retVal != 0) ? HASH_STATE_SMAC : HASH_STATE_EMPTY;
40  return retVal;
41  }
42
43  /*** CryptMacStart()
44  // Function to start either an HMAC or an SMAC. Cannot reuse the CryptHmacStart
45  // function because of the difference in number of parameters.
46  UINT16
47  CryptMacStart(HMAC_STATE* state,
48               TPMU_PUBLIC_PARMS* keyParameters,
49               TPM_ALG_ID macAlg, // IN: the type of MAC
50               TPM2B* key)
51  {
52      MemorySet(state, 0, sizeof(HMAC_STATE));
53      if(CryptHashIsValidAlg(macAlg, FALSE))
54      {
55          return CryptHmacStart(state, macAlg, key->size, key->buffer);
56      }
57      else if(CryptSmacIsValidAlg(macAlg, FALSE))
58      {
59          return CryptSmacStart(&state->hashState, keyParameters, macAlg, key);
60      }
61      else
62          return 0;
63  }
64
65  /*** CryptMacEnd()
66  // Dispatch to the MAC end function using a size and buffer pointer.
67  UINT16
68  CryptMacEnd(HMAC_STATE* state, UINT32 size, BYTE* buffer)
69  {
70      UINT16 retVal = 0;
71      if(state->hashState.type == HASH_STATE_SMAC)
72          retVal = (state->hashState.state.smac.smacMethods.end)(
73                  &state->hashState.state.smac.state, size, buffer);
74      else if(state->hashState.type == HASH_STATE_HMAC)
75          retVal = CryptHmacEnd(state, size, buffer);
76      state->hashState.type = HASH_STATE_EMPTY;
77      return retVal;
78  }
79
80  /*** CryptMacEnd2B()
81  // Dispatch to the MAC end function using a 2B.
82  UINT16
83  CryptMacEnd2B(HMAC_STATE* state, TPM2B* data)
84  {
85      return CryptMacEnd(state, data->size, data->buffer);
86  }
87  #endif // SMAC_IMPLEMENTED

```

7.150 /tpm/src/crypt/CryptSym.c

```

1  /*** Introduction
2  //
3  // This file contains the implementation of the symmetric block cipher modes
4  // allowed for a TPM. These functions only use the single block encryption functions
5  // of the selected symmetric crypto library.
6
7  /*** Includes, Defines, and Typedefs
8  #include "Tpm.h"

```



```

9
10 #include "CryptSym.h"
11
12 #define KEY_BLOCK_SIZES(ALG, alg) \
13     static const INT16 alg##KeyBlockSizes[] = {ALG##_KEY_SIZES_BITS, \
14                                                 -1, \
15                                                 ALG##_BLOCK_SIZES};
16
17 FOR_EACH_SYM(KEY_BLOCK_SIZES)
18
19 /** Initialization and Data Access Functions
20 //
21 /** CryptSymInit()
22 // This function is called to do _TPM_Init processing
23 BOOL CryptSymInit(void)
24 {
25     return TRUE;
26 }
27
28 /** CryptSymStartup()
29 // This function is called to do TPM2_Startup() processing
30 BOOL CryptSymStartup(void)
31 {
32     return TRUE;
33 }
34
35 /** CryptGetSymmetricBlockSize()
36 // This function returns the block size of the algorithm. The table of bit sizes has
37 // an entry for each allowed key size. The entry for a key size is 0 if the TPM does
38 // not implement that key size. The key size table is delimited with a negative number
39 // (-1). After the delimiter is a list of block sizes with each entry corresponding
40 // to the key bit size. For most symmetric algorithms, the block size is the same
41 // regardless of the key size but this arrangement allows them to be different.
42 // Return Type: INT16
43 // <= 0    cipher not supported
44 // > 0    the cipher block size in bytes
45 LIB_EXPORT INT16 CryptGetSymmetricBlockSize(
46     TPM_ALG_ID symmetricAlg, // IN: the symmetric algorithm
47     UINT16 keySizeInBits // IN: the key size
48 )
49 {
50     const INT16* sizes;
51     INT16 i;
52 #define ALG_CASE(SYM, sym) \
53     case TPM_ALG_##SYM: \
54         sizes = sym##KeyBlockSizes; \
55         break
56     switch(symmetricAlg)
57     {
58 #define GET_KEY_BLOCK_POINTER(SYM, sym) \
59     case TPM_ALG_##SYM: \
60         sizes = sym##KeyBlockSizes; \
61         break;
62         // Get the pointer to the block size array
63         FOR_EACH_SYM(GET_KEY_BLOCK_POINTER);
64
65         default:
66             return 0;
67     }
68     // Find the index of the indicated keySizeInBits
69     for(i = 0; *sizes >= 0; i++, sizes++)
70     {
71         if(*sizes == keySizeInBits)
72             break;
73     }
74     // If sizes is pointing at the end of the list of key sizes, then the desired

```

```

75     // key size was not found so set the block size to zero.
76     if(*sizes++ < 0)
77         return 0;
78     // Advance until the end of the list is found
79     while(*sizes++ >= 0)
80         ;
81     // sizes is pointing to the first entry in the list of block sizes. Use the
82     // ith index to find the block size for the corresponding key size.
83     return sizes[i];
84 }
85
86 /** Symmetric Encryption
87 // This function performs symmetric encryption based on the mode.
88 // Return Type: TPM_RC
89 //     TPM_RC_SIZE      'dSize' is not a multiple of the block size for an
90 //                      algorithm that requires it
91 //     TPM_RC_FAILURE    Fatal error
92 LIB_EXPORT TPM_RC CryptSymmetricEncrypt(
93     BYTE*      dOut,          // OUT:
94     TPM_ALG_ID algorithm,     // IN: the symmetric algorithm
95     UINT16     keySizeInBits, // IN: key size in bits
96     const BYTE* key,          // IN: key buffer. The size of this buffer
97                               // in bytes is (keySizeInBits + 7) / 8
98     TPM2B_IV*  ivInOut,       // IN/OUT: IV for decryption.
99     TPM_ALG_ID mode,          // IN: Mode to use
100    INT32       dSize,         // IN: data size (may need to be a
101                               // multiple of the blockSize)
102    const BYTE* dIn            // IN: data buffer
103 )
104 {
105     BYTE*      pIv;
106     int        i;
107     BYTE       tmp[MAX_SYM_BLOCK_SIZE];
108     BYTE*      pT;
109     tpmCryptKeySchedule_t keySchedule;
110     INT16       blockSize;
111     TpmCryptSetSymKeyCall_t encrypt;
112     BYTE*      iv;
113     BYTE       defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
114     //
115     pAssert(dOut != NULL && key != NULL && dIn != NULL);
116     if(dSize == 0)
117         return TPM_RC_SUCCESS;
118
119     TEST(algorithm);
120     blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
121     if(blockSize == 0)
122         return TPM_RC_FAILURE;
123     // If the iv is provided, then it is expected to be block sized. In some cases,
124     // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
125     // with no knowledge of the actual block size. This function will set it.
126     if((ivInOut != NULL) && (mode != TPM_ALG_ECB))
127     {
128         ivInOut->t.size = blockSize;
129         iv               = ivInOut->t.buffer;
130     }
131     else
132         iv = defaultIv;
133     pIv = iv;
134
135     // Create encrypt key schedule and set the encryption function pointer.
136     switch(algorithm)
137     {
138         FOR_EACH_SYM(ENCRYPT_CASE)
139
140         default:

```

```

141         return TPM_RC_SYMMETRIC;
142     }
143     switch(mode)
144     {
145 #if ALG_CTR
146         case TPM_ALG_CTR:
147             for(; dSize > 0; dSize -= blockSize)
148             {
149                 // Encrypt the current value of the IV(counter)
150                 ENCRYPT(&keySchedule, iv, tmp);
151
152                 //increment the counter (counter is big-endian so start at end)
153                 for(i = blockSize - 1; i >= 0; i--)
154                     if((iv[i] += 1) != 0)
155                         break;
156                 // XOR the encrypted counter value with input and put into output
157                 pT = tmp;
158                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
159                     *dOut++ = *dIn++ ^ *pT++;
160             }
161             break;
162 #endif
163 #if ALG_OFB
164         case TPM_ALG_OFB:
165             // This is written so that dIn and dOut may be the same
166             for(; dSize > 0; dSize -= blockSize)
167             {
168                 // Encrypt the current value of the "IV"
169                 ENCRYPT(&keySchedule, iv, iv);
170
171                 // XOR the encrypted IV into dIn to create the cipher text (dOut)
172                 pIv = iv;
173                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
174                     *dOut++ = (*pIv++ ^ *dIn++);
175             }
176             break;
177 #endif
178 #if ALG_CBC
179         case TPM_ALG_CBC:
180             // For CBC the data size must be an even multiple of the
181             // cipher block size
182             if((dSize % blockSize) != 0)
183                 return TPM_RC_SIZE;
184             // XOR the data block into the IV, encrypt the IV into the IV
185             // and then copy the IV to the output
186             for(; dSize > 0; dSize -= blockSize)
187             {
188                 pIv = iv;
189                 for(i = blockSize; i > 0; i--)
190                     *pIv++ ^= *dIn++;
191                 ENCRYPT(&keySchedule, iv, iv);
192                 pIv = iv;
193                 for(i = blockSize; i > 0; i--)
194                     *dOut++ = *pIv++;
195             }
196             break;
197 #endif
198         // CFB is not optional
199         case TPM_ALG_CFB:
200             // Encrypt the IV into the IV, XOR in the data, and copy to output
201             for(; dSize > 0; dSize -= blockSize)
202             {
203                 // Encrypt the current value of the IV
204                 ENCRYPT(&keySchedule, iv, iv);
205                 pIv = iv;
206                 for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)

```

```

207         // XOR the data into the IV to create the cipher text
208         // and put into the output
209         *dOut++ = *pIv++ ^= *dIn++;
210     }
211     // If the inner loop (i loop) was smaller than blockSize, then dSize
212     // would have been smaller than blockSize and it is now negative. If
213     // it is negative, then it indicates how many bytes are needed to pad
214     // out the IV for the next round.
215     for(; dSize < 0; dSize++)
216         *pIv++ = 0;
217     break;
218 #if ALG_ECB
219     case TPM_ALG_ECB:
220         // For ECB the data size must be an even multiple of the
221         // cipher block size
222         if((dSize % blockSize) != 0)
223             return TPM_RC_SIZE;
224         // Encrypt the input block to the output block
225         for(; dSize > 0; dSize -= blockSize)
226         {
227             ENCRYPT(&keySchedule, dIn, dOut);
228             dIn = &dIn[blockSize];
229             dOut = &dOut[blockSize];
230         }
231         break;
232 #endif
233     default:
234         return TPM_RC_FAILURE;
235 }
236 return TPM_RC_SUCCESS;
237 }
238
239 /*** CryptSymmetricDecrypt()
240 // This function performs symmetric decryption based on the mode.
241 // Return Type: TPM_RC
242 //     TPM_RC_FAILURE      A fatal error
243 //     TPM_RC_SIZE         'dSize' is not a multiple of the block size for an
244 //                         algorithm that requires it
245 LIB_EXPORT TPM_RC CryptSymmetricDecrypt(
246     BYTE*      dOut,           // OUT: decrypted data
247     TPM_ALG_ID algorithm,      // IN: the symmetric algorithm
248     UINT16     keySizeInBits,  // IN: key size in bits
249     const BYTE* key,           // IN: key buffer. The size of this buffer
250                               // in bytes is (keySizeInBits + 7) / 8
251     TPM2B_IV*  ivInOut,        // IN/OUT: IV for decryption.
252     TPM_ALG_ID mode,           // IN: Mode to use
253     INT32      dSize,          // IN: data size (may need to be a
254                               // multiple of the blockSize)
255     const BYTE* dIn            // IN: data buffer
256 )
257 {
258     BYTE*      pIv;
259     int        i;
260     BYTE       tmp[MAX_SYM_BLOCK_SIZE];
261     BYTE*      pT;
262     tpmCryptKeySchedule_t keySchedule;
263     INT16      blockSize;
264     BYTE*      iv;
265     TpmCryptSetSymKeyCall_t encrypt;
266     TpmCryptSetSymKeyCall_t decrypt;
267     BYTE       defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
268
269     // These are used but the compiler can't tell because they are initialized
270     // in case statements and it can't tell if they are always initialized
271     // when needed, so... Comment these out if the compiler can tell or doesn't
272     // care that these are initialized before use.

```

```

273     encrypt = NULL;
274     decrypt = NULL;
275
276     pAssert(dOut != NULL && key != NULL && dIn != NULL);
277     if(dSize == 0)
278         return TPM_RC_SUCCESS;
279
280     TEST(algorithm);
281     blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
282     if(blockSize == 0)
283         return TPM_RC_FAILURE;
284     // If the iv is provided, then it is expected to be block sized. In some cases,
285     // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
286     // with no knowledge of the actual block size. This function will set it.
287     if((ivInOut != NULL) && (mode != TPM_ALG_ECB))
288     {
289         ivInOut->t.size = blockSize;
290         iv               = ivInOut->t.buffer;
291     }
292     else
293         iv = defaultIv;
294
295     pIv = iv;
296     // Use the mode to select the key schedule to create. Encrypt always uses the
297     // encryption schedule. Depending on the mode, decryption might use either
298     // the decryption or encryption schedule.
299     switch(mode)
300     {
301     #if ALG_CBC || ALG_ECB
302         case TPM_ALG_CBC: // decrypt = decrypt
303         case TPM_ALG_ECB:
304             // For ECB and CBC, the data size must be an even multiple of the
305             // cipher block size
306             if((dSize % blockSize) != 0)
307                 return TPM_RC_SIZE;
308             switch(algorithm)
309             {
310                 FOR_EACH_SYM(DECRYPT_CASE)
311                 default:
312                     return TPM_RC_SYMMETRIC;
313             }
314             break;
315     #endif
316         default:
317             // For the remaining stream ciphers, use encryption to decrypt
318             switch(algorithm)
319             {
320                 FOR_EACH_SYM(ENCRYPT_CASE)
321                 default:
322                     return TPM_RC_SYMMETRIC;
323             }
324     }
325     // Now do the mode-dependent decryption
326     switch(mode)
327     {
328     #if ALG_CBC
329         case TPM_ALG_CBC:
330             // Copy the input data to a temp buffer, decrypt the buffer into the
331             // output, XOR in the IV, and copy the temp buffer to the IV and repeat.
332             for(; dSize > 0; dSize -= blockSize)
333             {
334                 pT = tmp;
335                 for(i = blockSize; i > 0; i--)
336                     *pT++ = *dIn++;
337                 DECRYPT(&keySchedule, tmp, dOut);
338                 pIv = iv;

```

```

339         pT = tmp;
340         for(i = blockSize; i > 0; i--)
341         {
342             *dOut++ ^= *pIv;
343             *pIv++ = *pT++;
344         }
345     }
346     break;
347 #endif
348     case TPM_ALG_CFB:
349         for(; dSize > 0; dSize -= blockSize)
350         {
351             // Encrypt the IV into the temp buffer
352             ENCRYPT(&keySchedule, iv, tmp);
353             pT = tmp;
354             pIv = iv;
355             for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
356                 // Copy the current cipher text to IV, XOR
357                 // with the temp buffer and put into the output
358                 *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
359         }
360         // If the inner loop (i loop) was smaller than blockSize, then dSize
361         // would have been smaller than blockSize and it is now negative
362         // If it is negative, then it indicates how many fill bytes
363         // are needed to pad out the IV for the next round.
364         for(; dSize < 0; dSize++)
365             *pIv++ = 0;
366     }
367     break;
368 #if ALG_CTR
369     case TPM_ALG_CTR:
370         for(; dSize > 0; dSize -= blockSize)
371         {
372             // Encrypt the current value of the IV(counter)
373             ENCRYPT(&keySchedule, iv, tmp);
374
375             //increment the counter (counter is big-endian so start at end)
376             for(i = blockSize - 1; i >= 0; i--)
377                 if((iv[i] += 1) != 0)
378                     break;
379             // XOR the encrypted counter value with input and put into output
380             pT = tmp;
381             for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
382                 *dOut++ = *dIn++ ^ *pT++;
383         }
384         break;
385 #endif
386 #if ALG_ECB
387     case TPM_ALG_ECB:
388         for(; dSize > 0; dSize -= blockSize)
389         {
390             DECRYPT(&keySchedule, dIn, dOut);
391             dIn = &dIn[blockSize];
392             dOut = &dOut[blockSize];
393         }
394         break;
395 #endif
396 #if ALG_OFB
397     case TPM_ALG_OFB:
398         // This is written so that dIn and dOut may be the same
399         for(; dSize > 0; dSize -= blockSize)
400         {
401             // Encrypt the current value of the "IV"
402             ENCRYPT(&keySchedule, iv, iv);
403
404             // XOR the encrypted IV into dIn to create the cipher text (dOut)

```



```

405         pIv = iv;
406         for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
407             *dOut++ = (*pIv++ ^ *dIn++);
408     }
409     break;
410 #endif
411     default:
412         return TPM_RC_FAILURE;
413 }
414 return TPM_RC_SUCCESS;
415 }
416
417 /*** CryptSymKeyValidate()
418 // Validate that a provided symmetric key meets the requirements of the TPM
419 // Return Type: TPM_RC
420 //     TPM_RC_KEY_SIZE      Key size specifiers do not match
421 //     TPM_RC_KEY           Key is not allowed
422 TPM_RC
423 CryptSymKeyValidate(TPMT_SYM_DEF_OBJECT* symDef, TPM2B_SYM_KEY* key)
424 {
425     if(key->t.size != BITS_TO_BYTES(symDef->keyBits.sym))
426         return TPM_RC_KEY_SIZE;
427     return TPM_RC_SUCCESS;
428 }

```

7.151 /tpm/src/crypt/CryptUtil.c

```

1  /*** Introduction
2  //
3  // This module contains the interfaces to the CryptoEngine and provides
4  // miscellaneous cryptographic functions in support of the TPM.
5  //
6
7  /*** Includes
8  #include "Tpm.h"
9  #include "Marshal.h"
10
11  /*******
12  /*** Hash/HMAC Functions
13  /*******
14
15  /*** CryptHmacSign()
16  // Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a
17  // message.
18  // Return Type: TPM_RC
19  //     TPM_RC_HASH      not a valid hash
20  static TPM_RC CryptHmacSign(TPMT_SIGNATURE* signature, // OUT: signature
21                             OBJECT*      signKey,      // IN: HMAC key sign the hash
22                             TPM2B_DIGEST* hashData     // IN: hash to be signed
23 )
24 {
25     HMAC_STATE hmacState;
26     UINT32      digestSize;
27
28     digestSize = CryptHmacStart2B(&hmacState,
29                                 signature->signature.any.hashAlg,
30                                 &signKey->sensitive.sensitive.bits.b);
31     CryptDigestUpdate2B(&hmacState.hashState, &hashData->b);
32     CryptHmacEnd(&hmacState, digestSize, (BYTE*)&signature->signature.hmac.digest);
33     return TPM_RC_SUCCESS;
34 }
35
36 /*** CryptHMACVerifySignature()
37 // This function will verify a signature signed by a HMAC key.
38 // Note that a caller needs to prepare 'signature' with the signature algorithm

```

```

39 // (TPM_ALG_HMAC) and the hash algorithm to use. This function then builds a
40 // signature of that type.
41 // Return Type: TPM_RC
42 //     TPM_RC_SCHEME          not the proper scheme for this key type
43 //     TPM_RC_SIGNATURE       if invalid input or signature is not genuine
44 static TPM_RC CryptHMACVerifySignature(
45     OBJECT*      signKey,    // IN: HMAC key signed the hash
46     TPM2B_DIGEST* hashData,  // IN: digest being verified
47     TPMT_SIGNATURE* signature // IN: signature to be verified
48 )
49 {
50     TPMT_SIGNATURE test;
51     TPMT_KEYEDHASH_SCHEME* keyScheme =
52         &signKey->publicArea.parameters.keyedHashDetail.scheme;
53     //
54     if((signature->sigAlg != TPM_ALG_HMAC)
55         || (signature->signature.hmac.hashAlg == TPM_ALG_NULL))
56         return TPM_RC_SCHEME;
57     // This check is not really needed for verification purposes. However, it does
58     // prevent someone from trying to validate a signature using a weaker hash
59     // algorithm than otherwise allowed by the key. That is, a key with a scheme
60     // other than TPM_ALG_NULL can only be used to validate signatures that have
61     // a matching scheme.
62     if((keyScheme->scheme != TPM_ALG_NULL)
63         && ((keyScheme->scheme != signature->sigAlg)
64             || (keyScheme->details.hmac.hashAlg != signature->signature.any.hashAlg)))
65         return TPM_RC_SIGNATURE;
66     test.sigAlg = signature->sigAlg;
67     test.signature.hmac.hashAlg = signature->signature.hmac.hashAlg;
68
69     CryptHmacSign(&test, signKey, hashData);
70
71     // Compare digest
72     if(!MemoryEqual(&test.signature.hmac.digest,
73                     &signature->signature.hmac.digest,
74                     CryptHashGetDigestSize(signature->signature.any.hashAlg)))
75         return TPM_RC_SIGNATURE;
76
77     return TPM_RC_SUCCESS;
78 }
79
80 /*** CryptGenerateKeyedHash()
81 // This function creates a keyedHash object.
82 // Return type: TPM_RC
83 //     TPM_RC_NO_RESULT    cannot get values from random number generator
84 //     TPM_RC_SIZE         sensitive data size is larger than allowed for
85 //                         the scheme
86 static TPM_RC CryptGenerateKeyedHash(
87     TPMT_PUBLIC* publicArea,                // IN/OUT: the public area template
88                                             //      for the new key.
89     TPMT_SENSITIVE* sensitive,              // OUT: sensitive area
90     TPMS_SENSITIVE_CREATE* sensitiveCreate, // IN: sensitive creation data
91     RAND_STATE* rand                       // IN: "entropy" source
92 )
93 {
94     TPMT_KEYEDHASH_SCHEME* scheme;
95     TPM_ALG_ID hashAlg;
96     UINT16 digestSize;
97
98     scheme = &publicArea->parameters.keyedHashDetail.scheme;
99
100     if(publicArea->type != TPM_ALG_KEYEDHASH)
101         return TPM_RC_FAILURE;
102
103     // Pick the limiting hash algorithm
104     if(scheme->scheme == TPM_ALG_NULL)

```

```

105     hashAlg = publicArea->nameAlg;
106     else if(scheme->scheme == TPM_ALG_XOR)
107         hashAlg = scheme->details.xor.hashAlg;
108     else
109         hashAlg = scheme->details.hmac.hashAlg;
110     digestSize = CryptHashGetDigestSize(hashAlg);
111
112     // if this is a signing or a decryption key, then the limit
113     // for the data size is the block size of the hash. This limit
114     // is set because larger values have lower entropy because of the
115     // HMAC function. The lower limit is 1/2 the size of the digest
116     //
117     // If the user provided the key, check that it is a proper size
118     if(sensitiveCreate->data.t.size != 0)
119     {
120         if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
121            || IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
122         {
123             if(sensitiveCreate->data.t.size > CryptHashGetBlockSize(hashAlg))
124                 return TPM_RC_SIZE;
125 #if 0 // May make this a FIPS-mode requirement
126             if(sensitiveCreate->data.t.size < (digestSize / 2))
127                 return TPM_RC_SIZE;
128 #endif
129         }
130         // If this is a data blob, then anything that will get past the unmarshaling
131         // is OK
132         MemoryCopy2B(&sensitive->sensitive.bits.b,
133                     &sensitiveCreate->data.b,
134                     sizeof(sensitive->sensitive.bits.t.buffer));
135     }
136     else
137     {
138         // The TPM is going to generate the data so set the size to be the
139         // size of the digest of the algorithm
140         sensitive->sensitive.bits.t.size =
141             DRBG_Generate(rand, sensitive->sensitive.bits.t.buffer, digestSize);
142         if(sensitive->sensitive.bits.t.size == 0)
143             return (g_inFailureMode) ? TPM_RC_FAILURE : TPM_RC_NO_RESULT;
144     }
145     return TPM_RC_SUCCESS;
146 }
147
148 /*** CryptIsSchemeAnonymous()
149 // This function is used to test a scheme to see if it is an anonymous scheme
150 // The only anonymous scheme is ECDSA. ECDSA can be used to do things
151 // like U-Prove.
152 BOOL CryptIsSchemeAnonymous(TPM_ALG_ID scheme // IN: the scheme algorithm to test
153 )
154 {
155     return scheme == TPM_ALG_ECDSA;
156 }
157
158 /*** Symmetric Functions
159 /*** ParmDecryptSym()
160 // This function performs parameter decryption using symmetric block cipher.
161 // (See Part 1 specification)
162 // Symmetric parameter decryption
163 // When parameter decryption uses a symmetric block cipher, a decryption
164 // key and IV will be generated from:
165 // KDFa(hash, sessionAuth, "CFB", nonceNewer, nonceOlder, bits) (24)
166 // Where:
167 // hash the hash function associated with the session

```

```

171 //      sessionAuth      the sessionAuth associated with the session
172 //      nonceNewer       nonceCaller for a command
173 //      nonceOlder       nonceTPM for a command
174 //      bits             the number of bits required for the symmetric key
175 //                      plus an IV
176 */
177 void ParmDecryptSym(TPM_ALG_ID symAlg,          // IN: the symmetric algorithm
178                   TPM_ALG_ID hash,            // IN: hash algorithm for KDFa
179                   UINT16   keySizeInBits,     // IN: the key size in bits
180                   TPM2B*   key,               // IN: KDF HMAC key
181                   TPM2B*   nonceCaller,       // IN: nonce caller
182                   TPM2B*   nonceTpm,         // IN: nonce TPM
183                   UINT32   dataSize,          // IN: size of parameter buffer
184                   BYTE*    data,              // OUT: buffer to be decrypted
185 )
186 {
187     // KDF output buffer
188     // It contains parameters for the CFB encryption
189     // From MSB to LSB, they are the key and iv
190     BYTE symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
191     // Symmetric key size in byte
192     UINT16 keySize = (keySizeInBits + 7) / 8;
193     TPM2B_IV iv;
194
195     iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
196     // If there is decryption to do...
197     if(iv.t.size > 0)
198     {
199         // Generate key and iv
200         CryptKDFa(hash,
201                 key,
202                 CFB_KEY,
203                 nonceCaller,
204                 nonceTpm,
205                 keySizeInBits + (iv.t.size * 8),
206                 symParmString,
207                 NULL,
208                 FALSE);
209         MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
210
211         CryptSymmetricDecrypt(data,
212                             symAlg,
213                             keySizeInBits,
214                             symParmString,
215                             &iv,
216                             TPM_ALG_CFB,
217                             dataSize,
218                             data);
219     }
220     return;
221 }
222
223 /*** ParmEncryptSym()
224 // This function performs parameter encryption using symmetric block cipher.
225 // (See part 1 specification)
226 // When parameter decryption uses a symmetric block cipher, an encryption
227 // key and IV will be generated from:
228 // KDFa(hash, sessionAuth, "CFB", nonceNewer, nonceOlder, bits)      (24)
229 // Where:
230 // hash      the hash function associated with the session
231 // sessionAuth the sessionAuth associated with the session
232 // nonceNewer nonceTPM for a response
233 // nonceOlder nonceCaller for a response
234 // bits      the number of bits required for the symmetric key
235 //          plus an IV
236 */

```

```

237 void ParmEncryptSym(TPM_ALG_ID symAlg,          // IN: symmetric algorithm
238                    TPM_ALG_ID hash,           // IN: hash algorithm for KDFa
239                    UINT16 keySizeInBits,      // IN: symmetric key size in bits
240                    TPM2B* key,                // IN: KDF HMAC key
241                    TPM2B* nonceCaller,        // IN: nonce caller
242                    TPM2B* nonceTpm,          // IN: nonce TPM
243                    UINT32 dataSize,           // IN: size of parameter buffer
244                    BYTE* data,                // OUT: buffer to be encrypted
245 )
246 {
247     // KDF output buffer
248     // It contains parameters for the CFB encryption
249     BYTE symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
250
251     // Symmetric key size in bytes
252     UINT16 keySize = (keySizeInBits + 7) / 8;
253
254     TPM2B_IV iv;
255
256     iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
257     // See if there is any encryption to do
258     if(iv.t.size > 0)
259     {
260         // Generate key and iv
261         CryptKDFa(hash,
262                  key,
263                  CFB_KEY,
264                  nonceTpm,
265                  nonceCaller,
266                  keySizeInBits + (iv.t.size * 8),
267                  symParmString,
268                  NULL,
269                  FALSE);
270         MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
271
272         CryptSymmetricEncrypt(data,
273                               symAlg,
274                               keySizeInBits,
275                               symParmString,
276                               &iv,
277                               TPM_ALG_CFB,
278                               dataSize,
279                               data);
280     }
281     return;
282 }
283
284 /*** CryptGenerateKeySymmetric()
285 // This function generates a symmetric cipher key. The derivation process is
286 // determined by the type of the provided 'rand'
287 // Return type: TPM_RC
288 //     TPM_RC_NO_RESULT    cannot get a random value
289 //     TPM_RC_KEY_SIZE     key size in the public area does not match the size
290 //                         in the sensitive creation area
291 //     TPM_RC_KEY          provided key value is not allowed
292 static TPM_RC CryptGenerateKeySymmetric(
293     TPMT_PUBLIC* publicArea,          // IN/OUT: The public area template
294                                         // for the new key.
295     TPMT_SENSITIVE* sensitive,        // OUT: sensitive area
296     TPMS_SENSITIVE_CREATE* sensitiveCreate, // IN: sensitive creation data
297     RAND_STATE* rand,                // IN: the "entropy" source for
298 )
299 {
300     UINT16 keyBits = publicArea->parameters.symDetail.sym.keyBits.sym;
301     TPM_RC result;
302     //

```

```

303 // only do multiples of RADIX_BITS
304 if((keyBits % RADIX_BITS) != 0)
305     return TPM_RC_KEY_SIZE;
306 // If this is not a new key, then the provided key data must be the right size
307 if(sensitiveCreate->data.t.size != 0)
308 {
309     result = CryptSymKeyValidate(&publicArea->parameters.symDetail.sym,
310                                 (TPM2B_SYM_KEY*)&sensitiveCreate->data);
311     if(result == TPM_RC_SUCCESS)
312         MemoryCopy2B(&sensitive->sensitive.sym.b,
313                     &sensitiveCreate->data.b,
314                     sizeof(sensitive->sensitive.sym.t.buffer));
315 }
316 else
317 {
318     sensitive->sensitive.sym.t.size = DRBG_Generate(
319         rand, sensitive->sensitive.sym.t.buffer, BITS_TO_BYTES(keyBits));
320     if(g_inFailureMode)
321         result = TPM_RC_FAILURE;
322     else if(sensitive->sensitive.sym.t.size == 0)
323         result = TPM_RC_NO_RESULT;
324     else
325         result = TPM_RC_SUCCESS;
326 }
327 return result;
328 }
329
330 /*** CryptXORObfuscation()
331 // This function implements XOR obfuscation. It should not be called if the
332 // hash algorithm is not implemented. The only return value from this function
333 // is TPM_RC_SUCCESS.
334 void CryptXORObfuscation(TPM_ALG_ID hash, // IN: hash algorithm for KDF
335                          TPM2B* key, // IN: KDF key
336                          TPM2B* contextU, // IN: contextU
337                          TPM2B* contextV, // IN: contextV
338                          UINT32 dataSize, // IN: size of data buffer
339                          BYTE* data // IN/OUT: data to be XORed in place
340 )
341 {
342     BYTE mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
343     BYTE* pm;
344     UINT32 i;
345     UINT32 counter = 0;
346     UINT16 hLen = CryptHashGetDigestSize(hash);
347     UINT32 requestSize = dataSize * 8;
348     INT32 remainBytes = (INT32)dataSize;
349
350     pAssert((key != NULL) && (data != NULL) && (hLen != 0));
351
352     // Call KDFa to generate XOR mask
353     for(; remainBytes > 0; remainBytes -= hLen)
354     {
355         // Make a call to KDFa to get next iteration
356         CryptKDFa(hash,
357                   key,
358                   XOR_KEY,
359                   contextU,
360                   contextV,
361                   requestSize,
362                   mask,
363                   &counter,
364                   TRUE);
365
366         // XOR next piece of the data
367         pm = mask;
368         for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)

```



```

369         *data++ ^= *pm++;
370     }
371     return;
372 }
373
374 //*****
375 /** Initialization and shut down
376 //*****
377
378 /** CryptInit()
379 // This function is called when the TPM receives a _TPM_Init indication.
380 //
381 // NOTE: The hash algorithms do not have to be tested, they just need to be
382 // available. They have to be tested before the TPM can accept HMAC authorization
383 // or return any result that relies on a hash algorithm.
384 // Return Type: BOOL
385 //     TRUE(1)      initializations succeeded
386 //     FALSE(0)     initialization failed and caller should place the TPM into
387 //                  Failure Mode
388 BOOL CryptInit(void)
389 {
390     BOOL ok;
391     // Initialize the vector of implemented algorithms
392     AlgorithmGetImplementedVector(&g_implementedAlgorithms);
393
394     // Indicate that all test are necessary
395     CryptInitializeToTest();
396
397     // Do any library initializations that are necessary. If any fails,
398     // the caller should go into failure mode;
399     ok = ExtMath_LibInit();
400     ok = ok && CryptSymInit();
401     ok = ok && CryptRandInit();
402     ok = ok && CryptHashInit();
403     #if ALG_RSA
404     ok = ok && CryptRsaInit();
405     #endif // ALG_RSA
406     #if ALG_ECC
407     ok = ok && CryptEccInit();
408     #endif // ALG_ECC
409     return ok;
410 }
411
412 /** CryptStartup()
413 // This function is called by TPM2_Startup() to initialize the functions in
414 // this cryptographic library and in the provided CryptoLibrary. This function
415 // and CryptUtlInit() are both provided so that the implementation may move the
416 // initialization around to get the best interaction.
417 // Return Type: BOOL
418 //     TRUE(1)      startup succeeded
419 //     FALSE(0)     startup failed and caller should place the TPM into
420 //                  Failure Mode
421 BOOL CryptStartup(STARTUP_TYPE type // IN: the startup type
422 )
423 {
424     BOOL OK;
425     NOT_REFERENCED(type);
426
427     OK = CryptSymStartup() && CryptRandStartup() && CryptHashStartup()
428     #if ALG_RSA
429         && CryptRsaStartup()
430     #endif // ALG_RSA
431     #if ALG_ECC
432         && CryptEccStartup()
433     #endif // ALG_ECC
434     ;

```

```

435 #if ALG_ECC
436     // Don't directly check for SU_RESET because that is the default
437     if(OK && (type != SU_RESTART) && (type != SU_RESUME))
438     {
439         // If the shutdown was orderly, then the values recovered from NV will
440         // be OK to use.
441         // Get a new random commit nonce
442         gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
443         CryptRandomGenerate(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
444         // Reset the counter and commit array
445         gr.commitCounter = 0;
446         MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
447     }
448 #endif // ALG_ECC
449     return OK;
450 }
451
452 //*****
453 /** Algorithm-Independent Functions
454 //*****
455 /*** Introduction
456 // These functions are used generically when a function of a general type
457 // (e.g., symmetric encryption) is required. The functions will modify the
458 // parameters as required to interface to the indicated algorithms.
459 //
460 /*** CryptIsAsymAlgorithm()
461 // This function indicates if an algorithm is an asymmetric algorithm.
462 // Return Type: BOOL
463 //     TRUE(1)         if it is an asymmetric algorithm
464 //     FALSE(0)        if it is not an asymmetric algorithm
465 BOOL CryptIsAsymAlgorithm(TPM_ALG_ID algID // IN: algorithm ID
466 )
467 {
468     switch(algID)
469     {
470 #if ALG_RSA
471         case TPM_ALG_RSA:
472 #endif
473 #if ALG_ECC
474         case TPM_ALG_ECC:
475 #endif
476         return TRUE;
477         break;
478     default:
479         break;
480     }
481     return FALSE;
482 }
483
484 /*** CryptSecretEncrypt()
485 // This function creates a secret value and its associated secret structure using
486 // an asymmetric algorithm.
487 //
488 // This function is used by TPM2_Rewrap() TPM2_MakeCredential(),
489 // and TPM2_Duplicate().
490 // Return Type: TPM_RC
491 //     TPM_RC_ATTRIBUTES    'keyHandle' does not reference a valid decryption key
492 //     TPM_RC_KEY           invalid ECC key (public point is not on the curve)
493 //     TPM_RC_SCHEME        RSA key with an unsupported padding scheme
494 //     TPM_RC_VALUE         numeric value of the data to be decrypted is greater
495 //                          than the RSA key modulus
496 TPM_RC
497 CryptSecretEncrypt(OBJECT*      encryptKey, // IN: encryption key object
498                   const TPM2B* label,      // IN: a null-terminated string as L
499                   TPM2B_DATA* data,        // OUT: secret value
500                   TPM2B_ENCRYPTED_SECRET* secret // OUT: secret structure

```

```

501 )
502 {
503     TPM_RC result = TPM_RC_SUCCESS;
504     //
505     if(data == NULL || secret == NULL)
506         return TPM_RC_FAILURE;
507
508     // CryptKDFe was fixed to not add a NULL byte as per NIST.SP.800-56Cr2.pdf
509     // (required for ACVP tests). This check ensures backwards compatibility with
510     // previous versions of the TPM reference code by verifying the label itself
511     // has a NULL terminator. Note the TPM spec specifies that the label must be NULL
512     // terminated. This is only a "new" failure path in the sense that it adds a
513     // runtime check of hardcoded constants; provided the code is correct it will
never
514     // fail, and running the compliance tests will verify this isn't hit.
515     if((label == NULL) || (label->size == 0) || (label->buffer[label->size - 1] != 0))
516         return TPM_RC_FAILURE;
517
518     // The output secret value has the size of the digest produced by the nameAlg.
519     data->t.size = CryptHashGetDigestSize(encryptKey->publicArea.nameAlg);
520
521     if(!IS_ATTRIBUTE(encryptKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
522         return TPM_RC_ATTRIBUTES;
523     switch(encryptKey->publicArea.type)
524     {
525 #if ALG_RSA
526         case TPM_ALG_RSA:
527         {
528             // The encryption scheme is OAEP using the nameAlg of the encrypt key.
529             TPMT_RSA_DECRYPT scheme;
530             scheme.scheme = TPM_ALG_OAEP;
531             scheme.details.anySig.hashAlg = encryptKey->publicArea.nameAlg;
532
533             // Create secret data from RNG
534             CryptRandomGenerate(data->t.size, data->t.buffer);
535
536             // Encrypt the data by RSA OAEP into encrypted secret
537             result = CryptRsaEncrypt((TPM2B_PUBLIC_KEY_RSA*)secret,
538                                     &data->b,
539                                     encryptKey,
540                                     &scheme,
541                                     label,
542                                     NULL);
543         }
544         break;
545 #endif // ALG_RSA
546
547 #if ALG_ECC
548         case TPM_ALG_ECC:
549         {
550             TPMS_ECC_POINT eccPublic;
551             TPM2B_ECC_PARAMETER eccPrivate;
552             TPMS_ECC_POINT eccSecret;
553             BYTE* buffer = secret->t.secret;
554
555             // Need to make sure that the public point of the key is on the
556             // curve defined by the key.
557             if(!CryptEccIsPointOnCurve(
558                 encryptKey->publicArea.parameters.eccDetail.curveID,
559                 &encryptKey->publicArea.unique.ecc))
560                 result = TPM_RC_KEY;
561             else
562             {
563                 // Call crypto engine to create an auxiliary ECC key
564                 // We assume crypt engine initialization should always success.
565                 // Otherwise, TPM should go to failure mode.

```

```

566
567     CryptEccNewKeyPair(
568         &eccPublic,
569         &eccPrivate,
570         encryptKey->publicArea.parameters.eccDetail.curveID);
571     // Marshal ECC public to secret structure. This will be used by the
572     // recipient to decrypt the secret with their private key.
573     secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
574
575     // Compute ECDH shared secret which is R = [d]Q where d is the
576     // private part of the ephemeral key and Q is the public part of a
577     // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
578     // because the auxiliary ECC key is just created according to the
579     // parameters of input ECC encrypt key.
580     if(CryptEccPointMultiply(
581         &eccSecret,
582         encryptKey->publicArea.parameters.eccDetail.curveID,
583         &encryptKey->publicArea.unique.ecc,
584         &eccPrivate,
585         NULL,
586         NULL)
587         != TPM_RC_SUCCESS)
588         result = TPM_RC_KEY;
589     else
590     {
591         // The secret value is computed from Z using KDFe as:
592         // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
593         // Where:
594         // HashID the nameAlg of the decrypt key
595         // Z the x coordinate (Px) of the product (P) of the point
596         // (Q) of the secret and the private x coordinate (de,V)
597         // of the decryption key
598         // Use a null-terminated string containing "SECRET"
599         // PartyUInfo the x coordinate of the point in the secret
600         // (Qe,U)
601         // PartyVInfo the x coordinate of the public key (Qs,V)
602         // bits the number of bits in the digest of HashID
603         // Retrieve seed from KDFe
604         CryptKDFe(encryptKey->publicArea.nameAlg,
605             &eccSecret.x.b,
606             label,
607             &eccPublic.x.b,
608             &encryptKey->publicArea.unique.ecc.x.b,
609             data->t.size * 8,
610             data->t.buffer);
611     }
612 }
613 }
614 break;
615 #endif // ALG_ECC
616 default:
617     FAIL(FATAL_ERROR_INTERNAL);
618     break;
619 }
620 return result;
621 }
622
623 /*** CryptSecretDecrypt()
624 // Decrypt a secret value by asymmetric (or symmetric) algorithm
625 // This function is used for ActivateCredential and Import for asymmetric
626 // decryption, and StartAuthSession for both asymmetric and symmetric
627 // decryption process
628 //
629 // Return Type: TPM_RC
630 // TPM_RC_ATTRIBUTES RSA key is not a decryption key
631 // TPM_RC_BINDING Invalid RSA key (public and private parts are not

```

```

632 //                                     cryptographically bound.
633 //                                     ECC point in the secret is not on the curve
634 //                                     TPM_RC_INSUFFICIENT failed to retrieve ECC point from the secret
635 //                                     TPM_RC_NO_RESULT multiplication resulted in ECC point at infinity
636 //                                     TPM_RC_SIZE data to decrypt is not of the same size as RSA key
637 //                                     TPM_RC_VALUE For RSA key, numeric value of the encrypted data is
638 //                                     greater than the modulus, or the recovered data is
639 //                                     larger than the output buffer.
640 //                                     For keyedHash or symmetric key, the secret is
641 //                                     larger than the size of the digest produced by
642 //                                     the name algorithm.
643 //                                     TPM_RC_FAILURE internal error
644 TPM_RC
645 CryptSecretDecrypt(OBJECT* decryptKey, // IN: decrypt key
646                   TPM2B_NONCE* nonceCaller, // IN: nonceCaller. It is needed for
647                   // symmetric decryption. For
648                   // asymmetric decryption, this
649                   // parameter is NULL
650                   const TPM2B* label, // IN: a value for L
651                   TPM2B_ENCRYPTED_SECRET* secret, // IN: input secret
652                   TPM2B_DATA* data // OUT: decrypted secret value
653 )
654 {
655     TPM_RC result = TPM_RC_SUCCESS;
656
657     // CryptKDFe was fixed to not add a NULL byte as per NIST.SP.800-56Cr2.pdf
658     // (required for ACPV tests). This check ensures backwards compatibility with
659     // previous versions of the TPM reference code by verifying the label itself
660     // has a NULL terminator. Note the TPM spec specifies that the label must be NULL
661     // terminated. This is only a "new" failure path in the sense that it adds a
662     // runtime check of hardcoded constants; provided the code is correct it will
663     // never
664     // fail, and running the compliance tests will verify this isn't hit.
665     if((label == NULL) || (label->size == 0) || (label->buffer[label->size - 1] != 0))
666         return TPM_RC_FAILURE;
667
668     // Decryption for secret
669     switch(decryptKey->publicArea.type)
670     {
671     #if ALG_RSA
672     case TPM_ALG_RSA:
673     {
674         TPMT_RSA_DECRYPT scheme;
675         TPMT_RSA_SCHEME* keyScheme =
676             &decryptKey->publicArea.parameters.rsaDetail.scheme;
677         UINT16 digestSize;
678
679         scheme = *(TPMT_RSA_DECRYPT*)keyScheme;
680         // If the key scheme is TPM_ALG_NULL, set the scheme to OAEP and
681         // set the algorithm to the name algorithm.
682         if(scheme.scheme == TPM_ALG_NULL)
683         {
684             // Use OAEP scheme
685             scheme.scheme = TPM_ALG_OAEP;
686             scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
687         }
688         // use the digestSize as an indicator of whether or not the scheme
689         // is using a supported hash algorithm.
690         // Note: depending on the scheme used for encryption, a hashAlg might
691         // not be needed. However, the return value has to have some upper
692         // limit on the size. In this case, it is the size of the digest of the
693         // hash algorithm. It is checked after the decryption is done but, there
694         // is no point in doing the decryption if the size is going to be
695         // 'wrong' anyway.
696         digestSize = CryptHashGetDigestSize(scheme.details.oaep.hashAlg);
697         if(scheme.scheme != TPM_ALG_OAEP || digestSize == 0)

```

```

697         return TPM_RC_SCHEME;
698
699     // Set the output buffer capacity
700     data->t.size = sizeof(data->t.buffer);
701
702     // Decrypt seed by RSA OAEP
703     result =
704         CryptRsaDecrypt(&data->b, &secret->b, decryptKey, &scheme, label);
705     if((result == TPM_RC_SUCCESS) && (data->t.size > digestSize))
706         result = TPM_RC_VALUE;
707     }
708     break;
709 #endif // ALG_RSA
710 #if ALG_ECC
711     case TPM_ALG_ECC:
712     {
713         TPMS_ECC_POINT eccPublic;
714         TPMS_ECC_POINT eccSecret;
715         BYTE*          buffer = secret->t.secret;
716         INT32           size   = secret->t.size;
717
718         // Retrieve ECC point from secret buffer
719         result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
720         if(result == TPM_RC_SUCCESS)
721         {
722             result = CryptEccPointMultiply(
723                 &eccSecret,
724                 decryptKey->publicArea.parameters.eccDetail.curveID,
725                 &eccPublic,
726                 &decryptKey->sensitive.sensitive.ecc,
727                 NULL,
728                 NULL);
729             if(result == TPM_RC_SUCCESS)
730             {
731                 // Set the size of the "recovered" secret value to be the size
732                 // of the digest produced by the nameAlg.
733                 data->t.size =
734                     CryptHashGetDigestSize(decryptKey->publicArea.nameAlg);
735
736                 // The secret value is computed from Z using KDFe as:
737                 // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
738                 // Where:
739                 // HashID -- the nameAlg of the decrypt key
740                 // Z -- the x coordinate (Px) of the product (P) of the point
741                 // (Q) of the secret and the private x coordinate (de,V)
742                 // of the decryption key
743                 // Use -- a null-terminated string containing "SECRET"
744                 // PartyUInfo -- the x coordinate of the point in the secret
745                 // (Qe,U )
746                 // PartyVInfo -- the x coordinate of the public key (Qs,V )
747                 // bits -- the number of bits in the digest of HashID
748                 // Retrieve seed from KDFe
749                 CryptKDFe(decryptKey->publicArea.nameAlg,
750                     &eccSecret.x.b,
751                     label,
752                     &eccPublic.x.b,
753                     &decryptKey->publicArea.unique.ecc.x.b,
754                     data->t.size * 8,
755                     data->t.buffer);
756             }
757         }
758     }
759     break;
760 #endif // ALG_ECC
761 #if !ALG_KEYEDHASH
762 # error "KEYEDHASH support is required"

```



```

763 #endif
764 case TPM_ALG_KEYEDHASH:
765     // The seed size can not be bigger than the digest size of nameAlg
766     if(secret->t.size
767         > CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
768         result = TPM_RC_VALUE;
769     else
770     {
771         // Retrieve seed by XOR Obfuscation:
772         // seed = XOR(secret, hash, key, nonceCaller, nullNonce)
773         // where:
774         // secret the secret parameter from the TPM2_StartAuthHMAC
775         // command that contains the seed value
776         // hash nameAlg of tpmKey
777         // key the key or data value in the object referenced by
778         // entityHandle in the TPM2_StartAuthHMAC command
779         // nonceCaller the parameter from the TPM2_StartAuthHMAC command
780         // nullNonce a zero-length nonce
781         // XOR Obfuscation in place
782         CryptXORObfuscation(decryptKey->publicArea.nameAlg,
783                             &decryptKey->sensitive.sensitive.bits.b,
784                             &nonceCaller->b,
785                             NULL,
786                             secret->t.size,
787                             secret->t.secret);
788         // Copy decrypted seed
789         MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
790     }
791     break;
792 case TPM_ALG_SYMCIPHER:
793     {
794         TPM2B_IV iv = {{0}};
795         TPMT_SYM_DEF_OBJECT* symDef;
796         // The seed size can not be bigger than the digest size of nameAlg
797         if(secret->t.size
798             > CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
799             result = TPM_RC_VALUE;
800         else
801         {
802             symDef = &decryptKey->publicArea.parameters.symDetail.sym;
803             iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
804                                                     symDef->keyBits.sym);
805             if(iv.t.size == 0)
806                 return TPM_RC_FAILURE;
807             if(nonceCaller->t.size >= iv.t.size)
808             {
809                 MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
810             }
811             else
812             {
813                 if(nonceCaller->t.size > sizeof(iv.t.buffer))
814                     return TPM_RC_FAILURE;
815                 MemoryCopy(
816                     iv.b.buffer, nonceCaller->t.buffer, nonceCaller->t.size);
817             }
818             // make sure secret will fit
819             if(secret->t.size > sizeof(data->t.buffer))
820                 return TPM_RC_FAILURE;
821             data->t.size = secret->t.size;
822             // CFB decrypt, using nonceCaller as iv
823             CryptSymmetricDecrypt(data->t.buffer,
824                                   symDef->algorithm,
825                                   symDef->keyBits.sym,
826                                   decryptKey->sensitive.sensitive.sym.t.buffer,
827                                   &iv,
828                                   TPM_ALG_CFB,

```

```

829         secret->t.size,
830         secret->t.secret);
831     }
832 }
833 break;
834 default:
835     FAIL(FATAL_ERROR_INTERNAL);
836     break;
837 }
838 return result;
839 }
840
841 /*** CryptParameterEncryption()
842 // This function does in-place encryption of a response parameter.
843 void CryptParameterEncryption(
844     TPM_HANDLE handle,           // IN: encrypt session handle
845     TPM2B* nonceCaller,         // IN: nonce caller
846     INT32 bufferSize,           // IN: size of parameter buffer
847     UINT16 leadingSizeInByte,    // IN: the size of the leading size field in
848                                     // bytes
849     TPM2B_AUTH* extraKey,        // IN: additional key material other than
850                                     // sessionAuth
851     BYTE* buffer                 // IN/OUT: parameter buffer to be encrypted
852 )
853 {
854     SESSION* session = SessionGet(handle); // encrypt session
855     TPM2B_TYPE(TEMP_KEY,
856         (sizeof(extraKey->t.buffer) + sizeof(session->sessionKey.t.buffer)));
857     TPM2B_TEMP_KEY key;           // encryption key
858     UINT16 cipherSize = 0;        // size of cipher text
859
860     if(bufferSize < leadingSizeInByte)
861     {
862         FAIL(FATAL_ERROR_INTERNAL);
863         return;
864     }
865
866     // Parameter encryption for a non-2B is not supported.
867     if(leadingSizeInByte != 2)
868     {
869         FAIL(FATAL_ERROR_INTERNAL);
870         return;
871     }
872
873     // Retrieve encrypted data size.
874     if(UINT16_Unmarshal(&cipherSize, &buffer, &bufferSize) != TPM_RC_SUCCESS)
875     {
876         FAIL(FATAL_ERROR_INTERNAL);
877         return;
878     }
879
880     if(cipherSize > bufferSize)
881     {
882         FAIL(FATAL_ERROR_INTERNAL);
883         return;
884     }
885
886     // Compute encryption key by concatenating sessionKey with extra key
887     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
888     MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
889
890     if(session->symmetric.algorithm == TPM_ALG_XOR)
891
892         // XOR parameter encryption formulation:
893         // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
894         CryptXORObfuscation(session->authHashAlg,

```

```

895         &(key.b),
896         &(session->nonceTPM.b),
897         nonceCaller,
898         (UINT32)cipherSize,
899         buffer);
900     else
901         ParmEncryptSym(session->symmetric.algorithm,
902             session->authHashAlg,
903             session->symmetric.keyBits.aes,
904             &(key.b),
905             nonceCaller,
906             &(session->nonceTPM.b),
907             (UINT32)cipherSize,
908             buffer);
909     return;
910 }
911
912 /*** CryptParameterDecryption()
913 // This function does in-place decryption of a command parameter.
914 // Return Type: TPM_RC
915 // TPM_RC_SIZE The number of bytes in the input buffer is less than
916 // the number of bytes to be decrypted.
917 TPM_RC
918 CryptParameterDecryption(
919     TPM_HANDLE handle, // IN: encrypted session handle
920     TPM2B* nonceCaller, // IN: nonce caller
921     INT32 bufferSize, // IN: size of parameter buffer
922     UINT16 leadingSizeInByte, // IN: the size of the leading size field in
923                             // byte
924     TPM2B_AUTH* extraKey, // IN: the authValue
925     BYTE* buffer // IN/OUT: parameter buffer to be decrypted
926 )
927 {
928     SESSION* session = SessionGet(handle); // encrypt session
929     // The HMAC key is going to be the concatenation of the session key and any
930     // additional key material (like the authValue). The size of both of these
931     // is the size of the buffer which can contain a TPMT_HA.
932     TPM2B_TYPE(HMAC_KEY,
933         (sizeof(extraKey->t.buffer) + sizeof(session->sessionKey.t.buffer)));
934     TPM2B_HMAC_KEY key; // decryption key
935     UINT16 cipherSize = 0; // size of ciphertext
936
937     if(bufferSize < leadingSizeInByte)
938     {
939         return TPM_RC_INSUFFICIENT;
940     }
941
942     // Parameter encryption for a non-2B is not supported.
943     if(leadingSizeInByte != 2)
944     {
945         FAIL_RC(FATAL_ERROR_INTERNAL);
946     }
947
948     // Retrieve encrypted data size.
949     if(UINT16_Unmarshal(&cipherSize, &buffer, &bufferSize) != TPM_RC_SUCCESS)
950     {
951         return TPM_RC_INSUFFICIENT;
952     }
953
954     if(cipherSize > bufferSize)
955     {
956         return TPM_RC_SIZE;
957     }
958
959     // Compute decryption key by concatenating sessionAuth with extra input key
960     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));

```

```

961     MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
962
963     if(session->symmetric.algorithm == TPM_ALG_XOR)
964         // XOR parameter decryption formulation:
965         // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
966         // Call XOR obfuscation function
967         CryptXORObfuscation(session->authHashAlg,
968                             &key.b,
969                             nonceCaller,
970                             &(session->nonceTPM.b),
971                             (UINT32)cipherSize,
972                             buffer);
973     else
974         // Assume that it is one of the symmetric block ciphers.
975         ParmDecryptSym(session->symmetric.algorithm,
976                       session->authHashAlg,
977                       session->symmetric.keyBits.sym,
978                       &key.b,
979                       nonceCaller,
980                       &(session->nonceTPM.b),
981                       (UINT32)cipherSize,
982                       buffer);
983
984     return TPM_RC_SUCCESS;
985 }
986
987 /*** CryptComputeSymmetricUnique()
988 // This function computes the unique field in public area for symmetric objects.
989 void CryptComputeSymmetricUnique(
990     TPMT_PUBLIC*    publicArea, // IN: the object's public area
991     TPMT_SENSITIVE* sensitive,  // IN: the associated sensitive area
992     TPM2B_DIGEST*   unique     // OUT: unique buffer
993 )
994 {
995     // For parents (symmetric and derivation), use an HMAC to compute
996     // the 'unique' field
997     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
998        && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt))
999     {
1000         // Unique field is HMAC(sensitive->seedValue, sensitive->sensitive)
1001         HMAC_STATE hmacState;
1002         unique->b.size = CryptHmacStart2B(
1003             &hmacState, publicArea->nameAlg, &sensitive->seedValue.b);
1004         CryptDigestUpdate2B(&hmacState.hashState, &sensitive->sensitive.any.b);
1005         CryptHmacEnd2B(&hmacState, &unique->b);
1006     }
1007     else
1008     {
1009         HASH_STATE hashState;
1010         // Unique := Hash(sensitive->seedValue || sensitive->sensitive)
1011         unique->t.size = CryptHashStart(&hashState, publicArea->nameAlg);
1012         CryptDigestUpdate2B(&hashState, &sensitive->seedValue.b);
1013         CryptDigestUpdate2B(&hashState, &sensitive->sensitive.any.b);
1014         CryptHashEnd2B(&hashState, &unique->b);
1015     }
1016     return;
1017 }
1018
1019 /*** CryptCreateObject()
1020 // This function creates an object.
1021 // For an asymmetric key, it will create a key pair and, for a parent key, a seed
1022 // value for child protections.
1023 //
1024 // For an symmetric object, (TPM_ALG_SYMCIPHER or TPM_ALG_KEYEDHASH), it will
1025 // create a secret key if the caller did not provide one. It will create a random
1026 // secret seed value that is hashed with the secret value to create the public

```

```

1027 // unique value.
1028 //
1029 // 'publicArea', 'sensitive', and 'sensitiveCreate' are the only required parameters
1030 // and are the only ones that are used by TPM2_Create(). The other parameters
1031 // are optional and are used when the generated Object needs to be deterministic.
1032 // This is the case for both Primary Objects and Derived Objects.
1033 //
1034 // When a seed value is provided, a RAND_STATE will be populated and used for
1035 // all operations in the object generation that require a random number. In the
1036 // simplest case, TPM2_CreatePrimary() will use 'seed', 'label' and 'context' with
1037 // context being the hash of the template. If the Primary Object is in
1038 // the Endorsement hierarchy, it will also populate 'proof' with ehProof.
1039 //
1040 // For derived keys, 'seed' will be the secret value from the parent, 'label' and
1041 // 'context' will be set according to the parameters of TPM2_CreateLoaded() and
1042 // 'hashAlg' will be set which causes the RAND_STATE to be a KDF generator.
1043 //
1044 // Return Type: TPM_RC
1045 //     TPM_RC_KEY           a provided key is not an allowed value
1046 //     TPM_RC_KEY_SIZE     key size in the public area does not match the size
1047 //                         in the sensitive creation area for a symmetric key
1048 //     TPM_RC_NO_RESULT    unable to get random values (only in derivation)
1049 //     TPM_RC_RANGE        for an RSA key, the exponent is not supported
1050 //     TPM_RC_SIZE        sensitive data size is larger than allowed for the
1051 //                         scheme for a keyed hash object
1052 //     TPM_RC_VALUE        exponent is not prime or could not find a prime using
1053 //                         the provided parameters for an RSA key;
1054 //                         unsupported name algorithm for an ECC key
1055 TPM_RC
1056 CryptCreateObject(OBJECT*          object, // IN: new object structure pointer
1057                  TPMS_SENSITIVE_CREATE* sensitiveCreate, // IN: sensitive creation
1058                  RAND_STATE*      rand // IN: the random number generator
1059                  // to use
1060 )
1061 {
1062     TPMT_PUBLIC*    publicArea = &object->publicArea;
1063     TPMT_SENSITIVE* sensitive = &object->sensitive;
1064     TPM_RC          result     = TPM_RC_SUCCESS;
1065     //
1066     // Set the sensitive type for the object
1067     sensitive->sensitiveType = publicArea->type;
1068
1069     // For all objects, copy the initial authorization data
1070     sensitive->authValue = sensitiveCreate->userAuth;
1071
1072     // If the TPM is the source of the data, set the size of the provided data to
1073     // zero so that there's no confusion about what to do.
1074     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sensitiveDataOrigin))
1075         sensitiveCreate->data.t.size = 0;
1076
1077     // Generate the key and unique fields for the asymmetric keys and just the
1078     // sensitive value for symmetric object
1079     switch(publicArea->type)
1080     {
1081 #if ALG_RSA
1082         // Create RSA key
1083         case TPM_ALG_RSA:
1084             // RSA uses full object so that it has a place to put the private
1085             // exponent
1086             result = CryptRsaGenerateKey(publicArea, sensitive, rand);
1087             break;
1088 #endif // ALG_RSA
1089
1090 #if ALG_ECC
1091         // Create ECC key
1092         case TPM_ALG_ECC:

```

```

1093         result = CryptEccGenerateKey(publicArea, sensitive, rand);
1094         break;
1095 #endif // ALG_ECC
1096 case TPM_ALG_SYMCIPHER:
1097     result = CryptGenerateKeySymmetric(
1098         publicArea, sensitive, sensitiveCreate, rand);
1099     break;
1100 case TPM_ALG_KEYEDHASH:
1101     result =
1102         CryptGenerateKeyedHash(publicArea, sensitive, sensitiveCreate, rand);
1103     break;
1104 default:
1105     FAIL(FATAL_ERROR_INTERNAL);
1106     break;
1107 }
1108 if(result != TPM_RC_SUCCESS)
1109     return result;
1110 // Create the sensitive seed value
1111 // If this is a primary key in the endorsement hierarchy, stir the DRBG state
1112 // This implementation uses both shProof and ehProof to make sure that there
1113 // is no leakage of either.
1114 if(object->attributes.primary && object->attributes.epsHierarchy)
1115 {
1116     DRBG_AdditionalData((DRBG_STATE*)rand, &gp.shProof.b);
1117     DRBG_AdditionalData((DRBG_STATE*)rand, &gp.ehProof.b);
1118 }
1119 // Generate a seedValue that is the size of the digest produced by nameAlg
1120 sensitive->seedValue.t.size =
1121     DRBG_Generate(rand,
1122         sensitive->seedValue.t.buffer,
1123         CryptHashGetDigestSize(publicArea->nameAlg));
1124 if(g_inFailureMode)
1125     return TPM_RC_FAILURE;
1126 else if(sensitive->seedValue.t.size == 0)
1127     return TPM_RC_NO_RESULT;
1128 // For symmetric objects, need to compute the unique value for the public area
1129 if(publicArea->type == TPM_ALG_SYMCIPHER || publicArea->type == TPM_ALG_KEYEDHASH)
1130 {
1131     CryptComputeSymmetricUnique(publicArea, sensitive, &publicArea->unique.sym);
1132 }
1133 else
1134 {
1135     // if this is an asymmetric key and it isn't a parent, then
1136     // get rid of the seed.
1137     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
1138         || !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
1139         memset(&sensitive->seedValue, 0, sizeof(sensitive->seedValue));
1140 }
1141 // Compute the name
1142 PublicMarshalAndComputeName(publicArea, &object->name);
1143 return result;
1144 }
1145
1146 /*** CryptGetSignHashAlg()
1147 // Get the hash algorithm of signature from a TPMT_SIGNATURE structure.
1148 // It assumes the signature is not NULL
1149 // This is a function for easy access
1150 TPMI_ALG_HASH
1151 CryptGetSignHashAlg(TPMT_SIGNATURE* auth // IN: signature
1152 )
1153 {
1154     if(auth->sigAlg == TPM_ALG_NULL)
1155         FAIL(FATAL_ERROR_INTERNAL);
1156
1157     // Get authHash algorithm based on signing scheme
1158     switch(auth->sigAlg)

```



```

1159     {
1160     #if ALG_RSA
1161         // If RSA is supported, both RSASSA and RSAPSS are required
1162         # if !defined TPM_ALG_RSASSA || !defined TPM_ALG_RSAPSS
1163         #     error "RSASSA and RSAPSS are required for RSA"
1164         # endif
1165         case TPM_ALG_RSASSA:
1166             return auth->signature.rsassa.hash;
1167         case TPM_ALG_RSAPSS:
1168             return auth->signature.rsapss.hash;
1169     #endif // ALG_RSA
1170
1171     #if ALG_ECC
1172         // If ECC is defined, ECDSA is mandatory
1173         # if !ALG_ECDSA
1174         #     error "ECDSA is required for ECC"
1175         # endif
1176         case TPM_ALG_ECDSA:
1177             // SM2 and ECSCHNORR are optional
1178
1179         # if ALG_SM2
1180             case TPM_ALG_SM2:
1181             # endif
1182         # if ALG_ECSCHNORR
1183             case TPM_ALG_ECSCHNORR:
1184             # endif
1185             //all ECC signatures look the same
1186             return auth->signature.ecdsa.hash;
1187
1188         # if ALG_ECDA
1189             // Don't know how to verify an ECDA signature
1190             case TPM_ALG_ECDA:
1191                 break;
1192         # endif
1193     #endif // ALG_ECC
1194
1195     case TPM_ALG_HMAC:
1196         return auth->signature.hmac.hashAlg;
1197
1198     default:
1199         break;
1200     }
1201     return TPM_ALG_NULL;
1202 }
1203
1204
1205 /*** CryptIsSplitSign()
1206 // This function is used to determine if the signing operation is a split
1207 // signing operation that required a TPM2_Commit().
1208 //
1209 BOOL CryptIsSplitSign(TPM_ALG_ID scheme // IN: the algorithm selector
1210 )
1211 {
1212     switch(scheme)
1213     {
1214     #if ALG_ECDA
1215         case TPM_ALG_ECDA:
1216             return TRUE;
1217             break;
1218     #endif // ALG_ECDA
1219     default:
1220         return FALSE;
1221         break;
1222     }
1223 }
1224

```

```

1225  /*** CryptIsAsymSignScheme()
1226  // This function indicates if a scheme algorithm is a sign algorithm.
1227  BOOL CryptIsAsymSignScheme(TPMI_ALG_PUBLIC publicType, // IN: Type of the object
1228                             TPMI_ALG_ASYNC_SCHEME scheme // IN: the scheme
1229  )
1230  {
1231      BOOL isSignScheme = TRUE;
1232
1233      switch(publicType)
1234      {
1235      #if ALG_RSA
1236          case TPM_ALG_RSA:
1237              switch(scheme)
1238              {
1239              # if !ALG_RSASSA || !ALG_RSAPSS
1240              # error "RSASSA and PSAPSS required if RSA used."
1241              # endif
1242                  case TPM_ALG_RSASSA:
1243                  case TPM_ALG_RSAPSS:
1244                      break;
1245                  default:
1246                      isSignScheme = FALSE;
1247                      break;
1248              }
1249              break;
1250      #endif // ALG_RSA
1251
1252      #if ALG_ECC
1253          // If ECC is implemented ECDSA is required
1254          case TPM_ALG_ECC:
1255              switch(scheme)
1256              {
1257                  // Support for ECDSA is required for ECC
1258                  case TPM_ALG_ECDSA:
1259              # if ALG_ECDAA // ECDAA is optional
1260                  case TPM_ALG_ECDAA:
1261              # endif
1262              # if ALG_ECSCHNORR // Schnorr is also optional
1263                  case TPM_ALG_ECSCHNORR:
1264              # endif
1265              # if ALG_SM2 // SM2 is optional
1266                  case TPM_ALG_SM2:
1267              # endif
1268                      break;
1269                  default:
1270                      isSignScheme = FALSE;
1271                      break;
1272              }
1273              break;
1274      #endif // ALG_ECC
1275          default:
1276              isSignScheme = FALSE;
1277              break;
1278      }
1279      return isSignScheme;
1280  }
1281
1282  /*** CryptIsAsymDecryptScheme()
1283  // This function indicate if a scheme algorithm is a decrypt algorithm.
1284  BOOL CryptIsAsymDecryptScheme(TPMI_ALG_PUBLIC publicType, // IN: Type of the object
1285                                TPMI_ALG_ASYNC_SCHEME scheme // IN: the scheme
1286  )
1287  {
1288      BOOL isDecryptScheme = TRUE;
1289
1290      switch(publicType)

```

```

1291     {
1292     #if ALG_RSA
1293         case TPM_ALG_RSA:
1294             switch (scheme)
1295             {
1296                 case TPM_ALG_RSAES:
1297                 case TPM_ALG_OAEP:
1298                     break;
1299                 default:
1300                     isDecryptScheme = FALSE;
1301                     break;
1302             }
1303             break;
1304     #endif // ALG_RSA
1305
1306     #if ALG_ECC
1307         // If ECC is implemented ECDH is required
1308         case TPM_ALG_ECC:
1309             switch (scheme)
1310             {
1311                 # if !ALG_ECDH
1312                 #   error "ECDH is required for ECC"
1313                 # endif
1314                 case TPM_ALG_ECDH:
1315                 # if ALG_SM2
1316                 case TPM_ALG_SM2:
1317                 # endif
1318                 # if ALG_ECMQV
1319                 case TPM_ALG_ECMQV:
1320                 # endif
1321                     break;
1322                 default:
1323                     isDecryptScheme = FALSE;
1324                     break;
1325             }
1326             break;
1327     #endif // ALG_ECC
1328     default:
1329         isDecryptScheme = FALSE;
1330         break;
1331     }
1332     return isDecryptScheme;
1333 }
1334
1335 /*** CryptSelectSignScheme()
1336 // This function is used by the attestation and signing commands. It implements
1337 // the rules for selecting the signature scheme to use in signing. This function
1338 // requires that the signing key either be TPM_RH_NULL or be loaded.
1339 //
1340 // If a default scheme is defined in object, the default scheme should be chosen,
1341 // otherwise, the input scheme should be chosen.
1342 // In the case that both object and input scheme has a non-NULL scheme
1343 // algorithm, if the schemes are compatible, the input scheme will be chosen.
1344 //
1345 // This function should not be called if 'signObject->publicArea.type' ==
1346 // ALG_SYMCIPHER.
1347 //
1348 // Return Type: BOOL
1349 //     TRUE(1)         scheme selected
1350 //     FALSE(0)        both 'scheme' and key's default scheme are empty; or
1351 //                     'scheme' is empty while key's default scheme requires
1352 //                     explicit input scheme (split signing); or
1353 //                     non-empty default key scheme differs from 'scheme'
1354 BOOL CryptSelectSignScheme(OBJECT*      signObject, // IN: signing key
1355                           TPMT_SIG_SCHEME* scheme    // IN/OUT: signing scheme
1356 )

```

```

1357 {
1358     TPMT_SIG_SCHEME* objectScheme;
1359     TPMT_PUBLIC*      publicArea;
1360     BOOL              OK;
1361
1362     // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
1363     // of the setting of scheme
1364     if(signObject == NULL)
1365     {
1366         OK = TRUE;
1367         scheme->scheme = TPM_ALG_NULL;
1368         scheme->details.any.hashAlg = TPM_ALG_NULL;
1369     }
1370     else
1371     {
1372         // assignment to save typing.
1373         publicArea = &signObject->publicArea;
1374
1375         // A symmetric cipher can be used to encrypt and decrypt but it can't
1376         // be used for signing
1377         if(publicArea->type == TPM_ALG_SYMCIPHER)
1378             return FALSE;
1379         // Point to the scheme object
1380         if(CryptIsAsymAlgorithm(publicArea->type))
1381             objectScheme =
1382                 (TPMT_SIG_SCHEME*)&publicArea->parameters.asymDetail.scheme;
1383         else
1384             objectScheme =
1385                 (TPMT_SIG_SCHEME*)&publicArea->parameters.keyedHashDetail.scheme;
1386
1387         // If the object doesn't have a default scheme, then use the
1388         // input scheme.
1389         if(objectScheme->scheme == TPM_ALG_NULL)
1390         {
1391             // Input and default can't both be NULL
1392             OK = (scheme->scheme != TPM_ALG_NULL);
1393             // Assume that the scheme is compatible with the key. If not,
1394             // an error will be generated in the signing operation.
1395         }
1396         else if(scheme->scheme == TPM_ALG_NULL)
1397         {
1398             // input scheme is NULL so use default
1399
1400             // First, check to see if the default requires that the caller
1401             // provided scheme data
1402             OK = !CryptIsSplitSign(objectScheme->scheme);
1403             if(OK)
1404             {
1405                 // The object has a scheme and the input is TPM_ALG_NULL so copy
1406                 // the object scheme as the final scheme. It is better to use a
1407                 // structure copy than a copy of the individual fields.
1408                 *scheme = *objectScheme;
1409             }
1410         }
1411         else
1412         {
1413             // Both input and object have scheme selectors
1414             // If the scheme and the hash are not the same then...
1415             // NOTE: the reason that there is no copy here is that the input
1416             // might contain extra data for a split signing scheme and that
1417             // data is not in the object so, it has to be preserved.
1418             OK =
1419                 (objectScheme->scheme == scheme->scheme)
1420                 && (objectScheme->details.any.hashAlg == scheme->details.any.hashAlg);
1421         }
1422     }

```

```

1423     return OK;
1424 }
1425
1426 /*** CryptSign()
1427 // Sign a digest with asymmetric key or HMAC.
1428 // This function is called by attestation commands and the generic TPM2_Sign
1429 // command.
1430 // This function checks the key scheme and digest size. It does not
1431 // check if the sign operation is allowed for restricted key. It should be
1432 // checked before the function is called.
1433 // The function will assert if the key is not a signing key.
1434 //
1435 // Return Type: TPM_RC
1436 //     TPM_RC_SCHEME      'signScheme' is not compatible with the signing key type
1437 //     TPM_RC_VALUE       'digest' value is greater than the modulus of
1438 //                        'signHandle' or size of 'hashData' does not match hash
1439 //                        algorithm in 'signScheme' (for an RSA key);
1440 //                        invalid commit status or failed to generate "r" value
1441 //                        (for an ECC key)
1442 TPM_RC
1443 CryptSign(OBJECT*      signKey,      // IN: signing key
1444           TPMT_SIG_SCHEME* signScheme, // IN: sign scheme.
1445           TPM2B_DIGEST* digest,      // IN: The digest being signed
1446           TPMT_SIGNATURE* signature // OUT: signature
1447 )
1448 {
1449     TPM_RC result = TPM_RC_SCHEME;
1450
1451     // Initialize signature scheme
1452     signature->sigAlg = signScheme->scheme;
1453
1454     // If the signature algorithm is TPM_ALG_NULL or the signing key is NULL,
1455     // then we are done
1456     if((signature->sigAlg == TPM_ALG_NULL) || (signKey == NULL))
1457         return TPM_RC_SUCCESS;
1458
1459     // Initialize signature hash
1460     // Note: need to do the check for TPM_ALG_NULL first because the null scheme
1461     // doesn't have a hashAlg member.
1462     signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
1463
1464     // perform sign operation based on different key type
1465     switch(signKey->publicArea.type)
1466     {
1467 #if ALG_RSA
1468         case TPM_ALG_RSA:
1469             result = CryptRsaSign(signature, signKey, digest, NULL);
1470             break;
1471 #endif // ALG_RSA
1472 #if ALG_ECC
1473         case TPM_ALG_ECC:
1474             // The reason that signScheme is passed to CryptEccSign but not to the
1475             // other signing methods is that the signing for ECC may be split and
1476             // need the 'r' value that is in the scheme but not in the signature.
1477             result = CryptEccSign(
1478                 signature, signKey, digest, (TPMT_ECC_SCHEME*)signScheme, NULL);
1479             break;
1480 #endif // ALG_ECC
1481         case TPM_ALG_KEYEDHASH:
1482             result = CryptHmacSign(signature, signKey, digest);
1483             break;
1484         default:
1485             FAIL(FATAL_ERROR_INTERNAL);
1486             break;
1487     }
1488     return result;

```

```

1489 }
1490
1491 /*** CryptValidateSignature()
1492 // This function is used to verify a signature. It is called by
1493 // TPM2_VerifySignature() and TPM2_PolicySigned.
1494 //
1495 // Since this operation only requires use of a public key, no consistency
1496 // checks are necessary for the key to signature type because a caller can load
1497 // any public key that they like with any scheme that they like. This routine
1498 // simply makes sure that the signature is correct, whatever the type.
1499 //
1500 // Return Type: TPM_RC
1501 //     TPM_RC_SIGNATURE      the signature is not genuine
1502 //     TPM_RC_SCHEME         the scheme is not supported
1503 //     TPM_RC_HANDLE         an HMAC key was selected but the
1504 //                           private part of the key is not loaded
1505 TPM_RC
1506 CryptValidateSignature(TPMI_DH_OBJECT keyHandle, // IN: The handle of sign key
1507                       TPM2B_DIGEST* digest,    // IN: The digest being validated
1508                       TPMT_SIGNATURE* signature // IN: signature
1509 )
1510 {
1511     // NOTE: HandleToObject will either return a pointer to a loaded object or
1512     // will assert. It will never return a non-valid value. This makes it safe
1513     // to initialize 'publicArea' with the return value from HandleToObject()
1514     // without checking it first.
1515     OBJECT*      signObject = HandleToObject(keyHandle);
1516     TPMT_PUBLIC* publicArea = &signObject->publicArea;
1517     TPM_RC      result      = TPM_RC_SCHEME;
1518
1519     // The input unmarshaling should prevent any input signature from being
1520     // a NULL signature, but just in case
1521     if(signature->sigAlg == TPM_ALG_NULL)
1522         return TPM_RC_SIGNATURE;
1523
1524     switch(publicArea->type)
1525     {
1526 #if ALG_RSA
1527         case TPM_ALG_RSA:
1528         {
1529             //
1530             // Call RSA code to verify signature
1531             result = CryptRsaValidateSignature(signature, signObject, digest);
1532             break;
1533         }
1534 #endif // ALG_RSA
1535
1536 #if ALG_ECC
1537         case TPM_ALG_ECC:
1538             result = CryptEccValidateSignature(signature, signObject, digest);
1539             break;
1540 #endif // ALG_ECC
1541
1542         case TPM_ALG_KEYEDHASH:
1543             if(signObject->attributes.publicOnly)
1544                 result = TPM_RC_HANDLE;
1545             else
1546                 result = CryptHMACVerifySignature(signObject, digest, signature);
1547             break;
1548         default:
1549             break;
1550     }
1551     return result;
1552 }
1553
1554 /*** CryptGetTestResult

```



```

1555 // This function returns the results of a self-test function.
1556 // Note: the behavior in this function is NOT the correct behavior for a real
1557 // TPM implementation. An artificial behavior is placed here due to the
1558 // limitation of a software simulation environment. For the correct behavior,
1559 // consult the part 3 specification for TPM2_GetTestResult().
1560 TPM_RC
1561 CryptGetTestResult(TPM2B_MAX_BUFFER* outData // OUT: test result data
1562 )
1563 {
1564     outData->t.size = 0;
1565     return TPM_RC_SUCCESS;
1566 }
1567
1568 /*** CryptValidateKeys()
1569 // This function is used to verify that the key material of and object is valid.
1570 // For a 'publicOnly' object, the key is verified for size and, if it is an ECC
1571 // key, it is verified to be on the specified curve. For a key with a sensitive
1572 // area, the binding between the public and private parts of the key are verified.
1573 // If the nameAlg of the key is TPM_ALG_NULL, then the size of the sensitive area
1574 // is verified but the public portion is not verified, unless the key is an RSA key.
1575 // For an RSA key, the reason for loading the sensitive area is to use it. The
1576 // only way to use a private RSA key is to compute the private exponent. To compute
1577 // the private exponent, the public modulus is used.
1578 // Return Type: TPM_RC
1579 //     TPM_RC_BINDING      the public and private parts are not cryptographically
1580 //                          bound
1581 //     TPM_RC_HASH         cannot have a publicOnly key with nameAlg of TPM_ALG_NULL
1582 //     TPM_RC_KEY          the public unique is not valid
1583 //     TPM_RC_KEY_SIZE     the private area key is not valid
1584 //     TPM_RC_TYPE         the types of the sensitive and private parts do not match
1585 TPM_RC
1586 CryptValidateKeys(TPMT_PUBLIC*    publicArea,
1587                  TPMT_SENSITIVE* sensitive,
1588                  TPM_RC          blamePublic,
1589                  TPM_RC          blameSensitive)
1590 {
1591     TPM_RC          result;
1592     UINT16          keySizeInBytes;
1593     UINT16          digestSize = CryptHashGetDigestSize(publicArea->nameAlg);
1594     TPMU_PUBLIC_PARMS* params    = &publicArea->parameters;
1595     TPMU_PUBLIC_ID*   unique     = &publicArea->unique;
1596
1597     if(sensitive != NULL)
1598     {
1599         // Make sure that the types of the public and sensitive are compatible
1600         if(publicArea->type != sensitive->sensitiveType)
1601             return TPM_RCS_TYPE + blameSensitive;
1602         // Make sure that the authValue is not bigger than allowed
1603         // If there is no name algorithm, then the size just needs to be less than
1604         // the maximum size of the buffer used for authorization. That size check
1605         // was made during unmarshaling of the sensitive area
1606         if((sensitive->authValue.t.size) > digestSize && (digestSize > 0))
1607             return TPM_RCS_SIZE + blameSensitive;
1608     }
1609     switch(publicArea->type)
1610     {
1611 #if ALG_RSA
1612         case TPM_ALG_RSA:
1613             keySizeInBytes = BITS_TO_BYTES(params->rsaDetail.keyBits);
1614
1615             // Regardless of whether there is a sensitive area, the public modulus
1616             // needs to have the correct size. Otherwise, it can't be used for
1617             // any public key operation nor can it be used to compute the private
1618             // exponent.
1619             // NOTE: This implementation only supports key sizes that are multiples
1620             // of 1024 bits which means that the MSb of the 0th byte will always be

```

```

1621 // SET in any prime and in the public modulus.
1622 if((unique->rsa.t.size != keySizeInBytes)
1623    || (unique->rsa.t.buffer[0] < 0x80))
1624     return TPM_RCS_KEY + blamePublic;
1625 if(params->rsaDetail.exponent != 0 && params->rsaDetail.exponent < 7)
1626     return TPM_RCS_VALUE + blamePublic;
1627 if(sensitive != NULL)
1628 {
1629     // If there is a sensitive area, it has to be the correct size
1630     // including having the correct high order bit SET.
1631     if(((sensitive->sensitive.rsa.t.size * 2) != keySizeInBytes)
1632        || (sensitive->sensitive.rsa.t.buffer[0] < 0x80))
1633         return TPM_RCS_KEY_SIZE + blameSensitive;
1634     }
1635     break;
1636 #endif
1637 #if ALG_ECC
1638     case TPM_ALG_ECC:
1639     {
1640         TPMI_ECC_CURVE curveId;
1641         curveId = params->eccDetail.curveID;
1642         keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(curveId));
1643         if(sensitive == NULL)
1644         {
1645             // Validate the public key size
1646             if(unique->ecc.x.t.size != keySizeInBytes
1647                || unique->ecc.y.t.size != keySizeInBytes)
1648                 return TPM_RCS_KEY + blamePublic;
1649             if(publicArea->nameAlg != TPM_ALG_NULL)
1650             {
1651                 if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))
1652                     return TPM_RCS_ECC_POINT + blamePublic;
1653             }
1654         }
1655         else
1656         {
1657             // If the nameAlg is TPM_ALG_NULL, then only verify that the
1658             // private part of the key is OK.
1659             if(!CryptEccIsValidPrivateKey(&sensitive->sensitive.ecc, curveId))
1660                 return TPM_RCS_KEY_SIZE;
1661             if(publicArea->nameAlg != TPM_ALG_NULL)
1662             {
1663                 // Full key load, verify that the public point belongs to the
1664                 // private key.
1665                 TPMS_ECC_POINT toCompare;
1666                 result = CryptEccPointMultiply(&toCompare,
1667                                                 curveId,
1668                                                 NULL,
1669                                                 &sensitive->sensitive.ecc,
1670                                                 NULL,
1671                                                 NULL);
1672                 if(result != TPM_RC_SUCCESS)
1673                     return TPM_RCS_BINDING;
1674             }
1675             else
1676             {
1677                 // Make sure that the private key generated the public key.
1678                 // The input values and the values produced by the point
1679                 // multiply may not be the same size so adjust the computed
1680                 // value to match the size of the input value by adding or
1681                 // removing zeros.
1682                 AdjustNumberB(&toCompare.x.b, unique->ecc.x.t.size);
1683                 AdjustNumberB(&toCompare.y.b, unique->ecc.y.t.size);
1684                 if(!MemoryEqual2B(&unique->ecc.x.b, &toCompare.x.b)
1685                    || !MemoryEqual2B(&unique->ecc.y.b, &toCompare.y.b))
1686                     return TPM_RCS_BINDING;
1687             }
1688         }
1689     }
1690 #endif

```

```

1687     }
1688 }
1689     break;
1690 }
1691 #endif
1692 default:
1693     // Checks for SYMCIPHER and KEYEDHASH are largely the same
1694     // If public area has a nameAlg, then validate the public area size
1695     // and if there is also a sensitive area, validate the binding
1696
1697     // For consistency, if the object is public-only just make sure that
1698     // the unique field is consistent with the name algorithm
1699     if(sensitive == NULL)
1700     {
1701         if(unique->sym.t.size != digestSize)
1702             return TPM_RCS_KEY + blamePublic;
1703     }
1704     else
1705     {
1706         // Make sure that the key size in the sensitive area is consistent.
1707         if(publicArea->type == TPM_ALG_SYMCIPHER)
1708         {
1709             result = CryptSymKeyValidate(&params->symDetail.sym,
1710                                         &sensitive->sensitive.sym);
1711             if(result != TPM_RC_SUCCESS)
1712                 return result + blameSensitive;
1713         }
1714         else
1715         {
1716             // For a keyed hash object, the key has to be less than the
1717             // smaller of the block size of the hash used in the scheme or
1718             // 128 bytes. The worst case value is limited by the
1719             // unmarshaling code so the only thing left to be checked is
1720             // that it does not exceed the block size of the hash.
1721             // by the hash algorithm of the scheme.
1722             TPMT_KEYEDHASH_SCHEME* scheme;
1723             UINT16 maxSize;
1724             scheme = &params->keyedHashDetail.scheme;
1725             if(scheme->scheme == TPM_ALG_XOR)
1726             {
1727                 maxSize = CryptHashGetBlockSize(scheme->details.xor.hashAlg);
1728             }
1729             else if(scheme->scheme == TPM_ALG_HMAC)
1730             {
1731                 maxSize = CryptHashGetBlockSize(scheme->details.hmac.hashAlg);
1732             }
1733             else if(scheme->scheme == TPM_ALG_NULL)
1734             {
1735                 // Not signing or xor so must be a data block
1736                 maxSize = 128;
1737             }
1738             else
1739                 return TPM_RCS_SCHEME + blamePublic;
1740             if(sensitive->sensitive.bits.t.size > maxSize)
1741                 return TPM_RCS_KEY_SIZE + blameSensitive;
1742         }
1743         // If there is a nameAlg, check the binding
1744         if(publicArea->nameAlg != TPM_ALG_NULL)
1745         {
1746             TPM2B_DIGEST compare;
1747             if(sensitive->seedValue.t.size != digestSize)
1748                 return TPM_RCS_KEY_SIZE + blameSensitive;
1749
1750             CryptComputeSymmetricUnique(publicArea, sensitive, &compare);
1751             if(!MemoryEqual2B(&unique->sym.b, &compare.b))
1752                 return TPM_RC_BINDING;

```

```

1753     }
1754 }
1755     break;
1756 }
1757 // For a parent, need to check that the seedValue is the correct size for
1758 // protections. It should be at least half the size of the nameAlg
1759 if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
1760    && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
1761    && sensitive != NULL && publicArea->nameAlg != TPM_ALG_NULL)
1762 {
1763     if((sensitive->seedValue.t.size < (digestSize / 2))
1764        || (sensitive->seedValue.t.size > digestSize))
1765         return TPM_RCS_SIZE + blameSensitive;
1766 }
1767 return TPM_RC_SUCCESS;
1768 }
1769
1770 /*** CryptSelectMac()
1771 // This function is used to set the MAC scheme based on the key parameters and
1772 // the input scheme.
1773 // Return Type: TPM_RC
1774 //     TPM_RC_SCHEME      the scheme is not a valid mac scheme
1775 //     TPM_RC_TYPE        the input key is not a type that supports a mac
1776 //     TPM_RC_VALUE        the input scheme and the key scheme are not compatible
1777 TPM_RC
1778 CryptSelectMac(TPMT_PUBLIC* publicArea, TPMI_ALG_MAC_SCHEME* inMac)
1779 {
1780     TPM_ALG_ID macAlg = TPM_ALG_NULL;
1781     switch(publicArea->type)
1782     {
1783     case TPM_ALG_KEYEDHASH:
1784     {
1785         // Local value to keep lines from getting too long
1786         TPMT_KEYEDHASH_SCHEME* scheme;
1787         scheme = &publicArea->parameters.keyedHashDetail.scheme;
1788         // Expect that the scheme is either HMAC or NULL
1789         if(scheme->scheme != TPM_ALG_NULL)
1790             macAlg = scheme->details.hmac.hashAlg;
1791         break;
1792     }
1793     case TPM_ALG_SYMCIPHER:
1794     {
1795         TPMT_SYM_DEF_OBJECT* scheme;
1796         scheme = &publicArea->parameters.symDetail.sym;
1797         // Expect that the scheme is either valid symmetric cipher or NULL
1798         if(scheme->algorithm != TPM_ALG_NULL)
1799             macAlg = scheme->mode.sym;
1800         break;
1801     }
1802     default:
1803         return TPM_RC_TYPE;
1804     }
1805     // If the input value is not TPM_ALG_NULL ...
1806     if(*inMac != TPM_ALG_NULL)
1807     {
1808         // ... then either the scheme in the key must be TPM_ALG_NULL or the input
1809         // value must match
1810         if((macAlg != TPM_ALG_NULL) && (*inMac != macAlg))
1811             return TPM_RC_VALUE;
1812     }
1813     else
1814     {
1815         // Since the input value is TPM_ALG_NULL, then the key value can't be
1816         // TPM_ALG_NULL
1817         if(macAlg == TPM_ALG_NULL)
1818             return TPM_RC_VALUE;

```

```

1819     *inMac = macAlg;
1820 }
1821 if(!CryptMacIsValidForKey(publicArea->type, *inMac, FALSE))
1822     return TPM_RCS_SCHEME;
1823 return TPM_RC_SUCCESS;
1824 }
1825
1826 /*** CryptMacIsValidForKey()
1827 // Check to see if the key type is compatible with the mac type
1828 BOOL CryptMacIsValidForKey(TPM_ALG_ID keyType, TPM_ALG_ID macAlg, BOOL flag)
1829 {
1830     switch(keyType)
1831     {
1832         case TPM_ALG_KEYEDHASH:
1833             return CryptHashIsValidAlg(macAlg, flag);
1834             break;
1835         case TPM_ALG_SYMCIPHER:
1836             return CryptSmacIsValidAlg(macAlg, flag);
1837             break;
1838         default:
1839             break;
1840     }
1841     return FALSE;
1842 }
1843
1844 /*** CryptSmacIsValidAlg()
1845 // This function is used to test if an algorithm is a supported SMAC algorithm. It
1846 // needs to be updated as new algorithms are added.
1847 BOOL CryptSmacIsValidAlg(TPM_ALG_ID alg,
1848                          BOOL FLAG // IN: Indicates if TPM_ALG_NULL is valid
1849 )
1850 {
1851     switch(alg)
1852     {
1853 #if ALG_CMAC
1854         case TPM_ALG_CMAC:
1855             return TRUE;
1856             break;
1857 #endif
1858         case TPM_ALG_NULL:
1859             return FLAG;
1860             break;
1861         default:
1862             return FALSE;
1863     }
1864 }
1865
1866 /*** CryptSymModeIsValid()
1867 // Function checks to see if an algorithm ID is a valid, symmetric block cipher
1868 // mode for the TPM. If 'flag' is SET, then TPM_ALG_NULL is a valid mode.
1869 // not include the modes used for SMAC
1870 BOOL CryptSymModeIsValid(TPM_ALG_ID mode, BOOL flag)
1871 {
1872     switch(mode)
1873     {
1874 #if ALG_CTR
1875         case TPM_ALG_CTR:
1876 #endif // ALG_CTR
1877 #if ALG_OFB
1878         case TPM_ALG_OFB:
1879 #endif // ALG_OFB
1880 #if ALG_CBC
1881         case TPM_ALG_CBC:
1882 #endif // ALG_CBC
1883 #if ALG_CFB
1884         case TPM_ALG_CFB:

```

```

1885 #endif // ALG_CFB
1886 #if ALG_ECB
1887     case TPM_ALG_ECB:
1888 #endif // ALG_ECB
1889     return TRUE;
1890     case TPM_ALG_NULL:
1891     return flag;
1892     break;
1893     default:
1894     break;
1895 }
1896 return FALSE;
1897 }

```

7.152 /tpm/src/crypt/PrimeData.c

```

1  #include "Tpm.h"
2
3  // This table is the product of all of the primes up to 1000.
4  // Checking to see if there is a GCD between a prime candidate
5  // and this number will eliminate many prime candidates from
6  // consideration before running Miller-Rabin on the result.
7
8  const CRYPT_INT_BUF(smallprimecomp, 43 * RADIX BITS) s_CompositeOfSmallPrimes =
9      {44, 44, {0x2ED42696, 0x2BBFA177, 0x4820594F, 0xF73F4841, 0xBFAC313A, 0xCAC3EB81,
10                0xF6F26BF8, 0x7FAB5061, 0x59746FB7, 0xF71377F6, 0x3B19855B, 0xCBD03132,
11                0xBB92EF1B, 0x3AC3152C, 0xE87C8273, 0xC0AE0E69, 0x74A9E295, 0x448CCE86,
12                0x63CA1907, 0x8A0BF944, 0xF8CC3BE0, 0xC26F0AF5, 0xC501C02F, 0x6579441A,
13                0xD1099CDA, 0x6BC76A00, 0xC81A3228, 0xBFBA1AB25, 0x70FA3841, 0x51B3D076,
14                0xCC2359ED, 0xD9EE0769, 0x75E47AF0, 0xD45FF31E, 0x52CCE4F6, 0x04DBC891,
15                0x96658ED2, 0x1753EFE5, 0x3AE4A5A6, 0x8FD4A97F, 0x8B15E7EB, 0x0243C3E1,
16                0xE0F0C31D, 0x0000000B}};
17
18  const Crypt_Int* s_CompositeOfSmallPrimes =
19      (const Crypt_Int*)&s_CompositeOfSmallPrimes;
20
21  // This table contains a bit for each of the odd values between 1 and 2^16 + 1.
22  // This table allows fast checking of the primes in that range.
23  // Don't change the size of this table unless you are prepared to do redo
24  // IsPrimeInt().
25
26  const uint32_t      s_LastPrimeInTable = 65537;
27  const uint32_t      s_PrimeTableSize   = 4097;
28  const uint32_t      s_PrimesInTable    = 6542;
29  const unsigned char s_PrimeTable[] =
30      {0x0e, 0xcb, 0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x81, 0x32, 0x4c, 0x4a, 0x86, 0x0d,
31        0x82, 0x96, 0x21, 0xc9, 0x34, 0x04, 0x5a, 0x20, 0x61, 0x89, 0xa4, 0x44, 0x11,
32        0x86, 0x29, 0xd1, 0x82, 0x28, 0x4a, 0x30, 0x40, 0x42, 0x32, 0x21, 0x99, 0x34,
33        0x08, 0x4b, 0x06, 0x25, 0x42, 0x84, 0x48, 0x8a, 0x14, 0x05, 0x42, 0x30, 0x6c,
34        0x08, 0xb4, 0x40, 0x0b, 0xa0, 0x08, 0x51, 0x12, 0x28, 0x89, 0x04, 0x65, 0x98,
35        0x30, 0x4c, 0x80, 0x96, 0x44, 0x12, 0x80, 0x21, 0x42, 0x12, 0x41, 0xc9, 0x04,
36        0x21, 0xc0, 0x32, 0x2d, 0x98, 0x00, 0x00, 0x49, 0x04, 0x08, 0x81, 0x96, 0x68,
37        0x82, 0xb0, 0x25, 0x08, 0x22, 0x48, 0x89, 0xa2, 0x40, 0x59, 0x26, 0x04, 0x90,
38        0x06, 0x40, 0x43, 0x30, 0x44, 0x92, 0x00, 0x69, 0x10, 0x82, 0x08, 0x08, 0xa4,
39        0x0d, 0x41, 0x12, 0x60, 0xc0, 0x00, 0x24, 0xd2, 0x22, 0x61, 0x08, 0x84, 0x04,
40        0x1b, 0x82, 0x01, 0xd3, 0x10, 0x01, 0x02, 0xa0, 0x44, 0xc0, 0x22, 0x60, 0x91,
41        0x14, 0x0c, 0x40, 0xa6, 0x04, 0xd2, 0x94, 0x20, 0x09, 0x94, 0x20, 0x52, 0x00,
42        0x08, 0x10, 0xa2, 0x4c, 0x00, 0x82, 0x01, 0x51, 0x10, 0x08, 0x8b, 0xa4, 0x25,
43        0x9a, 0x30, 0x44, 0x81, 0x10, 0x4c, 0x03, 0x02, 0x25, 0x52, 0x80, 0x08, 0x49,
44        0x84, 0x20, 0x50, 0x32, 0x00, 0x18, 0xa2, 0x40, 0x11, 0x24, 0x28, 0x01, 0x84,
45        0x01, 0x01, 0xa0, 0x41, 0x0a, 0x12, 0x45, 0x00, 0x36, 0x08, 0x00, 0x26, 0x29,
46        0x83, 0x82, 0x61, 0xc0, 0x80, 0x04, 0x10, 0x10, 0x6d, 0x00, 0x22, 0x48, 0x58,
47        0x26, 0x0c, 0xc2, 0x08, 0x48, 0x89, 0x24, 0x20, 0x58, 0x20, 0x45, 0x88, 0x24,
48        0x00, 0x19, 0x02, 0x25, 0xc0, 0x10, 0x68, 0x08, 0x14, 0x01, 0xca, 0x32, 0x28,
49        0x80, 0x00, 0x04, 0x4b, 0x26, 0x00, 0x13, 0x90, 0x60, 0x82, 0x80, 0x25, 0xd0,

```


50 0x00, 0x01, 0x10, 0x32, 0x0c, 0x43, 0x86, 0x21, 0x11, 0x00, 0x08, 0x43, 0x24,
51 0x04, 0x48, 0x10, 0x0c, 0x90, 0x92, 0x00, 0x43, 0x20, 0x2d, 0x00, 0x06, 0x09,
52 0x88, 0x24, 0x40, 0xc0, 0x32, 0x09, 0x09, 0x82, 0x00, 0x53, 0x80, 0x08, 0x80,
53 0x96, 0x41, 0x81, 0x00, 0x40, 0x48, 0x10, 0x48, 0x08, 0x96, 0x48, 0x58, 0x20,
54 0x29, 0xc3, 0x80, 0x20, 0x02, 0x94, 0x60, 0x92, 0x00, 0x20, 0x81, 0x22, 0x44,
55 0x10, 0xa0, 0x05, 0x40, 0x90, 0x01, 0x49, 0x20, 0x04, 0x0a, 0x00, 0x24, 0x89,
56 0x34, 0x48, 0x13, 0x80, 0x2c, 0xc0, 0x82, 0x29, 0x00, 0x24, 0x45, 0x08, 0x00,
57 0x08, 0x98, 0x36, 0x04, 0x52, 0x84, 0x04, 0xd0, 0x04, 0x00, 0x8a, 0x90, 0x44,
58 0x82, 0x32, 0x65, 0x18, 0x90, 0x00, 0x0a, 0x02, 0x01, 0x40, 0x02, 0x28, 0x40,
59 0xa4, 0x04, 0x92, 0x30, 0x04, 0x11, 0x86, 0x08, 0x42, 0x00, 0x2c, 0x52, 0x04,
60 0x08, 0xc9, 0x84, 0x60, 0x48, 0x12, 0x09, 0x99, 0x24, 0x44, 0x00, 0x24, 0x00,
61 0x03, 0x14, 0x21, 0x00, 0x10, 0x01, 0x1a, 0x32, 0x05, 0x88, 0x20, 0x40, 0x40,
62 0x06, 0x09, 0xc3, 0x84, 0x40, 0x01, 0x30, 0x60, 0x18, 0x02, 0x68, 0x11, 0x90,
63 0x0c, 0x02, 0xa2, 0x04, 0x00, 0x86, 0x29, 0x89, 0x14, 0x24, 0x82, 0x02, 0x41,
64 0x08, 0x80, 0x04, 0x19, 0x80, 0x08, 0x10, 0x12, 0x68, 0x42, 0xa4, 0x04, 0x00,
65 0x02, 0x61, 0x10, 0x06, 0x0c, 0x10, 0x00, 0x01, 0x12, 0x10, 0x20, 0x03, 0x94,
66 0x21, 0x42, 0x12, 0x65, 0x18, 0x94, 0x0c, 0x0a, 0x04, 0x28, 0x01, 0x14, 0x29,
67 0x0a, 0xa4, 0x40, 0xd0, 0x00, 0x40, 0x01, 0x90, 0x04, 0x41, 0x20, 0x2d, 0x40,
68 0x82, 0x48, 0xc1, 0x20, 0x00, 0x10, 0x30, 0x01, 0x08, 0x24, 0x04, 0x59, 0x84,
69 0x24, 0x00, 0x02, 0x29, 0x82, 0x00, 0x61, 0x58, 0x02, 0x48, 0x81, 0x16, 0x48,
70 0x10, 0x00, 0x21, 0x11, 0x06, 0x00, 0xca, 0xa0, 0x40, 0x02, 0x00, 0x04, 0x91,
71 0xb0, 0x00, 0x42, 0x04, 0x0c, 0x81, 0x06, 0x09, 0x48, 0x14, 0x25, 0x92, 0x20,
72 0x25, 0x11, 0xa0, 0x00, 0x0a, 0x86, 0x0c, 0xc1, 0x02, 0x48, 0x00, 0x20, 0x45,
73 0x08, 0x32, 0x00, 0x98, 0x06, 0x04, 0x13, 0x22, 0x00, 0x82, 0x04, 0x48, 0x81,
74 0x14, 0x44, 0x82, 0x12, 0x24, 0x18, 0x10, 0x40, 0x43, 0x80, 0x28, 0xd0, 0x04,
75 0x20, 0x81, 0x24, 0x64, 0xd8, 0x00, 0x2c, 0x09, 0x12, 0x08, 0x41, 0xa2, 0x00,
76 0x00, 0x02, 0x41, 0xca, 0x20, 0x41, 0xc0, 0x10, 0x01, 0x18, 0xa4, 0x04, 0x18,
77 0xa4, 0x20, 0x12, 0x94, 0x20, 0x83, 0xa0, 0x40, 0x02, 0x32, 0x44, 0x80, 0x04,
78 0x00, 0x18, 0x00, 0x0c, 0x40, 0x86, 0x60, 0x8a, 0x00, 0x64, 0x88, 0x12, 0x05,
79 0x01, 0x82, 0x00, 0x4a, 0xa2, 0x01, 0xc1, 0x10, 0x61, 0x09, 0x04, 0x01, 0x88,
80 0x00, 0x60, 0x01, 0xb4, 0x40, 0x08, 0x06, 0x01, 0x03, 0x80, 0x08, 0x40, 0x94,
81 0x04, 0x8a, 0x20, 0x29, 0x80, 0x02, 0x0c, 0x52, 0x02, 0x01, 0x42, 0x84, 0x00,
82 0x80, 0x84, 0x64, 0x02, 0x32, 0x48, 0x00, 0x30, 0x44, 0x40, 0x22, 0x21, 0x00,
83 0x02, 0x08, 0xc3, 0xa0, 0x04, 0xd0, 0x20, 0x40, 0x18, 0x16, 0x40, 0x40, 0x00,
84 0x28, 0x52, 0x90, 0x08, 0x82, 0x14, 0x01, 0x18, 0x10, 0x08, 0x09, 0x82, 0x40,
85 0x0a, 0xa0, 0x20, 0x93, 0x80, 0x08, 0xc0, 0x00, 0x20, 0x52, 0x00, 0x05, 0x01,
86 0x10, 0x40, 0x11, 0x06, 0x0c, 0x82, 0x00, 0x00, 0x4b, 0x90, 0x44, 0x9a, 0x00,
87 0x28, 0x80, 0x90, 0x04, 0x4a, 0x06, 0x09, 0x43, 0x02, 0x28, 0x00, 0x34, 0x01,
88 0x18, 0x00, 0x65, 0x09, 0x80, 0x44, 0x03, 0x00, 0x24, 0x02, 0x82, 0x61, 0x48,
89 0x14, 0x41, 0x00, 0x12, 0x28, 0x00, 0x34, 0x08, 0x51, 0x04, 0x05, 0x12, 0x90,
90 0x28, 0x89, 0x84, 0x60, 0x12, 0x10, 0x49, 0x10, 0x26, 0x40, 0x49, 0x82, 0x00,
91 0x91, 0x10, 0x01, 0x0a, 0x24, 0x40, 0x88, 0x10, 0x4c, 0x10, 0x04, 0x00, 0x50,
92 0xa2, 0x2c, 0x40, 0x90, 0x48, 0x0a, 0xb0, 0x01, 0x50, 0x12, 0x08, 0x00, 0xa4,
93 0x04, 0x09, 0xa0, 0x28, 0x92, 0x02, 0x00, 0x43, 0x10, 0x21, 0x02, 0x20, 0x41,
94 0x81, 0x32, 0x00, 0x08, 0x04, 0x0c, 0x52, 0x00, 0x21, 0x49, 0x84, 0x20, 0x10,
95 0x02, 0x01, 0x81, 0x10, 0x48, 0x40, 0x22, 0x01, 0x01, 0x84, 0x69, 0xc1, 0x30,
96 0x01, 0xc8, 0x02, 0x44, 0x88, 0x00, 0x0c, 0x01, 0x02, 0x2d, 0xc0, 0x12, 0x61,
97 0x00, 0xa0, 0x00, 0xc0, 0x30, 0x40, 0x01, 0x12, 0x08, 0x0b, 0x20, 0x00, 0x80,
98 0x94, 0x40, 0x01, 0x84, 0x40, 0x00, 0x32, 0x00, 0x10, 0x84, 0x00, 0x0b, 0x24,
99 0x00, 0x01, 0x06, 0x29, 0x8a, 0x84, 0x41, 0x80, 0x10, 0x08, 0x08, 0x94, 0x4c,
100 0x03, 0x80, 0x01, 0x40, 0x96, 0x40, 0x41, 0x20, 0x20, 0x50, 0x22, 0x25, 0x89,
101 0xa2, 0x40, 0x40, 0xa4, 0x20, 0x02, 0x86, 0x28, 0x01, 0x20, 0x21, 0x4a, 0x10,
102 0x08, 0x00, 0x14, 0x08, 0x40, 0x04, 0x25, 0x42, 0x02, 0x21, 0x43, 0x10, 0x04,
103 0x92, 0x00, 0x21, 0x11, 0xa0, 0x4c, 0x18, 0x22, 0x09, 0x03, 0x84, 0x41, 0x89,
104 0x10, 0x04, 0x82, 0x22, 0x24, 0x01, 0x14, 0x08, 0x08, 0x84, 0x08, 0xc1, 0x00,
105 0x09, 0x42, 0xb0, 0x41, 0x8a, 0x02, 0x00, 0x80, 0x36, 0x04, 0x49, 0xa0, 0x24,
106 0x91, 0x00, 0x00, 0x02, 0x94, 0x41, 0x92, 0x02, 0x01, 0x08, 0x06, 0x08, 0x09,
107 0x00, 0x01, 0xd0, 0x16, 0x28, 0x89, 0x80, 0x60, 0x00, 0x00, 0x68, 0x01, 0x90,
108 0x0c, 0x50, 0x20, 0x01, 0x40, 0x80, 0x40, 0x42, 0x30, 0x41, 0x00, 0x20, 0x25,
109 0x81, 0x06, 0x40, 0x49, 0x00, 0x08, 0x01, 0x12, 0x49, 0x00, 0xa0, 0x20, 0x18,
110 0x30, 0x05, 0x01, 0xa6, 0x00, 0x10, 0x24, 0x28, 0x00, 0x02, 0x20, 0xc8, 0x20,
111 0x00, 0x88, 0x12, 0x0c, 0x90, 0x92, 0x00, 0x02, 0x26, 0x01, 0x42, 0x16, 0x49,
112 0x00, 0x04, 0x24, 0x42, 0x02, 0x01, 0x88, 0x80, 0x0c, 0x1a, 0x80, 0x08, 0x10,
113 0x00, 0x60, 0x02, 0x94, 0x44, 0x88, 0x00, 0x69, 0x11, 0x30, 0x08, 0x12, 0xa0,
114 0x24, 0x13, 0x84, 0x00, 0x82, 0x00, 0x65, 0xc0, 0x10, 0x28, 0x00, 0x30, 0x04,
115 0x03, 0x20, 0x01, 0x11, 0x06, 0x01, 0xc8, 0x80, 0x00, 0xc2, 0x20, 0x08, 0x10,

116 0x82, 0x0c, 0x13, 0x02, 0x0c, 0x52, 0x06, 0x40, 0x00, 0xb0, 0x61, 0x40, 0x10,
117 0x01, 0x98, 0x86, 0x04, 0x10, 0x84, 0x08, 0x92, 0x14, 0x60, 0x41, 0x80, 0x41,
118 0x1a, 0x10, 0x04, 0x81, 0x22, 0x40, 0x41, 0x20, 0x29, 0x52, 0x00, 0x41, 0x08,
119 0x34, 0x60, 0x10, 0x00, 0x28, 0x01, 0x10, 0x40, 0x00, 0x84, 0x08, 0x42, 0x90,
120 0x20, 0x48, 0x04, 0x04, 0x52, 0x02, 0x00, 0x08, 0x20, 0x04, 0x00, 0x82, 0x0d,
121 0x00, 0x82, 0x40, 0x02, 0x10, 0x05, 0x48, 0x20, 0x40, 0x99, 0x00, 0x00, 0x01,
122 0x06, 0x24, 0xc0, 0x00, 0x68, 0x82, 0x04, 0x21, 0x12, 0x10, 0x44, 0x08, 0x04,
123 0x00, 0x40, 0xa6, 0x20, 0xd0, 0x16, 0x09, 0xc9, 0x24, 0x41, 0x02, 0x20, 0x0c,
124 0x09, 0x92, 0x40, 0x12, 0x00, 0x00, 0x40, 0x00, 0x09, 0x43, 0x84, 0x20, 0x98,
125 0x02, 0x01, 0x11, 0x24, 0x00, 0x43, 0x24, 0x00, 0x03, 0x90, 0x08, 0x41, 0x30,
126 0x24, 0x58, 0x20, 0x4c, 0x80, 0x82, 0x08, 0x10, 0x24, 0x25, 0x81, 0x06, 0x41,
127 0x09, 0x10, 0x20, 0x18, 0x10, 0x44, 0x80, 0x10, 0x00, 0x4a, 0x24, 0x0d, 0x01,
128 0x94, 0x28, 0x80, 0x30, 0x00, 0xc0, 0x02, 0x60, 0x10, 0x84, 0x0c, 0x02, 0x00,
129 0x09, 0x02, 0x82, 0x01, 0x08, 0x10, 0x04, 0xc2, 0x20, 0x68, 0x09, 0x06, 0x04,
130 0x18, 0x00, 0x00, 0x11, 0x90, 0x08, 0x0b, 0x10, 0x21, 0x82, 0x02, 0x0c, 0x10,
131 0xb6, 0x08, 0x00, 0x26, 0x00, 0x41, 0x02, 0x01, 0x4a, 0x24, 0x21, 0x1a, 0x20,
132 0x24, 0x80, 0x00, 0x44, 0x02, 0x00, 0x2d, 0x40, 0x02, 0x00, 0x8b, 0x94, 0x20,
133 0x10, 0x00, 0x20, 0x90, 0xa6, 0x40, 0x13, 0x00, 0x2c, 0x11, 0x86, 0x61, 0x01,
134 0x80, 0x41, 0x10, 0x02, 0x04, 0x81, 0x30, 0x48, 0x48, 0x20, 0x28, 0x50, 0x80,
135 0x21, 0x8a, 0x10, 0x04, 0x08, 0x10, 0x09, 0x10, 0x10, 0x48, 0x42, 0xa0, 0x0c,
136 0x82, 0x92, 0x60, 0xc0, 0x20, 0x05, 0xd2, 0x20, 0x40, 0x01, 0x00, 0x04, 0x08,
137 0x82, 0x2d, 0x82, 0x02, 0x00, 0x48, 0x80, 0x41, 0x48, 0x10, 0x00, 0x91, 0x04,
138 0x04, 0x03, 0x84, 0x00, 0xc2, 0x04, 0x68, 0x00, 0x00, 0x64, 0xc0, 0x22, 0x40,
139 0x08, 0x32, 0x44, 0x09, 0x86, 0x00, 0x91, 0x02, 0x28, 0x01, 0x00, 0x64, 0x48,
140 0x00, 0x24, 0x10, 0x90, 0x00, 0x43, 0x00, 0x21, 0x52, 0x86, 0x41, 0x8b, 0x90,
141 0x20, 0x40, 0x20, 0x08, 0x88, 0x04, 0x44, 0x13, 0x20, 0x00, 0x02, 0x84, 0x60,
142 0x81, 0x90, 0x24, 0x40, 0x30, 0x00, 0x08, 0x10, 0x08, 0x08, 0x02, 0x01, 0x10,
143 0x04, 0x20, 0x43, 0xb4, 0x40, 0x90, 0x12, 0x68, 0x01, 0x80, 0x4c, 0x18, 0x00,
144 0x08, 0xc0, 0x12, 0x49, 0x40, 0x10, 0x24, 0x1a, 0x00, 0x41, 0x89, 0x24, 0x4c,
145 0x10, 0x00, 0x04, 0x52, 0x10, 0x09, 0x4a, 0x20, 0x41, 0x48, 0x22, 0x69, 0x11,
146 0x14, 0x08, 0x10, 0x06, 0x24, 0x80, 0x84, 0x28, 0x00, 0x10, 0x00, 0x40, 0x10,
147 0x01, 0x08, 0x26, 0x08, 0x48, 0x06, 0x28, 0x00, 0x14, 0x01, 0x42, 0x84, 0x04,
148 0x0a, 0x20, 0x00, 0x01, 0x82, 0x08, 0x00, 0x82, 0x24, 0x12, 0x04, 0x40, 0x40,
149 0xa0, 0x40, 0x90, 0x10, 0x04, 0x90, 0x22, 0x40, 0x10, 0x20, 0x2c, 0x80, 0x10,
150 0x28, 0x43, 0x00, 0x04, 0x58, 0x00, 0x01, 0x81, 0x10, 0x48, 0x09, 0x20, 0x21,
151 0x83, 0x04, 0x00, 0x42, 0xa4, 0x44, 0x00, 0x00, 0x6c, 0x10, 0xa0, 0x44, 0x48,
152 0x80, 0x00, 0x83, 0x80, 0x48, 0xc9, 0x00, 0x00, 0x02, 0x05, 0x10, 0xb0,
153 0x04, 0x13, 0x04, 0x29, 0x10, 0x92, 0x40, 0x08, 0x04, 0x44, 0x82, 0x22, 0x00,
154 0x19, 0x20, 0x00, 0x19, 0x20, 0x01, 0x81, 0x90, 0x60, 0x8a, 0x00, 0x41, 0xc0,
155 0x02, 0x45, 0x10, 0x04, 0x00, 0x02, 0xa2, 0x09, 0x40, 0x10, 0x21, 0x49, 0x20,
156 0x01, 0x42, 0x30, 0x2c, 0x00, 0x14, 0x44, 0x01, 0x22, 0x04, 0x02, 0x92, 0x08,
157 0x89, 0x04, 0x21, 0x80, 0x10, 0x05, 0x01, 0x20, 0x40, 0x41, 0x80, 0x04, 0x00,
158 0x12, 0x09, 0x40, 0xb0, 0x64, 0x58, 0x32, 0x01, 0x08, 0x90, 0x00, 0x41, 0x04,
159 0x09, 0xc1, 0x80, 0x61, 0x08, 0x90, 0x00, 0x9a, 0x00, 0x24, 0x01, 0x12, 0x08,
160 0x02, 0x26, 0x05, 0x82, 0x06, 0x08, 0x08, 0x00, 0x20, 0x48, 0x20, 0x00, 0x18,
161 0x24, 0x48, 0x03, 0x02, 0x00, 0x11, 0x00, 0x09, 0x00, 0x84, 0x01, 0x4a, 0x10,
162 0x01, 0x98, 0x00, 0x04, 0x18, 0x86, 0x00, 0xc0, 0x00, 0x20, 0x81, 0x80, 0x04,
163 0x10, 0x30, 0x05, 0x00, 0xb4, 0x0c, 0x4a, 0x82, 0x29, 0x91, 0x02, 0x28, 0x00,
164 0x20, 0x44, 0xc0, 0x00, 0x2c, 0x91, 0x80, 0x40, 0x01, 0xa2, 0x00, 0x12, 0x04,
165 0x09, 0xc3, 0x20, 0x00, 0x08, 0x02, 0x0c, 0x10, 0x22, 0x04, 0x00, 0x00, 0x2c,
166 0x11, 0x86, 0x00, 0xc0, 0x00, 0x00, 0x12, 0x32, 0x40, 0x89, 0x80, 0x40, 0x40,
167 0x02, 0x05, 0x50, 0x86, 0x60, 0x82, 0xa4, 0x60, 0x0a, 0x12, 0x4d, 0x80, 0x90,
168 0x08, 0x12, 0x80, 0x09, 0x02, 0x14, 0x48, 0x01, 0x24, 0x20, 0x8a, 0x00, 0x44,
169 0x90, 0x04, 0x04, 0x01, 0x02, 0x00, 0xd1, 0x12, 0x00, 0x0a, 0x04, 0x40, 0x00,
170 0x32, 0x21, 0x81, 0x24, 0x08, 0x19, 0x84, 0x20, 0x02, 0x04, 0x08, 0x89, 0x80,
171 0x24, 0x02, 0x02, 0x68, 0x18, 0x82, 0x44, 0x42, 0x00, 0x21, 0x40, 0x00, 0x28,
172 0x01, 0x80, 0x45, 0x82, 0x20, 0x40, 0x11, 0x80, 0x0c, 0x02, 0x00, 0x24, 0x40,
173 0x90, 0x01, 0x40, 0x20, 0x20, 0x50, 0x20, 0x28, 0x19, 0x00, 0x40, 0x09, 0x20,
174 0x08, 0x80, 0x04, 0x60, 0x40, 0x80, 0x20, 0x08, 0x30, 0x49, 0x09, 0x34, 0x00,
175 0x11, 0x24, 0x24, 0x82, 0x00, 0x41, 0xc2, 0x00, 0x04, 0x92, 0x02, 0x24, 0x80,
176 0x00, 0x0c, 0x02, 0xa0, 0x00, 0x01, 0x06, 0x60, 0x41, 0x04, 0x21, 0xd0, 0x00,
177 0x01, 0x01, 0x00, 0x48, 0x12, 0x84, 0x04, 0x91, 0x12, 0x08, 0x00, 0x24, 0x44,
178 0x00, 0x12, 0x41, 0x18, 0x26, 0x0c, 0x41, 0x80, 0x00, 0x52, 0x04, 0x20, 0x09,
179 0x00, 0x24, 0x90, 0x20, 0x48, 0x18, 0x02, 0x00, 0x03, 0xa2, 0x09, 0xd0, 0x14,
180 0x00, 0x8a, 0x84, 0x25, 0x4a, 0x00, 0x20, 0x98, 0x14, 0x40, 0x00, 0xa2, 0x05,
181 0x00, 0x00, 0x00, 0x40, 0x14, 0x01, 0x58, 0x20, 0x2c, 0x80, 0x84, 0x00, 0x09,

182 0x20, 0x20, 0x91, 0x02, 0x08, 0x02, 0x02, 0xb0, 0x41, 0x08, 0x30, 0x00, 0x09, 0x10,
183 0x00, 0x18, 0x02, 0x21, 0x02, 0x02, 0x00, 0x00, 0x24, 0x44, 0x08, 0x12, 0x60,
184 0x00, 0xb2, 0x44, 0x12, 0x02, 0x0c, 0xc0, 0x80, 0x40, 0xc8, 0x20, 0x04, 0x50,
185 0x20, 0x05, 0x00, 0xb0, 0x04, 0x0b, 0x04, 0x29, 0x53, 0x00, 0x61, 0x48, 0x30,
186 0x00, 0x82, 0x20, 0x29, 0x00, 0x16, 0x00, 0x53, 0x22, 0x20, 0x43, 0x10, 0x48,
187 0x00, 0x80, 0x04, 0xd2, 0x00, 0x40, 0x00, 0xa2, 0x44, 0x03, 0x80, 0x29, 0x00,
188 0x04, 0x08, 0xc0, 0x04, 0x64, 0x40, 0x30, 0x28, 0x09, 0x84, 0x44, 0x50, 0x80,
189 0x21, 0x02, 0x92, 0x00, 0xc0, 0x10, 0x60, 0x88, 0x22, 0x08, 0x80, 0x00, 0x00,
190 0x18, 0x84, 0x04, 0x83, 0x96, 0x00, 0x81, 0x20, 0x05, 0x02, 0x00, 0x45, 0x88,
191 0x84, 0x00, 0x51, 0x20, 0x20, 0x51, 0x86, 0x41, 0x4b, 0x94, 0x00, 0x80, 0x00,
192 0x08, 0x11, 0x20, 0x4c, 0x58, 0x80, 0x04, 0x03, 0x06, 0x20, 0x89, 0x00, 0x05,
193 0x08, 0x22, 0x05, 0x90, 0x00, 0x40, 0x00, 0x82, 0x09, 0x50, 0x00, 0x00, 0x00,
194 0xa0, 0x41, 0xc2, 0x20, 0x08, 0x00, 0x16, 0x08, 0x40, 0x26, 0x21, 0xd0, 0x90,
195 0x08, 0x81, 0x90, 0x41, 0x00, 0x02, 0x44, 0x08, 0x10, 0x0c, 0x0a, 0x86, 0x09,
196 0x90, 0x04, 0x00, 0xc8, 0xa0, 0x04, 0x08, 0x30, 0x20, 0x89, 0x84, 0x00, 0x11,
197 0x22, 0x2c, 0x40, 0x00, 0x08, 0x02, 0xb0, 0x01, 0x48, 0x02, 0x01, 0x09, 0x20,
198 0x04, 0x03, 0x04, 0x00, 0x80, 0x02, 0x60, 0x42, 0x30, 0x21, 0x4a, 0x10, 0x44,
199 0x09, 0x02, 0x00, 0x01, 0x24, 0x00, 0x12, 0x82, 0x21, 0x80, 0xa4, 0x20, 0x10,
200 0x02, 0x04, 0x91, 0xa0, 0x40, 0x18, 0x04, 0x00, 0x02, 0x06, 0x69, 0x09, 0x00,
201 0x05, 0x58, 0x02, 0x01, 0x00, 0x00, 0x48, 0x00, 0x00, 0x00, 0x03, 0x92, 0x20,
202 0x00, 0x34, 0x01, 0xc8, 0x20, 0x48, 0x08, 0x30, 0x08, 0x42, 0x80, 0x20, 0x91,
203 0x90, 0x68, 0x01, 0x04, 0x40, 0x12, 0x02, 0x61, 0x00, 0x12, 0x08, 0x01, 0xa0,
204 0x00, 0x11, 0x04, 0x21, 0x48, 0x04, 0x24, 0x92, 0x00, 0x0c, 0x01, 0x84, 0x04,
205 0x00, 0x00, 0x01, 0x12, 0x96, 0x40, 0x01, 0xa0, 0x41, 0x88, 0x22, 0x28, 0x88,
206 0x00, 0x44, 0x42, 0x80, 0x24, 0x12, 0x14, 0x01, 0x42, 0x90, 0x60, 0x1a, 0x10,
207 0x04, 0x81, 0x10, 0x48, 0x08, 0x06, 0x29, 0x83, 0x02, 0x40, 0x02, 0x24, 0x64,
208 0x80, 0x10, 0x05, 0x80, 0x10, 0x40, 0x02, 0x02, 0x08, 0x42, 0x84, 0x01, 0x09,
209 0x20, 0x04, 0x50, 0x00, 0x60, 0x11, 0x30, 0x40, 0x13, 0x02, 0x04, 0x81, 0x00,
210 0x09, 0x08, 0x20, 0x45, 0x4a, 0x10, 0x61, 0x90, 0x26, 0x0c, 0x08, 0x02, 0x21,
211 0x91, 0x00, 0x60, 0x02, 0x04, 0x00, 0x02, 0x00, 0x0c, 0x08, 0x06, 0x08, 0x48,
212 0x84, 0x08, 0x11, 0x02, 0x00, 0x80, 0xa4, 0x00, 0x5a, 0x20, 0x00, 0x88, 0x04,
213 0x04, 0x02, 0x00, 0x09, 0x00, 0x14, 0x08, 0x49, 0x14, 0x20, 0xc8, 0x00, 0x04,
214 0x91, 0xa0, 0x40, 0x59, 0x80, 0x00, 0x12, 0x10, 0x00, 0x80, 0x80, 0x65, 0x00,
215 0x00, 0x04, 0x00, 0x80, 0x40, 0x19, 0x00, 0x21, 0x03, 0x84, 0x60, 0xc0, 0x04,
216 0x24, 0x1a, 0x12, 0x61, 0x80, 0x80, 0x08, 0x02, 0x04, 0x09, 0x42, 0x12, 0x20,
217 0x08, 0x34, 0x04, 0x90, 0x20, 0x01, 0x01, 0xa0, 0x00, 0x0b, 0x00, 0x08, 0x91,
218 0x92, 0x40, 0x02, 0x34, 0x40, 0x88, 0x10, 0x61, 0x19, 0x02, 0x00, 0x40, 0x04,
219 0x25, 0xc0, 0x80, 0x68, 0x08, 0x04, 0x21, 0x80, 0x22, 0x04, 0x00, 0xa0, 0x0c,
220 0x01, 0x84, 0x20, 0x41, 0x00, 0x08, 0x8a, 0x00, 0x20, 0x8a, 0x00, 0x48, 0x88,
221 0x04, 0x04, 0x11, 0x82, 0x08, 0x40, 0x86, 0x09, 0x49, 0xa4, 0x40, 0x00, 0x10,
222 0x01, 0x01, 0xa2, 0x04, 0x50, 0x80, 0x0c, 0x80, 0x00, 0x48, 0x82, 0xa0, 0x01,
223 0x18, 0x12, 0x41, 0x01, 0x04, 0x48, 0x41, 0x00, 0x24, 0x01, 0x00, 0x00, 0x88,
224 0x14, 0x00, 0x02, 0x00, 0x68, 0x01, 0x20, 0x08, 0x4a, 0x22, 0x08, 0x83, 0x80,
225 0x00, 0x89, 0x04, 0x01, 0xc2, 0x00, 0x00, 0x00, 0x34, 0x04, 0x00, 0x82, 0x28,
226 0x02, 0x02, 0x41, 0x4a, 0x90, 0x05, 0x82, 0x02, 0x09, 0x80, 0x24, 0x04, 0x41,
227 0x00, 0x01, 0x92, 0x80, 0x28, 0x01, 0x14, 0x00, 0x50, 0x20, 0x4c, 0x10, 0xb0,
228 0x04, 0x43, 0xa4, 0x21, 0x90, 0x04, 0x01, 0x02, 0x00, 0x44, 0x48, 0x00, 0x64,
229 0x08, 0x06, 0x00, 0x42, 0x20, 0x08, 0x02, 0x92, 0x01, 0x4a, 0x00, 0x20, 0x50,
230 0x32, 0x25, 0x90, 0x22, 0x04, 0x09, 0x00, 0x08, 0x11, 0x80, 0x21, 0x01, 0x10,
231 0x05, 0x00, 0x32, 0x08, 0x88, 0x94, 0x08, 0x08, 0x24, 0x0d, 0xc1, 0x80, 0x40,
232 0x0b, 0x20, 0x40, 0x18, 0x12, 0x04, 0x00, 0x22, 0x40, 0x10, 0x26, 0x05, 0xc1,
233 0x82, 0x00, 0x01, 0x30, 0x24, 0x02, 0x22, 0x41, 0x08, 0x24, 0x48, 0x1a, 0x00,
234 0x25, 0xd2, 0x12, 0x28, 0x42, 0x00, 0x04, 0x40, 0x30, 0x41, 0x00, 0x02, 0x00,
235 0x13, 0x20, 0x24, 0xd1, 0x84, 0x08, 0x89, 0x80, 0x04, 0x52, 0x00, 0x44, 0x18,
236 0xa4, 0x00, 0x00, 0x06, 0x20, 0x91, 0x10, 0x09, 0x42, 0x20, 0x24, 0x40, 0x30,
237 0x28, 0x00, 0x84, 0x40, 0x40, 0x80, 0x08, 0x10, 0x04, 0x09, 0x08, 0x04, 0x40,
238 0x08, 0x22, 0x00, 0x19, 0x02, 0x00, 0x00, 0x80, 0x2c, 0x02, 0x02, 0x21, 0x01,
239 0x90, 0x20, 0x40, 0x00, 0x0c, 0x00, 0x34, 0x48, 0x58, 0x20, 0x01, 0x43, 0x04,
240 0x20, 0x80, 0x14, 0x00, 0x90, 0x00, 0x6d, 0x11, 0x00, 0x00, 0x40, 0x20, 0x00,
241 0x03, 0x10, 0x40, 0x88, 0x30, 0x05, 0x4a, 0x00, 0x65, 0x10, 0x24, 0x08, 0x18,
242 0x84, 0x28, 0x03, 0x80, 0x20, 0x42, 0xb0, 0x40, 0x00, 0x10, 0x69, 0x19, 0x04,
243 0x00, 0x00, 0x80, 0x04, 0xc2, 0x04, 0x00, 0x01, 0x00, 0x05, 0x00, 0x22, 0x25,
244 0x08, 0x96, 0x04, 0x02, 0x22, 0x00, 0xd0, 0x10, 0x29, 0x01, 0xa0, 0x60, 0x08,
245 0x10, 0x04, 0x01, 0x16, 0x44, 0x10, 0x02, 0x28, 0x02, 0x82, 0x48, 0x40, 0x84,
246 0x20, 0x90, 0x22, 0x28, 0x80, 0x04, 0x00, 0x40, 0x04, 0x24, 0x00, 0x80, 0x29,
247 0x03, 0x10, 0x60, 0x48, 0x00, 0x00, 0x81, 0xa0, 0x00, 0x51, 0x20, 0x0c, 0xd1,

248 0x00, 0x01, 0x41, 0x20, 0x04, 0x92, 0x00, 0x00, 0x10, 0x92, 0x00, 0x42, 0x04,
249 0x05, 0x01, 0x86, 0x40, 0x80, 0x10, 0x20, 0x52, 0x20, 0x21, 0x00, 0x10, 0x48,
250 0x0a, 0x02, 0x00, 0xd0, 0x12, 0x41, 0x48, 0x80, 0x04, 0x00, 0x00, 0x48, 0x09,
251 0x22, 0x04, 0x00, 0x24, 0x00, 0x43, 0x10, 0x60, 0x0a, 0x00, 0x44, 0x12, 0x20,
252 0x2c, 0x08, 0x20, 0x44, 0x00, 0x84, 0x09, 0x40, 0x06, 0x08, 0xc1, 0x00, 0x40,
253 0x80, 0x20, 0x00, 0x98, 0x12, 0x48, 0x10, 0xa2, 0x20, 0x00, 0x84, 0x48, 0xc0,
254 0x10, 0x20, 0x90, 0x12, 0x08, 0x98, 0x82, 0x00, 0x0a, 0xa0, 0x04, 0x03, 0x00,
255 0x28, 0xc3, 0x00, 0x44, 0x42, 0x10, 0x04, 0x08, 0x04, 0x40, 0x00, 0x00, 0x05,
256 0x10, 0x00, 0x21, 0x03, 0x80, 0x04, 0x88, 0x12, 0x69, 0x10, 0x00, 0x04, 0x08,
257 0x04, 0x04, 0x02, 0x84, 0x48, 0x49, 0x04, 0x20, 0x18, 0x02, 0x64, 0x80, 0x30,
258 0x08, 0x01, 0x02, 0x00, 0x52, 0x12, 0x49, 0x08, 0x20, 0x41, 0x88, 0x10, 0x48,
259 0x08, 0x34, 0x00, 0x01, 0x86, 0x05, 0xd0, 0x00, 0x00, 0x83, 0x84, 0x21, 0x40,
260 0x02, 0x41, 0x10, 0x80, 0x48, 0x40, 0xa2, 0x20, 0x51, 0x00, 0x00, 0x49, 0x00,
261 0x01, 0x90, 0x20, 0x40, 0x18, 0x02, 0x40, 0x02, 0x22, 0x05, 0x40, 0x80, 0x08,
262 0x82, 0x10, 0x20, 0x18, 0x00, 0x05, 0x01, 0x82, 0x40, 0x58, 0x00, 0x04, 0x81,
263 0x90, 0x29, 0x01, 0xa0, 0x64, 0x00, 0x22, 0x40, 0x01, 0xa2, 0x00, 0x18, 0x04,
264 0x0d, 0x00, 0x00, 0x60, 0x80, 0x94, 0x60, 0x82, 0x10, 0x0d, 0x80, 0x30, 0x0c,
265 0x12, 0x20, 0x00, 0x00, 0x12, 0x40, 0xc0, 0x20, 0x21, 0x58, 0x02, 0x41, 0x10,
266 0x80, 0x44, 0x03, 0x02, 0x04, 0x13, 0x90, 0x29, 0x08, 0x00, 0x44, 0xc0, 0x00,
267 0x21, 0x00, 0x26, 0x00, 0x1a, 0x80, 0x01, 0x13, 0x14, 0x20, 0x0a, 0x14, 0x20,
268 0x00, 0x32, 0x61, 0x08, 0x00, 0x40, 0x42, 0x20, 0x09, 0x80, 0x06, 0x01, 0x81,
269 0x80, 0x60, 0x42, 0x00, 0x68, 0x90, 0x82, 0x08, 0x42, 0x80, 0x04, 0x02, 0x80,
270 0x09, 0x0b, 0x04, 0x00, 0x98, 0x00, 0x0c, 0x81, 0x06, 0x44, 0x48, 0x84, 0x28,
271 0x03, 0x92, 0x00, 0x01, 0x80, 0x40, 0x0a, 0x00, 0x0c, 0x81, 0x02, 0x08, 0x51,
272 0x04, 0x28, 0x90, 0x02, 0x20, 0x09, 0x10, 0x60, 0x00, 0x00, 0x09, 0x81, 0xa0,
273 0x0c, 0x00, 0xa4, 0x09, 0x00, 0x02, 0x28, 0x80, 0x20, 0x00, 0x02, 0x02, 0x04,
274 0x81, 0x14, 0x04, 0x00, 0x04, 0x09, 0x11, 0x12, 0x60, 0x40, 0x20, 0x01, 0x48,
275 0x30, 0x40, 0x11, 0x00, 0x08, 0x0a, 0x86, 0x00, 0x00, 0x04, 0x60, 0x81, 0x04,
276 0x01, 0xd0, 0x02, 0x41, 0x18, 0x90, 0x00, 0x0a, 0x20, 0x00, 0xc1, 0x06, 0x01,
277 0x08, 0x80, 0x64, 0xca, 0x10, 0x04, 0x99, 0x80, 0x48, 0x01, 0x82, 0x20, 0x50,
278 0x90, 0x48, 0x80, 0x84, 0x20, 0x90, 0x22, 0x00, 0x19, 0x00, 0x04, 0x18, 0x20,
279 0x24, 0x10, 0x86, 0x40, 0xc2, 0x00, 0x24, 0x12, 0x10, 0x44, 0x00, 0x16, 0x08,
280 0x10, 0x24, 0x00, 0x12, 0x06, 0x01, 0x08, 0x90, 0x00, 0x12, 0x02, 0x4d, 0x10,
281 0x80, 0x40, 0x50, 0x22, 0x00, 0x43, 0x10, 0x01, 0x00, 0x30, 0x21, 0x0a, 0x00,
282 0x00, 0x01, 0x14, 0x00, 0x10, 0x84, 0x04, 0xc1, 0x10, 0x29, 0x0a, 0x00, 0x01,
283 0x8a, 0x00, 0x20, 0x01, 0x12, 0x0c, 0x49, 0x20, 0x04, 0x81, 0x00, 0x48, 0x01,
284 0x04, 0x60, 0x80, 0x12, 0x0c, 0x08, 0x10, 0x48, 0x4a, 0x04, 0x28, 0x10, 0x00,
285 0x28, 0x40, 0x84, 0x45, 0x50, 0x10, 0x60, 0x10, 0x06, 0x44, 0x01, 0x80, 0x09,
286 0x00, 0x86, 0x01, 0x42, 0xa0, 0x00, 0x90, 0x00, 0x05, 0x90, 0x22, 0x40, 0x41,
287 0x00, 0x08, 0x80, 0x02, 0x08, 0xc0, 0x00, 0x01, 0x58, 0x30, 0x49, 0x09, 0x14,
288 0x00, 0x41, 0x02, 0x0c, 0x02, 0x80, 0x40, 0x89, 0x00, 0x24, 0x08, 0x10, 0x05,
289 0x90, 0x32, 0x40, 0x0a, 0x82, 0x08, 0x00, 0x12, 0x61, 0x00, 0x04, 0x21, 0x00,
290 0x22, 0x04, 0x10, 0x24, 0x08, 0x0a, 0x04, 0x01, 0x10, 0x00, 0x20, 0x40, 0x84,
291 0x04, 0x88, 0x22, 0x20, 0x90, 0x12, 0x00, 0x53, 0x06, 0x24, 0x01, 0x04, 0x40,
292 0x0b, 0x14, 0x60, 0x82, 0x02, 0x0d, 0x10, 0x90, 0x0c, 0x08, 0x20, 0x09, 0x00,
293 0x14, 0x09, 0x80, 0x80, 0x24, 0x82, 0x00, 0x40, 0x01, 0x02, 0x44, 0x01, 0x20,
294 0x0c, 0x40, 0x84, 0x40, 0x0a, 0x10, 0x41, 0x00, 0x30, 0x05, 0x09, 0x80, 0x44,
295 0x08, 0x20, 0x20, 0x02, 0x00, 0x49, 0x43, 0x20, 0x21, 0x00, 0x20, 0x00, 0x01,
296 0xb6, 0x08, 0x40, 0x04, 0x08, 0x02, 0x80, 0x01, 0x41, 0x80, 0x40, 0x08, 0x10,
297 0x24, 0x00, 0x20, 0x04, 0x12, 0x86, 0x09, 0xc0, 0x12, 0x21, 0x81, 0x14, 0x04,
298 0x00, 0x02, 0x20, 0x89, 0xb4, 0x44, 0x12, 0x80, 0x00, 0xd1, 0x00, 0x69, 0x40,
299 0x80, 0x00, 0x42, 0x12, 0x00, 0x18, 0x04, 0x00, 0x49, 0x06, 0x21, 0x02, 0x04,
300 0x28, 0x02, 0x84, 0x01, 0xc0, 0x10, 0x68, 0x00, 0x20, 0x08, 0x40, 0x00, 0x08,
301 0x91, 0x10, 0x01, 0x81, 0x24, 0x04, 0xd2, 0x10, 0x4c, 0x88, 0x86, 0x00, 0x10,
302 0x80, 0x0c, 0x02, 0x14, 0x00, 0x8a, 0x90, 0x40, 0x18, 0x20, 0x21, 0x80, 0xa4,
303 0x00, 0x58, 0x24, 0x20, 0x10, 0x10, 0x60, 0xc1, 0x30, 0x41, 0x48, 0x02, 0x48,
304 0x09, 0x00, 0x40, 0x09, 0x02, 0x05, 0x11, 0x82, 0x20, 0x4a, 0x20, 0x24, 0x18,
305 0x02, 0x0c, 0x10, 0x22, 0x0c, 0x0a, 0x04, 0x00, 0x03, 0x06, 0x48, 0x48, 0x04,
306 0x04, 0x02, 0x00, 0x21, 0x80, 0x84, 0x00, 0x18, 0x00, 0x0c, 0x02, 0x12, 0x01,
307 0x00, 0x14, 0x05, 0x82, 0x10, 0x41, 0x89, 0x12, 0x08, 0x40, 0xa4, 0x21, 0x01,
308 0x84, 0x48, 0x02, 0x10, 0x60, 0x40, 0x02, 0x28, 0x00, 0x14, 0x08, 0x40, 0xa0,
309 0x20, 0x51, 0x12, 0x00, 0xc2, 0x00, 0x01, 0x1a, 0x30, 0x40, 0x89, 0x12, 0x4c,
310 0x02, 0x80, 0x00, 0x00, 0x14, 0x01, 0x01, 0xa0, 0x21, 0x18, 0x22, 0x21, 0x18,
311 0x06, 0x40, 0x01, 0x80, 0x00, 0x90, 0x04, 0x48, 0x02, 0x30, 0x04, 0x08, 0x00,
312 0x05, 0x88, 0x24, 0x08, 0x48, 0x04, 0x24, 0x02, 0x06, 0x00, 0x80, 0x00, 0x00,
313 0x00, 0x10, 0x65, 0x11, 0x90, 0x00, 0x0a, 0x82, 0x04, 0xc3, 0x04, 0x60, 0x48,

```

314     0x24, 0x04, 0x92, 0x02, 0x44, 0x88, 0x80, 0x40, 0x18, 0x06, 0x29, 0x80, 0x10,
315     0x01, 0x00, 0x00, 0x44, 0xc8, 0x10, 0x21, 0x89, 0x30, 0x00, 0x4b, 0xa0, 0x01,
316     0x10, 0x14, 0x00, 0x02, 0x94, 0x40, 0x00, 0x20, 0x65, 0x00, 0xa2, 0x0c, 0x40,
317     0x22, 0x20, 0x81, 0x12, 0x20, 0x82, 0x04, 0x01, 0x10, 0x00, 0x08, 0x88, 0x00,
318     0x00, 0x11, 0x80, 0x04, 0x42, 0x80, 0x40, 0x41, 0x14, 0x00, 0x40, 0x32, 0x2c,
319     0x80, 0x24, 0x04, 0x19, 0x00, 0x00, 0x91, 0x00, 0x20, 0x83, 0x00, 0x05, 0x40,
320     0x20, 0x09, 0x01, 0x84, 0x40, 0x40, 0x20, 0x20, 0x11, 0x00, 0x40, 0x41, 0x90,
321     0x20, 0x00, 0x00, 0x40, 0x90, 0x92, 0x48, 0x18, 0x06, 0x08, 0x81, 0x80, 0x48,
322     0x01, 0x34, 0x24, 0x10, 0x20, 0x04, 0x00, 0x20, 0x04, 0x18, 0x06, 0x2d, 0x90,
323     0x10, 0x01, 0x00, 0x90, 0x00, 0x0a, 0x22, 0x01, 0x00, 0x22, 0x00, 0x11, 0x84,
324     0x01, 0x01, 0x00, 0x20, 0x88, 0x00, 0x44, 0x00, 0x22, 0x01, 0x00, 0xa6, 0x40,
325     0x02, 0x06, 0x20, 0x11, 0x00, 0x01, 0xc8, 0xa0, 0x04, 0x8a, 0x00, 0x28, 0x19,
326     0x80, 0x00, 0x52, 0xa0, 0x24, 0x12, 0x12, 0x09, 0x08, 0x24, 0x01, 0x48, 0x00,
327     0x04, 0x00, 0x24, 0x40, 0x02, 0x84, 0x08, 0x00, 0x04, 0x48, 0x40, 0x90, 0x60,
328     0x0a, 0x22, 0x01, 0x88, 0x14, 0x08, 0x01, 0x02, 0x08, 0xd3, 0x00, 0x20, 0xc0,
329     0x90, 0x24, 0x10, 0x00, 0x00, 0x01, 0xb0, 0x08, 0x0a, 0xa0, 0x00, 0x80, 0x00,
330     0x01, 0x09, 0x00, 0x20, 0x52, 0x02, 0x25, 0x00, 0x24, 0x04, 0x02, 0x84, 0x24,
331     0x10, 0x92, 0x40, 0x02, 0xa0, 0x40, 0x00, 0x22, 0x08, 0x11, 0x04, 0x08, 0x01,
332     0x22, 0x00, 0x42, 0x14, 0x00, 0x09, 0x90, 0x21, 0x00, 0x30, 0x6c, 0x00, 0x00,
333     0x0c, 0x00, 0x22, 0x09, 0x90, 0x10, 0x28, 0x40, 0x00, 0x20, 0xc0, 0x20, 0x00,
334     0x90, 0x00, 0x40, 0x01, 0x82, 0x05, 0x12, 0x12, 0x09, 0xc1, 0x04, 0x61, 0x80,
335     0x02, 0x28, 0x81, 0x24, 0x00, 0x49, 0x04, 0x08, 0x10, 0x86, 0x29, 0x41, 0x80,
336     0x21, 0x0a, 0x30, 0x49, 0x88, 0x90, 0x00, 0x41, 0x04, 0x29, 0x81, 0x80, 0x41,
337     0x09, 0x00, 0x40, 0x12, 0x10, 0x40, 0x00, 0x10, 0x40, 0x48, 0x02, 0x05, 0x80,
338     0x02, 0x21, 0x40, 0x20, 0x00, 0x58, 0x20, 0x60, 0x00, 0x90, 0x48, 0x00, 0x80,
339     0x28, 0xc0, 0x80, 0x48, 0x00, 0x00, 0x44, 0x80, 0x02, 0x00, 0x09, 0x06, 0x00,
340     0x12, 0x02, 0x01, 0x00, 0x10, 0x08, 0x83, 0x10, 0x45, 0x12, 0x00, 0x2c, 0x08,
341     0x04, 0x44, 0x00, 0x20, 0x20, 0xc0, 0x10, 0x20, 0x01, 0x00, 0x05, 0xc8, 0x20,
342     0x04, 0x98, 0x10, 0x08, 0x10, 0x00, 0x24, 0x02, 0x16, 0x40, 0x88, 0x00, 0x61,
343     0x88, 0x12, 0x24, 0x80, 0xa6, 0x00, 0x42, 0x00, 0x08, 0x10, 0x06, 0x48, 0x40,
344     0xa0, 0x00, 0x50, 0x20, 0x04, 0x81, 0xa4, 0x40, 0x18, 0x00, 0x08, 0x10, 0x80,
345     0x01, 0x01};
346
347 #if RSA_KEY_SIEVE && SIMULATION && RSA_INSTRUMENT
348 UINT32 PrimeIndex = 0;
349 UINT32 failedAtIteration[10] = {0};
350 UINT32 PrimeCounts[3] = {0};
351 UINT32 MillerRabinTrials[3] = {0};
352 UINT32 totalFieldsSieved[3] = {0};
353 UINT32 bitsInFieldAfterSieve[3] = {0};
354 UINT32 emptyFieldsSieved[3] = {0};
355 UINT32 noPrimeFields[3] = {0};
356 UINT32 primesChecked[3] = {0};
357 UINT16 lastSievePrime = 0;
358 #endif

```

7.153 /tpm/src/crypt/RsaKeyCache.c

```

1  /** Introduction
2  // This file contains the functions to implement the RSA key cache that can be used
3  // to speed up simulation.
4  //
5  // Only one key is created for each supported key size and it is returned whenever
6  // a key of that size is requested.
7  //
8  // If desired, the key cache can be populated from a file. This allows multiple
9  // TPM to run with the same RSA keys. Also, when doing simulation, the DRBG will
10 // use preset sequences so it is not too hard to repeat sequences for debug or
11 // profile or stress.
12 //
13 // When the key cache is enabled, a call to CryptRsaGenerateKey() will call the
14 // GetCachedRsaKey(). If the cache is enabled and populated, then the cached key
15 // of the requested size is returned. If a key of the requested size is not
16 // available, the no key is loaded and the requested key will need to be generated.
17 // If the cache is not populated, the TPM will open a file that has the appropriate

```

```

18 // name for the type of keys required (CRT or no-CRT). If the file is the right
19 // size, it is used. If the file doesn't exist or the file does not have the correct
20 // size, the TMP will populate the cache with new keys of the required size and
21 // write the cache data to the file so that they will be available the next time.
22 //
23 // Currently, if two simulations are being run with TPM's that have different RSA
24 // key sizes (e.g., one with 1024 and 2048 and another with 2048 and 3072, then the
25 // files will not match for the both of them and they will both try to overwrite
26 // the other's cache file. I may try to do something about this if necessary.
27
28 /** Includes, Types, Locals, and Defines
29
30 #include "Tpm.h"
31
32 #if USE_RSA_KEY_CACHE
33
34 # include <stdio.h>
35 # include "RsaKeyCache_fp.h"
36
37 # if CRT_FORMAT_RSA == YES
38 #   define CACHE_FILE_NAME "RsaKeyCacheCrt.data"
39 # else
40 #   define CACHE_FILE_NAME "RsaKeyCacheNoCrt.data"
41 # endif
42
43 typedef struct _RSA_KEY_CACHE_
44 {
45     TPM2B_PUBLIC_KEY_RSA publicModulus;
46     TPM2B_PRIVATE_KEY_RSA privateExponent;
47 } RSA_KEY_CACHE;
48
49 // Determine the number of RSA key sizes for the cache
50 TPMI_RSA_KEY_BITS SupportedRsaKeySizes[] = {
51 # if RSA_1024
52     1024,
53 # endif
54 # if RSA_2048
55     2048,
56 # endif
57 # if RSA_3072
58     3072,
59 # endif
60 # if RSA_4096
61     4096,
62 # endif
63     0};
64
65 # define RSA_KEY_CACHE_ENTRIES (RSA_1024 + RSA_2048 + RSA_3072 + RSA_4096)
66
67 // The key cache holds one entry for each of the supported key sizes
68 RSA_KEY_CACHE s_rsaKeyCache[RSA_KEY_CACHE_ENTRIES];
69 // Indicates if the key cache is loaded. It can be loaded and enabled or disabled.
70 BOOL s_keyCacheLoaded = 0;
71
72 // Indicates if the key cache is enabled
73 int s_rsaKeyCacheEnabled = FALSE;
74
75 /*** RsaKeyCacheControl()
76 // Used to enable and disable the RSA key cache.
77 LIB_EXPORT void RsaKeyCacheControl(int state)
78 {
79     s_rsaKeyCacheEnabled = state;
80 }
81
82 /*** InitializeKeyCache()
83 // This will initialize the key cache and attempt to write it to a file for later

```



```

84 // use.
85 // Return Type: BOOL
86 //     TRUE(1)      success
87 //     FALSE(0)     failure
88 static BOOL InitializeKeyCache(TPMT_PUBLIC*   publicArea,
89                               TPMT_SENSITIVE* sensitive,
90                               RAND_STATE* rand // IN: if not NULL, the deterministic
91                                              // RNG state
92 )
93 {
94     int index;
95     TPM_KEY_BITS keySave = publicArea->parameters.rsaDetail.keyBits;
96     BOOL OK = TRUE;
97     //
98     s_rsaKeyCacheEnabled = FALSE;
99     for(index = 0; OK && index < RSA_KEY_CACHE_ENTRIES; index++)
100     {
101         publicArea->parameters.rsaDetail.keyBits = SupportedRsaKeySizes[index];
102         OK = (CryptRsaGenerateKey(publicArea, sensitive, rand) == TPM_RC_SUCCESS);
103         if(OK)
104         {
105             s_rsaKeyCache[index].publicModulus = publicArea->unique.rsa;
106             s_rsaKeyCache[index].privateExponent = sensitive->sensitive.rsa;
107         }
108     }
109     publicArea->parameters.rsaDetail.keyBits = keySave;
110     s_keyCacheLoaded = OK;
111 # if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
112     if(OK)
113     {
114         FILE* cacheFile;
115         const char* fn = CACHE_FILE_NAME;
116
117 # if defined _MSC_VER
118         if(fopen_s(&cacheFile, fn, "w+b") != 0)
119 # else
120         cacheFile = fopen(fn, "w+b");
121         if(NULL == cacheFile)
122 # endif
123         {
124             printf("Can't open %s for write.\n", fn);
125         }
126         else
127         {
128             fseek(cacheFile, 0, SEEK_SET);
129             if(fwrite(s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
130                != sizeof(s_rsaKeyCache))
131             {
132                 printf("Error writing cache to %s.", fn);
133             }
134         }
135         if(cacheFile)
136             fclose(cacheFile);
137     }
138 # endif
139     return s_keyCacheLoaded;
140 }
141
142 /*** KeyCacheLoaded()
143 // Checks that key cache is loaded.
144 // Return Type: BOOL
145 //     TRUE(1)      cache loaded
146 //     FALSE(0)     cache not loaded
147 static BOOL KeyCacheLoaded(TPMT_PUBLIC*   publicArea,
148                           TPMT_SENSITIVE* sensitive,
149                           RAND_STATE* rand // IN: if not NULL, the deterministic

```

```

150                                     //      RNG state
151 )
152 {
153 # if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
154     if(!s_keyCacheLoaded)
155     {
156         FILE*      cacheFile;
157         const char* fn = CACHE_FILE_NAME;
158 # if defined _MSC_VER && 1
159         if(fopen_s(&cacheFile, fn, "r+b") == 0)
160 # else
161         cacheFile = fopen(fn, "r+b");
162         if(NULL != cacheFile)
163 # endif
164         {
165             fseek(cacheFile, 0L, SEEK_END);
166             if(ftell(cacheFile) == sizeof(s_rsaKeyCache))
167             {
168                 fseek(cacheFile, 0L, SEEK_SET);
169                 s_keyCacheLoaded =
170                     (fread(&s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
171                      == sizeof(s_rsaKeyCache));
172             }
173             fclose(cacheFile);
174         }
175     }
176 # endif
177     if(!s_keyCacheLoaded)
178         s_rsaKeyCacheEnabled = InitializeKeyCache(publicArea, sensitive, rand);
179     return s_keyCacheLoaded;
180 }
181
182 /*** GetCachedRsaKey()
183 // Return Type: BOOL
184 //     TRUE(1)      key loaded
185 //     FALSE(0)     key not loaded
186 BOOL GetCachedRsaKey(TPMT_PUBLIC*   publicArea,
187                     TPMT_SENSITIVE* sensitive,
188                     RAND_STATE*    rand // IN: if not NULL, the deterministic
189                                     //      RNG state
190 )
191 {
192     int keyBits = publicArea->parameters.rsaDetail.keyBits;
193     int index;
194     //
195     if(KeyCacheLoaded(publicArea, sensitive, rand))
196     {
197         for(index = 0; index < RSA_KEY_CACHE_ENTRIES; index++)
198         {
199             if((s_rsaKeyCache[index].publicModulus.t.size * 8) == keyBits)
200             {
201                 publicArea->unique.rsa = s_rsaKeyCache[index].publicModulus;
202                 sensitive->sensitive.rsa = s_rsaKeyCache[index].privateExponent;
203                 return TRUE;
204             }
205         }
206         return FALSE;
207     }
208     return s_keyCacheLoaded;
209 }
210 #endif // defined SIMULATION && defined USE_RSA_KEY_CACHE

```

7.154 /tpm/src/crypt/Ticket.c

```

1  /*** Introduction

```

```

2  /*
3   This clause contains the functions used for ticket computations.
4  */
5
6  /** Includes
7  #include "Tpm.h"
8  #include "Marshal.h"
9
10 /** Functions
11
12 /** TicketIsSafe()
13 // This function indicates if producing a ticket is safe.
14 // It checks if the leading bytes of an input buffer is TPM_GENERATED_VALUE
15 // or its substring of canonical form. If so, it is not safe to produce ticket
16 // for an input buffer claiming to be TPM generated buffer
17 // Return Type: BOOL
18 // TRUE(1)      safe to produce ticket
19 // FALSE(0)     not safe to produce ticket
20 BOOL TicketIsSafe(TPM2B* buffer)
21 {
22     TPM_CONSTANTS32 valueToCompare = TPM_GENERATED_VALUE;
23     BYTE             bufferToCompare[sizeof(valueToCompare)];
24     BYTE*            marshalBuffer;
25     //
26     // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume
27     // it is not safe to generate a ticket
28     if(buffer->size < sizeof(valueToCompare))
29         return FALSE;
30     marshalBuffer = bufferToCompare;
31     TPM_CONSTANTS32 Marshal(&valueToCompare, &marshalBuffer, NULL);
32     if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
33         return FALSE;
34     else
35         return TRUE;
36 }
37
38 /** TicketComputeVerified()
39 // This function creates a TPMT_TK_VERIFIED ticket.
40 /*(See part 2 specification)
41 // The ticket is computed as:
42 // HMAC(proof, (TPM_ST_VERIFIED | digest | keyName))
43 // Where:
44 // HMAC()          an HMAC using the hash of proof
45 // proof           a TPM secret value associated with the hierarchy
46 //                 associated with keyName
47 // TPM_ST_VERIFIED a value to differentiate the tickets
48 // digest          the signed digest
49 // keyName         the Name of the key that signed digest
50 */
51 TPM_RC TicketComputeVerified(
52     TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy constant for ticket
53     TPM2B_DIGEST* digest,       // IN: digest
54     TPM2B_NAME* keyName,        // IN: name of key that signed the values
55     TPMT_TK_VERIFIED* ticket    // OUT: verified ticket
56 )
57 {
58     TPM_RC result = TPM_RC_SUCCESS;
59     TPM2B_PROOF proof;
60     HMAC_STATE hmacState;
61     //
62     // Fill in ticket fields
63     ticket->tag = TPM_ST_VERIFIED;
64     ticket->hierarchy = hierarchy;
65     result = HierarchyGetProof(hierarchy, &proof);
66     if(result != TPM_RC_SUCCESS)
67         return result;

```

```

68
69 // Start HMAC using the proof value of the hierarchy as the HMAC key
70 ticket->digest.t.size =
71     CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG, &proof.b);
72 MemorySet(proof.b.buffer, 0, proof.b.size);
73
74 // TPM_ST_VERIFIED
75 CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
76 // digest
77 CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
78 // key name
79 CryptDigestUpdate2B(&hmacState.hashState, &keyName->b);
80 // done
81 CryptHmacEnd2B(&hmacState, &ticket->digest.b);
82
83 return TPM_RC_SUCCESS;
84 }
85
86 /*** TicketComputeAuth()
87 // This function creates a TPMT_TK_AUTH ticket.
88 /*(See part 2 specification)
89 // The ticket is computed as:
90 //     HMAC(proof, (type || timeout || timeEpoch || cpHash
91 //                 || policyRef || keyName))
92 // where:
93 //     HMAC()      an HMAC using the hash of proof
94 //     proof       a TPM secret value associated with the hierarchy of the key
95 //                 associated with keyName.
96 //     type        a value to differentiate the tickets. It could be either
97 //                 TPM_ST_AUTH_SECRET or TPM_ST_AUTH_SIGNED
98 //     timeout     TPM-specific value indicating when the authorization expires
99 //     timeEpoch  TPM-specific value indicating the epoch for the timeout
100 //     cpHash      optional hash (digest only) of the authorized command
101 //     policyRef   optional reference to a policy value
102 //     keyName     name of the key that signed the authorization
103 */
104 TPM_RC TicketComputeAuth(
105     TPM_ST type, // IN: the type of ticket.
106     TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy constant for ticket
107     UINT64 timeout, // IN: timeout
108     BOOL expiresOnReset, // IN: flag to indicate if ticket expires on
109                          // TPM Reset
110     TPM2B_DIGEST* cpHashA, // IN: input cpHashA
111     TPM2B_NONCE* policyRef, // IN: input policyRef
112     TPM2B_NAME* entityName, // IN: name of entity
113     TPMT_TK_AUTH* ticket // OUT: Created ticket
114 )
115 {
116     TPM_RC result = TPM_RC_SUCCESS;
117     TPM2B_PROOF proof;
118     HMAC_STATE hmacState;
119     //
120     // Get proper proof
121     result = HierarchyGetProof(hierarchy, &proof);
122     if(result != TPM_RC_SUCCESS)
123         return result;
124
125     // Fill in ticket fields
126     ticket->tag = type;
127     ticket->hierarchy = hierarchy;
128
129     // Start HMAC with hierarchy proof as the HMAC key
130     ticket->digest.t.size =
131         CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG, &proof.b);
132     MemorySet(proof.b.buffer, 0, proof.b.size);
133

```

```

134 // TPM_ST_AUTH_SECRET or TPM_ST_AUTH_SIGNED,
135 CryptDigestUpdateInt(&hmacState, sizeof(UINT16), ticket->tag);
136 // cpHash
137 CryptDigestUpdate2B(&hmacState.hashState, &cpHashA->b);
138 // policyRef
139 CryptDigestUpdate2B(&hmacState.hashState, &policyRef->b);
140 // keyName
141 CryptDigestUpdate2B(&hmacState.hashState, &entityName->b);
142 // timeout
143 CryptDigestUpdateInt(&hmacState, sizeof(timeout), timeout);
144 if(timeout != 0)
145 {
146     // epoch
147     CryptDigestUpdateInt(&hmacState.hashState, sizeof(CLOCK_NONCE), g_timeEpoch);
148     // reset count
149     if(expiresOnReset)
150         CryptDigestUpdateInt(
151             &hmacState.hashState, sizeof(gp.totalResetCount), gp.totalResetCount);
152 }
153 // done
154 CryptHmacEnd2B(&hmacState, &ticket->digest.b);
155
156 return TPM_RC_SUCCESS;
157 }
158
159 /** TicketComputeHashCheck()
160 // This function creates a TPMT_TK_HASHCHECK ticket.
161 /* (See part 2 specification)
162 // The ticket is computed as:
163 // HMAC(proof, (TPM_ST_HASHCHECK || digest))
164 // where:
165 // HMAC() an HMAC using the hash of proof
166 // proof a TPM secret value associated with the hierarchy
167 // TPM_ST_HASHCHECK
168 // a value to differentiate the tickets
169 // digest the digest of the data
170 */
171 TPM_RC TicketComputeHashCheck(
172     TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy constant for ticket
173     TPM_ALG_ID hashAlg, // IN: the hash algorithm for 'digest'
174     TPM2B_DIGEST* digest, // IN: input digest
175     TPMT_TK_HASHCHECK* ticket // OUT: Created ticket
176 )
177 {
178     TPM_RC result = TPM_RC_SUCCESS;
179     TPM2B_PROOF proof;
180     HMAC_STATE hmacState;
181     //
182     // Get proper proof
183     result = HierarchyGetProof(hierarchy, &proof);
184     if(result != TPM_RC_SUCCESS)
185         return result;
186
187     // Fill in ticket fields
188     ticket->tag = TPM_ST_HASHCHECK;
189     ticket->hierarchy = hierarchy;
190
191     // Start HMAC using hierarchy proof as HMAC key
192     ticket->digest.t.size =
193         CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG, &proof.b);
194     MemorySet(proof.b.buffer, 0, proof.b.size);
195
196     // TPM_ST_HASHCHECK
197     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
198     // hash algorithm
199     CryptDigestUpdateInt(&hmacState, sizeof(hashAlg), hashAlg);

```

```

200     // digest
201     CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
202     // done
203     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
204
205     return TPM_RC_SUCCESS;
206 }
207
208 /** TicketComputeCreation()
209  * This function creates a TPMT_TK_CREATION ticket.
210  * (See part 2 specification)
211  * The ticket is computed as:
212  *     HMAC(proof, (TPM_ST_CREATION || Name || hash(TPMS_CREATION_DATA)))
213  * Where:
214  *     HMAC() an HMAC using the hash of proof
215  *     proof a TPM secret value associated with the hierarchy associated with Name
216  *     TPM_ST_VERIFIED a value to differentiate the tickets
217  *     Name the Name of the object to which the creation data is to be associated
218  *     TPMS_CREATION_DATA the creation data structure associated with Name
219  */
220 TPM_RC TicketComputeCreation(TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy for ticket
221                             TPM2B_NAME* name,           // IN: object name
222                             TPM2B_DIGEST* creation,     // IN: creation hash
223                             TPMT_TK_CREATION* ticket,   // OUT: created ticket
224 )
225 {
226     TPM_RC result = TPM_RC_SUCCESS;
227     TPM2B_PROOF proof;
228     HMAC_STATE hmacState;
229
230     // Get proper proof
231     result = HierarchyGetProof(hierarchy, &proof);
232     if(result != TPM_RC_SUCCESS)
233         return result;
234
235     // Fill in ticket fields
236     ticket->tag = TPM_ST_CREATION;
237     ticket->hierarchy = hierarchy;
238
239     // Start HMAC using hierarchy proof as HMAC key
240     ticket->digest.t.size =
241         CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG, &proof.b);
242     MemorySet(proof.b.buffer, 0, proof.b.size);
243
244     // TPM_ST CREATION
245     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
246     // name if provided
247     if(name != NULL)
248         CryptDigestUpdate2B(&hmacState.hashState, &name->b);
249     // creation hash
250     CryptDigestUpdate2B(&hmacState.hashState, &creation->b);
251     // Done
252     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
253
254     return TPM_RC_SUCCESS;
255 }

```

7.155 /tpm/src/crypt/ecc/TpmEcc_Signature_ECDSA.c

```

1  #include "Tpm.h"
2  #include "TpmEcc_Signature_ECDSA_fp.h"
3  #include "TpmEcc_Signature_Util_fp.h"
4  #include "TpmMath_Debug_fp.h"
5  #include "TpmMath_Util_fp.h"
6

```



```

7  #if ALG_ECC && ALG_ECDA
8
9  /*** TpmEcc_SignEcdaa()
10 //
11 // This function performs 's' = 'r' + 'T' * 'd' mod 'q' where
12 // 1) 'r' is a random, or pseudo-random value created in the commit phase
13 // 2) 'nonceK' is a TPM-generated, random value 0 < 'nonceK' < 'n'
14 // 3) 'T' is mod 'q' of "Hash"('nonceK' || 'digest'), and
15 // 4) 'd' is a private key.
16 //
17 // The signature is the tuple ('nonceK', 's')
18 //
19 // Regrettably, the parameters in this function kind of collide with the parameter
20 // names used in ECSCHNORR making for a lot of confusion.
21 // Return Type: TPM_RC
22 //     TPM_RC_SCHEME      unsupported hash algorithm
23 //     TPM_RC_NO_RESULT   cannot get values from random number generator
24 TPM_RC TpmEcc_SignEcdaa(
25     TPM2B_ECC_PARAMETER* nonceK, // OUT: 'nonce' component of the signature
26     Crypt_Int* bnS, // OUT: 's' component of the signature
27     const Crypt_EccCurve* E, // IN: the curve used in signing
28     Crypt_Int* bnD, // IN: the private key
29     const TPM2B_DIGEST* digest, // IN: the value to sign (mod 'q')
30     TPMT_ECC_SCHEME* scheme, // IN: signing scheme (contains the
31                             // commit count value).
32     OBJECT* eccKey, // IN: The signing key
33     RAND_STATE* rand // IN: a random number state
34 )
35 {
36     TPM_RC retVal;
37     TPM2B_ECC_PARAMETER r;
38     HASH_STATE state;
39     TPM2B_DIGEST T;
40     CRYPT_INT_MAX(bnT);
41     //
42     NOT_REFERENCED(rand);
43     if(!CryptGeneratorR(&r,
44                         &scheme->details.ecdaa.count,
45                         eccKey->publicArea.parameters.eccDetail.curveID,
46                         &eccKey->name))
47         retVal = TPM_RC_VALUE;
48     else
49     {
50         // This allocation is here because 'r' doesn't have a value until
51         // CryptGeneratorR() is done.
52         CRYPT_ECC_INITIALIZED(bnR, &r);
53         do
54         {
55             // generate nonceK such that 0 < nonceK < n
56             // use bnT as a temp.
57             if(!TpmEcc_GenPrivateScalar(bnT, E, rand))
58             {
59                 retVal = TPM_RC_NO_RESULT;
60                 break;
61             }
62             TpmMath_IntTo2B(bnT, &nonceK->b, 0);
63
64             T.t.size = CryptHashStart(&state, scheme->details.ecdaa.hashAlg);
65             if(T.t.size == 0)
66             {
67                 retVal = TPM_RC_SCHEME;
68             }
69             else
70             {
71                 CryptDigestUpdate2B(&state, &nonceK->b);
72                 CryptDigestUpdate2B(&state, &digest->b);

```

```

73         CryptHashEnd2B(&state, &T.b);
74         TpmMath_IntFrom2B(bnT, &T.b);
75         // Watch out for the name collisions in this call!!
76         retVal = TpmEcc_SchnorrCalculates(
77             bnS,
78             bnR,
79             bnT,
80             bnD,
81             ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E)));
82     }
83     } while(retVal == TPM_RC_NO_RESULT);
84     // Because the rule is that internal state is not modified if the command
85     // fails, only end the commit if the command succeeds.
86     // NOTE that if the result of the Schnorr computation was zero
87     // it will probably not be worthwhile to run the same command again because
88     // the result will still be zero. This means that the Commit command will
89     // need to be run again to get a new commit value for the signature.
90     if(retVal == TPM_RC_SUCCESS)
91         CryptEndCommit(scheme->details.ecdaa.count);
92 }
93 return retVal;
94 }
95
96 #endif // ALG_ECC && ALG_ECDSA

```

7.156 /tpm/src/crypt/ecc/TpmEcc_Signature_ECDSA.c

```

1  #include "Tpm.h"
2  #include "TpmEcc_Signature_ECDSA_fp.h"
3  #include "TpmMath_Debug_fp.h"
4  #include "TpmMath_Util_fp.h"
5
6  #if ALG_ECC && ALG_ECDSA
7  /*** TpmEcc_AdjustEcdsaDigest()
8  // Function to adjust the digest so that it is no larger than the order of the
9  // curve. This is used for ECDSA sign and verification.
10 static Crypt_Int* TpmEcc_AdjustEcdsaDigest(
11     Crypt_Int*      bnD,      // OUT: the adjusted digest
12     const TPM2B_DIGEST* digest, // IN: digest to adjust
13     const Crypt_Int* max      // IN: value that indicates the maximum
14                               // number of bits in the results
15 )
16 {
17     int bitsInMax = ExtMath_SizeInBits(max);
18     int shift;
19     //
20     if(digest == NULL)
21         ExtMath_SetWord(bnD, 0);
22     else
23     {
24         ExtMath_IntFromBytes(bnD,
25                               digest->t.buffer,
26                               (NUMBYTES)MIN(digest->t.size, BITS_TO_BYTES(bitsInMax)));
27         shift = ExtMath_SizeInBits(bnD) - bitsInMax;
28         if(shift > 0)
29             ExtMath_ShiftRight(bnD, bnD, shift);
30     }
31     return bnD;
32 }
33
34 /*** TpmEcc_SignEcdsa()
35 // This function implements the ECDSA signing algorithm. The method is described
36 // in the comments below.
37 TPM_RC
38 TpmEcc_SignEcdsa(Crypt_Int*      bnR,      // OUT: 'r' component of the signature

```

```

39         Crypt_Int*      bnS,      // OUT: 's' component of the signature
40         const Crypt_EccCurve* E,    // IN: the curve used in the signature
41                                     // process
42         Crypt_Int*      bnD,      // IN: private signing key
43         const TPM2B_DIGEST* digest, // IN: the digest to sign
44         RAND_STATE*      rand     // IN: used in debug of signing
45     )
46 {
47     CRYPT_ECC_NUM(bnK);
48     CRYPT_ECC_NUM(bnIk);
49     CRYPT_INT_VAR(bnE, MAX_ECC_KEY_BITS);
50     CRYPT_POINT_VAR(ecR);
51     CRYPT_ECC_NUM(bnX);
52     const Crypt_Int* order = ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E));
53     TPM_RC          retVal = TPM_RC_SUCCESS;
54     INT32           tries  = 10;
55     BOOL            OK     = FALSE;
56     //
57     pAssert(digest != NULL);
58     // The algorithm as described in "Suite B Implementer's Guide to FIPS
59     // 186-3(ECDSA)"
60     // 1. Use one of the routines in Appendix A.2 to generate (k, k^-1), a
61     //    per-message secret number and its inverse modulo n. Since n is prime,
62     //    the output will be invalid only if there is a failure in the RBG.
63     // 2. Compute the elliptic curve point R = [k]G = (xR, yR) using EC scalar
64     //    multiplication (see [Routines]), where G is the base point included in
65     //    the set of domain parameters.
66     // 3. Compute r = xR mod n. If r = 0, then return to Step 1. 1.
67     // 4. Use the selected hash function to compute H = Hash(M).
68     // 5. Convert the bit string H to an integer e as described in Appendix B.2.
69     // 6. Compute s = (k^-1 * (e + d * r)) mod q. If s = 0, return to Step 1.2.
70     // 7. Return (r, s).
71     // In the code below, q is n (that is, the order of the curve is p)
72
73     do // This implements the loop at step 6. If s is zero, start over.
74     {
75         for(; tries > 0; tries--)
76         {
77             // Step 1 and 2 -- generate an ephemeral key and the modular inverse
78             // of the private key.
79             if(!TpmEcc_GenerateKeyPair(bnK, ecR, E, rand))
80                 continue;
81             // get mutable copy of X coordinate
82             ExtMath_Copy(bnX, ExtEcc_PointX(ecR));
83             // x coordinate is mod p. Make it mod q
84             ExtMath_Mod(bnX, order);
85             // Make sure that it is not zero;
86             if(ExtMath_IsZero(bnX))
87                 continue;
88             // write the modular reduced version of r as part of the signature
89             ExtMath_Copy(bnR, bnX);
90             // Make sure that a modular inverse exists and try again if not
91             OK = (ExtMath_ModInverse(bnIk, bnK, order));
92             if(OK)
93                 break;
94         }
95         if(!OK)
96             goto Exit;
97
98         TpmEcc_AdjustEcDSA_Digest(bnE, digest, order);
99
100        // now have inverse of K (bnIk), e (bnE), r (bnR), d (bnD) and
101        // ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E))
102        // Compute s = k^-1 (e + r*d) (mod q)
103        // first do s = r*d mod q
104        ExtMath_ModMult(bnS, bnR, bnD, order);

```

```

105         // s = e + s = e + r * d
106         ExtMath_Add(bnS, bnE, bnS);
107         // s = k^(-1)s (mod n) = k^(-1)(e + r * d) (mod n)
108         ExtMath_ModMult(bnS, bnIk, bnS, order);
109
110         // If S is zero, try again
111     } while(ExtMath_IsZero(bnS));
112 Exit:
113     return retVal;
114 }
115
116 /*** TpmEcc_ValidateSignatureEcdsa()
117 // This function validates an ECDSA signature. rIn and sIn should have been checked
118 // to make sure that they are in the range 0 < 'v' < 'n'
119 // Return Type: TPM_RC
120 //     TPM_RC_SIGNATURE      signature not valid
121 TPM_RC
122 TpmEcc_ValidateSignatureEcdsa(
123     Crypt_Int*      bnR, // IN: 'r' component of the signature
124     Crypt_Int*      bnS, // IN: 's' component of the signature
125     const Crypt_EccCurve* E, // IN: the curve used in the signature
126                             // process
127     const Crypt_Point* ecQ, // IN: the public point of the key
128     const TPM2B_DIGEST* digest // IN: the digest that was signed
129 )
130 {
131     // Make sure that the allocation for the digest is big enough for a maximum
132     // digest
133     CRYPT_INT_VAR(bnE, MAX_ECC_KEY_BITS);
134     CRYPT_POINT_VAR(ecR);
135     CRYPT_ECC_NUM(bnU1);
136     CRYPT_ECC_NUM(bnU2);
137     CRYPT_ECC_NUM(bnW);
138     CRYPT_ECC_NUM(bnV);
139     const Crypt_Int* order = ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E));
140     TPM_RC      retVal = TPM_RC_SIGNATURE;
141     //
142     // Get adjusted digest
143     TpmEcc_AdjustEcdsaDigest(bnE, digest, order);
144     // 1. If r and s are not both integers in the interval [1, n - 1], output
145     //     INVALID.
146     // bnR and bnS were validated by the caller
147     // 2. Use the selected hash function to compute H0 = Hash(M0).
148     // This is an input parameter
149     // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
150     // Done at entry
151     // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
152     if(!ExtMath_ModInverse(bnW, bnS, order))
153         goto Exit;
154     // 5. Compute u1 = (e' * w) mod n, and compute u2 = (r' * w) mod n.
155     ExtMath_ModMult(bnU1, bnE, bnW, order);
156     ExtMath_ModMult(bnU2, bnR, bnW, order);
157     // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
158     //     scalar multiplication and EC addition (see [Routines]). If R is equal to
159     //     the point at infinity O, output INVALID.
160     if(TpmEcc_PointMult(
161         ecR, ExtEcc_CurveGetG(ExtEcc_CurveGetCurveId(E)), bnU1, ecQ, bnU2, E)
162        != TPM_RC_SUCCESS)
163         goto Exit;
164     // 7. Compute v = Rx mod n.
165     ExtMath_Copy(bnV, ExtEcc_PointX(ecR));
166     ExtMath_Mod(bnV, order);
167     // 8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
168     if(ExtMath_UnsignedCmp(bnV, bnR) != 0)
169         goto Exit;
170

```

```

171     retVal = TPM_RC_SUCCESS;
172 Exit:
173     return retVal;
174 }
175
176 #endif // ALG_ECC && ALG_ECDSA

```

7.157 /tpm/src/crypt/ecc/TpmEcc_Signature_Schnorr.c

```

1  #include "Tpm.h"
2  #include "TpmEcc_Signature_Schnorr_fp.h"
3  #include "TpmEcc_Signature_Util_fp.h"
4  #include "TpmMath_Debug_fp.h"
5  #include "TpmMath_Util_fp.h"
6
7  #if ALG_ECC && ALG_ECSCHNORR
8
9  /*** SchnorrReduce()
10 // Function to reduce a hash result if it's magnitude is too large. The size of
11 // 'number' is set so that it has no more bytes of significance than 'reference'
12 // value. If the resulting number can have more bits of significance than
13 // 'reference'.
14 static void SchnorrReduce(TPM2B* number, // IN/OUT: Value to reduce
15                          const Crypt_Int* reference // IN: the reference value
16 )
17 {
18     UINT16 maxBytes = (UINT16)BITS_TO_BYTES(ExtMath_SizeInBits(reference));
19     if(number->size > maxBytes)
20         number->size = maxBytes;
21 }
22
23 /*** SchnorrEcc()
24 // This function is used to perform a modified Schnorr signature.
25 //
26 // This function will generate a random value 'k' and compute
27 // a) ('xR', 'yR') = ['k']'G'
28 // b) 'r' = "Hash"('xR' || 'P') (mod 'q')
29 // c) 'rT' = truncated 'r'
30 // d) 's' = 'k' + 'rT' * 'ds' (mod 'q')
31 // e) return the tuple 'rT', 's'
32 //
33 // Return Type: TPM_RC
34 //     TPM_RC_NO_RESULT failure in the Schnorr sign process
35 //     TPM_RC_SCHEME hashAlg can't produce zero-length digest
36 TPM_RC TpmEcc_SignEcSchnorr(
37     Crypt_Int* bnR, // OUT: 'r' component of the signature
38     Crypt_Int* bnS, // OUT: 's' component of the signature
39     const Crypt_EccCurve* E, // IN: the curve used in signing
40     Crypt_Int* bnD, // IN: the signing key
41     const TPM2B_DIGEST* digest, // IN: the digest to sign
42     TPM_ALG_ID hashAlg, // IN: signing scheme (contains a hash)
43     RAND_STATE* rand // IN: non-NULL when testing
44 )
45 {
46     HASH_STATE hashState;
47     UINT16 digestSize = CryptHashGetDigestSize(hashAlg);
48     TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_KEY_BYTES));
49     TPM2B_T T2b;
50     TPM2B* e = &T2b.b;
51     TPM_RC retVal = TPM_RC_NO_RESULT;
52     const Crypt_Int* order;
53     const Crypt_Int* prime;
54     CRYPT_ECC_NUM(bnK);
55     CRYPT_POINT_VAR(ecR);
56     //

```

```

57     // Parameter checks
58     if (E == NULL)
59         ERROR_EXIT(TPM_RC_VALUE);
60
61     order = ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E));
62     prime = ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E));
63
64     // If the digest does not produce a hash, then null the signature and return
65     // a failure.
66     if (digestSize == 0)
67     {
68         ExtMath_SetWord(bnR, 0);
69         ExtMath_SetWord(bnS, 0);
70         ERROR_EXIT(TPM_RC_SCHEME);
71     }
72     do
73     {
74         // Generate a random key pair
75         if (!TpmEcc_GenerateKeyPair(bnK, ecR, E, rand))
76             break;
77         // Convert R.x to a string
78         TpmMath_IntTo2B(ExtEcc_PointX(ecR),
79             e,
80             (NUMBYTES)BITS_TO_BYTES(ExtMath_SizeInBits(prime)));
81
82         // f) compute r = Hash(e || P) (mod n)
83         CryptHashStart(&hashState, hashAlg);
84         CryptDigestUpdate2B(&hashState, e);
85         CryptDigestUpdate2B(&hashState, &digest->b);
86         e->size = CryptHashEnd(&hashState, digestSize, e->buffer);
87         // Reduce the hash size if it is larger than the curve order
88         SchnorrReduce(e, order);
89         // Convert hash to number
90         TpmMath_IntFrom2B(bnR, e);
91         // Do the Schnorr computation
92         retVal = TpmEcc_SchnorrCalculateS(
93             bnS, bnK, bnR, bnD, ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E)));
94     } while (retVal == TPM_RC_NO_RESULT);
95 Exit:
96     return retVal;
97 }
98
99 /*** TpmEcc_ValidateSignatureEcSchnorr()
100 // This function is used to validate an EC Schnorr signature.
101 // Return Type: TPM_RC
102 //     TPM_RC_SIGNATURE      signature not valid
103 TPM_RC TpmEcc_ValidateSignatureEcSchnorr(
104     Crypt_Int*      bnR,      // IN: 'r' component of the signature
105     Crypt_Int*      bnS,      // IN: 's' component of the signature
106     TPM_ALG_ID      hashAlg,  // IN: hash algorithm of the signature
107     const Crypt_EccCurve* E,   // IN: the curve used in the signature
108                               // process
109     Crypt_Point*    ecQ,      // IN: the public point of the key
110     const TPM2B_DIGEST* digest // IN: the digest that was signed
111 )
112 {
113     CRYPT_INT_MAX(bnRn);
114     CRYPT_POINT_VAR(ecE);
115     CRYPT_INT_MAX(bnEx);
116     const Crypt_Int* order = ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E));
117     UINT16 digestSize = CryptHashGetDigestSize(hashAlg);
118     HASH_STATE hashState;
119     TPM2B_TYPE(BUFFER, MAX(ECC_PARAMETER_BYTES, MAX_DIGEST_SIZE));
120     TPM2B_BUFFER Ex2 = {{sizeof(Ex2.t.buffer), {0}}};
121     BOOL OK;
122     //

```



```

123 // E = [s]G - [r]Q
124 ExtMath_Mod(bnR, order);
125 // Make -r = n - r
126 ExtMath_Subtract(bnRn, order, bnR);
127 // E = [s]G + [-r]Q
128 OK = TpmEcc_PointMult(
129     ecE, ExtEcc_CurveGetG(ExtEcc_CurveGetCurveId(E)), bnS, ecQ, bnRn, E)
130 == TPM_RC_SUCCESS;
131 // reduce the x portion of E mod q
132 // OK = OK && ExtMath_Mod(ecE->x, order);
133 // Convert to byte string
134 OK = OK
135     && TpmMath_IntTo2B(ExtEcc_PointX(ecE),
136                        &Ex2.b,
137                        (NUMBYTES)(BITS_TO_BYTES(ExtMath_SizeInBits(order))));
138 if(OK)
139 {
140     // Ex = h(pE.x || digest)
141     CryptHashStart(&hashState, hashAlg);
142     CryptDigestUpdate(&hashState, Ex2.t.size, Ex2.t.buffer);
143     CryptDigestUpdate(&hashState, digest->t.size, digest->t.buffer);
144     Ex2.t.size = CryptHashEnd(&hashState, digestSize, Ex2.t.buffer);
145     SchnorrReduce(&Ex2.b, order);
146     TpmMath_IntFrom2B(bnEx, &Ex2.b);
147     // see if Ex matches R
148     OK = ExtMath_UnsignedCmp(bnEx, bnR) == 0;
149 }
150 return (OK) ? TPM_RC_SUCCESS : TPM_RC_SIGNATURE;
151 }
152
153 #endif // ALG_ECC && ALG_ECSCNORR

```

7.158 /tpm/src/crypt/ecc/TpmEcc_Signature_SM2.c

```

1 #include "Tpm.h"
2 #include "TpmEcc_Signature_SM2_fp.h"
3 #include "TpmMath_Debug_fp.h"
4 #include "TpmMath_Util_fp.h"
5
6 #if ALG_ECC && ALG_SM2
7
8 /*** TpmEcc_SignEcSm2()
9 // This function signs a digest using the method defined in SM2 Part 2. The method
10 // in the standard will add a header to the message to be signed that is a hash of
11 // the values that define the key. This then hashed with the message to produce a
12 // digest ('e'). This function signs 'e'.
13 // Return Type: TPM_RC
14 // TPM_RC_VALUE bad curve
15 TPM_RC TpmEcc_SignEcSm2(Crypt_Int* bnR, // OUT: 'r' component of the signature
16                        Crypt_Int* bnS, // OUT: 's' component of the signature
17                        const Crypt_EccCurve* E, // IN: the curve used in signing
18                        Crypt_Int* bnD, // IN: the private key
19                        const TPM2B_DIGEST* digest, // IN: the digest to sign
20                        RAND_STATE* rand // IN: random number generator (mostly for
21                                         // debug)
22 )
23 {
24     CRYPT_INT_MAX_INITIALIZED(bnE, digest); // Don't know how big digest might be
25     CRYPT_ECC_NUM(bnN);
26     CRYPT_ECC_NUM(bnK);
27     CRYPT_ECC_NUM(bnT); // temp
28     CRYPT_POINT_VAR(Q1);
29     const Crypt_Int* order =
30         (E != NULL) ? ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E)) : NULL;
31 //

```

```

32 # ifdef _SM2_SIGN_DEBUG
33     TpmEccDebug_FromHex(bnE,
34         "B524F552CD82B8B028476E005C377FB1"
35         "9A87E6FC682D48BB5D42E3D9B9E7FE76",
36         MAX_ECC_KEY_BYTES);
37     TpmEccDebug_FromHex(bnD,
38         "128B2FA8BD433C6C068C8D803DFF7979"
39         "2A519A55171B1B650C23661D15897263",
40         MAX_ECC_KEY_BYTES);
41 # endif
42 // A3: Use random number generator to generate random number 1 <= k <= n-1;
43 // NOTE: Ax: numbers are from the SM2 standard
44 loop:
45 {
46     // Get a random number 0 < k < n
47     TpmMath_GetRandomInRange(bnK, order, rand);
48 # ifdef _SM2_SIGN_DEBUG
49     TpmEccDebug_FromHex(bnK,
50         "6CB28D99385C175C94F94E934817663F"
51         "C176D925DD72B727260DBAAE1FB2F96F",
52         MAX_ECC_KEY_BYTES);
53 # endif
54 // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
55 // to details specified in 4.2.7 in Part 1 of this document, transform the
56 // data type of x1 into an integer;
57 if(!ExtEcc_PointMultiply(Q1, NULL, bnK, E))
58     goto loop;
59 // A5: Figure out 'r' = ('e' + 'x1') mod 'n',
60 ExtMath_Add(bnR, bnE, ExtEcc_PointX(Q1));
61 ExtMath_Mod(bnR, order);
62 # ifdef _SM2_SIGN_DEBUG
63 pAssert(TpmEccDebug_HexEqual(bnR,
64     "40F1EC59F793D9F49E09DCEF49130D41"
65     "94F79FB1EED2CAA55BACDB49C4E755D1"));
66 # endif
67 // if r=0 or r+k=n, return to A3;
68 if(ExtMath_IsZero(bnR))
69     goto loop;
70 ExtMath_Add(bnT, bnK, bnR);
71 if(ExtMath_UnsignedCmp(bnT, bnN) == 0)
72     goto loop;
73 // A6: Figure out s = ((1 + dA)^-1 (k - r dA)) mod n,
74 // if s=0, return to A3;
75 // compute t = (1+dA)^-1
76 ExtMath_AddWord(bnT, bnD, 1);
77 ExtMath_ModInverse(bnT, bnT, order);
78 # ifdef _SM2_SIGN_DEBUG
79 pAssert(TpmEccDebug_HexEqual(bnT,
80     "79BFCF3052C80DA7B939E0C6914A18CB"
81     "B2D96D8555256E83122743A7D4F5F956"));
82 # endif
83 // compute s = t * (k - r * dA) mod n
84 ExtMath_ModMult(bnS, bnR, bnD, order);
85 // k - r * dA mod n = k + n - ((r * dA) mod n)
86 ExtMath_Subtract(bnS, order, bnS);
87 ExtMath_Add(bnS, bnK, bnS);
88 ExtMath_ModMult(bnS, bnS, bnT, order);
89 # ifdef _SM2_SIGN_DEBUG
90 pAssert(TpmEccDebug_HexEqual(bnS,
91     "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
92     "67A457872FB09EC56327A67EC7DEEBE7"));
93 # endif
94 if(ExtMath_IsZero(bnS))
95     goto loop;
96 }
97 // A7: According to details specified in 4.2.1 in Part 1 of this document,

```

```

98 // transform the data type of r, s into bit strings, signature of message M
99 // is (r, s).
100 // This is handled by the common return code
101 # ifdef _SM2_SIGN_DEBUG
102     pAssert(TpmEccDebug_HexEqual(bnR,
103                                   "40F1EC59F793D9F49E09DCEF49130D41"
104                                   "94F79FB1EED2CAA55BACDB49C4E755D1"));
105     pAssert(TpmEccDebug_HexEqual(bnS,
106                                   "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
107                                   "67A457872FB09EC56327A67EC7DEEBE7"));
108 # endif
109     return TPM_RC_SUCCESS;
110 }
111
112 /*** TpmEcc_ValidateSignatureEcSm2()
113 // This function is used to validate an SM2 signature.
114 // Return Type: TPM_RC
115 // TPM_RC_SIGNATURE signature not valid
116 TPM_RC TpmEcc_ValidateSignatureEcSm2(
117     Crypt_Int* bnR, // IN: 'r' component of the signature
118     Crypt_Int* bnS, // IN: 's' component of the signature
119     const Crypt_EccCurve* E, // IN: the curve used in the signature
120                             // process
121     Crypt_Point* ecQ, // IN: the public point of the key
122     const TPM2B_DIGEST* digest // IN: the digest that was signed
123 )
124 {
125     CRYPT_POINT VAR(P);
126     CRYPT_ECC_NUM(bnRp);
127     CRYPT_ECC_NUM(bnT);
128     CRYPT_INT_MAX_INITIALIZED(bnE, digest);
129     BOOL OK;
130     const Crypt_Int* order = ExtEcc_CurveGetOrder(ExtEcc_CurveGetCurveId(E));
131
132 # ifdef _SM2_SIGN_DEBUG
133     // Make sure that the input signature is the test signature
134     pAssert(TpmEccDebug_HexEqual(bnR,
135                                   "40F1EC59F793D9F49E09DCEF49130D41"
136                                   "94F79FB1EED2CAA55BACDB49C4E755D1"));
137     pAssert(TpmEccDebug_HexEqual(bnS,
138                                   "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
139                                   "67A457872FB09EC56327A67EC7DEEBE7"));
140 # endif
141     // b) compute t := (r + s) mod n
142     ExtMath_Add(bnT, bnR, bnS);
143     ExtMath_Mod(bnT, order);
144 # ifdef _SM2_SIGN_DEBUG
145     pAssert(TpmEccDebug_HexEqual(bnT,
146                                   "2B75F07ED7ECE7CCC1C8986B991F441A"
147                                   "D324D6D619FE06DD63ED32E0C997C801"));
148 # endif
149     // c) verify that t > 0
150     OK = !ExtMath_IsZero(bnT);
151     if(!OK)
152         // set T to a value that should allow rest of the computations to run
153         // without trouble
154         ExtMath_Copy(bnT, bnS);
155     // d) compute (x, y) := [s]G + [t]Q
156     OK = ExtEcc_PointMultiplyAndAdd(P, NULL, bnS, ecQ, bnT, E);
157 # ifdef _SM2_SIGN_DEBUG
158     pAssert(OK
159             && TpmEccDebug_HexEqual(ExtEcc_PointX(P),
160                                     "110FCDA57615705D5E7B9324AC4B856D"
161                                     "23E6D9188B2AE47759514657CE25D112"));
162 # endif
163     // e) compute r' := (e + x) mod n (the x coordinate is in bnT)

```

```

164     OK = OK && ExtMath_Add(bnRp, bnE, ExtEcc_PointX(P));
165     OK = OK && ExtMath_Mod(bnRp, order);
166
167     // f) verify that r' = r
168     OK = OK && (ExtMath_UnsignedCmp(bnR, bnRp) == 0);
169
170     if(!OK)
171         return TPM_RC_SIGNATURE;
172     else
173         return TPM_RC_SUCCESS;
174 }
175
176 #endif // ALG_ECC && ALG_SM2

```

7.159 /tpm/src/crypt/ecc/TpmEcc_Signature_Util.c

```

1  // functions shared by multiple signature algorithms
2  #include "Tpm.h"
3  #include "TpmEcc_Signature_Util_fp.h"
4  #include "TpmMath_Debug_fp.h"
5  #include "TpmMath_Util_fp.h"
6
7  #if(ALG_ECC && (ALG_ECSCHNORR || ALG_ECDA))
8
9  /*** TpmEcc_SchnorrCalculateS()
10 // This contains the Schnorr signature (S) computation. It is used by both ECDA and
11 // Schnorr signing. The result is computed as: ['s' = 'k' + 'r' * 'd' (mod 'n')]
12 // where
13 // 1) 's' is the signature
14 // 2) 'k' is a random value
15 // 3) 'r' is the value to sign
16 // 4) 'd' is the private EC key
17 // 5) 'n' is the order of the curve
18 // Return Type: TPM_RC
19 // TPM_RC_NO_RESULT the result of the operation was zero or 'r' (mod 'n')
20 // is zero
21 TPM_RC TpmEcc_SchnorrCalculateS(
22     Crypt_Int* bnS, // OUT: 's' component of the signature
23     const Crypt_Int* bnK, // IN: a random value
24     Crypt_Int* bnR, // IN: the signature 'r' value
25     const Crypt_Int* bnD, // IN: the private key
26     const Crypt_Int* bnN // IN: the order of the curve
27 )
28 {
29     // Need a local temp value to store the intermediate computation because product
30     // size can be larger than will fit in bnS.
31     CRYPT_INT_VAR(bnT1, MAX_ECC_PARAMETER_BYTES * 2 * 8);
32     //
33     // Reduce bnR without changing the input value
34     ExtMath_Divide(NULL, bnT1, bnR, bnN);
35     if(ExtMath_IsZero(bnT1))
36         return TPM_RC_NO_RESULT;
37     // compute s = (k + r * d) (mod n)
38     // r * d
39     ExtMath_Multiply(bnT1, bnT1, bnD);
40     // k + r * d
41     ExtMath_Add(bnT1, bnT1, bnK);
42     // k + r * d (mod n)
43     ExtMath_Divide(NULL, bnS, bnT1, bnN);
44     return (ExtMath_IsZero(bnS)) ? TPM_RC_NO_RESULT : TPM_RC_SUCCESS;
45 }
46
47 #endif // (ALG_ECC && (ALG_ECSCHNORR || ALG_ECDA))

```

7.160 /tpm/src/crypt/ecc/TpmEcc_Util.c

```

1  /** Introduction
2  // This file contains utility functions to help using the external Math library
3  // for Ecc functions.
4  #include "Tpm.h"
5  #include "TpmMath_Util_fp.h"
6
7  #if ALG_ECC
8
9  /**
10 // TpmEcc_PointFrom2B() Function to create a Crypt_Point structure from a 2B
11 // point. The target point is expected to have memory allocated and
12 // uninitialized. A TPMS_ECC_POINT is going to be two ECC values in the same
13 // buffer. The values are going to be the size of the modulus. They are in
14 // modular form.
15 //
16 // NOTE: This function considers both parameters optional because of use
17 // cases where points may not be specified in the calling function. If the
18 // initializer or point buffer is NULL, then NULL is returned. As a result, the
19 // only error detection when the initializer value is invalid is to return NULL
20 // in that error case as well. If a caller wants to handle that error case
21 // differently, then the caller must perform the correct validation before/after
22 // this function.
23 LIB_EXPORT Crypt_Point* TpmEcc_PointFrom2B(
24     Crypt_Point* ecP, // OUT: the preallocated point structure
25     TPMS_ECC_POINT* p // IN: the number to convert
26 )
27 {
28     if(p == NULL)
29         return NULL;
30
31     if(ecP != NULL)
32     {
33         return ExtEcc_PointFromBytes(
34             ecP, p->x.t.buffer, p->x.t.size, p->y.t.buffer, p->y.t.size);
35     }
36     return ecP; // will return NULL if ecP is NULL.
37 }
38
39 /** TpmEcc_PointTo2B()
40 // This function converts a BIG_POINT into a TPMS_ECC_POINT. A TPMS_ECC_POINT
41 // contains two TPM2B_ECC_PARAMETER values. The maximum size of the parameters
42 // is dependent on the maximum EC key size used in an implementation.
43 // The presumption is that the TPMS_ECC_POINT is large enough to hold 2 TPM2B
44 // values, each as large as a MAX_ECC_PARAMETER_BYTES
45 LIB_EXPORT BOOL TpmEcc_PointTo2B(
46     TPMS_ECC_POINT* p, // OUT: the converted 2B structure
47     const Crypt_Point* ecP, // IN: the values to be converted
48     const Crypt_EccCurve* E // IN: curve descriptor for the point
49 )
50 {
51     pAssert(p && ecP && E);
52     TPM_ECC_CURVE curveId = ExtEcc_CurveGetCurveId(E);
53     NUMBYTES size = CryptEccGetKeySizeForCurve(curveId);
54     size = (UINT16)BITS_TO_BYTES(size);
55     MemorySet(p, 0, sizeof(*p));
56     p->x.t.size = size;
57     p->y.t.size = size;
58     return ExtEcc_PointToBytes(
59         ecP, p->x.t.buffer, &p->x.t.size, p->y.t.buffer, &p->y.t.size);
60 }
61
62 #endif // ALG_ECC

```

7.161 /tpm/src/crypt/math/TpmMath_Debug.c

```

1  /*** Introduction
2  // This file contains debug utility functions to help testing Ecc.
3  #include "Tpm.h"
4  #include "TpmEcc_Util_fp.h"
5  #include "TpmMath_Debug_fp.h"
6
7  #if ALG_SM2
8  #   ifdef _SM2_SIGN_DEBUG
9
10 /*** SafeGetStringLength()
11 // self-implemented version of strlen_s. This is necessary because
12 // some environments don't have a C-runtime library, or are limited to
13 // C99, and strlen_s was standardized in C11.
14 static size_t SafeGetStringLength(const char* string, size_t maxsize)
15 {
16     // strlen_s has two boundary conditions:
17     // return 0 if pointer is nullptr, or
18     // maxsize if no null character is found.
19     if(string == NULL)
20         return 0;
21
22     const char* pos = string;
23     size_t      size = 0;
24
25     while(*pos != '\0' && size < maxsize)
26     {
27         pos++;
28         size++;
29     }
30     return size;
31 }
32
33 // convert from hex value. If invalid, result will be out of range.
34 static LIB_EXPORT BYTE FromHex(unsigned char c)
35 {
36     // hack for the ASCII characters we care about
37     BYTE upper = (c & (~0x20));
38     if(c >= '0' && c <= '9')
39         return c - '0';
40     else if(c >= 'A' && c <= 'F')
41         return c - 'A';
42
43     return 255;
44 }
45
46 /*** TpmEccDebug_FromHex()
47 // Convert a hex string into a Crypt_Int*. This is primarily used in debugging.
48 LIB_EXPORT Crypt_Int* TpmEccDebug_FromHex(
49     Crypt_Int*      bn,           // OUT:
50     const unsigned char* hex,     // IN:
51     size_t          maxsizeHex   // IN: maximum size of hex
52 )
53 {
54     // if value is larger than this, then fail
55     BYTE tempBuf[MAX_ECC_KEY_BYTES];
56     MemorySet(tempBuf, 0, sizeof(tempBuf));
57     ExtMath_SetWord(bn, 0);
58
59     size_t len = SafeGetStringLength(hex, maxsizeHex);
60     BOOL    OK = FALSE;
61     if((len % 2) == 0)
62     {
63         OK = TRUE;
64         for(size_t i = 0; i < len; i += 2)

```



```

65     {
66         BYTE highNibble = FromHex(*hex);
67         hex++;
68         BYTE lowNibble = FromHex(*hex);
69         hex++;
70         // unsigned, no need to check zero
71         if(highNibble > 15 || lowNibble > 15)
72         {
73             OK = FALSE;
74             break;
75         }
76         BYTE b = ((highNibble << 4) | lowNibble);
77         tempBuf[i / 2] = b;
78     }
79     if(OK)
80     {
81         ExtMath_IntFromBytes(bn, tempBuf, (NUMBYTES)(len / 2));
82     }
83 }
84
85 if(!OK)
86 {
87     // this should only be called in testing, so any
88     // errors are fatal.
89     FAIL(FATAL_ERROR_INTERNAL);
90 }
91 return bn;
92 }
93
94 /*** TpmEccDebug_HexEqual()
95 // This function compares a bignum value to a hex string.
96 // using TpmEcc namespace because code assumes the max size
97 // is correct for ECC.
98 // Return Type: BOOL
99 // TRUE(1) values equal
100 // FALSE(0) values not equal
101 BOOL TpmEccDebug_HexEqual(const Crypt_Int* bn, //IN: big number value
102                          const char* c //IN: character string number
103 )
104 {
105     CRYPT_ECC_NUM(bnC);
106     TpmEccDebug_FromHex(bnC, c, MAX_ECC_KEY_BYTES * 2 + 1);
107     return (ExtMath_UnsignedCmp(bn, bnC) == 0);
108 }
109 #endif // _SM2_SIGN_DEBUG
110 #endif // ALG_SM2

```

7.162 /tpm/src/crypt/math/TpmMath_Util.c

```

1  /*** Introduction
2  // This file contains utility functions to help using the external Math library
3  #include "Tpm.h"
4  #include "TpmMath_Util_fp.h"
5
6  /*** TpmMath_IntFrom2B()
7  // Convert an TPM2B to a Crypt_Int.
8  // If the input value does not exist, or the output does not exist, or the input
9  // will not fit into the output the function returns NULL
10 LIB_EXPORT Crypt_Int* TpmMath_IntFrom2B(Crypt_Int* value, // OUT:
11                                         const TPM2B* a2B // IN: number to convert
12 )
13 {
14     if(value != NULL && a2B != NULL)
15         return ExtMath_IntFromBytes(value, a2B->buffer, a2B->size);
16     return NULL;

```

```

17 }
18
19 /*** TpmMath_IntTo2B()
20 //
21 // Function to convert a Crypt_Int to TPM2B. The TPM2B bytes are
22 // always in big-endian ordering (most significant byte first). If 'size' is
23 // non-zero and less than required by `value` then an error is returned. If
24 // `size` is non-zero and larger than `value`, the result buffer is padded
25 // with zeros. If `size` is zero, then the TPM2B is assumed to be large enough
26 // for the data and a2b->size will be adjusted accordingly.
27 LIB_EXPORT BOOL TpmMath_IntTo2B(
28     const Crypt_Int* value, // IN: value to convert
29     TPM2B*          a2B,    // OUT: buffer for output
30     NUMBYTES        size   // IN: Size of output buffer - see comments.
31 )
32 {
33     // Set the output size
34     if(value && a2B)
35     {
36         a2B->size = size;
37         return ExtMath_IntToBytes(value, a2B->buffer, &a2B->size);
38     }
39     return FALSE;
40 }
41
42 /*** TpmMath_GetRandomBits()
43 // This function gets random bits for use in various places.
44 //
45 // One consequence of the generation scheme is that, if the number of bits requested
46 // is not a multiple of 8, then the high-order bits are set to zero. This would come
47 // into play when generating a 521-bit ECC key. A 66-byte (528-bit) value is
48 // generated and the high order 7 bits are masked off (CLEAR).
49 // In this situation, the highest order byte is the first byte (big-endian/TPM2B
50 // format)
51 // Return Type: BOOL
52 // TRUE(1)      success
53 // FALSE(0)     failure
54 LIB_EXPORT BOOL TpmMath_GetRandomBits(BYTE* pBuffer, size_t bits, RAND_STATE* rand)
55 {
56     // buffer is assumed to be large enough for the number of bits rounded up to
57     // bytes.
58     NUMBYTES byteCount = (NUMBYTES)BITS_TO_BYTES(bits);
59     if(DRBG_Generate(rand, pBuffer, byteCount) == byteCount)
60     {
61         // now flip the buffer order - this exists only to maintain
62         // compatibility with existing Known-value tests that expect the
63         // GetRandomInteger behavior of generating the value in little-endian
64         // order.
65         BYTE* pFrom = pBuffer + byteCount - 1;
66         BYTE* pTo   = pBuffer;
67         while(pTo < pFrom)
68         {
69             BYTE t = *pTo;
70             *pTo   = *pFrom;
71             *pFrom = t;
72             pTo++;
73             pFrom--;
74         }
75         // For a little-endian machine, the conversion is a straight byte
76         // reversal, done above. For a big-endian machine, we have to put the
77         // words in big-endian byte order. COMPATIBILITY NOTE: This code does
78         // not exactly reproduce the original code, because the original big-num
79         // code always generated data in units of crypt_word_t sizes. I.e. you
80         // couldn't generate just 9 bits for example. This revised version of
81         // the function could; and would generate 2 bytes with the first byte
82         // masked to 1 bit. In order to avoid running over the buffer when

```

```

82         // swapping crypt_uword_t blocks, this loop intentionally doesn't swap
83         // the last word if it is smaller than crypt_word_t size (which is the
84         // same as saying the buffer isn't an integral number of crypt_word_t
85         // units.) This is okay in this particular case because this whole
86         // block of swapping code is to maintain compatibility with existing
87         // KNOWN ANSWER TESTS, and said existing tests use sizes that this
88         // assumption is true for. Any new code with a different size where
89         // this last partial value isn't swapped will be creating a new KAT, and
90         // thus any (cryptographically valid) value is still random; swapping
91         // doesn't make a cryptographic random buffer more or less random, so
92         // the failure to swap is fine.
93 #if BIG_ENDIAN_TPM
94     crypt_uword_t* pTemp = pBuffer;
95     for(size_t t = 0; t < (byteCount / sizeof(crypt_uword_t)); t++)
96         *pTemp = SWAP_CRYPT_WORD(*pTemp);
97 #endif
98     // if the number of bits % 8 != 0, mask the high order (first) byte to the
99     // relevant number of bits
100     // bits % 8      desired mask    right-shift of 0xFF
101     // 0             0xFF           0 = (8 - 0) % 8
102     // 1             0x01           7 = (8 - 1) % 8
103     // 2             0x03           6 = (8 - 2) % 8
104     // ... etc ...
105     // 7             0x7F           1 = (8 - 7) % 8
106     int excessBits = bits % 8;
107     int shift      = (8 - excessBits) % 8;
108     BYTE mask      = ~(0xFF >> shift);
109     pBuffer[0]     = pBuffer[0] & mask;
110     return TRUE;
111 }
112 }
113
114 /*** TpmMath_GetRandomInteger()
115 // This function gets random bits for use in various places. To make sure that the
116 // number is generated in a portable format, it is created as a TPM2B and then
117 // converted to the internal format.
118 //
119 // One consequence of the generation scheme is that, if the number of bits requested
120 // is not a multiple of 8, then the high-order bits are set to zero. This would come
121 // into play when generating a 521-bit ECC key. A 66-byte (528-bit) value is
122 // generated and the high order 7 bits are masked off (CLEAR).
123 // Return Type: BOOL
124 // TRUE(1)      success
125 // FALSE(0)     failure
126 LIB_EXPORT BOOL TpmMath_GetRandomInteger(Crypt_Int* n, size_t bits, RAND_STATE* rand)
127 {
128     // Since this could be used for ECC key generation using the extra bits method,
129     // make sure that the value is large enough
130     TPM2B_TYPE(LARGEST, LARGEST_NUMBER + 8);
131     TPM2B_LARGEST large;
132     //
133     large.b.size = (UINT16)BITS_TO_BYTES(bits);
134     if(DRBG_Generate(rand, large.t.buffer, large.t.size) == large.t.size)
135     {
136         if(TpmMath_IntFrom2B(n, &large.b) != NULL)
137         {
138             if(ExtMath_MaskBits(n, (crypt_uword_t)bits))
139                 return TRUE;
140         }
141     }
142     return FALSE;
143 }
144
145 /*** BnGenerateRandomInRange()
146 // This function is used to generate a random number r in the range 1 <= r < limit.

```

```

147 // The function gets a random number of bits that is the size of limit. There is some
148 // some probability that the returned number is going to be greater than or equal
149 // to the limit. If it is, try again. There is no more than 50% chance that the
150 // next number is also greater, so try again. We keep trying until we get a
151 // value that meets the criteria. Since limit is very often a number with a LOT of
152 // high order ones, this rarely would need a second try.
153 // Return Type: BOOL
154 //     TRUE(1)          success
155 //     FALSE(0)         failure ('limit' is too small)
156 LIB_EXPORT BOOL TpmMath_GetRandomInRange(
157     Crypt_Int* dest, const Crypt_Int* limit, RAND_STATE* rand)
158 {
159     size_t bits = ExtMath_SizeInBits(limit);
160     //
161     if(bits < 2)
162     {
163         ExtMath_SetWord(dest, 0);
164         return FALSE;
165     }
166     else
167     {
168         while(TpmMath_GetRandomInteger(dest, bits, rand)
169             && (ExtMath_IsZero(dest) || (ExtMath_UnsignedCmp(dest, limit) >= 0)))
170             ;
171     }
172     return !g_inFailureMode;
173 }

```

7.163 /tpm/src/events/_TPM_Hash_Data.c

```

1  #include "Tpm.h"
2
3  // This function is called to process a _TPM_Hash_Data indication.
4  LIB_EXPORT void _TPM_Hash_Data(uint32_t dataSize, // IN: size of data to be extend
5                                unsigned char* data // IN: data buffer
6  )
7  {
8      UINT32 i;
9      HASH_OBJECT* hashObject;
10     TPMI_DH_PCR pcrHandle = TPMIsStarted() ? PCR_FIRST + DRTM_PCR
11                                         : PCR_FIRST + HCRTM_PCR;
12
13     // If there is no DRTM sequence object, then _TPM_Hash_Start
14     // was not called so this function returns without doing
15     // anything.
16     if(g_DRTMHandle == TPM_RH_UNASSIGNED)
17         return;
18
19     hashObject = (HASH_OBJECT*)HandleToObject(g_DRTMHandle);
20     pAssert(hashObject->attributes.eventSeq);
21
22     // For each of the implemented hash algorithms, update the digest with the
23     // data provided.
24     for(i = 0; i < HASH_COUNT; i++)
25     {
26         // make sure that the PCR is implemented for this algorithm
27         if(PcrIsAllocated(pcrHandle, hashObject->state.hashState[i].hashAlg))
28             // Update sequence object
29             CryptDigestUpdate(&hashObject->state.hashState[i], dataSize, data);
30     }
31
32     return;
33 }

```

7.164 /tpm/src/events/_TPM_Hash_End.c

```

1  #include "Tpm.h"
2
3  // This function is called to process a _TPM_Hash_End indication.
4  LIB_EXPORT void _TPM_Hash_End(void)
5  {
6      UINT32      i;
7      TPM2B_DIGEST digest;
8      HASH_OBJECT* hashObject;
9      TPMI_DH_PCR pcrHandle;
10
11     // If the DRTM handle is not being used, then either _TPM_Hash_Start has not
12     // been called, _TPM_Hash_End was previously called, or some other command
13     // was executed and the sequence was aborted.
14     if(g_DRTMHandle == TPM_RH_UNASSIGNED)
15         return;
16
17     // Get DRTM sequence object
18     hashObject = (HASH_OBJECT*)HandleToObject(g_DRTMHandle);
19
20     // Is this _TPM_Hash_End after Startup or before
21     if(TPMIsStarted())
22     {
23         // After
24
25         // Reset the DRTM PCR
26         PCRResetDynamics();
27
28         // Extend the DRTM PCR.
29         pcrHandle = PCR_FIRST + DRTM_PCR;
30
31         // DRTM sequence increments restartCount
32         gr.restartCount++;
33     }
34     else
35     {
36         pcrHandle = PCR_FIRST + HCRTM_PCR;
37         g_DrtmPreStartup = TRUE;
38     }
39
40     // Complete hash and extend PCR, or if this is an HCRTM, complete
41     // the hash, reset the H-CRTM register (PCR[0]) to 0...04, and then
42     // extend the H-CRTM data
43     for(i = 0; i < HASH_COUNT; i++)
44     {
45         TPMI_ALG_HASH hash = CryptHashGetAlgByIndex(i);
46         // make sure that the PCR is implemented for this algorithm
47         if(PcrIsAllocated(pcrHandle, hashObject->state.hashState[i].hashAlg))
48         {
49             // Complete hash
50             digest.t.size = CryptHashGetDigestSize(hash);
51             CryptHashEnd2B(&hashObject->state.hashState[i], &digest.b);
52
53             PcrDrtm(pcrHandle, hash, &digest);
54         }
55     }
56
57     // Flush sequence object.
58     FlushObject(g_DRTMHandle);
59
60     g_DRTMHandle = TPM_RH_UNASSIGNED;
61
62     return;
63 }

```

7.165 /tpm/src/events/_TPM_Hash_Start.c

```

1  #include "Tpm.h"
2
3  // This function is called to process a _TPM_Hash_Start indication.
4  LIB_EXPORT void _TPM_Hash_Start(void)
5  {
6      TPM_RC      result;
7      TPMI_DH_OBJECT handle;
8
9      // If a DRTM sequence object exists, free it up
10     if(g_DRTMHandle != TPM_RH_UNASSIGNED)
11     {
12         FlushObject(g_DRTMHandle);
13         g_DRTMHandle = TPM_RH_UNASSIGNED;
14     }
15
16     // Create an event sequence object and store the handle in global
17     // g_DRTMHandle. A TPM_RC_OBJECT_MEMORY error may be returned at this point
18     // The NULL value for the first parameter will cause the sequence structure to
19     // be allocated without being set as present. This keeps the sequence from
20     // being left behind if the sequence is terminated early.
21     result = ObjectCreateEventSequence(NULL, &g_DRTMHandle);
22
23     // If a free slot was not available, then free up a slot.
24     if(result != TPM_RC_SUCCESS)
25     {
26         // An implementation does not need to have a fixed relationship between
27         // slot numbers and handle numbers. To handle the general case, scan for
28         // a handle that is assigned and free it for the DRTM sequence.
29         // In the reference implementation, the relationship between handles and
30         // slots is fixed. So, if the call to ObjectCreateEventSequence()
31         // failed indicating that all slots are occupied, then the first handle we
32         // are going to check (TRANSIENT_FIRST) will be occupied. It will be freed
33         // so that it can be assigned for use as the DRTM sequence object.
34         for(handle = TRANSIENT_FIRST; handle < TRANSIENT_LAST; handle++)
35         {
36             // try to flush the first object
37             if(IsObjectPresent(handle))
38                 break;
39         }
40         // If the first call to find a slot fails but none of the slots is occupied
41         // then there's a big problem
42         pAssert(handle < TRANSIENT_LAST);
43
44         // Free the slot
45         FlushObject(handle);
46
47         // Try to create an event sequence object again. This time, we must
48         // succeed.
49         result = ObjectCreateEventSequence(NULL, &g_DRTMHandle);
50         if(result != TPM_RC_SUCCESS)
51             FAIL(FATAL_ERROR_INTERNAL);
52     }
53
54     return;
55 }

```

7.166 /tpm/src/events/_TPM_Init.c

```

1  #include "Tpm.h"
2  // TODO_RENAME_INC_FOLDER:platform_interface refers to the TPM_CoreLib platform
   interface
3  #include <platform_interface/prototypes/_TPM_Init_fp.h>
4

```



```

5  // This function is used to process a _TPM_Init indication.
6  LIB_EXPORT void _TPM_Init(void)
7  {
8      g_powerWasLost = g_powerWasLost | _plat__WasPowerLost();
9
10     #if SIMULATION && DEBUG
11         // If power was lost and this was a simulation, put canary in RAM used by NV
12         // so that uninitialized memory can be detected more easily
13         if(g_powerWasLost)
14         {
15             memset(&gc, 0xbb, sizeof(gc));
16             memset(&gr, 0xbb, sizeof(gr));
17             memset(&gp, 0xbb, sizeof(gp));
18             memset(&go, 0xbb, sizeof(go));
19         }
20     #endif
21
22     #if ALLOW_FORCE_FAILURE_MODE
23         // Clear the flag that forces failure on self-test
24         g_forceFailureMode = FALSE;
25     #endif
26
27     // Disable the tick processing
28     #if ACT_SUPPORT
29         _plat__ACT_EnableTicks(FALSE);
30     #endif
31
32     // Set initialization state
33     TPMInit();
34
35     // Set g_DRTMHandle as unassigned
36     g_DRTMHandle = TPM_RH_UNASSIGNED;
37
38     // No H-CRTM, yet.
39     g_DrtmPreStartup = FALSE;
40
41     // Initialize the NvEnvironment.
42     g_nvOk = NvPowerOn();
43
44     // Initialize cryptographic functions
45     g_inFailureMode = (g_nvOk == FALSE) || (CryptInit() == FALSE);
46     if(!g_inFailureMode)
47     {
48         // Load the persistent data
49         NvReadPersistent();
50
51         // Load the orderly data (clock and DRBG state).
52         // If this is not done here, things break
53         NvRead(&go, NV_ORDERLY_DATA, sizeof(go));
54
55         // Start clock. Need to do this after NV has been restored.
56         TimePowerOn();
57     }
58     return;
59 }

```

7.167 /tpm/src/main/CommandDispatcher.c

```

1  /* Includes and Typedefs
2  #include "Tpm.h"
3  #include "Marshal.h"
4
5  #if TABLE_DRIVEN_DISPATCH || TABLE_DRIVEN_MARSHAL
6
7  typedef TPM_RC (NoFlagFunction) (void* target, BYTE** buffer, INT32* size);

```

```

8  typedef TPM_RC(FlagFunction)(void* target, BYTE** buffer, INT32* size, BOOL flag);
9
10 typedef FlagFunction* UNMARSHAL_t;
11
12 typedef INT16(MarshalFunction)(void* source, BYTE** buffer, INT32* size);
13 typedef MarshalFunction* MARSHAL_t;
14
15 typedef TPM_RC(COMMAND_NO_ARGS)(void);
16 typedef TPM_RC(COMMAND_IN_ARG)(void* in);
17 typedef TPM_RC(COMMAND_OUT_ARG)(void* out);
18 typedef TPM_RC(COMMAND_INOUT_ARG)(void* in, void* out);
19
20 typedef union COMMAND_t
21 {
22     COMMAND_NO_ARGS*   noArgs;
23     COMMAND_IN_ARG*    inArg;
24     COMMAND_OUT_ARG*   outArg;
25     COMMAND_INOUT_ARG* inOutArg;
26 } COMMAND_t;
27
28 // This structure is used by ParseHandleBuffer() and CommandDispatcher(). The
29 // parameters in this structure are unique for each command. The parameters are:
30 // command      holds the address of the command processing function that is called
31 //              by Command Dispatcher
32 // inSize       This is the size of the command-dependent input structure. The
33 //              input structure holds the unmarshaled handles and command
34 //              parameters. If the command takes no arguments (handles or
35 //              parameters) then inSize will have a value of 0.
36 // outSize      This is the size of the command-dependent output structure. The
37 //              output structure holds the results of the command in an unmarshaled
38 //              form. When command processing is completed, these values are
39 //              marshaled into the output buffer. It is always the case that the
40 //              unmarshaled version of an output structure is larger than the
41 //              marshaled version. This is because the marshaled version contains
42 //              the exact same number of significant bytes but with padding removed.
43 // typesOffsets This parameter points to the list of data types that are to be
44 //              marshaled or unmarshaled. The list of types follows the 'offsets'
45 //              array. The offsets array is variable sized so the typesOffset field
46 //              is necessary for the handle and command processing to be able to
47 //              find the types that are being handled. The 'offsets' array may be
48 //              empty. The 'types' structure is described below.
49 // offsets      This is an array of offsets of each of the parameters in the
50 //              command or response. When processing the command parameters (not
51 //              handles) the list contains the offset of the next parameter. For
52 //              example, if the first command parameter has a size of 4 and there is
53 //              a second command parameter, then the offset would be 4, indicating
54 //              that the second parameter starts at 4. If the second parameter has
55 //              a size of 8, and there is a third parameter, then the second entry
56 //              in offsets is 12 (4 for the first parameter and 8 for the second).
57 //              An offset value of 0 in the list indicates the start of the response
58 //              parameter list. When CommandDispatcher hits this value, it will stop
59 //              unmarshaling the parameters and call 'command'. If a command has no
60 //              response parameters and only one command parameter, then offsets can
61 //              be an empty list.
62
63 typedef struct COMMAND_DESCRIPTOR_t
64 {
65     COMMAND_t command;      // Address of the command
66     UINT16 inSize;          // Maximum size of the input structure
67     UINT16 outSize;         // Maximum size of the output structure
68     UINT16 typesOffset;     // address of the types field
69     UINT16 offsets[1];
70 } COMMAND_DESCRIPTOR_t;
71
72 // The 'types' list is an encoded byte array. The byte value has two parts. The most
73 // significant bit is used when a parameter takes a flag and indicates if the flag

```

```

74 // should be SET or not. The remaining 7 bits are an index into an array of
75 // addresses of marshaling and unmarshaling functions.
76 // The array of functions is divided into 6 sections with a value assigned
77 // to denote the start of that section (and the end of the previous section). The
78 // defined offset values for each section are:
79 // 0                                unmarshaling for handles that do not take flags
80 // HANDLE_FIRST_FLAG_TYPE          unmarshaling for handles that take flags
81 // PARAMETER_FIRST_TYPE            unmarshaling for parameters that do not take flags
82 // PARAMETER_FIRST_FLAG_TYPE       unmarshaling for parameters that take flags
83 // PARAMETER_LAST_TYPE + 1         marshaling for handles
84 // RESPONSE_PARAMETER_FIRST_TYPE   marshaling for parameters
85 // RESPONSE_PARAMETER_LAST_TYPE    is the last value in the list of marshaling and
86 //                                unmarshaling functions.
87 //
88 // The types list is constructed with a byte of 0xff at the end of the command
89 // parameters and with an 0xff at the end of the response parameters.
90
91 # if COMPRESSED_LISTS
92 #   define PAD_LIST 0
93 # else
94 #   define PAD_LIST 1
95 # endif
96 # define _COMMAND_TABLE_DISPATCH_
97 # include "CommandDispatchData.h"
98
99 # define TEST_COMMAND TPM_CC_Startup
100
101 # define NEW_CC
102
103 #else
104
105 # include "Commands.h"
106
107 #endif
108
109 /* Marshal/Unmarshal Functions
110
111 /** ParseHandleBuffer()
112 // This is the table-driven version of the handle buffer unmarshaling code
113 TPM_RC
114 ParseHandleBuffer(COMMAND* command)
115 {
116     TPM_RC result;
117 #if TABLE_DRIVEN_DISPATCH || TABLE_DRIVEN_MARSHAL
118     COMMAND_DESCRIPTOR_t* desc;
119     BYTE* types;
120     BYTE type;
121     BYTE dType;
122
123     // Make sure that nothing strange has happened
124     pAssert(
125         command->index < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t));
126     // Get the address of the descriptor for this command
127     desc = s_CommandDataArray[command->index];
128
129     pAssert(desc != NULL);
130     // Get the associated list of unmarshaling data types.
131     types = &(BYTE*)desc[desc->typesOffset];
132
133     // if(s_ccAttr[commandIndex].commandIndex == TEST_COMMAND)
134     //     commandIndex = commandIndex;
135     // No handles yet
136     command->handleNum = 0;
137
138     // Get the first type value
139     for(type = *types++;

```

```

140         // check each byte to make sure that we have not hit the start
141         // of the parameters
142         (dType = (type & 0x7F)) < PARAMETER_FIRST_TYPE;
143         // get the next type
144         type = *types++;
145     {
146 # if TABLE_DRIVEN_MARSHAL
147         marshalIndex_t index;
148         index = unmarshalArray[dType] | ((type & 0x80) ? NULL_FLAG : 0);
149         result = Unmarshal(index,
150                             &(command->handles[command->handleNum]),
151                             &command->parameterBuffer,
152                             &command->parameterSize);
153 # else
154         // See if unmarshaling of this handle type requires a flag
155         if(dType < HANDLE_FIRST_FLAG_TYPE)
156         {
157             // Look up the function to do the unmarshaling
158             NoFlagFunction* f = (NoFlagFunction*)unmarshalArray[dType];
159             // call it
160             result = f(&(command->handles[command->handleNum]),
161                     &command->parameterBuffer,
162                     &command->parameterSize);
163         }
164         else
165         {
166             // Look up the function
167             FlagFunction* f = unmarshalArray[dType];
168             // Call it setting the flag to the appropriate value
169             result = f(&(command->handles[command->handleNum]),
170                     &command->parameterBuffer,
171                     &command->parameterSize,
172                     (type & 0x80) != 0);
173         }
174     }
175 # endif
176
177     // Got a handle
178     // We do this first so that the match for the handle offset of the
179     // response code works correctly.
180     command->handleNum += 1;
181     if(result != TPM_RC_SUCCESS)
182     // if the unmarshaling failed, return the response code with the
183     // handle indication set
184     return result + TPM_RC_H + (command->handleNum * TPM_RC_1);
185 }
186 #else
187 BYTE**      handleBufferStart      = &command->parameterBuffer;
188 INT32*      bufferRemainingSize    = &command->parameterSize;
189 TPM_HANDLE* handles                = &command->handles[0];
190 UINT32*     handleCount            = &command->handleNum;
191 *handleCount = 0;
192 switch(command->code)
193 {
194 # include "HandleProcess.h"
195 # undef handles
196     default:
197         FAIL(FATAL_ERROR_INTERNAL);
198         break;
199 }
200 #endif
201 return TPM_RC_SUCCESS;
202 }
203
204 /** CommandDispatcher()

```

```

206 // Function to unmarshal the command parameters, call the selected action code, and
207 // marshal the response parameters.
208 TPM_RC
209 CommandDispatcher(COMMAND* command)
210 {
211     #if !TABLE_DRIVEN_DISPATCH || TABLE_DRIVEN_MARSHAL
212         TPM_RC result;
213         BYTE** paramBuffer = &command->parameterBuffer;
214         INT32* paramBufferSize = &command->parameterSize;
215         BYTE** responseBuffer = &command->responseBuffer;
216         INT32* respParamSize = &command->parameterSize;
217         INT32 rSize;
218         TPM_HANDLE* handles = &command->handles[0];
219         //
220         command->handleNum = 0; // The command-specific code knows how
221                                 // many handles there are. This is for
222                                 // cataloging the number of response
223                                 // handles
224         MemoryIoBufferAllocationReset(); // Initialize so that allocation will
225                                         // work properly
226         switch (GetCommandCode (command->index))
227         {
228             # include "CommandDispatcher.h"
229
230             default:
231                 FAIL(FATAL_ERROR_INTERNAL);
232                 break;
233         }
234     Exit:
235         MemoryIoBufferZero();
236         return result;
237     #else
238         COMMAND_DESCRIPTOR_t* desc;
239         BYTE* types;
240         BYTE type;
241         UINT16* offsets;
242         UINT16 offset = 0;
243         UINT32 maxInSize;
244         BYTE* commandIn;
245         INT32 maxOutSize;
246         BYTE* commandOut;
247         COMMAND_t cmd;
248         TPM_HANDLE* handles;
249         UINT32 hasInParameters = 0;
250         BOOL hasOutParameters = FALSE;
251         UINT32 pNum = 0;
252         BYTE dType; // dispatch type
253         TPM_RC result;
254         //
255         // Get the address of the descriptor for this command
256         pAssert(
257             command->index < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t));
258         desc = s_CommandDataArray[command->index];
259
260         // Get the list of parameter types for this command
261         pAssert(desc != NULL);
262         types = &(BYTE*)desc[desc->typesOffset];
263
264         // Get a pointer to the list of parameter offsets
265         offsets = &desc->offsets[0];
266         // pointer to handles
267         handles = command->handles;
268
269         // Get the size required to hold all the unmarshaled parameters for this command
270         maxInSize = desc->inSize;
271         // and the size of the output parameter structure returned by this command

```

```

272     maxOutSize = desc->outSize;
273
274     MemoryIoBufferAllocationReset();
275     // Get a buffer for the input parameters
276     commandIn = MemoryGetInBuffer(maxInSize);
277     // And the output parameters
278     commandOut = (BYTE*)MemoryGetOutBuffer((UINT32)maxOutSize);
279
280     // Get the address of the action code dispatch
281     cmd = desc->command;
282
283     // Copy any handles into the input buffer
284     for(type = *types++; (type & 0x7F) < PARAMETER_FIRST_TYPE; type = *types++)
285     {
286         // 'offset' was initialized to zero so the first unmarshaling will always
287         // be to the start of the data structure
288         *(TPM_HANDLE*)&(commandIn[offset]) = *handles++;
289         // This check is used so that we don't have to add an additional offset
290         // value to the offsets list to correspond to the stop value in the
291         // command parameter list.
292         if(*types != 0xFF)
293             offset = *offsets++;
294         // maxInSize -= sizeof(TPM_HANDLE);
295         hasInParameters++;
296     }
297     // Exit loop with type containing the last value read from types
298     // maxInSize has the amount of space remaining in the command action input
299     // buffer. Make sure that we don't have more data to unmarshal than is going to
300     // fit.
301
302     // type contains the last value read from types so it is not necessary to
303     // reload it, which is good because *types now points to the next value
304     for(; (dType = (type & 0x7F)) <= PARAMETER_LAST_TYPE; type = *types++)
305     {
306         pNum++;
307     # if TABLE_DRIVEN_UNMARSHAL
308     {
309         marshalIndex t index = unmarshalArray[dType];
310         index |= (type & 0x80) ? NULL_FLAG : 0;
311         result = Unmarshal(index,
312                             &commandIn[offset],
313                             &command->parameterBuffer,
314                             &command->parameterSize);
315     }
316     # else
317     if(dType < PARAMETER_FIRST_FLAG_TYPE)
318     {
319         NoFlagFunction* f = (NoFlagFunction*)unmarshalArray[dType];
320         result = f(&commandIn[offset],
321                   &command->parameterBuffer,
322                   &command->parameterSize);
323     }
324     else
325     {
326         FlagFunction* f = unmarshalArray[dType];
327         result = f(&commandIn[offset],
328                   &command->parameterBuffer,
329                   &command->parameterSize,
330                   (type & 0x80) != 0);
331     }
332     # endif
333     if(result != TPM_RC_SUCCESS)
334     {
335         result += TPM_RC_P + (TPM_RC_1 * pNum);
336         goto Exit;
337     }

```



```

338     // This check is used so that we don't have to add an additional offset
339     // value to the offsets list to correspond to the stop value in the
340     // command parameter list.
341     if(*types != 0xFF)
342         offset = *offsets++;
343     hasInParameters++;
344 }
345 // Should have used all the bytes in the input
346 if(command->parameterSize != 0)
347 {
348     result = TPM_RC_SIZE;
349     goto Exit;
350 }
351
352 // The command parameter unmarshaling stopped when it hit a value that was out
353 // of range for unmarshaling values and left *types pointing to the first
354 // marshaling type. If that type happens to be the STOP value, then there
355 // are no response parameters. So, set the flag to indicate if there are
356 // output parameters.
357 hasOutParameters = *types != 0xFF;
358
359 // There are four cases for calling, with and without input parameters and with
360 // and without output parameters.
361 if(hasInParameters > 0)
362 {
363     if(hasOutParameters)
364         result = cmd.inOutArg(commandIn, commandOut);
365     else
366         result = cmd.inArg(commandIn);
367 }
368 else
369 {
370     if(hasOutParameters)
371         result = cmd.outArg(commandOut);
372     else
373         result = cmd.noArgs();
374 }
375 if(result != TPM_RC_SUCCESS)
376     goto Exit;
377
378 // Offset in the marshaled output structure
379 offset = 0;
380
381 // Process the return handles, if any
382 command->handleNum = 0;
383
384 // Could make this a loop to process output handles but there is only ever
385 // one handle in the outputs (for now).
386 type = *types++;
387 if((dType = (type & 0x7F)) < RESPONSE_PARAMETER_FIRST_TYPE)
388 {
389     // The out->handle value was referenced as TPM_HANDLE in the
390     // action code so it has to be properly aligned.
391     command->handles[command->handleNum++] =
392         *((TPM_HANDLE*) &(commandOut[offset]));
393     maxOutSize -= sizeof(UINT32);
394     type = *types++;
395     offset = *offsets++;
396 }
397 // Use the size of the command action output buffer as the maximum for the
398 // number of bytes that can get marshaled. Since the marshaling code has
399 // no pointers to data, all of the data being returned has to be in the
400 // command action output buffer. If we try to marshal more bytes than
401 // could fit into the output buffer, we need to fail.
402 for(; (dType = (type & 0x7F)) <= RESPONSE_PARAMETER_LAST_TYPE && !g_inFailureMode;
403     type = *types++)

```

```

404     {
405     # if TABLE_DRIVEN_MARSHAL
406         marshalIndex_t index = marshalArray[dType];
407         command->parameterSize += Marshal(
408             index, &commandOut[offset], &command->responseBuffer, &maxOutSize);
409     # else
410         const MARSHAL_t f = marshalArray[dType];
411
412         command->parameterSize +=
413             f(&commandOut[offset], &command->responseBuffer, &maxOutSize);
414     # endif
415         offset = *offsets++;
416     }
417     result = (maxOutSize < 0) ? TPM_RC_FAILURE : TPM_RC_SUCCESS;
418 Exit:
419     MemoryIoBufferZero();
420     return result;
421 #endif
422 }

```

7.168 /tpm/src/main/ExecCommand.c

```

1  /** Introduction
2  //
3  // This file contains the entry function ExecuteCommand() which provides the main
4  // control flow for TPM command execution.
5
6  /** Includes
7
8  #include "Tpm.h"
9  #include "Marshal.h"
10 // TODO_RENAME_INC_FOLDER:platform_interface refers to the TPM_CoreLib platform
11 // interface
12 #include <platform_interface/prototypes/ExecCommand_fp.h>
13
14 // Uncomment this next #include if doing static command/response buffer sizing
15 // #include "CommandResponseSizes_fp.h"
16
17 /** ExecuteCommand()
18 //
19 // The function performs the following steps.
20 //
21 // a) Parses the command header from input buffer.
22 // b) Calls ParseHandleBuffer() to parse the handle area of the command.
23 // c) Validates that each of the handles references a loaded entity.
24 // d) Calls ParseSessionBuffer() to:
25 //     1) unmarshal and parse the session area;
26 //     2) check the authorizations; and
27 //     3) when necessary, decrypt a parameter.
28 // e) Calls CommandDispatcher() to:
29 //     1) unmarshal the command parameters from the command buffer;
30 //     2) call the routine that performs the command actions; and
31 //     3) marshal the responses into the response buffer.
32 // f) If any error occurs in any of the steps above create the error response
33 //     and return.
34 // g) Calls BuildResponseSession() to:
35 //     1) when necessary, encrypt a parameter
36 //     2) build the response authorization sessions
37 //     3) update the audit sessions and nonces
38 // h) Calls BuildResponseHeader() to complete the construction of the response.
39 //
40 // 'responseSize' is set by the caller to the maximum number of bytes available in
41 // the output buffer. ExecuteCommand will adjust the value and return the number
42 // of bytes placed in the buffer.

```

```

43 // 'response' is also set by the caller to indicate the buffer into which
44 // ExecuteCommand is to place the response.
45 //
46 // 'request' and 'response' may point to the same buffer
47 //
48 // Note: As of February, 2016, the failure processing has been moved to the
49 // platform-specific code. When the TPM code encounters an unrecoverable failure, it
50 // will SET g_inFailureMode and call _plat_Fail(). That function should not return
51 // but may call ExecuteCommand().
52 //
53 LIB_EXPORT void ExecuteCommand(
54     uint32_t      requestSize,    // IN: command buffer size
55     unsigned char* request,       // IN: command buffer
56     uint32_t*     responseSize,   // IN/OUT: response buffer size
57     unsigned char** response      // IN/OUT: response buffer
58 )
59 {
60     // Command local variables
61     UINT32 commandSize;
62     COMMAND command;
63
64     // Response local variables
65     UINT32 maxResponse = *responseSize;
66     TPM_RC result; // return code for the command
67
68     // This next function call is used in development to size the command and response
69     // buffers. The values printed are the sizes of the internal structures and
70     // not the sizes of the canonical forms of the command response structures. Also,
71     // the sizes do not include the tag, command.code, requestSize, or the
authorization
72     // fields.
73     //CommandResponseSizes();
74     // Set flags for NV access state. This should happen before any other
75     // operation that may require a NV write. Note, that this needs to be done
76     // even when in failure mode. Otherwise, g_updateNV would stay SET while in
77     // Failure mode and the NV would be written on each call.
78     g_updateNV = UT_NONE;
79     g_clearOrderly = FALSE;
80     if(g_inFailureMode)
81     {
82         // Do failure mode processing
83         TpmFailureMode(requestSize, request, responseSize, response);
84         return;
85     }
86     // Query platform to get the NV state. The result state is saved internally
87     // and will be reported by NvIsAvailable(). The reference code requires that
88     // accessibility of NV does not change during the execution of a command.
89     // Specifically, if NV is available when the command execution starts and then
90     // is not available later when it is necessary to write to NV, then the TPM
91     // will go into failure mode.
92     NvCheckState();
93
94     // Due to the limitations of the simulation, TPM clock must be explicitly
95     // synchronized with the system clock whenever a command is received.
96     // This function call is not necessary in a hardware TPM. However, taking
97     // a snapshot of the hardware timer at the beginning of the command allows
98     // the time value to be consistent for the duration of the command execution.
99     TimeUpdateToCurrent();
100
101     // Any command through this function will unceremoniously end the
102     // _TPM_Hash_Data/_TPM_Hash_End sequence.
103     if(g_DRTMHandle != TPM_RH_UNASSIGNED)
104         ObjectTerminateEvent();
105
106     // Get command buffer size and command buffer.
107     command.parameterBuffer = request;

```

```

108     command.parameterSize = requestSize;
109
110     // Parse command header: tag, commandSize and command.code.
111     // First parse the tag. The unmarshaling routine will validate
112     // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
113     result = TPMI_ST_COMMAND_TAG_Unmarshal(
114         &command.tag, &command.parameterBuffer, &command.parameterSize);
115     if(result != TPM_RC_SUCCESS)
116         goto Cleanup;
117     // Unmarshal the commandSize indicator.
118     result = UINT32_Unmarshal(
119         &commandSize, &command.parameterBuffer, &command.parameterSize);
120     if(result != TPM_RC_SUCCESS)
121         goto Cleanup;
122     // On a TPM that receives bytes on a port, the number of bytes that were
123     // received on that port is requestSize it must be identical to commandSize.
124     // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
125     // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
126     // as the input processing (the function that receives the command bytes and
127     // places them in the input buffer) would likely have the input truncated when
128     // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
129     if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
130     {
131         result = TPM_RC_COMMAND_SIZE;
132         goto Cleanup;
133     }
134     // Unmarshal the command code.
135     result = TPM_CC_Unmarshal(
136         &command.code, &command.parameterBuffer, &command.parameterSize);
137     if(result != TPM_RC_SUCCESS)
138         goto Cleanup;
139     // Check to see if the command is implemented.
140     command.index = CommandCodeToCommandIndex(command.code);
141     if(UNIMPLEMENTED_COMMAND_INDEX == command.index)
142     {
143         result = TPM_RC_COMMAND_CODE;
144         goto Cleanup;
145     }
146     #if FIELD_UPGRADE_IMPLEMENTED == YES
147     // If the TPM is in FUM, then the only allowed command is
148     // TPM_CC_FieldUpgradeData.
149     if(IsFieldUpgradeMode() && (command.code != TPM_CC_FieldUpgradeData))
150     {
151         result = TPM_RC_UPGRADE;
152         goto Cleanup;
153     }
154     else
155     #endif
156     // Excepting FUM, the TPM only accepts TPM2_Startup() after
157     // TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
158     // is no longer allowed.
159     if((!TPMIsStarted() && command.code != TPM_CC_Startup)
160        || (TPMIsStarted() && command.code == TPM_CC_Startup))
161     {
162         result = TPM_RC_INITIALIZE;
163         goto Cleanup;
164     }
165     // Start regular command process.
166     NvIndexCacheInit();
167     // Parse Handle buffer.
168     result = ParseHandleBuffer(&command);
169     if(result != TPM_RC_SUCCESS)
170         goto Cleanup;
171     // All handles in the handle area are required to reference TPM-resident
172     // entities.
173     result = EntityGetLoadStatus(&command);

```

```

174     if(result != TPM_RC_SUCCESS)
175         goto Cleanup;
176     // Authorization session handling for the command.
177     ClearCpRpHashes(&command);
178     if(command.tag == TPM_ST_SESSIONS)
179     {
180         // Find out session buffer size.
181         result = UINT32_Unmarshal((UINT32*)&command.authSize,
182                                 &command.parameterBuffer,
183                                 &command.parameterSize);
184         if(result != TPM_RC_SUCCESS)
185             goto Cleanup;
186         // Perform sanity check on the unmarshaled value. If it is smaller than
187         // the smallest possible session or larger than the remaining size of
188         // the command, then it is an error. NOTE: This check could pass but the
189         // session size could still be wrong. That will be determined after the
190         // sessions are unmarshaled.
191         if(command.authSize < 9 || command.authSize > command.parameterSize)
192         {
193             result = TPM_RC_SIZE;
194             goto Cleanup;
195         }
196         command.parameterSize -= command.authSize;
197
198         // The actions of ParseSessionBuffer() are described in the introduction.
199         // As the sessions are parsed command.parameterBuffer is advanced so, on a
200         // successful return, command.parameterBuffer should be pointing at the
201         // first byte of the parameters.
202         result = ParseSessionBuffer(&command);
203         if(result != TPM_RC_SUCCESS)
204             goto Cleanup;
205     }
206     else
207     {
208         command.authSize = 0;
209         // The command has no authorization sessions.
210         // If the command requires authorizations, then CheckAuthNoSession() will
211         // return an error.
212         result = CheckAuthNoSession(&command);
213         if(result != TPM_RC_SUCCESS)
214             goto Cleanup;
215     }
216     // Set up the response buffer pointers. CommandDispatch will marshal the
217     // response parameters starting at the address in command.responseBuffer.
218     // *response = MemoryGetResponseBuffer(command.index);
219     // leave space for the command header
220     command.responseBuffer = *response + STD_RESPONSE_HEADER;
221
222     // leave space for the parameter size field if needed
223     if(command.tag == TPM_ST_SESSIONS)
224         command.responseBuffer += sizeof(UINT32);
225     if(IsHandleInResponse(command.index))
226         command.responseBuffer += sizeof(TPM_HANDLE);
227
228     // CommandDispatcher returns a response handle buffer and a response parameter
229     // buffer if it succeeds. It will also set the parameterSize field in the
230     // buffer if the tag is TPM_RC_SESSIONS.
231     result = CommandDispatcher(&command);
232     if(result != TPM_RC_SUCCESS)
233         goto Cleanup;
234
235     // Build the session area at the end of the parameter area.
236     result = BuildResponseSession(&command);
237     if(result != TPM_RC_SUCCESS)
238     {
239         goto Cleanup;

```



```

240     }
241
242 Cleanup:
243     if(g_clearOrderly == TRUE && NV_IS_ORDERLY)
244     {
245 #if USE_DA_USED
246         gp.orderlyState = g_daUsed ? SU_DA_USED_VALUE : SU_NONE_VALUE;
247 #else
248         gp.orderlyState = SU_NONE_VALUE;
249 #endif
250         NV_SYNC_PERSISTENT(orderlyState);
251     }
252     // This implementation loads an "evict" object to a transient object slot in
253     // RAM whenever an "evict" object handle is used in a command so that the
254     // access to any object is the same. These temporary objects need to be
255     // cleared from RAM whether the command succeeds or fails.
256     ObjectCleanupEvict();
257
258     // The parameters and sessions have been marshaled. Now tack on the header and
259     // set the sizes
260     BuildResponseHeader(&command, *response, result);
261
262     // Try to commit all the writes to NV if any NV write happened during this
263     // command execution. This check should be made for both succeeded and failed
264     // commands, because a failed one may trigger a NV write in DA logic as well.
265     // This is the only place in the command execution path that may call the NV
266     // commit. If the NV commit fails, the TPM should be put in failure mode.
267     if((g_updateNV != UT_NONE) && !g_inFailureMode)
268     {
269         if(g_updateNV == UT_ORDERLY)
270             NvUpdateIndexOrderlyData();
271         if(!NvCommit())
272             FAIL(FATAL_ERROR_INTERNAL);
273         g_updateNV = UT_NONE;
274     }
275     pAssert((UINT32)command.parameterSize <= maxResponse);
276
277     // Clear unused bits in response buffer.
278     MemorySet(*response + *responseSize, 0, maxResponse - *responseSize);
279
280     // as a final act, and not before, update the response size.
281     *responseSize = (UINT32)command.parameterSize;
282
283     return;
284 }

```

7.169 /tpm/src/main/SessionProcess.c

```

1  /** Introduction
2  // This file contains the subsystem that process the authorization sessions
3  // including implementation of the Dictionary Attack logic. ExecCommand() uses
4  // ParseSessionBuffer() to process the authorization session area of a command and
5  // BuildResponseSession() to create the authorization session area of a response.
6
7  /** Includes and Data Definitions
8
9  #define SESSION_PROCESS_C
10
11 #include "Tpm.h"
12 #include "ACT.h"
13 #include "Marshal.h"
14
15 //
16 /** Authorization Support Functions
17 //

```



```

18
19  /*** IsDAExempted()
20  // This function indicates if a handle is exempted from DA logic.
21  // A handle is exempted if it is:
22  //  a) a primary seed handle;
23  //  b) an object with noDA bit SET;
24  //  c) an NV Index with TPMA_NV_NO_DA bit SET; or
25  //  d) a PCR handle.
26  //
27  // Return Type: BOOL
28  //     TRUE(1)      handle is exempted from DA logic
29  //     FALSE(0)     handle is not exempted from DA logic
30  BOOL IsDAExempted(TPM_HANDLE handle // IN: entity handle
31  )
32  {
33      BOOL result = FALSE;
34      //
35      switch(HandleGetType(handle))
36      {
37          case TPM_HT_PERMANENT:
38              // All permanent handles, other than TPM_RH_LOCKOUT, are exempt from
39              // DA protection.
40              result = (handle != TPM_RH_LOCKOUT);
41              break;
42              // When this function is called, a persistent object will have been loaded
43              // into an object slot and assigned a transient handle.
44          case TPM_HT_TRANSIENT:
45              {
46                  TPMA_OBJECT attributes = ObjectGetPublicAttributes(handle);
47                  result = IS_ATTRIBUTE(attributes, TPMA_OBJECT, noDA);
48                  break;
49              }
50          case TPM_HT_NV_INDEX:
51              {
52                  NV_INDEX* nvIndex = NvGetIndexInfo(handle, NULL);
53                  result = IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, NO_DA);
54                  break;
55              }
56          case TPM_HT_PCR:
57              // PCRs are always exempted from DA.
58              result = TRUE;
59              break;
60          default:
61              break;
62      }
63      return result;
64  }
65
66  /*** IncrementLockout()
67  // This function is called after an authorization failure that involves use of
68  // an authValue. If the entity referenced by the handle is not exempt from DA
69  // protection, then the failedTries counter will be incremented.
70  //
71  // Return Type: TPM_RC
72  //     TPM_RC_AUTH_FAIL    authorization failure that caused DA lockout to increment
73  //     TPM_RC_BAD_AUTH     authorization failure did not cause DA lockout to
74  //                          increment
75  static TPM_RC IncrementLockout(UINT32 sessionIndex)
76  {
77      TPM_HANDLE handle = s_associatedHandles[sessionIndex];
78      TPM_HANDLE sessionHandle = s_sessionHandles[sessionIndex];
79      SESSION* session = NULL;
80      //
81      // Don't increment lockout unless the handle associated with the session
82      // is DA protected or the session is bound to a DA protected entity.
83      if(sessionHandle == TPM_RS_PW)

```

```

84     {
85         if(IsDAExempted(handle))
86             return TPM_RC_BAD_AUTH;
87     }
88     else
89     {
90         session = SessionGet(sessionHandle);
91         // If the session is bound to lockout, then use that as the relevant
92         // handle. This means that an authorization failure with a bound session
93         // bound to lockoutAuth will take precedence over any other
94         // lockout check
95         if(session->attributes.isLockoutBound == SET)
96             handle = TPM_RH_LOCKOUT;
97         if(session->attributes.isDaBound == CLEAR
98             && (IsDAExempted(handle) || session->attributes.includeAuth == CLEAR))
99             // If the handle was changed to TPM_RH_LOCKOUT, this will not return
100             // TPM_RC_BAD_AUTH
101             return TPM_RC_BAD_AUTH;
102     }
103     if(handle == TPM_RH_LOCKOUT)
104     {
105         pAssert(gp.lockOutAuthEnabled == TRUE);
106
107         // lockout is no longer enabled
108         gp.lockOutAuthEnabled = FALSE;
109
110         // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
111         // the lockout authorization will be reset at startup.
112         if(gp.lockoutRecovery != 0)
113         {
114             if(NV_IS_AVAILABLE)
115                 // Update NV.
116                 NV_SYNC_PERSISTENT(lockOutAuthEnabled);
117             else
118                 // No NV access for now. Put the TPM in pending mode.
119                 s_DAPendingOnNV = TRUE;
120         }
121     }
122     else
123     {
124         if(gp.recoveryTime != 0)
125         {
126             gp.failedTries++;
127             if(NV_IS_AVAILABLE)
128                 // Record changes to NV. NvWrite will SET g_updateNV
129                 NV_SYNC_PERSISTENT(failedTries);
130             else
131                 // No NV access for now. Put the TPM in pending mode.
132                 s_DAPendingOnNV = TRUE;
133         }
134     }
135     // Register a DA failure and reset the timers.
136     DARegisterFailure(handle);
137
138     return TPM_RC_AUTH_FAIL;
139 }
140
141 /*** IsSessionBindEntity()
142 // This function indicates if the entity associated with the handle is the entity,
143 // to which this session is bound. The binding would occur by making the "bind"
144 // parameter in TPM2_StartAuthSession() not equal to TPM_RH_NULL. The binding only
145 // occurs if the session is an HMAC session. The bind value is a combination of
146 // the Name and the authValue of the entity.
147 //
148 // Return Type: BOOL
149 //     TRUE(1)         handle points to the session start entity

```

```

150 // FALSE(0) handle does not point to the session start entity
151 static BOOL IsSessionBindEntity(
152     TPM_HANDLE associatedHandle, // IN: handle to be authorized
153     SESSION* session // IN: associated session
154 )
155 {
156     TPM2B_NAME entity; // The bind value for the entity
157     //
158     // If the session is not bound, return FALSE.
159     if(session->attributes.isBound)
160     {
161         // Compute the bind value for the entity.
162         SessionComputeBoundEntity(associatedHandle, &entity);
163
164         // Compare to the bind value in the session.
165         return MemoryEqual2B(&entity.b, &session->ul.boundEntity.b);
166     }
167     return FALSE;
168 }
169
170 /*** IsPolicySessionRequired()
171 // Checks if a policy session is required for a command. If a command requires
172 // DUP or ADMIN role authorization, then the handle that requires that role is the
173 // first handle in the command. This simplifies this checking. If a new command
174 // is created that requires multiple ADMIN role authorizations, then it will
175 // have to be special-cased in this function.
176 // A policy session is required if:
177 // a) the command requires the DUP role;
178 // b) the command requires the ADMIN role and the authorized entity
179 // is an object and its adminWithPolicy bit is SET;
180 // c) the command requires the ADMIN role and the authorized entity
181 // is a permanent handle or an NV Index; or
182 // d) the authorized entity is a PCR belonging to a policy group, and
183 // has its policy initialized
184 // Return Type: BOOL
185 // TRUE(1) policy session is required
186 // FALSE(0) policy session is not required
187 static BOOL IsPolicySessionRequired(COMMAND_INDEX commandIndex, // IN: command index
188     UINT32 sessionIndex // IN: session index
189 )
190 {
191     AUTH_ROLE role = CommandAuthRole(commandIndex, sessionIndex);
192     TPM_HT type = HandleGetType(s_associatedHandles[sessionIndex]);
193     //
194     if(role == AUTH_DUP)
195         return TRUE;
196     if(role == AUTH_ADMIN)
197     {
198         // We allow an exception for ADMIN role in a transient object. If the object
199         // allows ADMIN role actions with authorization, then policy is not
200         // required. For all other cases, there is no way to override the command
201         // requirement that a policy be used
202         if(type == TPM_HT_TRANSIENT)
203         {
204             OBJECT* object = HandleToObject(s_associatedHandles[sessionIndex]);
205
206             if(!IS_ATTRIBUTE(
207                 object->publicArea.objectAttributes, TPMA_OBJECT, adminWithPolicy))
208                 return FALSE;
209         }
210         return TRUE;
211     }
212
213     if(type == TPM_HT_PCR)
214     {
215         if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))

```

```

216     {
217         TPM2B_DIGEST policy;
218         TPMI_ALG_HASH policyAlg;
219         policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex], &policy);
220         if(policyAlg != TPM_ALG_NULL)
221             return TRUE;
222     }
223 }
224 return FALSE;
225 }
226
227 /*** IsAuthValueAvailable()
228 // This function indicates if authValue is available and allowed for USER role
229 // authorization of an entity.
230 //
231 // This function is similar to IsAuthPolicyAvailable() except that it does not
232 // check the size of the authValue as IsAuthPolicyAvailable() does (a null
233 // authValue is a valid authorization, but a null policy is not a valid policy).
234 //
235 // This function does not check that the handle reference is valid or if the entity
236 // is in an enabled hierarchy. Those checks are assumed to have been performed
237 // during the handle unmarshaling.
238 //
239 // Return Type: BOOL
240 //     TRUE(1)         authValue is available
241 //     FALSE(0)        authValue is not available
242 static BOOL IsAuthValueAvailable(TPM_HANDLE handle, // IN: handle of entity
243                                 COMMAND_INDEX commandIndex, // IN: command index
244                                 UINT32 sessionIndex // IN: session index
245 )
246 {
247     BOOL result = FALSE;
248     //
249     switch(HandleGetType(handle))
250     {
251     case TPM_HT_PERMANENT:
252         switch(handle)
253         {
254             // At this point hierarchy availability has already been
255             // checked so primary seed handles are always available here
256             case TPM_RH_OWNER:
257             case TPM_RH_ENDORSEMENT:
258             case TPM_RH_PLATFORM:
259 #if VENDOR_PERMANENT_AUTH_ENABLED == YES
260                 // This vendor defined handle associated with the
261                 // manufacturer's shared secret
262             case VENDOR_PERMANENT_AUTH_HANDLE:
263 #endif
264                 // The DA checking has been performed on LockoutAuth but we
265                 // bypass the DA logic if we are using lockout policy. The
266                 // policy would allow execution to continue an lockoutAuth
267                 // could be used, even if direct use of lockoutAuth is disabled
268             case TPM_RH_LOCKOUT:
269                 // NullAuth is always available.
270             case TPM_RH_NULL:
271                 result = TRUE;
272                 break;
273
274 #if ACT_SUPPORT
275             FOR_EACH_ACT(CASE_ACT_HANDLE)
276             {
277                 // The ACT auth value is not available if the platform is
278                 disabled
279                 result = g_phEnable == SET;
280                 break;
281             }
282 #endif
283         }
284     }
285 }

```

```

281 #endif // ACT_SUPPORT
282
283         default:
284             // Otherwise authValue is not available.
285             break;
286     }
287     break;
288 case TPM_HT_TRANSIENT:
289     // A persistent object has already been loaded and the internal
290     // handle changed.
291     {
292         OBJECT*    object;
293         TPMA_OBJECT attributes;
294         //
295         object      = HandleToObject(handle);
296         attributes = object->publicArea.objectAttributes;
297
298         // authValue is always available for a sequence object.
299         // An alternative for this is to
300         // SET_ATTRIBUTE(object->publicArea, TPMA_OBJECT, userWithAuth) when
the
301         // sequence is started.
302         if(ObjectIsSequence(object))
303         {
304             result = TRUE;
305             break;
306         }
307         // authValue is available for an object if it has its sensitive
308         // portion loaded and
309         // a) userWithAuth bit is SET, or
310         // b) ADMIN role is required
311         if(object->attributes.publicOnly == CLEAR
312            && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, userWithAuth)
313              || (CommandAuthRole(commandIndex, sessionIndex) == AUTH_ADMIN
314                && !IS_ATTRIBUTE(
315                    attributes, TPMA_OBJECT, adminWithPolicy))))
316             result = TRUE;
317     }
318     break;
319 case TPM_HT_NV_INDEX:
320     // NV Index.
321     {
322         NV_REF    locator;
323         NV_INDEX* nvIndex = NvGetIndexInfo(handle, &locator);
324         TPMA_NV    nvAttributes;
325         //
326         pAssert(nvIndex != 0);
327
328         nvAttributes = nvIndex->publicArea.attributes;
329
330         if(IsWriteOperation(commandIndex))
331         {
332             // AuthWrite can't be set for a PIN index
333             if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHWRITE))
334                 result = TRUE;
335         }
336         else
337         {
338             // A "read" operation
339             // For a PIN Index, the authValue is available as long as the
340             // Index has been written and the pinCount is less than pinLimit
341             if(IsNvPinFailIndex(nvAttributes)
342                || IsNvPinPassIndex(nvAttributes))
343             {
344                 NV_PIN pin;
345                 if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))

```

```

346         break; // return false
347         // get the index values
348         pin.intVal = NvGetUINT64Data(nvIndex, locator);
349         if(pin.pin.pinCount < pin.pin.pinLimit)
350             result = TRUE;
351     }
352     // For non-PIN Indexes, need to allow use of the authValue
353     else if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHREAD))
354         result = TRUE;
355     }
356     }
357     break;
358     case TPM_HT_PCR:
359         // PCR handle.
360         // authValue is always allowed for PCR
361         result = TRUE;
362         break;
363     default:
364         // Otherwise, authValue is not available
365         break;
366 }
367 return result;
368 }
369
370 /*** IsAuthPolicyAvailable()
371 // This function indicates if an authPolicy is available and allowed.
372 //
373 // This function does not check that the handle reference is valid or if the entity
374 // is in an enabled hierarchy. Those checks are assumed to have been performed
375 // during the handle unmarshaling.
376 //
377 // Return Type: BOOL
378 //     TRUE(1)         authPolicy is available
379 //     FALSE(0)        authPolicy is not available
380 static BOOL IsAuthPolicyAvailable(TPM_HANDLE handle, // IN: handle of entity
381                                  COMMAND_INDEX commandIndex, // IN: command index
382                                  UINT32 sessionIndex // IN: session index
383 )
384 {
385     BOOL result = FALSE;
386     //
387     switch(HandleGetType(handle))
388     {
389     case TPM_HT_PERMANENT:
390         switch(handle)
391         {
392             // At this point hierarchy availability has already been checked.
393             case TPM_RH_OWNER:
394                 if(gp.ownerPolicy.t.size != 0)
395                     result = TRUE;
396                 break;
397             case TPM_RH_ENDORSEMENT:
398                 if(gp.endorsementPolicy.t.size != 0)
399                     result = TRUE;
400                 break;
401             case TPM_RH_PLATFORM:
402                 if(gc.platformPolicy.t.size != 0)
403                     result = TRUE;
404                 break;
405 #if ACT_SUPPORT
406 # define ACT_GET_POLICY(N) \
407     case TPM_RH_ACT_##N: \
408         if(go.ACT_##N.authPolicy.t.size != 0) \
409             result = TRUE; \
410         break;
411 
```



```

412
413         FOR_EACH_ACT (ACT_GET_POLICY)
414 #endif // ACT_SUPPORT
415
416         case TPM_RH_LOCKOUT:
417             if (gp.lockoutPolicy.t.size != 0)
418                 result = TRUE;
419             break;
420         default:
421             break;
422     }
423     break;
424 case TPM_HT_TRANSIENT:
425 {
426     // Object handle.
427     // An evict object would already have been loaded and given a
428     // transient object handle by this point.
429     OBJECT* object = HandleToObject(handle);
430     // Policy authorization is not available for an object with only
431     // public portion loaded.
432     if (object->attributes.publicOnly == CLEAR)
433     {
434         // Policy authorization is always available for an object but
435         // is never available for a sequence.
436         if (!ObjectIsSequence(object))
437             result = TRUE;
438     }
439     break;
440 }
441 case TPM_HT_NV_INDEX:
442     // An NV Index.
443     {
444         NV_INDEX* nvIndex = NvGetIndexInfo(handle, NULL);
445         TPMA_NV nvAttributes = nvIndex->publicArea.attributes;
446         //
447         // If the policy size is not zero, check if policy can be used.
448         if (nvIndex->publicArea.authPolicy.t.size != 0)
449         {
450             // If policy session is required for this handle, always
451             // uses policy regardless of the attributes bit setting
452             if (IsPolicySessionRequired(commandIndex, sessionIndex))
453                 result = TRUE;
454             // Otherwise, the presence of the policy depends on the NV
455             // attributes.
456             else if (IsWriteOperation(commandIndex))
457             {
458                 if (IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYWRITE))
459                     result = TRUE;
460             }
461             else
462             {
463                 if (IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYREAD))
464                     result = TRUE;
465             }
466         }
467     }
468     break;
469 case TPM_HT_PCR:
470     // PCR handle.
471     if (PCRPolicyIsAvailable(handle))
472         result = TRUE;
473     break;
474 default:
475     break;
476 }
477 return result;

```

```

478 }
479
480 /** Session Parsing Functions
481
482 /*** ClearCpRpHashes()
483 void ClearCpRpHashes(COMMAND* command)
484 {
485     // The macros expand according to the implemented hash algorithms. An IDE may
486     // complain that COMMAND does not contain SHA1CpHash or SHA1RpHash because of the
487     // complexity of the macro expansion where the data space is defined; but, if SHA1
488     // is implemented, it actually does and the compiler is happy.
489     #define CLEAR_CP_HASH(HASH, Hash) command->Hash##CpHash.b.size = 0;
490     FOR_EACH_HASH(CLEAR_CP_HASH)
491     #define CLEAR_RP_HASH(HASH, Hash) command->Hash##RpHash.b.size = 0;
492     FOR_EACH_HASH(CLEAR_RP_HASH)
493 }
494
495 /*** GetCpHashPointer()
496 // Function to get a pointer to the cpHash of the command
497 static TPM2B_DIGEST* GetCpHashPointer(COMMAND* command, TPMI_ALG_HASH hashAlg)
498 {
499     TPM2B_DIGEST* retVal;
500     //
501     // Define the macro that will expand for each implemented algorithm in the switch
502     // statement below.
503     #define GET_CP_HASH_POINTER(HASH, Hash) \
504     case ALG_ ##HASH##_VALUE: \
505         retVal = (TPM2B_DIGEST*)&command->Hash##CpHash; \
506         break; \
507
508     switch(hashAlg)
509     {
510         // For each implemented hash, this will expand as defined above
511         // by GET_CP_HASH_POINTER. Your IDE may complain that
512         // 'struct "COMMAND" has no field "SHA1CpHash"' but the compiler says
513         // it does, so...
514         FOR_EACH_HASH(GET_CP_HASH_POINTER)
515         default:
516             retVal = NULL;
517             break;
518     }
519     return retVal;
520 }
521
522 /*** GetRpHashPointer()
523 // Function to get a pointer to the RpHash of the command
524 static TPM2B_DIGEST* GetRpHashPointer(COMMAND* command, TPMI_ALG_HASH hashAlg)
525 {
526     TPM2B_DIGEST* retVal;
527     //
528     // Define the macro that will expand for each implemented algorithm in the switch
529     // statement below.
530     #define GET_RP_HASH_POINTER(HASH, Hash) \
531     case ALG_ ##HASH##_VALUE: \
532         retVal = (TPM2B_DIGEST*)&command->Hash##RpHash; \
533         break; \
534
535     switch(hashAlg)
536     {
537         // For each implemented hash, this will expand as defined above
538         // by GET_RP_HASH_POINTER. Your IDE may complain that
539         // 'struct "COMMAND" has no field 'SHA1RpHash'' but the compiler says
540         // it does, so...
541         FOR_EACH_HASH(GET_RP_HASH_POINTER)
542         default:
543             retVal = NULL;

```

```

544         break;
545     }
546     return retVal;
547 }
548
549 /*** ComputeCpHash()
550 // This function computes the cpHash as defined in Part 2 and described in Part 1.
551 static TPM2B_DIGEST* ComputeCpHash(COMMAND* command, // IN: command parsing structure
552                                     TPML_ALG_HASH hashAlg // IN: hash algorithm
553 )
554 {
555     UINT32      i;
556     HASH_STATE  hashState;
557     TPM2B_NAME  name;
558     TPM2B_DIGEST* cpHash;
559     //
560     // cpHash = hash(commandCode [ || authName1
561     //                               [ || authName2
562     //                               [ || authName 3 ]])
563     //                               [ || parameters])
564     // A cpHash can contain just a commandCode only if the lone session is
565     // an audit session.
566     // Get pointer to the hash value
567     cpHash = GetCpHashPointer(command, hashAlg);
568     if(cpHash->t.size == 0)
569     {
570         cpHash->t.size = CryptHashStart(&hashState, hashAlg);
571         // Add commandCode.
572         CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
573         // Add authNames for each of the handles.
574         for(i = 0; i < command->handleNum; i++)
575             CryptDigestUpdate2B(&hashState,
576                                &EntityGetName(command->handles[i], &name)->b);
577         // Add the parameters.
578         CryptDigestUpdate(
579             &hashState, command->parameterSize, command->parameterBuffer);
580         // Complete the hash.
581         CryptHashEnd2B(&hashState, &cpHash->b);
582     }
583     return cpHash;
584 }
585
586 /*** GetCpHash()
587 // This function is used to access a precomputed cpHash.
588 static TPM2B_DIGEST* GetCpHash(COMMAND* command, TPML_ALG_HASH hashAlg)
589 {
590     TPM2B_DIGEST* cpHash = GetCpHashPointer(command, hashAlg);
591     //
592     pAssert(cpHash->t.size != 0);
593     return cpHash;
594 }
595
596 /*** CompareTemplateHash()
597 // This function computes the template hash and compares it to the session
598 // templateHash. It is the hash of the second parameter
599 // assuming that the command is TPM2_Create(), TPM2_CreatePrimary(), or
600 // TPM2_CreateLoaded()
601 // Return Type: BOOL
602 //     TRUE(1)           template hash equal to session->templateHash
603 //     FALSE(0)          template hash not equal to session->templateHash
604 static BOOL CompareTemplateHash(COMMAND* command, // IN: parsing structure
605                                 SESSION* session // IN: session data
606 )
607 {
608     BYTE*      pBuffer = command->parameterBuffer;
609     INT32      pSize   = command->parameterSize;

```

```

610     TPM2B_DIGEST tHash;
611     UINT16      size;
612     //
613     // Only try this for the three commands for which it is intended
614     if(command->code != TPM_CC_Create && command->code != TPM_CC_CreatePrimary
615 #if CC_CreateLoaded
616     && command->code != TPM_CC_CreateLoaded
617 #endif
618     )
619         return FALSE;
620     // Assume that the first parameter is a TPM2B and unmarshal the size field
621     // Note: this will not affect the parameter buffer and size in the calling
622     // function.
623     if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
624         return FALSE;
625     // reduce the space in the buffer.
626     // NOTE: this could make pSize go negative if the parameters are not correct but
627     // the unmarshaling code does not try to unmarshal if the remaining size is
628     // negative.
629     pSize -= size;
630
631     // Advance the pointer
632     pBuffer += size;
633
634     // Get the size of what should be the template
635     if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
636         return FALSE;
637     // See if this is reasonable
638     if(size > pSize)
639         return FALSE;
640     // Hash the template data
641     tHash.t.size = CryptHashBlock(
642         session->authHashAlg, size, pBuffer, sizeof(tHash.t.buffer), tHash.t.buffer);
643     return (MemoryEqual2B(&session->ul.templateHash.b, &tHash.b));
644 }
645
646 /*** CompareNameHash()
647 // This function computes the name hash and compares it to the nameHash in the
648 // session data, returning true if they are equal.
649 BOOL CompareNameHash(COMMAND* command, // IN: main parsing structure
650                     SESSION* session // IN: session structure with nameHash
651 )
652 {
653     HASH_STATE hashState;
654     TPM2B_DIGEST nameHash;
655     UINT32 i;
656     TPM2B_NAME name;
657     //
658     nameHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
659     // Add names.
660     for(i = 0; i < command->handleNum; i++)
661         CryptDigestUpdate2B(&hashState,
662                             &EntityGetName(command->handles[i], &name)->b);
663     // Complete hash.
664     CryptHashEnd2B(&hashState, &nameHash.b);
665     // and compare
666     return MemoryEqual(
667         session->ul.nameHash.t.buffer, nameHash.t.buffer, nameHash.t.size);
668 }
669
670 /*** CompareParametersHash()
671 // This function computes the parameters hash and compares it to the pHash in
672 // the session data, returning true if they are equal.
673 BOOL CompareParametersHash(COMMAND* command, // IN: main parsing structure
674                          SESSION* session // IN: session structure with pHash
675 )

```

```

676 {
677     HASH_STATE    hashState;
678     TPM2B_DIGEST  pHash;
679     //
680     pHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
681     // Add commandCode.
682     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
683     // Add the parameters.
684     CryptDigestUpdate(&hashState, command->parameterSize, command->parameterBuffer);
685     // Complete hash.
686     CryptHashEnd2B(&hashState, &pHash.b);
687     // and compare
688     return MemoryEqual2B(&session->ul.pHash.b, &pHash.b);
689 }
690
691 /*** CheckPWAAuthSession()
692 // This function validates the authorization provided in a PWAP session. It
693 // compares the input value to authValue of the authorized entity. Argument
694 // sessionIndex is used to get handles handle of the referenced entities from
695 // s_inputAuthValues[] and s_associatedHandles[].
696 //
697 // Return Type: TPM_RC
698 //             TPM_RC_AUTH_FAIL           authorization fails and increments DA failure
699 //                                         count
700 //             TPM_RC_BAD_AUTH           authorization fails but DA does not apply
701 //
702 static TPM_RC CheckPWAAuthSession(
703     UINT32 sessionIndex // IN: index of session to be processed
704 )
705 {
706     TPM2B_AUTH authValue;
707     TPM_HANDLE associatedHandle = s_associatedHandles[sessionIndex];
708     //
709     // Strip trailing zeros from the password.
710     MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
711
712     // Get the authValue with trailing zeros removed
713     EntityGetAuthValue(associatedHandle, &authValue);
714
715     // Success if the values are identical.
716     if(MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &authValue.b))
717     {
718         return TPM_RC_SUCCESS;
719     }
720     else // if the digests are not identical
721     {
722         // Invoke DA protection if applicable.
723         return IncrementLockout(sessionIndex);
724     }
725 }
726
727 /*** ComputeCommandHMAC()
728 // This function computes the HMAC for an authorization session in a command.
729 /*(See part 1 specification -- this tag keeps this comment from showing up in
730 // merged document which is probably good because this comment doesn't look right.
731 // The sessionAuth value
732 //     authHMAC := HMACsHash((sessionKey | authValue),
733 //                             (pHash | nonceNewer | nonceOlder | nonceTPMencrypt-only
734 //                             | nonceTPMAudit | sessionAttributes))
735 // Where:
736 //     HMACsHash() The HMAC algorithm using the hash algorithm specified
737 //                 when the session was started.
738 //
739 //     sessionKey A value that is computed in a protocol-dependent way,
740 //                 using KDFa. When used in an HMAC or KDF, the size field
741 //                 for this value is not included.

```

```

742 //
743 //     authValue      A value that is found in the sensitive area of an entity.
744 //                   When used in an HMAC or KDF, the size field for this
745 //                   value is not included.
746 //
747 //     pHash          Hash of the command (cpHash) using the session hash.
748 //                   When using a pHash in an HMAC computation, only the
749 //                   digest is used.
750 //
751 //     nonceNewer      A value that is generated by the entity using the
752 //                   session. A new nonce is generated on each use of the
753 //                   session. For a command, this will be nonceCaller.
754 //                   When used in an HMAC or KDF, the size field is not used.
755 //
756 //     nonceOlder      A TPM2B_NONCE that was received the previous time the
757 //                   session was used. For a command, this is nonceTPM.
758 //                   When used in an HMAC or KDF, the size field is not used.
759 //
760 //     nonceTPMdecrypt The nonceTPM of the decrypt session is included in
761 //                   the HMAC, but only in the command.
762 //
763 //     nonceTPMencrypt The nonceTPM of the encrypt session is included in
764 //                   the HMAC but only in the command.
765 //
766 //     sessionAttributes A byte indicating the attributes associated with the
767 //                   particular use of the session.
768 */
769 static TPM2B_DIGEST* ComputeCommandHMAC(
770     COMMAND*      command,      // IN: primary control structure
771     UINT32         sessionIndex, // IN: index of session to be processed
772     TPM2B_DIGEST* hmac          // OUT: authorization HMAC
773 )
774 {
775     TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
776     TPM2B_KEY      key;
777     BYTE           marshalBuffer[sizeof(TPMA_SESSION)];
778     BYTE*          buffer;
779     UINT32         marshalSize;
780     HMAC_STATE     hmacState;
781     TPM2B_NONCE*   nonceDecrypt;
782     TPM2B_NONCE*   nonceEncrypt;
783     SESSION*       session;
784     //
785     nonceDecrypt = NULL;
786     nonceEncrypt = NULL;
787
788     // Determine if extra nonceTPM values are going to be required.
789     // If this is the first session (sessionIndex = 0) and it is an authorization
790     // session that uses an HMAC, then check if additional session nonces are to be
791     // included.
792     if(sessionIndex == 0 && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
793     {
794         // If there is a decrypt session and if this is not the decrypt session,
795         // then an extra nonce may be needed.
796         if(s_decryptSessionIndex != UNDEFINED_INDEX
797            && s_decryptSessionIndex != sessionIndex)
798         {
799             // Will add the nonce for the decrypt session.
800             SESSION* decryptSession =
801                 SessionGet(s_sessionHandles[s_decryptSessionIndex]);
802             nonceDecrypt = &decryptSession->nonceTPM;
803         }
804         // Now repeat for the encrypt session.
805         if(s_encryptSessionIndex != UNDEFINED_INDEX
806            && s_encryptSessionIndex != sessionIndex
807            && s_encryptSessionIndex != s_decryptSessionIndex)

```



```

808     {
809         // Have to have the nonce for the encrypt session.
810         SESSION* encryptSession =
811             SessionGet(s_sessionHandles[s_encryptSessionIndex]);
812         nonceEncrypt = &encryptSession->nonceTPM;
813     }
814 }
815
816 // Continue with the HMAC processing.
817 session = SessionGet(s_sessionHandles[sessionIndex]);
818
819 // Generate HMAC key.
820 MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
821
822 // Check if the session has an associated handle and if the associated entity
823 // is the one to which the session is bound. If not, add the authValue of
824 // this entity to the HMAC key.
825 // If the session is bound to the object or the session is a policy session
826 // with no authValue required, do not include the authValue in the HMAC key.
827 // Note: For a policy session, its isBound attribute is CLEARED.
828 //
829 // Include the entity authValue if it is needed
830 if(session->attributes.includeAuth == SET)
831 {
832     TPM2B_AUTH authValue;
833     // Get the entity authValue with trailing zeros removed
834     EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
835     // add the authValue to the HMAC key
836     MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
837 }
838 // if the HMAC key size is 0, a NULL string HMAC is allowed
839 if(key.t.size == 0 && s_inputAuthValues[sessionIndex].t.size == 0)
840 {
841     hmac->t.size = 0;
842     return hmac;
843 }
844 // Start HMAC
845 hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
846
847 // Add cpHash
848 CryptDigestUpdate2B(&hmacState.hashState,
849                     &ComputeCpHash(command, session->authHashAlg)->b);
850 // Add nonces as required
851 CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
852 CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
853 if(nonceDecrypt != NULL)
854     CryptDigestUpdate2B(&hmacState.hashState, &nonceDecrypt->b);
855 if(nonceEncrypt != NULL)
856     CryptDigestUpdate2B(&hmacState.hashState, &nonceEncrypt->b);
857 // Add sessionAttributes
858 buffer = marshalBuffer;
859 marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]), &buffer, NULL);
860 CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
861 // Complete the HMAC computation
862 CryptHmacEnd2B(&hmacState, &hmac->b);
863
864 return hmac;
865 }
866
867 /*** CheckSessionHMAC()
868 // This function checks the HMAC of in a session. It uses ComputeCommandHMAC()
869 // to compute the expected HMAC value and then compares the result with the
870 // HMAC in the authorization session. The authorization is successful if they
871 // are the same.
872 //
873 // If the authorizations are not the same, IncrementLockout() is called. It will

```

```

874 // return TPM_RC_AUTH_FAIL if the failure caused the failureCount to increment.
875 // Otherwise, it will return TPM_RC_BAD_AUTH.
876 //
877 // Return Type: TPM_RC
878 //     TPM_RC_AUTH_FAIL      authorization failure caused failureCount increment
879 //     TPM_RC_BAD_AUTH       authorization failure did not cause failureCount
880 //                             increment
881 //
882 static TPM_RC CheckSessionHMAC(
883     COMMAND* command,          // IN: primary control structure
884     UINT32 sessionIndex // IN: index of session to be processed
885 )
886 {
887     TPM2B_DIGEST hmac; // authHMAC for comparing
888                        //
889     // Compute authHMAC
890     ComputeCommandHMAC(command, sessionIndex, &hmac);
891
892     // Compare the input HMAC with the authHMAC computed above.
893     if(!MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &hmac.b))
894     {
895         // If an HMAC session has a failure, invoke the anti-hammering
896         // if it applies to the authorized entity or the session.
897         // Otherwise, just indicate that the authorization is bad.
898         return IncrementLockout(sessionIndex);
899     }
900     return TPM_RC_SUCCESS;
901 }
902
903 /*** CheckPolicyAuthSession()
904 // This function is used to validate the authorization in a policy session.
905 // This function performs the following comparisons to see if a policy
906 // authorization is properly provided. The check are:
907 // a) compare policyDigest in session with authPolicy associated with
908 //     the entity to be authorized;
909 // b) compare timeout if applicable;
910 // c) compare commandCode if applicable;
911 // d) compare cpHash if applicable; and
912 // e) see if PCR values have changed since computed.
913 //
914 // If all the above checks succeed, the handle is authorized.
915 // The order of these comparisons is not important because any failure will
916 // result in the same error code.
917 //
918 // Return Type: TPM_RC
919 //     TPM_RC_PCR_CHANGED      PCR value is not current
920 //     TPM_RC_POLICY_FAIL     policy session fails
921 //     TPM_RC_LOCALITY        command locality is not allowed
922 //     TPM_RC_POLICY_CC       CC doesn't match
923 //     TPM_RC_EXPIRED         policy session has expired
924 //     TPM_RC_PP              PP is required but not asserted
925 //     TPM_RC_NV_UNAVAILABLE  NV is not available for write
926 //     TPM_RC_NV_RATE         NV is rate limiting
927 static TPM_RC CheckPolicyAuthSession(
928     COMMAND* command,          // IN: primary parsing structure
929     UINT32 sessionIndex // IN: index of session to be processed
930 )
931 {
932     SESSION* session;
933     TPM2B_DIGEST authPolicy;
934     TPMI_ALG_HASH policyAlg;
935     UINT8 locality;
936     //
937     // Initialize pointer to the authorization session.
938     session = SessionGet(s_sessionHandles[sessionIndex]);
939

```

```

940 // If the command is TPM2_PolicySecret(), make sure that
941 // either password or authValue is required
942 if(command->code == TPM_CC_PolicySecret
943    && session->attributes.isPasswordNeeded == CLEAR
944    && session->attributes.isAuthValueNeeded == CLEAR)
945     return TPM_RC_MODE;
946 // See if the PCR counter for the session is still valid.
947 if(!SessionPCRValueIsCurrent(session))
948     return TPM_RC_PCR_CHANGED;
949 // Get authPolicy.
950 policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex], &authPolicy);
951 // Compare authPolicy.
952 if(!MemoryEqual2B(&session->u2.policyDigest.b, &authPolicy.b))
953     return TPM_RC_POLICY_FAIL;
954 // Policy is OK so check if the other factors are correct
955
956 // Compare policy hash algorithm.
957 if(policyAlg != session->authHashAlg)
958     return TPM_RC_POLICY_FAIL;
959
960 // Compare timeout.
961 if(session->timeout != 0)
962 {
963     // Cannot compare time if clock stop advancing. An TPM_RC_NV_UNAVAILABLE
964     // or TPM_RC_NV_RATE error may be returned here. This doesn't mean that
965     // a new nonce will be created just that, because TPM time can't advance
966     // we can't do time-based operations.
967     RETURN_IF_NV_IS_NOT_AVAILABLE;
968
969     if((session->timeout < g_time) || (session->epoch != g_timeEpoch))
970         return TPM_RC_EXPIRED;
971 }
972 // If command code is provided it must match
973 if(session->commandCode != 0)
974 {
975     if(session->commandCode != command->code)
976         return TPM_RC_POLICY_CC;
977 }
978 else
979 {
980     // If command requires a DUP or ADMIN authorization, the session must have
981     // command code set.
982     AUTH_ROLE role = CommandAuthRole(command->index, sessionIndex);
983     if(role == AUTH_ADMIN || role == AUTH_DUP)
984         return TPM_RC_POLICY_FAIL;
985 }
986 // Check command locality.
987 {
988     BYTE sessionLocality[sizeof(TPMA_LOCALITY)];
989     BYTE* buffer = sessionLocality;
990
991     // Get existing locality setting in canonical form
992     sessionLocality[0] = 0;
993     TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
994
995     // See if the locality has been set
996     if(sessionLocality[0] != 0)
997     {
998         // If so, get the current locality
999         locality = _plat__LocalityGet();
1000         if(locality < 5)
1001         {
1002             if(((sessionLocality[0] & (1 << locality)) == 0)
1003                || sessionLocality[0] > 31)
1004                 return TPM_RC_LOCALITY;
1005         }
1006     }

```

```

1006         else if(locality > 31)
1007         {
1008             if(sessionLocality[0] != locality)
1009                 return TPM_RC_LOCALITY;
1010         }
1011         else
1012         {
1013             // Could throw an assert here but a locality error is just
1014             // as good. It just means that, whatever the locality is, it isn't
1015             // the locality requested so...
1016             return TPM_RC_LOCALITY;
1017         }
1018     }
1019 } // end of locality check
1020 // Check physical presence.
1021 if(session->attributes.isPPRequired == SET && !_plat__PhysicalPresenceAsserted())
1022     return TPM_RC_PP;
1023 // Compare cpHash/nameHash/pHash/templateHash if defined.
1024 if(session->ul.cpHash.b.size != 0)
1025 {
1026     BOOL OK = FALSE;
1027     if(session->attributes.isCpHashDefined)
1028         // Compare cpHash.
1029         OK = MemoryEqual2B(&session->ul.cpHash.b,
1030                             &ComputeCpHash(command, session->authHashAlg)->b);
1031     else if(session->attributes.isNameHashDefined)
1032         OK = CompareNameHash(command, session);
1033     else if(session->attributes.isParametersHashDefined)
1034         OK = CompareParametersHash(command, session);
1035     else if(session->attributes.isTemplateHashDefined)
1036         OK = CompareTemplateHash(command, session);
1037     if(!OK)
1038         return TPM_RCS_POLICY_FAIL;
1039 }
1040 if(session->attributes.checkNvWritten)
1041 {
1042     NV_REF locator;
1043     NV_INDEX* nvIndex;
1044     //
1045     // If this is not an NV index, the policy makes no sense so fail it.
1046     if(HandleGetType(s_associatedHandles[sessionIndex]) != TPM_HT_NV_INDEX)
1047         return TPM_RC_POLICY_FAIL;
1048     // Get the index data
1049     nvIndex = NvGetIndexInfo(s_associatedHandles[sessionIndex], &locator);
1050
1051     // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
1052     if((IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
1053         != (session->attributes.nvWrittenState == SET))
1054         return TPM_RC_POLICY_FAIL;
1055 }
1056 return TPM_RC_SUCCESS;
1057 }
1058
1059 /*** RetrieveSessionData()
1060 // This function will unmarshal the sessions in the session area of a command. The
1061 // values are placed in the arrays that are defined at the beginning of this file.
1062 // The normal unmarshaling errors are possible.
1063 //
1064 // Return Type: TPM_RC
1065 //     TPM_RC_SUCCSS      unmarshaled without error
1066 //     TPM_RC_SIZE        the number of bytes unmarshaled is not the same
1067 //                        as the value for authorizationSize in the command
1068 //
1069 static TPM_RC RetrieveSessionData(
1070     COMMAND* command // IN: main parsing structure for command
1071 )

```

```

1072 {
1073     int            i;
1074     TPM_RC         result;
1075     SESSION*       session;
1076     TPMA_SESSION   sessionAttributes;
1077     TPM_HT         sessionType;
1078     INT32          sessionIndex;
1079     TPM_RC         errorIndex;
1080     //
1081     s_decryptSessionIndex = UNDEFINED_INDEX;
1082     s_encryptSessionIndex = UNDEFINED_INDEX;
1083     s_auditSessionIndex   = UNDEFINED_INDEX;
1084
1085     for(sessionIndex = 0; command->authSize > 0; sessionIndex++)
1086     {
1087         errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
1088
1089         // If maximum allowed number of sessions has been parsed, return a size
1090         // error with a session number that is larger than the number of allowed
1091         // sessions
1092         if(sessionIndex == MAX_SESSION_NUM)
1093             return TPM_RCS_SIZE + errorIndex;
1094         // make sure that the associated handle for each session starts out
1095         // unassigned
1096         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
1097
1098         // First parameter: Session handle.
1099         result = TPMI_SH_AUTH_SESSION_Unmarshal(&s_sessionHandles[sessionIndex],
1100                                                 &command->parameterBuffer,
1101                                                 &command->authSize,
1102                                                 TRUE);
1103
1104         if(result != TPM_RC_SUCCESS)
1105             return result + TPM_RC_S + g_rcIndex[sessionIndex];
1106         // Second parameter: Nonce.
1107         result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
1108                                       &command->parameterBuffer,
1109                                       &command->authSize);
1110
1111         if(result != TPM_RC_SUCCESS)
1112             return result + TPM_RC_S + g_rcIndex[sessionIndex];
1113         // Third parameter: sessionAttributes.
1114         result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
1115                                       &command->parameterBuffer,
1116                                       &command->authSize);
1117
1118         if(result != TPM_RC_SUCCESS)
1119             return result + TPM_RC_S + g_rcIndex[sessionIndex];
1120         // Fourth parameter: authValue (PW or HMAC).
1121         result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
1122                                       &command->parameterBuffer,
1123                                       &command->authSize);
1124
1125         if(result != TPM_RC_SUCCESS)
1126             return result + errorIndex;
1127
1128         sessionAttributes = s_attributes[sessionIndex];
1129         if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1130         {
1131             // A PWAP session needs additional processing.
1132             // Can't have any attributes set other than continueSession bit
1133             if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt)
1134                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt)
1135                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit)
1136                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
1137                || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset))
1138                 return TPM_RCS_ATTRIBUTES + errorIndex;
1139             // The nonce size must be zero.
1140             if(s_nonceCaller[sessionIndex].t.size != 0)
1141                 return TPM_RCS_NONCE + errorIndex;
1142         }
1143     }

```



```

1138         continue;
1139     }
1140     // For not password sessions...
1141     // Find out if the session is loaded.
1142     if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
1143         return TPM_RC_REFERENCE_S0 + sessionIndex;
1144     sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
1145     session      = SessionGet(s_sessionHandles[sessionIndex]);
1146
1147     // Check if the session is an HMAC/policy session.
1148     if((session->attributes.isPolicy == SET && sessionType == TPM_HT_HMAC_SESSION)
1149         || (session->attributes.isPolicy == CLEAR
1150             && sessionType == TPM_HT_POLICY_SESSION))
1151         return TPM_RCS_HANDLE + errorIndex;
1152     // Check that this handle has not previously been used.
1153     for(i = 0; i < sessionIndex; i++)
1154     {
1155         if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
1156             return TPM_RCS_HANDLE + errorIndex;
1157     }
1158     // If the session is used for parameter encryption or audit as well, set
1159     // the corresponding indexes.
1160
1161     // First process decrypt.
1162     if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt))
1163     {
1164         // Check if the commandCode allows command parameter encryption.
1165         if(DecryptSize(command->index) == 0)
1166             return TPM_RCS_ATTRIBUTES + errorIndex;
1167         // Encrypt attribute can only appear in one session
1168         if(s_decryptSessionIndex != UNDEFINED_INDEX)
1169             return TPM_RCS_ATTRIBUTES + errorIndex;
1170         // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
1171         if(session->symmetric.algorithm == TPM_ALG_NULL)
1172             return TPM_RCS_SYMMETRIC + errorIndex;
1173         // All checks passed, so set the index for the session used to decrypt
1174         // a command parameter.
1175         s_decryptSessionIndex = sessionIndex;
1176     }
1177     // Now process encrypt.
1178     if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt))
1179     {
1180         // Check if the commandCode allows response parameter encryption.
1181         if(EncryptSize(command->index) == 0)
1182             return TPM_RCS_ATTRIBUTES + errorIndex;
1183         // Encrypt attribute can only appear in one session.
1184         if(s_encryptSessionIndex != UNDEFINED_INDEX)
1185             return TPM_RCS_ATTRIBUTES + errorIndex;
1186         // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
1187         if(session->symmetric.algorithm == TPM_ALG_NULL)
1188             return TPM_RCS_SYMMETRIC + errorIndex;
1189         // All checks passed, so set the index for the session used to encrypt
1190         // a response parameter.
1191         s_encryptSessionIndex = sessionIndex;
1192     }
1193     // At last process audit.
1194     if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit))
1195     {
1196         // Audit attribute can only appear in one session.
1197         if(s_auditSessionIndex != UNDEFINED_INDEX)
1198             return TPM_RCS_ATTRIBUTES + errorIndex;
1199         // An audit session can not be policy session.
1200         if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION)
1201             return TPM_RCS_ATTRIBUTES + errorIndex;
1202         // If this is a reset of the audit session, or the first use
1203         // of the session as an audit session, it doesn't matter what

```



```

1204         // the exclusive state is. The session will become exclusive.
1205         if(!IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset)
1206             && session->attributes.isAudit == SET)
1207         {
1208             // Not first use or reset. If auditExclusive is SET, then this
1209             // session must be the current exclusive session.
1210             if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
1211                 && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
1212                 return TPM_RC_EXCLUSIVE;
1213         }
1214         s_auditSessionIndex = sessionIndex;
1215     }
1216     // Initialize associated handle as undefined. This will be changed when
1217     // the handles are processed.
1218     s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
1219 }
1220 command->sessionNum = sessionIndex;
1221 return TPM_RC_SUCCESS;
1222 }
1223
1224 /*** CheckLockedOut()
1225 // This function checks to see if the TPM is in lockout. This function should only
1226 // be called if the entity being checked is subject to DA protection. The TPM
1227 // is in lockout if the NV is not available and a DA write is pending. Otherwise
1228 // the TPM is locked out if checking for lockoutAuth ('lockoutAuthCheck' == TRUE)
1229 // and use of lockoutAuth is disabled, or 'failedTries' >= 'maxTries'
1230 // Return Type: TPM_RC
1231 //     TPM_RC_NV_RATE           NV is rate limiting
1232 //     TPM_RC_NV_UNAVAILABLE    NV is not available at this time
1233 //     TPM_RC_LOCKOUT           TPM is in lockout
1234 static TPM_RC CheckLockedOut(
1235     BOOL lockoutAuthCheck // IN: TRUE if checking is for lockoutAuth
1236 )
1237 {
1238     // If NV is unavailable, and current cycle state recorded in NV is not
1239     // SU_NONE_VALUE, refuse to check any authorization because we would
1240     // not be able to handle a DA failure.
1241     if(!NV_IS_AVAILABLE && NV_IS_ORDERLY)
1242         return g_NvStatus;
1243     // Check if DA info needs to be updated in NV.
1244     if(s_DAPendingOnNV)
1245     {
1246         // If NV is accessible,
1247         RETURN_IF_NV_IS_NOT_AVAILABLE;
1248
1249         // ... write the pending DA data and proceed.
1250         NV_SYNC_PERSISTENT(lockOutAuthEnabled);
1251         NV_SYNC_PERSISTENT(failedTries);
1252         s_DAPendingOnNV = FALSE;
1253     }
1254     // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
1255     // is disabled...
1256     if(lockoutAuthCheck)
1257     {
1258         if(gp.lockOutAuthEnabled == FALSE)
1259             return TPM_RC_LOCKOUT;
1260     }
1261     else
1262     {
1263         // ... or if the number of failed tries has been maxed out.
1264         if(gp.failedTries >= gp.maxTries)
1265             return TPM_RC_LOCKOUT;
1266     }
1267 #if USE_DA_USED
1268     // If the daUsed flag is not SET, then no DA validation until the
1269     // daUsed state is written to NV
1270     if(!g_daUsed)

```

```

1270     {
1271         RETURN_IF_NV_IS_NOT_AVAILABLE;
1272         g_daUsed = TRUE;
1273         gp.orderlyState = SU_DA_USED_VALUE;
1274         NV_SYNC_PERSISTENT(orderlyState);
1275         return TPM_RC_RETRY;
1276     }
1277 #endif
1278     }
1279     return TPM_RC_SUCCESS;
1280 }
1281
1282 /*** CheckAuthSession()
1283 // This function checks that the authorization session properly authorizes the
1284 // use of the associated handle.
1285 //
1286 // Return Type: TPM_RC
1287 //     TPM_RC_LOCKOUT           entity is protected by DA and TPM is in
1288 //                             lockout, or TPM is locked out on NV update
1289 //                             pending on DA parameters
1290 //
1291 //     TPM_RC_PP               Physical Presence is required but not provided
1292 //     TPM_RC_AUTH_FAIL        HMAC or PW authorization failed
1293 //                             with DA side-effects (can be a policy session)
1294 //
1295 //     TPM_RC_BAD_AUTH         HMAC or PW authorization failed without DA
1296 //                             side-effects (can be a policy session)
1297 //
1298 //     TPM_RC_POLICY_FAIL     if policy session fails
1299 //     TPM_RC_POLICY_CC       command code of policy was wrong
1300 //     TPM_RC_EXPIRED         the policy session has expired
1301 //     TPM_RC_PCR
1302 //     TPM_RC_AUTH_UNAVAILABLE authValue or authPolicy unavailable
1303 static TPM_RC CheckAuthSession(
1304     COMMAND* command, // IN: primary parsing structure
1305     UINT32 sessionIndex // IN: index of session to be processed
1306 )
1307 {
1308     TPM_RC result = TPM_RC_SUCCESS;
1309     SESSION* session = NULL;
1310     TPM_HANDLE sessionHandle = s_sessionHandles[sessionIndex];
1311     TPM_HANDLE associatedHandle = s_associatedHandles[sessionIndex];
1312     TPM_HT sessionHandleType = HandleGetType(sessionHandle);
1313     BOOL authUsed;
1314     //
1315     pAssert(sessionHandle != TPM_RH_UNASSIGNED);
1316
1317     // Take care of physical presence
1318     if(associatedHandle == TPM_RH_PLATFORM)
1319     {
1320         // If the physical presence is required for this command, check for PP
1321         // assertion. If it isn't asserted, no point going any further.
1322         if(PhysicalPresenceIsRequired(command->index)
1323             && !_plat__PhysicalPresenceAsserted())
1324             return TPM_RC_PP;
1325     }
1326     if(sessionHandle != TPM_RS_PW)
1327     {
1328         session = SessionGet(sessionHandle);
1329
1330         // Set includeAuth to indicate if DA checking will be required and if the
1331         // authValue will be included in any HMAC.
1332         if(sessionHandleType == TPM_HT_POLICY_SESSION)
1333         {
1334             // For a policy session, will check the DA status of the entity if either
1335             // isAuthValueNeeded or isPasswordNeeded is SET.

```

```

1336         session->attributes.includeAuth = session->attributes.isAuthValueNeeded
1337                                     || session->attributes.isPasswordNeeded;
1338     }
1339     else
1340     {
1341         // For an HMAC session, need to check unless the session
1342         // is bound.
1343         session->attributes.includeAuth =
1344             !IsSessionBindEntity(s_associatedHandles[sessionIndex], session);
1345     }
1346     authUsed = session->attributes.includeAuth;
1347 }
1348 else
1349     // Password session
1350     authUsed = TRUE;
1351 // If the authorization session is going to use an authValue, then make sure
1352 // that access to that authValue isn't locked out.
1353 if(authUsed)
1354 {
1355     // See if entity is subject to lockout.
1356     if(!IsDAExempted(associatedHandle))
1357     {
1358         // See if in lockout
1359         result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
1360         if(result != TPM_RC_SUCCESS)
1361             return result;
1362     }
1363 }
1364 // Policy or HMAC+PW?
1365 if(sessionHandleType != TPM_HT_POLICY_SESSION)
1366 {
1367     // for non-policy session make sure that a policy session is not required
1368     if(IsPolicySessionRequired(command->index, sessionIndex))
1369         return TPM_RC_AUTH_TYPE;
1370     // The authValue must be available.
1371     // Note: The authValue is going to be "used" even if it is an EmptyAuth.
1372     // and the session is bound.
1373     if(!IsAuthValueAvailable(associatedHandle, command->index, sessionIndex))
1374         return TPM_RC_AUTH_UNAVAILABLE;
1375 }
1376 else
1377 {
1378     // ... see if the entity has a policy, ...
1379     // Note: IsAuthPolicyAvailable will return FALSE if the sensitive area of the
1380     // object is not loaded
1381     if(!IsAuthPolicyAvailable(associatedHandle, command->index, sessionIndex))
1382         return TPM_RC_AUTH_UNAVAILABLE;
1383     // ... and check the policy session.
1384     result = CheckPolicyAuthSession(command, sessionIndex);
1385     if(result != TPM_RC_SUCCESS)
1386         return result;
1387 }
1388 // Check authorization according to the type
1389 if((TPM_RS_PW == sessionHandle) || (session->attributes.isPasswordNeeded == SET))
1390     result = CheckPWAuthSession(sessionIndex);
1391 else
1392     result = CheckSessionHMAC(command, sessionIndex);
1393 // Do processing for PIN Indexes are only three possibilities for 'result' at
1394 // this point: TPM_RC_SUCCESS, TPM_RC_AUTH_FAIL, and TPM_RC_BAD_AUTH.
1395 // For all these cases, we would have to process a PIN index if the
1396 // authValue of the index was used for authorization.
1397 if((TPM_HT_NV_INDEX == HandleGetType(associatedHandle)) && authUsed)
1398 {
1399     NV_REF    locator;
1400     NV_INDEX* nvIndex = NvGetIndexInfo(associatedHandle, &locator);
1401     NV_PIN    pinData;

```

```

1402     TPMA_NV    nvAttributes;
1403     //
1404     pAssert(nvIndex != NULL);
1405     nvAttributes = nvIndex->publicArea.attributes;
1406     // If this is a PIN FAIL index and the value has been written
1407     // then we can update the counter (increment or clear)
1408     if(IsNvPinFailIndex(nvAttributes)
1409        && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))
1410     {
1411         pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1412         if(result != TPM_RC_SUCCESS)
1413             pinData.pin.pinCount++;
1414         else
1415             pinData.pin.pinCount = 0;
1416         NvWriteUINT64Data(nvIndex, pinData.intVal);
1417     }
1418     // If this is a PIN PASS Index, increment if we have used the
1419     // authorization value.
1420     // NOTE: If the counter has already hit the limit, then we
1421     // would not get here because the authorization value would not
1422     // be available and the TPM would have returned before it gets here
1423     else if(IsNvPinPassIndex(nvAttributes)
1424             && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN)
1425             && result == TPM_RC_SUCCESS)
1426     {
1427         // If the access is valid, then increment the use counter
1428         pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1429         pinData.pin.pinCount++;
1430         NvWriteUINT64Data(nvIndex, pinData.intVal);
1431     }
1432 }
1433 return result;
1434 }
1435
1436 #if CC_GetCommandAuditDigest
1437 /*** CheckCommandAudit()
1438 // This function is called before the command is processed if audit is enabled
1439 // for the command. It will check to see if the audit can be performed and
1440 // will ensure that the cpHash is available for the audit.
1441 // Return Type: TPM_RC
1442 //     TPM_RC_NV_UNAVAILABLE    NV is not available for write
1443 //     TPM_RC_NV_RATE           NV is rate limiting
1444 static TPM_RC CheckCommandAudit(COMMAND* command)
1445 {
1446     // If the audit digest is clear and command audit is required, NV must be
1447     // available so that TPM2_GetCommandAuditDigest() is able to increment
1448     // audit counter. If NV is not available, the function bails out to prevent
1449     // the TPM from attempting an operation that would fail anyway.
1450     if(gp.commandAuditDigest.t.size == 0
1451        || GetCommandCode(command->index) == TPM_CC_GetCommandAuditDigest)
1452     {
1453         RETURN_IF_NV_IS_NOT_AVAILABLE;
1454     }
1455     // Make sure that the cpHash is computed for the algorithm
1456     ComputeCpHash(command, gp.auditHashAlg);
1457     return TPM_RC_SUCCESS;
1458 }
1459 #endif
1460
1461 /*** ParseSessionBuffer()
1462 // This function is the entry function for command session processing.
1463 // It iterates sessions in session area and reports if the required authorization
1464 // has been properly provided. It also processes audit session and passes the
1465 // information of encryption sessions to parameter encryption module.
1466 //
1467 // Return Type: TPM_RC

```

```

1468 //          various          parsing failure or authorization failure
1469 //
1470 TPM_RC
1471 ParseSessionBuffer(COMMAND* command // IN: the structure that contains
1472 )
1473 {
1474     TPM_RC    result;
1475     UINT32    i;
1476     INT32     size = 0;
1477     TPM2B_AUTH extraKey;
1478     UINT32    sessionIndex;
1479     TPM_RC    errorIndex;
1480     SESSION*  session = NULL;
1481     //
1482     // Check if a command allows any session in its session area.
1483     if(!IsSessionAllowed(command->index))
1484         return TPM_RC_AUTH_CONTEXT;
1485     // Default-initialization.
1486     command->sessionNum = 0;
1487
1488     result = RetrieveSessionData(command);
1489     if(result != TPM_RC_SUCCESS)
1490         return result;
1491     // There is no command in the TPM spec that has more handles than
1492     // MAX_SESSION_NUM.
1493     pAssert(command->handleNum <= MAX_SESSION_NUM);
1494
1495     // Associate the session with an authorization handle.
1496     for(i = 0; i < command->handleNum; i++)
1497     {
1498         if(CommandAuthRole(command->index, i) != AUTH_NONE)
1499         {
1500             // If the received session number is less than the number of handles
1501             // that requires authorization, an error should be returned.
1502             // Note: for all the TPM 2.0 commands, handles requiring
1503             // authorization come first in a command input and there are only ever
1504             // two values requiring authorization
1505             if(i > (command->sessionNum - 1))
1506                 return TPM_RC_AUTH_MISSING;
1507             // Record the handle associated with the authorization session
1508             s_associatedHandles[i] = HierarchyNormalizeHandle(command->handles[i]);
1509         }
1510     }
1511     // Consistency checks are done first to avoid authorization failure when the
1512     // command will not be executed anyway.
1513     for(sessionIndex = 0; sessionIndex < command->sessionNum; sessionIndex++)
1514     {
1515         errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
1516         // PW session must be an authorization session
1517         if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1518         {
1519             if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1520                 return TPM_RCS_HANDLE + errorIndex;
1521             // a password session can't be audit, encrypt or decrypt
1522             if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1523                 || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1524                 || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1525                 return TPM_RCS_ATTRIBUTES + errorIndex;
1526             session = NULL;
1527         }
1528         else
1529         {
1530             session = SessionGet(s_sessionHandles[sessionIndex]);
1531
1532             // A trial session can not appear in session area, because it cannot
1533             // be used for authorization, audit or encrypt/decrypt.

```



```

1534     if(session->attributes.isTrialPolicy == SET)
1535         return TPM_RCS_ATTRIBUTES + errorIndex;
1536
1537     // See if the session is bound to a DA protected entity
1538     // NOTE: Since a policy session is never bound, a policy is still
1539     // usable even if the object is DA protected and the TPM is in
1540     // lockout.
1541     if(session->attributes.isDaBound == SET)
1542     {
1543         result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1544         if(result != TPM_RC_SUCCESS)
1545             return result;
1546     }
1547     // If this session is for auditing, make sure the cpHash is computed.
1548     if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit))
1549         ComputeCpHash(command, session->authHashAlg);
1550 }
1551
1552 // if the session has an associated handle, check the authorization
1553 if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1554 {
1555     result = CheckAuthSession(command, sessionIndex);
1556     if(result != TPM_RC_SUCCESS)
1557         return RcSafeAddToResult(result, errorIndex);
1558 }
1559 else
1560 {
1561     // a session that is not for authorization must either be encrypt,
1562     // decrypt, or audit
1563     if(!IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1564        && !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1565        && !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1566         return TPM_RCS_ATTRIBUTES + errorIndex;
1567
1568     // no authValue included in any of the HMAC computations
1569     pAssert(session != NULL);
1570     session->attributes.includeAuth = CLEAR;
1571
1572     // check HMAC for encrypt/decrypt/audit only sessions
1573     result = CheckSessionHMAC(command, sessionIndex);
1574     if(result != TPM_RC_SUCCESS)
1575         return RcSafeAddToResult(result, errorIndex);
1576 }
1577 }
1578 #if CC_GetCommandAuditDigest
1579 // Check if the command should be audited. Need to do this before any parameter
1580 // encryption so that the cpHash for the audit is correct
1581 if(CommandAuditIsRequired(command->index))
1582 {
1583     result = CheckCommandAudit(command);
1584     if(result != TPM_RC_SUCCESS)
1585         return result; // No session number to reference
1586 }
1587 #endif
1588 // Decrypt the first parameter if applicable. This should be the last operation
1589 // in session processing.
1590 // If the encrypt session is associated with a handle and the handle's
1591 // authValue is available, then authValue is concatenated with sessionKey to
1592 // generate encryption key, no matter if the handle is the session bound entity
1593 // or not.
1594 if(s_decryptSessionIndex != UNDEFINED_INDEX)
1595 {
1596     // If this is an authorization session, include the authValue in the
1597     // generation of the decryption key
1598     if(s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED)
1599     {

```



```

1600         EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex], &extraKey);
1601     }
1602     else
1603     {
1604         extraKey.b.size = 0;
1605     }
1606     size = DecryptSize(command->index);
1607     result = CryptParameterDecryption(s_sessionHandles[s_decryptSessionIndex],
1608                                     &s_nonceCaller[s_decryptSessionIndex].b,
1609                                     command->parameterSize,
1610                                     (UINT16)size,
1611                                     &extraKey,
1612                                     command->parameterBuffer);
1613     if(result != TPM_RC_SUCCESS)
1614         return RcSafeAddToResult(result,
1615                                   TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1616 }
1617
1618 return TPM_RC_SUCCESS;
1619 }
1620
1621 /*** CheckAuthNoSession()
1622 // Function to process a command with no session associated.
1623 // The function makes sure all the handles in the command require no authorization.
1624 //
1625 // Return Type: TPM_RC
1626 //     TPM_RC_AUTH_MISSING    failure - one or more handles require
1627 //                             authorization
1628 TPM_RC
1629 CheckAuthNoSession(COMMAND* command // IN: command parsing structure
1630 )
1631 {
1632     UINT32 i;
1633     #if CC_GetCommandAuditDigest
1634         TPM_RC result = TPM_RC_SUCCESS;
1635     #endif
1636     //
1637     // Check if the command requires authorization
1638     for(i = 0; i < command->handleNum; i++)
1639     {
1640         if(CommandAuthRole(command->index, i) != AUTH_NONE)
1641             return TPM_RC_AUTH_MISSING;
1642     }
1643     #if CC_GetCommandAuditDigest
1644         // Check if the command should be audited.
1645         if(CommandAuditIsRequired(command->index))
1646         {
1647             result = CheckCommandAudit(command);
1648             if(result != TPM_RC_SUCCESS)
1649                 return result;
1650         }
1651     #endif
1652     // Initialize number of sessions to be 0
1653     command->sessionNum = 0;
1654
1655     return TPM_RC_SUCCESS;
1656 }
1657
1658 /*** Response Session Processing
1659 /*** Introduction
1660 //
1661 // The following functions build the session area in a response and handle
1662 // the audit sessions (if present).
1663 //
1664
1665 /*** ComputeRpHash()

```

```

1666 // Function to compute rpHash (Response Parameter Hash). The rpHash is only
1667 // computed if there is an HMAC authorization session and the return code is
1668 // TPM_RC_SUCCESS.
1669 static TPM2B_DIGEST* ComputeRpHash(
1670     COMMAND* command, // IN: command structure
1671     TPM_ALG_ID hashAlg // IN: hash algorithm to compute rpHash
1672 )
1673 {
1674     TPM2B_DIGEST* rpHash = GetRpHashPointer(command, hashAlg);
1675     HASH_STATE hashState;
1676     //
1677     if(rpHash->t.size == 0)
1678     {
1679         // rpHash := hash(responseCode || commandCode || parameters)
1680
1681         // Initiate hash creation.
1682         rpHash->t.size = CryptHashStart(&hashState, hashAlg);
1683
1684         // Add hash constituents.
1685         CryptDigestUpdateInt(&hashState, sizeof(TPM_RC), TPM_RC_SUCCESS);
1686         CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
1687         CryptDigestUpdate(
1688             &hashState, command->parameterSize, command->parameterBuffer);
1689         // Complete hash computation.
1690         CryptHashEnd2B(&hashState, &rpHash->b);
1691     }
1692     return rpHash;
1693 }
1694
1695 /*** InitAuditSession()
1696 // This function initializes the audit data in an audit session.
1697 static void InitAuditSession(SESSION* session // session to be initialized
1698 )
1699 {
1700     // Mark session as an audit session.
1701     session->attributes.isAudit = SET;
1702
1703     // Audit session can not be bound.
1704     session->attributes.isBound = CLEAR;
1705
1706     // Size of the audit log is the size of session hash algorithm digest.
1707     session->u2.auditDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
1708
1709     // Set the original digest value to be 0.
1710     MemorySet(&session->u2.auditDigest.t.buffer, 0, session->u2.auditDigest.t.size);
1711     return;
1712 }
1713
1714 /*** UpdateAuditDigest
1715 // Function to update an audit digest
1716 static void UpdateAuditDigest(
1717     COMMAND* command, TPMI_ALG_HASH hashAlg, TPM2B_DIGEST* digest)
1718 {
1719     HASH_STATE hashState;
1720     TPM2B_DIGEST* cpHash = GetCpHash(command, hashAlg);
1721     TPM2B_DIGEST* rpHash = ComputeRpHash(command, hashAlg);
1722     //
1723     pAssert(cpHash != NULL);
1724
1725     // digestNew := hash (digestOld || cpHash || rpHash)
1726     // Start hash computation.
1727     digest->t.size = CryptHashStart(&hashState, hashAlg);
1728     // Add old digest.
1729     CryptDigestUpdate2B(&hashState, &digest->b);
1730     // Add cpHash
1731     CryptDigestUpdate2B(&hashState, &cpHash->b);

```

```

1732     // Add rpHash
1733     CryptDigestUpdate2B(&hashState, &rpHash->b);
1734     // Finalize the hash.
1735     CryptHashEnd2B(&hashState, &digest->b);
1736 }
1737
1738 /*** Audit()
1739 //This function updates the audit digest in an audit session.
1740 static void Audit(COMMAND* command,      // IN: primary control structure
1741                  SESSION* auditSession // IN: loaded audit session
1742 )
1743 {
1744     UpdateAuditDigest(
1745         command, auditSession->authHashAlg, &auditSession->u2.auditDigest);
1746     return;
1747 }
1748
1749 #if CC_GetCommandAuditDigest
1750 /*** CommandAudit()
1751 // This function updates the command audit digest.
1752 static void CommandAudit(COMMAND* command // IN:
1753 )
1754 {
1755     // If the digest.size is one, it indicates the special case of changing
1756     // the audit hash algorithm. For this case, no audit is done on exit.
1757     // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1758     // force an update to the NV on exit so that the change in digest will
1759     // be recorded. So, it is safe to exit here without setting any flags
1760     // because the digest change will be written to NV when this code exits.
1761     if(gr.commandAuditDigest.t.size == 1)
1762     {
1763         gr.commandAuditDigest.t.size = 0;
1764         return;
1765     }
1766     // If the digest size is zero, need to start a new digest and increment
1767     // the audit counter.
1768     if(gr.commandAuditDigest.t.size == 0)
1769     {
1770         gr.commandAuditDigest.t.size = CryptHashGetDigestSize(gp.auditHashAlg);
1771         MemorySet(gr.commandAuditDigest.t.buffer, 0, gr.commandAuditDigest.t.size);
1772
1773         // Bump the counter and save its value to NV.
1774         gp.auditCounter++;
1775         NV_SYNC_PERSISTENT(auditCounter);
1776     }
1777     UpdateAuditDigest(command, gp.auditHashAlg, &gr.commandAuditDigest);
1778     return;
1779 }
1780 #endif
1781
1782 /*** UpdateAuditSessionStatus()
1783 // This function updates the internal audit related states of a session. It will:
1784 // a) initialize the session as audit session and set it to be exclusive if this
1785 //    is the first time it is used for audit or audit reset was requested;
1786 // b) report exclusive audit session;
1787 // c) extend audit log; and
1788 // d) clear exclusive audit session if no audit session found in the command.
1789 static void UpdateAuditSessionStatus(
1790     COMMAND* command // IN: primary control structure
1791 )
1792 {
1793     UINT32 i;
1794     TPM_HANDLE auditSession = TPM_RH_UNASSIGNED;
1795     //
1796     // Iterate through sessions
1797     for(i = 0; i < command->sessionNum; i++)

```

```

1798 {
1799     SESSION* session;
1800     //
1801     // PW session do not have a loaded session and can not be an audit
1802     // session either. Skip it.
1803     if(s_sessionHandles[i] == TPM_RS_PW)
1804         continue;
1805     session = SessionGet(s_sessionHandles[i]);
1806
1807     // If a session is used for audit
1808     if(IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, audit))
1809     {
1810         // An audit session has been found
1811         auditSession = s_sessionHandles[i];
1812
1813         // If the session has not been an audit session yet, or
1814         // the auditSetting bits indicate a reset, initialize it and set
1815         // it to be the exclusive session
1816         if(session->attributes.isAudit == CLEAR
1817            || IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditReset))
1818         {
1819             InitAuditSession(session);
1820             g_exclusiveAuditSession = auditSession;
1821         }
1822         else
1823         {
1824             // Check if the audit session is the current exclusive audit
1825             // session and, if not, clear previous exclusive audit session.
1826             if(g_exclusiveAuditSession != auditSession)
1827                 g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1828         }
1829         // Report audit session exclusivity.
1830         if(g_exclusiveAuditSession == auditSession)
1831         {
1832             SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1833         }
1834         else
1835         {
1836             CLEAR_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1837         }
1838         // Extend audit log.
1839         Audit(command, session);
1840     }
1841 }
1842 // If no audit session is found in the command, and the command allows
1843 // a session then, clear the current exclusive
1844 // audit session.
1845 if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(command->index))
1846 {
1847     g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1848 }
1849 return;
1850 }
1851
1852 /*** ComputeResponseHMAC()
1853 // Function to compute HMAC for authorization session in a response.
1854 /*(See part 1 specification)
1855 // Function: Compute HMAC for response sessions
1856 //     The sessionAuth value
1857 //     authHMAC := HMACsHASH((sessionAuth | authValue),
1858 //                          (pHash | nonceTPM | nonceCaller | sessionAttributes))
1859 // Where:
1860 //     HMACsHASH()     The HMAC algorithm using the hash algorithm specified when
1861 //                     the session was started.
1862 //
1863 //     sessionAuth     A TPMB_MEDIUM computed in a protocol-dependent way, using

```

```

1864 //          KDFa. In an HMAC or KDF, only sessionAuth.buffer is used.
1865 //
1866 //      authValue      A TPM2B_AUTH that is found in the sensitive area of an
1867 //                      object. In an HMAC or KDF, only authValue.buffer is used
1868 //                      and all trailing zeros are removed.
1869 //
1870 //      pHash          Response parameters (rpHash) using the session hash. When
1871 //                      using a pHash in an HMAC computation, both the algorithm ID
1872 //                      and the digest are included.
1873 //
1874 //      nonceTPM        A TPM2B_NONCE that is generated by the entity using the
1875 //                      session. In an HMAC or KDF, only nonceTPM.buffer is used.
1876 //
1877 //      nonceCaller     a TPM2B_NONCE that was received the previous time the
1878 //                      session was used. In an HMAC or KDF, only
1879 //                      nonceCaller.buffer is used.
1880 //
1881 //      sessionAttributes A TPMA_SESSION that indicates the attributes associated
1882 //                      with a particular use of the session.
1883 */
1884 static void ComputeResponseHMAC(
1885     COMMAND*      command,      // IN: command structure
1886     UINT32        sessionIndex,  // IN: session index to be processed
1887     SESSION*      session,      // IN: loaded session
1888     TPM2B_DIGEST* hmac         // OUT: authHMAC
1889 )
1890 {
1891     TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1892     TPM2B_KEY      key; // HMAC key
1893     BYTE           marshalBuffer[sizeof(TPMA_SESSION)];
1894     BYTE*          buffer;
1895     UINT32          marshalSize;
1896     HMAC_STATE      hmacState;
1897     TPM2B_DIGEST*  rpHash = ComputeRpHash(command, session->authHashAlg);
1898     //
1899     // Generate HMAC key
1900     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1901
1902     // Add the object authValue if required
1903     if(session->attributes.includeAuth == SET)
1904     {
1905         // Note: includeAuth may be SET for a policy that is used in
1906         // UndefineSpaceSpecial(). At this point, the Index has been deleted
1907         // so the includeAuth will have no meaning. However, the
1908         // s_associatedHandles[] value for the session is now set to TPM_RH_NULL so
1909         // this will return the authValue associated with TPM_RH_NULL and that is
1910         // and empty buffer.
1911         TPM2B_AUTH authValue;
1912         //
1913         // Get the authValue with trailing zeros removed
1914         EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
1915
1916         // Add it to the key
1917         MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
1918     }
1919
1920     // if the HMAC key size is 0, the response HMAC is computed according to the
1921     // input HMAC
1922     if(key.t.size == 0 && s_inputAuthValues[sessionIndex].t.size == 0)
1923     {
1924         hmac->t.size = 0;
1925         return;
1926     }
1927     // Start HMAC computation.
1928     hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
1929

```

```

1930 // Add hash components.
1931 CryptDigestUpdate2B(&hmacState.hashState, &rpHash->b);
1932 CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
1933 CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
1934
1935 // Add session attributes.
1936 buffer = marshalBuffer;
1937 marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1938 CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
1939
1940 // Finalize HMAC.
1941 CryptHmacEnd2B(&hmacState, &hmac->b);
1942
1943 return;
1944 }
1945
1946 /*** UpdateInternalSession()
1947 // This function updates internal sessions by:
1948 // a) restarting session time; and
1949 // b) clearing a policy session since nonce is rolling.
1950 static void UpdateInternalSession(SESSION* session, // IN: the session structure
1951                                UINT32 i // IN: session number
1952 )
1953 {
1954     // If nonce is rolling in a policy session, the policy related data
1955     // will be re-initialized.
1956     if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION
1957         && IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1958     {
1959         // When the nonce rolls it starts a new timing interval for the
1960         // policy session.
1961         SessionResetPolicyData(session);
1962         SessionSetStartTime(session);
1963     }
1964     return;
1965 }
1966
1967 /*** BuildSingleResponseAuth()
1968 // Function to compute response HMAC value for a policy or HMAC session.
1969 static TPM2B_NONCE* BuildSingleResponseAuth(
1970     COMMAND* command, // IN: command structure
1971     UINT32 sessionIndex, // IN: session index to be processed
1972     TPM2B_AUTH* auth // OUT: authHMAC
1973 )
1974 {
1975     // Fill in policy/HMAC based session response.
1976     SESSION* session = SessionGet(s_sessionHandles[sessionIndex]);
1977     //
1978     // If the session is a policy session with isPasswordNeeded SET, the
1979     // authorization field is empty.
1980     if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1981         && session->attributes.isPasswordNeeded == SET)
1982         auth->t.size = 0;
1983     else
1984         // Compute response HMAC.
1985         ComputeResponseHMAC(command, sessionIndex, session, auth);
1986
1987     UpdateInternalSession(session, sessionIndex);
1988     return &session->nonceTPM;
1989 }
1990
1991 /*** UpdateAllNonceTPM()
1992 // Updates TPM nonce for all sessions in command.
1993 static void UpdateAllNonceTPM(COMMAND* command // IN: controlling structure
1994 )
1995 {

```



```

1996     UINT32    i;
1997     SESSION* session;
1998     //
1999     for(i = 0; i < command->sessionNum; i++)
2000     {
2001         // If not a PW session, compute the new nonceTPM.
2002         if(s_sessionHandles[i] != TPM_RS_PW)
2003         {
2004             session = SessionGet(s_sessionHandles[i]);
2005             // Update nonceTPM in both internal session and response.
2006             CryptRandomGenerate(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
2007         }
2008     }
2009     return;
2010 }
2011
2012 /*** BuildResponseSession()
2013 // Function to build Session buffer in a response. The authorization data is added
2014 // to the end of command->responseBuffer. The size of the authorization area is
2015 // accumulated in command->authSize.
2016 // When this is called, command->responseBuffer is pointing at the next location
2017 // in the response buffer to be filled. This is where the authorization sessions
2018 // will go, if any. command->parameterSize is the number of bytes that have been
2019 // marshaled as parameters in the output buffer.
2020 TPM_RC
2021 BuildResponseSession(COMMAND* command // IN: structure that has relevant command
2022                      // information
2023 )
2024 {
2025     TPM_RC result = TPM_RC_SUCCESS;
2026
2027     pAssert(command->authSize == 0);
2028
2029     // Reset the parameter buffer to point to the start of the parameters so that
2030     // there is a starting point for any rpHash that might be generated and so there
2031     // is a place where parameter encryption would start
2032     command->parameterBuffer = command->responseBuffer - command->parameterSize;
2033
2034     // Session nonces should be updated before parameter encryption
2035     if(command->tag == TPM_ST_SESSIONS)
2036     {
2037         UpdateAllNonceTPM(command);
2038
2039         // Encrypt first parameter if applicable. Parameter encryption should
2040         // happen after nonce update and before any rpHash is computed.
2041         // If the encrypt session is associated with a handle, the authValue of
2042         // this handle will be concatenated with sessionKey to generate
2043         // encryption key, no matter if the handle is the session bound entity
2044         // or not. The authValue is added to sessionKey only when the authValue
2045         // is available.
2046         if(s_encryptSessionIndex != UNDEFINED_INDEX)
2047         {
2048             UINT32    size;
2049             TPM2B_AUTH extraKey;
2050             //
2051             extraKey.b.size = 0;
2052             // If this is an authorization session, include the authValue in the
2053             // generation of the encryption key
2054             if(s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED)
2055             {
2056                 EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
2057                                   &extraKey);
2058             }
2059             size = EncryptSize(command->index);
2060             // This function operates on internally-generated data that is
2061             // expected to be well-formed for parameter encryption.

```

```

2062     // In the event that there is a bug elsewhere in the code and the
2063     // input data is not well-formed, CryptParameterEncryption will
2064     // put the TPM into failure mode instead of allowing the out-of-
2065     // band write.
2066     CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
2067                             &s_nonceCaller[s_encryptSessionIndex].b,
2068                             command->parameterSize,
2069                             (UINT16)size,
2070                             &extraKey,
2071                             command->parameterBuffer);
2072
2073     if(g_inFailureMode)
2074     {
2075         result = TPM_RC_FAILURE;
2076         goto Cleanup;
2077     }
2078 }
2079 // Audit sessions should be processed regardless of the tag because
2080 // a command with no session may cause a change of the exclusivity state.
2081 UpdateAuditSessionStatus(command);
2082 #if CC_GetCommandAuditDigest
2083     // Command Audit
2084     if(CommandAuditIsRequired(command->index))
2085         CommandAudit(command);
2086 #endif
2087 // Process command with sessions.
2088 if(command->tag == TPM_ST_SESSIONS)
2089 {
2090     UINT32 i;
2091     //
2092     pAssert(command->sessionNum > 0);
2093
2094     // Iterate over each session in the command session area, and create
2095     // corresponding sessions for response.
2096     for(i = 0; i < command->sessionNum; i++)
2097     {
2098         TPM2B_NONCE* nonceTPM;
2099         TPM2B_DIGEST responseAuth;
2100         // Make sure that continueSession is SET on any Password session.
2101         // This makes it marginally easier for the management software
2102         // to keep track of the closed sessions.
2103         if(s_sessionHandles[i] == TPM_RS_PW)
2104         {
2105             SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession);
2106             responseAuth.t.size = 0;
2107             nonceTPM = (TPM2B_NONCE*)&responseAuth;
2108         }
2109         else
2110         {
2111             // Compute the response HMAC and get a pointer to the nonce used.
2112             // This function will also update the values if needed. Note, the
2113             nonceTPM = BuildSingleResponseAuth(command, i, &responseAuth);
2114         }
2115         command->authSize +=
2116             TPM2B_NONCE_Marshal(nonceTPM, &command->responseBuffer, NULL);
2117         command->authSize += TPMA_SESSION_Marshal(
2118             &s_attributes[i], &command->responseBuffer, NULL);
2119         command->authSize +=
2120             TPM2B_DIGEST_Marshal(&responseAuth, &command->responseBuffer, NULL);
2121         if(!IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
2122             SessionFlush(s_sessionHandles[i]);
2123     }
2124 }
2125
2126 Cleanup:
2127     return result;

```

```

2128 }
2129
2130 /*** SessionRemoveAssociationToHandle()
2131 // This function deals with the case where an entity associated with an authorization
2132 // is deleted during command processing. The primary use of this is to support
2133 // UndefineSpaceSpecial().
2134 void SessionRemoveAssociationToHandle(TPM_HANDLE handle)
2135 {
2136     UINT32 i;
2137     //
2138     for(i = 0; i < MAX_SESSION_NUM; i++)
2139     {
2140         if(s_associatedHandles[i] == HierarchyNormalizeHandle(handle))
2141         {
2142             s_associatedHandles[i] = TPM_RH_NULL;
2143         }
2144     }
2145 }

```

7.170 /tpm/src/subsystem/CommandAudit.c

```

1  /*** Introduction
2  // This file contains the functions that support command audit.
3
4  /*** Includes
5  #include "Tpm.h"
6
7  /*** Functions
8
9  /*** CommandAuditPreInstall_Init()
10 // This function initializes the command audit list. This function simulates
11 // the behavior of manufacturing. A function is used instead of a structure
12 // definition because this is easier than figuring out the initialization value
13 // for a bit array.
14 //
15 // This function would not be implemented outside of a manufacturing or
16 // simulation environment.
17 void CommandAuditPreInstall_Init(void)
18 {
19     // Clear all the audit commands
20     MemorySet(gp.auditCommands, 0x00, sizeof(gp.auditCommands));
21
22     // TPM_CC_SetCommandCodeAuditStatus always being audited
23     CommandAuditSet(TPM_CC_SetCommandCodeAuditStatus);
24
25     // Set initial command audit hash algorithm to be context integrity hash
26     // algorithm
27     gp.auditHashAlg = CONTEXT_INTEGRITY_HASH_ALG;
28
29     // Set up audit counter to be 0
30     gp.auditCounter = 0;
31
32     // Write command audit persistent data to NV
33     NV_SYNC_PERSISTENT(auditCommands);
34     NV_SYNC_PERSISTENT(auditHashAlg);
35     NV_SYNC_PERSISTENT(auditCounter);
36
37     return;
38 }
39
40 /*** CommandAuditStartup()
41 // This function clears the command audit digest on a TPM Reset.
42 BOOL CommandAuditStartup(STARTUP_TYPE type // IN: start up type
43 )
44 {

```

```

45     if((type != SU_RESTART) && (type != SU_RESUME))
46     {
47         // Reset the digest size to initialize the digest
48         gr.commandAuditDigest.t.size = 0;
49     }
50     return TRUE;
51 }
52
53 /*** CommandAuditSet()
54 // This function will SET the audit flag for a command. This function
55 // will not SET the audit flag for a command that is not implemented. This
56 // ensures that the audit status is not SET when TPM2_GetCapability() is
57 // used to read the list of audited commands.
58 //
59 // This function is only used by TPM2_SetCommandCodeAuditStatus().
60 //
61 // The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the
62 // changes to be saved to NV after it is setting and clearing bits.
63 // Return Type: BOOL
64 //     TRUE(1)         command code audit status was changed
65 //     FALSE(0)        command code audit status was not changed
66 BOOL CommandAuditSet(TPM_CC commandCode // IN: command code
67 )
68 {
69     COMMAND_INDEX commandIndex = CommandCodeToCommandIndex(commandCode);
70
71     // Only SET a bit if the corresponding command is implemented
72     if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
73     {
74         // Can't audit shutdown
75         if(commandCode != TPM_CC_Shutdown)
76         {
77             if(!TEST_BIT(commandIndex, gp.auditCommands))
78             {
79                 // Set bit
80                 SET_BIT(commandIndex, gp.auditCommands);
81                 return TRUE;
82             }
83         }
84     }
85     // No change
86     return FALSE;
87 }
88
89 /*** CommandAuditClear()
90 // This function will CLEAR the audit flag for a command. It will not CLEAR the
91 // audit flag for TPM_CC_SetCommandCodeAuditStatus().
92 //
93 // This function is only used by TPM2_SetCommandCodeAuditStatus().
94 //
95 // The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the
96 // changes to be saved to NV after it is setting and clearing bits.
97 // Return Type: BOOL
98 //     TRUE(1)         command code audit status was changed
99 //     FALSE(0)        command code audit status was not changed
100 BOOL CommandAuditClear(TPM_CC commandCode // IN: command code
101 )
102 {
103     COMMAND_INDEX commandIndex = CommandCodeToCommandIndex(commandCode);
104
105     // Do nothing if the command is not implemented
106     if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
107     {
108         // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
109         // cleared
110         if(commandCode != TPM_CC_SetCommandCodeAuditStatus)

```

```

111     {
112         if(TEST_BIT(commandIndex, gp.auditCommands))
113         {
114             // Clear bit
115             CLEAR_BIT(commandIndex, gp.auditCommands);
116             return TRUE;
117         }
118     }
119 }
120 // No change
121 return FALSE;
122 }
123
124 /*** CommandAuditIsRequired()
125 // This function indicates if the audit flag is SET for a command.
126 // Return Type: BOOL
127 //     TRUE(1)         command is audited
128 //     FALSE(0)        command is not audited
129 BOOL CommandAuditIsRequired(COMMAND_INDEX commandIndex // IN: command index
130 )
131 {
132     // Check the bit map. If the bit is SET, command audit is required
133     return (TEST_BIT(commandIndex, gp.auditCommands));
134 }
135
136 /*** CommandAuditCapGetCCList()
137 // This function returns a list of commands that have their audit bit SET.
138 //
139 // The list starts at the input commandCode.
140 // Return Type: TPMI_YES_NO
141 //     YES         if there are more command code available
142 //     NO          all the available command code has been returned
143 TPMI_YES_NO
144 CommandAuditCapGetCCList(TPM_CC commandCode, // IN: start command code
145                          UINT32 count,       // IN: count of returned TPM_CC
146                          TPML_CC* commandList // OUT: list of TPM_CC
147 )
148 {
149     TPMI_YES_NO more = NO;
150     COMMAND_INDEX commandIndex;
151
152     // Initialize output handle list
153     commandList->count = 0;
154
155     // The maximum count of command we may return is MAX_CAP_CC
156     if(count > MAX_CAP_CC)
157         count = MAX_CAP_CC;
158
159     // Find the implemented command that has a command code that is the same or
160     // higher than the input
161     // Collect audit commands
162     for(commandIndex = GetClosestCommandIndex(commandCode);
163         commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
164         commandIndex = GetNextCommandIndex(commandIndex))
165     {
166         if(CommandAuditIsRequired(commandIndex))
167         {
168             if(commandList->count < count)
169             {
170                 // If we have not filled up the return list, add this command
171                 // code to its
172                 TPM_CC cc =
173                     GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex);
174                 if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
175                     cc += (1 << 29);
176                 commandList->commandCodes[commandList->count] = cc;

```

```

177         commandList->count++;
178     }
179     else
180     {
181         // If the return list is full but we still have command
182         // available, report this and stop iterating
183         more = YES;
184         break;
185     }
186 }
187 }
188
189 return more;
190 }
191
192 /*** CommandAuditCapGetOneCC()
193 // This function returns true if a command has its audit bit set.
194 BOOL CommandAuditCapGetOneCC(TPM_CC commandCode) // IN: command code
195 {
196     COMMAND_INDEX commandIndex = CommandCodeToCommandIndex(commandCode);
197     if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
198     {
199         return CommandAuditIsRequired(commandIndex);
200     }
201     return FALSE;
202 }
203
204 /*** CommandAuditGetDigest
205 // This command is used to create a digest of the commands being audited. The
206 // commands are processed in ascending numeric order with a list of TPM_CC being
207 // added to a hash. This operates as if all the audited command codes were
208 // concatenated and then hashed.
209 void CommandAuditGetDigest(TPM2B_DIGEST* digest // OUT: command digest
210 )
211 {
212     TPM_CC      commandCode;
213     COMMAND_INDEX commandIndex;
214     HASH_STATE   hashState;
215
216     // Start hash
217     digest->t.size = CryptHashStart(&hashState, gp.auditHashAlg);
218
219     // Add command code
220     for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
221     {
222         if(CommandAuditIsRequired(commandIndex))
223         {
224             commandCode = GetCommandCode(commandIndex);
225             CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
226         }
227     }
228
229     // Complete hash
230     CryptHashEnd2B(&hashState, &digest->b);
231
232     return;
233 }

```

7.171 /tpm/src/subsystem/DA.c

```

1  /*** Introduction
2  // This file contains the functions and data definitions relating to the
3  // dictionary attack logic.
4
5  /*** Includes and Data Definitions

```



```

6  #define DA_C
7  #include "Tpm.h"
8
9  /** Functions
10
11  /** DAPreInstall_Init()
12  // This function initializes the DA parameters to their manufacturer-default
13  // values. The default values are determined by a platform-specific specification.
14  //
15  // This function should not be called outside of a manufacturing or simulation
16  // environment.
17  //
18  // The DA parameters will be restored to these initial values by TPM2_Clear().
19  void DAPreInstall_Init(void)
20  {
21      gp.failedTries      = 0;
22      gp.maxTries         = 3;
23      gp.recoveryTime     = 1000; // in seconds (~16.67 minutes)
24      gp.lockoutRecovery  = 1000; // in seconds
25      gp.lockOutAuthEnabled = TRUE; // Use of lockoutAuth is enabled
26
27      // Record persistent DA parameter changes to NV
28      NV_SYNC_PERSISTENT(failedTries);
29      NV_SYNC_PERSISTENT(maxTries);
30      NV_SYNC_PERSISTENT(recoveryTime);
31      NV_SYNC_PERSISTENT(lockoutRecovery);
32      NV_SYNC_PERSISTENT(lockOutAuthEnabled);
33
34      return;
35  }
36
37  /** DASTartup()
38  // This function is called by TPM2_Startup() to initialize the DA parameters.
39  // In the case of Startup(CLEAR), use of lockoutAuth will be enabled if the
40  // lockout recovery time is 0. Otherwise, lockoutAuth will not be enabled until
41  // the TPM has been continuously powered for the lockoutRecovery time.
42  //
43  // This function requires that NV be available and not rate limiting.
44  BOOL DASTartup(STARTUP_TYPE type // IN: startup type
45  )
46  {
47      NOT_REFERENCED(type);
48      #if !ACCUMULATE_SELF_HEAL_TIMER
49          _plat_TimerWasReset();
50          s_selfHealTimer = 0;
51          s_lockoutTimer  = 0;
52      #else
53          if(_plat_TimerWasReset())
54          {
55              if(!NV_IS_ORDERLY)
56              {
57                  // If shutdown was not orderly, then don't really know if go.time has
58                  // any useful value so reset the timer to 0. This is what the tick
59                  // was reset to
60                  s_selfHealTimer = 0;
61                  s_lockoutTimer  = 0;
62              }
63              else
64              {
65                  // If we know how much time was accumulated at the last orderly shutdown
66                  // subtract that from the saved timer values so that they effectively
67                  // have the accumulated values
68                  s_selfHealTimer -= go.time;
69                  s_lockoutTimer  -= go.time;
70              }
71          }

```

```

72  #endif
73
74      // For any Startup(), if lockoutRecovery is 0, enable use of lockoutAuth.
75      if(gp.lockoutRecovery == 0)
76      {
77          gp.lockOutAuthEnabled = TRUE;
78          // Record the changes to NV
79          NV_SYNC_PERSISTENT(lockOutAuthEnabled);
80      }
81
82      // If DA has not been disabled and the previous shutdown is not orderly
83      // failedTries is not already at its maximum then increment 'failedTries'
84      if(gp.recoveryTime != 0 && gp.failedTries < gp.maxTries
85         && !IS_ORDERLY(g_prevOrderlyState))
86      {
87  #if USE_DA_USED
88          gp.failedTries += g_daUsed;
89          g_daUsed = FALSE;
90  #else
91          gp.failedTries++;
92  #endif
93          // Record the change to NV
94          NV_SYNC_PERSISTENT(failedTries);
95      }
96      // Before Startup, the TPM will not do clock updates. At startup, need to
97      // do a time update which will do the DA update.
98      TimeUpdate();
99
100     return TRUE;
101 }
102
103 /*** DRegisterFailure()
104 // This function is called when an authorization failure occurs on an entity
105 // that is subject to dictionary-attack protection. When a DA failure is
106 // triggered, register the failure by resetting the relevant self-healing
107 // timer to the current time.
108 void DRegisterFailure(TPM_HANDLE handle // IN: handle for failure
109 )
110 {
111     // Reset the timer associated with lockout if the handle is the lockoutAuth.
112     if(handle == TPM_RH_LOCKOUT)
113         s_lockoutTimer = g_time;
114     else
115         s_selfHealTimer = g_time;
116     return;
117 }
118
119 /*** DASelfHeal()
120 // This function is called to check if sufficient time has passed to allow
121 // decrement of failedTries or to re-enable use of lockoutAuth.
122 //
123 // This function should be called when the time interval is updated.
124 void DASelfHeal(void)
125 {
126     // Regular authorization self healing logic
127     // If no failed authorization tries, do nothing. Otherwise, try to
128     // decrease failedTries
129     if(gp.failedTries != 0)
130     {
131         // if recovery time is 0, DA logic has been disabled. Clear failed tries
132         // immediately
133         if(gp.recoveryTime == 0)
134         {
135             gp.failedTries = 0;
136             // Update NV record
137             NV_SYNC_PERSISTENT(failedTries);

```

```

138     }
139     else
140     {
141         UINT64 decreaseCount;
142         #if 0
143             // Errata eliminates this code
144             // In the unlikely event that failedTries should become larger than
145             // maxTries
146             if(gp.failedTries > gp.maxTries)
147                 gp.failedTries = gp.maxTries;
148         #endif
149         // How much can failedTries be decreased
150
151         // Cast s_selfHealTimer to an int in case it became negative at
152         // startup
153         decreaseCount =
154             ((g_time - (INT64)s_selfHealTimer) / 1000) / gp.recoveryTime;
155
156         if(gp.failedTries <= (UINT32)decreaseCount)
157             // should not set failedTries below zero
158             gp.failedTries = 0;
159         else
160             gp.failedTries -= (UINT32)decreaseCount;
161
162         // the cast prevents overflow of the product
163         s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
164         if(decreaseCount != 0)
165             // If there was a change to the failedTries, record the changes
166             // to NV
167             NV_SYNC_PERSISTENT(failedTries);
168     }
169 }
170
171 // LockoutAuth self healing logic
172 // If lockoutAuth is enabled, do nothing. Otherwise, try to see if we
173 // may enable it
174 if(!gp.lockOutAuthEnabled)
175 {
176     // if lockout authorization recovery time is 0, a reboot is required to
177     // re-enable use of lockout authorization. Self-healing would not
178     // apply in this case.
179     if(gp.lockoutRecovery != 0)
180     {
181         if(((g_time - (INT64)s_lockoutTimer) / 1000) >= gp.lockoutRecovery)
182         {
183             gp.lockOutAuthEnabled = TRUE;
184             // Record the changes to NV
185             NV_SYNC_PERSISTENT(lockOutAuthEnabled);
186         }
187     }
188 }
189 return;
190 }

```

7.172 /tpm/src/subsystem/Hierarchy.c

```

1  /** Introduction
2  // This file contains the functions used for managing and accessing the
3  // hierarchy-related values.
4
5  /** Includes
6
7  #include "Tpm.h"
8
9  /**HIERARCHY_MODIFIER_TYPE

```

```

10 // This enumerates the possible hierarchy modifiers.
11 typedef enum
12 {
13     HM_NONE = 0,
14     HM_FW_LIMITED, // Hierarchy is firmware-limited.
15     HM_SVN_LIMITED // Hierarchy is SVN-limited.
16 } HIERARCHY_MODIFIER_TYPE;
17
18 /*** HIERARCHY_MODIFIER Structure
19 // A HIERARCHY_MODIFIER structure holds metadata about an OBJECT's
20 // hierarchy modifier.
21 typedef struct HIERARCHY_MODIFIER
22 {
23     HIERARCHY_MODIFIER_TYPE type; // The type of modification.
24     uint16_t min_svn; // The minimum SVN to which the hierarchy is limited.
25                     // Only valid if 'type' is HM_SVN_LIMITED.
26 } HIERARCHY_MODIFIER;
27
28 /*** Functions
29
30 /*** HierarchyPreInstall()
31 // This function performs the initialization functions for the hierarchy
32 // when the TPM is simulated. This function should not be called if the
33 // TPM is not in a manufacturing mode at the manufacturer, or in a simulated
34 // environment.
35 void HierarchyPreInstall_Init(void)
36 {
37     // Allow lockout clear command
38     gp.disableClear = FALSE;
39
40     // Initialize Primary Seeds
41     gp.EPSeed.t.size = sizeof(gp.EPSeed.t.buffer);
42     gp.SPSeed.t.size = sizeof(gp.SPSeed.t.buffer);
43     gp.PPSeed.t.size = sizeof(gp.PPSeed.t.buffer);
44 #if (defined USE_PLATFORM_EPS) && (USE_PLATFORM_EPS != NO)
45     _plat_GetEPS(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
46 #else
47     CryptRandomGenerate(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
48 #endif
49     CryptRandomGenerate(gp.SPSeed.t.size, gp.SPSeed.t.buffer);
50     CryptRandomGenerate(gp.PPSeed.t.size, gp.PPSeed.t.buffer);
51
52     // Initialize owner, endorsement and lockout authorization
53     gp.ownerAuth.t.size = 0;
54     gp.endorsementAuth.t.size = 0;
55     gp.lockoutAuth.t.size = 0;
56
57     // Initialize owner, endorsement, and lockout policy
58     gp.ownerAlg = TPM_ALG_NULL;
59     gp.ownerPolicy.t.size = 0;
60     gp.endorsementAlg = TPM_ALG_NULL;
61     gp.endorsementPolicy.t.size = 0;
62     gp.lockoutAlg = TPM_ALG_NULL;
63     gp.lockoutPolicy.t.size = 0;
64
65     // Initialize ehProof, shProof and phProof
66     gp.phProof.t.size = sizeof(gp.phProof.t.buffer);
67     gp.shProof.t.size = sizeof(gp.shProof.t.buffer);
68     gp.ehProof.t.size = sizeof(gp.ehProof.t.buffer);
69     CryptRandomGenerate(gp.phProof.t.size, gp.phProof.t.buffer);
70     CryptRandomGenerate(gp.shProof.t.size, gp.shProof.t.buffer);
71     CryptRandomGenerate(gp.ehProof.t.size, gp.ehProof.t.buffer);
72
73     // Write hierarchy data to NV
74     NV_SYNC_PERSISTENT(disableClear);
75     NV_SYNC_PERSISTENT(EPSeed);

```

```

76     NV_SYNC_PERSISTENT(SPSeed);
77     NV_SYNC_PERSISTENT(PPSeed);
78     NV_SYNC_PERSISTENT(ownerAuth);
79     NV_SYNC_PERSISTENT(endorsementAuth);
80     NV_SYNC_PERSISTENT(lockoutAuth);
81     NV_SYNC_PERSISTENT(ownerAlg);
82     NV_SYNC_PERSISTENT(ownerPolicy);
83     NV_SYNC_PERSISTENT(endorsementAlg);
84     NV_SYNC_PERSISTENT(endorsementPolicy);
85     NV_SYNC_PERSISTENT(lockoutAlg);
86     NV_SYNC_PERSISTENT(lockoutPolicy);
87     NV_SYNC_PERSISTENT(phProof);
88     NV_SYNC_PERSISTENT(shProof);
89     NV_SYNC_PERSISTENT(ehProof);
90
91     return;
92 }
93
94 /*** HierarchyStartup()
95 // This function is called at TPM2_Startup() to initialize the hierarchy
96 // related values.
97 BOOL HierarchyStartup(STARTUP_TYPE type // IN: start up type
98 )
99 {
100     // phEnable is SET on any startup
101     g_phEnable = TRUE;
102
103     // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
104     // TPM_RESTART
105     if(type != SU_RESUME)
106     {
107         gc.platformAuth.t.size = 0;
108         gc.platformPolicy.t.size = 0;
109         gc.platformAlg = TPM_ALG_NULL;
110
111         // enable the storage and endorsement hierarchies and the platformNV
112         gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
113     }
114
115     // nullProof and nullSeed are updated at every TPM_RESET
116     if((type != SU_RESTART) && (type != SU_RESUME))
117     {
118         gr.nullProof.t.size = sizeof(gr.nullProof.t.buffer);
119         CryptRandomGenerate(gr.nullProof.t.size, gr.nullProof.t.buffer);
120         gr.nullSeed.t.size = sizeof(gr.nullSeed.t.buffer);
121         CryptRandomGenerate(gr.nullSeed.t.size, gr.nullSeed.t.buffer);
122     }
123
124     return TRUE;
125 }
126
127 /*** DecomposeHandle()
128 // This function extracts the base hierarchy and modifier from a given handle.
129 // Returns the base hierarchy.
130 static TPMI_RH_HIERARCHY DecomposeHandle(TPMI_RH_HIERARCHY handle, // IN
131                                          HIERARCHY_MODIFIER* modifier // OUT
132 )
133 {
134     TPMI_RH_HIERARCHY base_hierarchy = handle;
135
136     modifier->type = HM_NONE;
137
138     // See if the handle is firmware-bound.
139     switch(handle)
140     {
141         case TPMI_RH_FW_OWNER:

```

```

142     {
143         modifier->type = HM_FW_LIMITED;
144         base_hierarchy = TPM_RH_OWNER;
145         break;
146     }
147     case TPM_RH_FW_ENDORSEMENT:
148     {
149         modifier->type = HM_FW_LIMITED;
150         base_hierarchy = TPM_RH_ENDORSEMENT;
151         break;
152     }
153     case TPM_RH_FW_PLATFORM:
154     {
155         modifier->type = HM_FW_LIMITED;
156         base_hierarchy = TPM_RH_PLATFORM;
157         break;
158     }
159     case TPM_RH_FW_NULL:
160     {
161         modifier->type = HM_FW_LIMITED;
162         base_hierarchy = TPM_RH_NULL;
163         break;
164     }
165 }
166
167 if(modifier->type == HM_FW_LIMITED)
168 {
169     return base_hierarchy;
170 }
171
172 // See if the handle is SVN-bound.
173 switch(handle & 0xFFFF0000)
174 {
175     case TPM_RH_SVN_OWNER_BASE:
176         modifier->type = HM_SVN_LIMITED;
177         base_hierarchy = TPM_RH_OWNER;
178         break;
179     case TPM_RH_SVN_ENDORSEMENT_BASE:
180         modifier->type = HM_SVN_LIMITED;
181         base_hierarchy = TPM_RH_ENDORSEMENT;
182         break;
183     case TPM_RH_SVN_PLATFORM_BASE:
184         modifier->type = HM_SVN_LIMITED;
185         base_hierarchy = TPM_RH_PLATFORM;
186         break;
187     case TPM_RH_SVN_NULL_BASE:
188         modifier->type = HM_SVN_LIMITED;
189         base_hierarchy = TPM_RH_NULL;
190         break;
191 }
192
193 if(modifier->type == HM_SVN_LIMITED)
194 {
195     modifier->min_svn = handle & 0x0000FFFF;
196     return base_hierarchy;
197 }
198
199 // Handle is neither FW- nor SVN-bound; return it unmodified.
200 return handle;
201 }
202
203 /*** GetAdditionalSecret()
204 // Retrieve the additional secret for the given hierarchy modifier, along with the
205 // label that should be used when mixing the secret into a KDF. If the hierarchy
206 // needs no additional secret, secret_buffer's size is set to zero and secret_label
207 // is set to NULL.

```



```

208 //
209 // Return Type: TPM_RC
210 //     TPM_RC_FW_LIMITED           The requested hierarchy is FW-limited, but the TPM
211 //                                does not support FW-limited objects or the TPM failed
212 //                                to derive the Firmware Secret.
213 //     TPM_RC_SVN_LIMITED         The requested hierarchy is SVN-limited, but the TPM
214 //                                does not support SVN-limited objects or the TPM failed
215 //                                to derive the Firmware SVN Secret for the requested
216 //                                SVN.
217 static TPM_RC GetAdditionalSecret(const HIERARCHY_MODIFIER* modifier,           // IN
218                                 TPM2B_SEED* secret_buffer,                   // OUT
219                                 const TPM2B** secret_label                    // OUT
220 )
221 {
222     switch(modifier->type)
223     {
224         case HM_FW_LIMITED:
225         {
226             #if FW_LIMITED_SUPPORT
227                 if(_plat__GetTpmFirmwareSecret(sizeof(secret_buffer->t.buffer),
228                                                 secret_buffer->t.buffer,
229                                                 &secret_buffer->t.size)
230                     != 0)
231                 {
232                     return TPM_RC_FW_LIMITED;
233                 }
234
235                 *secret_label = HIERARCHY_FW_SECRET_LABEL;
236                 break;
237             #else
238                 return TPM_RC_FW_LIMITED;
239             #endif // FW_LIMITED_SUPPORT
240         }
241         case HM_SVN_LIMITED:
242         {
243             #if SVN_LIMITED_SUPPORT
244                 if(_plat__GetTpmFirmwareSvnSecret(modifier->min_svn,
245                                                    sizeof(secret_buffer->t.buffer),
246                                                    secret_buffer->t.buffer,
247                                                    &secret_buffer->t.size)
248                     != 0)
249                 {
250                     return TPM_RC_SVN_LIMITED;
251                 }
252
253                 *secret_label = HIERARCHY_SVN_SECRET_LABEL;
254                 break;
255             #else
256                 return TPM_RC_SVN_LIMITED;
257             #endif // SVN_LIMITED_SUPPORT
258         }
259         case HM_NONE:
260         default:
261         {
262             secret_buffer->t.size = 0;
263             *secret_label          = NULL;
264             break;
265         }
266     }
267
268     return TPM_RC_SUCCESS;
269 }
270
271 /***MixAdditionalSecret()
272 // This function obtains the additional secret for the hierarchy and
273 // mixes it into the base secret. The output buffer must have the same

```

```

274 // capacity as the base secret. The output buffer's size is set to the
275 // base secret size. If no additional secret is needed, the base secret
276 // is copied to the output buffer.
277 //
278 // Return Type: TPM_RC
279 //     TPM_RC_FW_LIMITED      The requested hierarchy is FW-limited, but the TPM
280 //                             does not support FW-limited objects or the TPM failed
281 //                             to derive the Firmware Secret.
282 //     TPM_RC_SVN_LIMITED     The requested hierarchy is SVN-limited, but the TPM
283 //                             does not support SVN-limited objects or the TPM failed
284 //                             to derive the Firmware SVN Secret for the requested
285 //                             SVN.
286 TPM_RC MixAdditionalSecret(const HIERARCHY_MODIFIER* modifier,          // IN
287                           const TPM2B* base_secret_label,            // IN
288                           const TPM2B* base_secret,                  // IN
289                           TPM2B* output_secret                       // OUT
290 )
291 {
292     TPM_RC result = TPM_RC_SUCCESS;
293     TPM2B_SEED additional_secret;
294     const TPM2B* additional_secret_label = NULL;
295
296     result =
297         GetAdditionalSecret(modifier, &additional_secret, &additional_secret_label);
298     if(result != TPM_RC_SUCCESS)
299         return result;
300
301     output_secret->size = base_secret->size;
302
303     if(additional_secret.b.size == 0)
304     {
305         memcpy(output_secret->buffer, base_secret->buffer, base_secret->size);
306     }
307     else
308     {
309         CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG,
310                 base_secret,
311                 base_secret_label,
312                 &additional_secret.b,
313                 additional_secret_label,
314                 base_secret->size * 8,
315                 output_secret->buffer,
316                 NULL,
317                 FALSE);
318     }
319
320     MemorySet(additional_secret.b.buffer, 0, additional_secret.b.size);
321
322     return TPM_RC_SUCCESS;
323 }
324
325 /*** HierarchyGetProof()
326 // This function derives the proof value associated with a hierarchy. It returns a
327 // buffer containing the proof value.
328 TPM_RC HierarchyGetProof(TPMI_RH HIERARCHY hierarchy, // IN: hierarchy constant
329                         TPM2B_PROOF* proof           // OUT: proof buffer
330 )
331 {
332     TPM2B_PROOF* base_proof = NULL;
333     HIERARCHY_MODIFIER modifier;
334
335     switch(DecomposeHandle(hierarchy, &modifier))
336     {
337     case TPMI_RH_PLATFORM:
338         // phProof for TPMI_RH_PLATFORM
339         base_proof = &gp.phProof;

```

```

340         break;
341     case TPM_RH_ENDORSEMENT:
342         // ehProof for TPM_RH_ENDORSEMENT
343         base_proof = &gp.ehProof;
344         break;
345     case TPM_RH_OWNER:
346         // shProof for TPM_RH_OWNER
347         base_proof = &gp.shProof;
348         break;
349     default:
350         // nullProof for TPM_RH_NULL or anything else
351         base_proof = &gr.nullProof;
352         break;
353 }
354
355 return MixAdditionalSecret(
356     &modifier, HIERARCHY_PROOF_SECRET_LABEL, &base_proof->b, &proof->b);
357 }
358
359 /*** HierarchyGetPrimarySeed()
360 // This function derives the primary seed of a hierarchy.
361 TPM_RC HierarchyGetPrimarySeed(TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy
362                                TPM2B_SEED* seed // OUT: seed buffer
363 )
364 {
365     TPM2B_SEED* base_seed = NULL;
366     HIERARCHY_MODIFIER modifier;
367
368     switch(DecomposeHandle(hierarchy, &modifier))
369     {
370     case TPM_RH_PLATFORM:
371         base_seed = &gp.PPSeed;
372         break;
373     case TPM_RH_OWNER:
374         base_seed = &gp.SPSeed;
375         break;
376     case TPM_RH_ENDORSEMENT:
377         base_seed = &gp.EPSeed;
378         break;
379     default:
380         base_seed = &gr.nullSeed;
381         break;
382     }
383
384     return MixAdditionalSecret(
385         &modifier, HIERARCHY_SEED_SECRET_LABEL, &base_seed->b, &seed->b);
386 }
387
388 /*** ValidateHierarchy()
389 // This function ensures a given hierarchy is valid and enabled.
390 // Return Type: TPM_RC
391 // TPM_RC_HIERARCHY Hierarchy is disabled
392 // TPM_RC_FW_LIMITED The requested hierarchy is FW-limited, but the TPM
393 // does not support FW-limited objects.
394 // TPM_RC_SVN_LIMITED The requested hierarchy is SVN-limited, but the TPM
395 // does not support SVN-limited objects or the given SVN
396 // is greater than the TPM's current SVN.
397 // TPM_RC_VALUE Hierarchy is not valid
398 TPM_RC ValidateHierarchy(TPMI_RH_HIERARCHY hierarchy // IN: hierarchy
399 )
400 {
401     BOOL enabled;
402     HIERARCHY_MODIFIER modifier;
403
404     hierarchy = DecomposeHandle(hierarchy, &modifier);
405

```

```

406     // Modifier-specific checks.
407     switch(modifier.type)
408     {
409         case HM_NONE:
410             break;
411         case HM_FW_LIMITED:
412             {
413 #if FW_LIMITED_SUPPORT
414                 break;
415 #else
416                 return TPM_RC_FW_LIMITED;
417 #endif // FW_LIMITED_SUPPORT
418             }
419         case HM_SVN_LIMITED:
420             {
421 #if SVN_LIMITED_SUPPORT
422                 // SVN-limited hierarchies are only enabled for SVN's less than or
423                 // equal to the current firmware's SVN.
424                 if(modifier.min_svn > _plat__GetTpmFirmwareSvn())
425                 {
426                     return TPM_RC_SVN_LIMITED;
427                 }
428                 break;
429 #else
430                 return TPM_RC_SVN_LIMITED;
431 #endif // SVN_LIMITED_SUPPORT
432             }
433     }
434
435     switch(hierarchy)
436     {
437         case TPM_RH_PLATFORM:
438             enabled = g_phEnable;
439             break;
440         case TPM_RH_OWNER:
441             enabled = gc.shEnable;
442             break;
443         case TPM_RH_ENDORSEMENT:
444             enabled = gc.ehEnable;
445             break;
446         case TPM_RH_NULL:
447             enabled = TRUE;
448             break;
449         default:
450             return TPM_RC_VALUE;
451     }
452
453     return enabled ? TPM_RC_SUCCESS : TPM_RC_HIERARCHY;
454 }
455
456 /*** HierarchyIsEnabled()
457 // This function checks to see if a hierarchy is enabled.
458 // NOTE: The TPM_RH_NULL hierarchy is always enabled.
459 // Return Type: BOOL
460 //     TRUE(1)         hierarchy is enabled
461 //     FALSE(0)        hierarchy is disabled
462 BOOL HierarchyIsEnabled(TPMI_RH_HIERARCHY hierarchy // IN: hierarchy
463 )
464 {
465     return ValidateHierarchy(hierarchy) == TPM_RC_SUCCESS;
466 }
467
468 /*** HierarchyNormalizeHandle
469 // This function accepts a handle that may or may not be FW- or SVN-bound,
470 // and returns the base hierarchy to which the handle refers.
471 TPMI_RH_HIERARCHY HierarchyNormalizeHandle(TPMI_RH_HIERARCHY handle // IN: handle

```

```

472 )
473 {
474     HIERARCHY_MODIFIER unused_modifier;
475
476     return DecomposeHandle(handle, &unused_modifier);
477 }
478
479 /*** HierarchyIsFirmwareLimited
480 // This function accepts a hierarchy handle and returns whether it is firmware-
481 // limited.
482 BOOL HierarchyIsFirmwareLimited(TPMI_RH_HIERARCHY handle // IN
483 )
484 {
485     HIERARCHY_MODIFIER modifier;
486
487     DecomposeHandle(handle, &modifier);
488
489     return modifier.type == HM_FW_LIMITED;
490 }
491 /*** HierarchyIsSvnLimited
492 // This function accepts a hierarchy handle and returns whether it is SVN-
493 // limited.
494 BOOL HierarchyIsSvnLimited(TPMI_RH_HIERARCHY handle // IN
495 )
496 {
497     HIERARCHY_MODIFIER modifier;
498
499     DecomposeHandle(handle, &modifier);
500
501     return modifier.type == HM_SVN_LIMITED;
502 }

```

7.173 /tpm/src/subsystem/NvDynamic.c

```

1  /*** Introduction
2
3  // The NV memory is divided into two areas: dynamic space for user defined NV
4  // indexes and evict objects, and reserved space for TPM persistent and state save
5  // data.
6  //
7  // The entries in dynamic space are a linked list of entries. Each entry has, as its
8  // first field, a size. If the size field is zero, it marks the end of the
9  // list.
10 //
11 // An Index allocation will contain an NV_INDEX structure. If the Index does not
12 // have the orderly attribute, the NV_INDEX is followed immediately by the NV data.
13 //
14 // An evict object entry contains a handle followed by an OBJECT structure. This
15 // results in both the Index and Evict Object having an identifying handle as the
16 // first field following the size field.
17 //
18 // When an Index has the orderly attribute, the data is kept in RAM. This RAM is
19 // saved to backing store in NV memory on any orderly shutdown. The entries in
20 // orderly memory are also a linked list using a size field as the first entry.
21 //
22 // The attributes of an orderly index are maintained in RAM memory in order to
23 // reduce the number of NV writes needed for orderly data. When an orderly index
24 // is created, an entry is made in the dynamic NV memory space that holds the Index
25 // authorizations (authPolicy and authValue) and the size of the data. This entry is
26 // only modified if the authValue of the index is changed. The more volatile data
27 // of the index is kept in RAM. When an orderly Index is created or deleted, the
28 // RAM data is copied to NV backing store so that the image in the backing store
29 // matches the layout of RAM. In normal operation, the RAM data is also copied on
30 // any orderly shutdown. In normal operation, the only other reason for writing
31 // to the backing store for RAM is when a counter is first written (TPMA_NV_WRITTEN

```

```

32 // changes from CLEAR to SET) or when a counter "rolls over".
33 //
34 // Static space contains items that are individually modifiable. The values are in
35 // the 'gp' PERSISTENT_DATA structure in RAM and mapped to locations in NV.
36 //
37
38 /** Includes, Defines and Data Definitions
39 #define NV_C
40 #include "Tpm.h"
41 #include "Marshal.h"
42
43 /** Local Functions
44
45 /*** NvNext()
46 // This function provides a method to traverse every data entry in NV dynamic
47 // area.
48 //
49 // To begin with, parameter 'iter' should be initialized to NV_REF_INIT
50 // indicating the first element. Every time this function is called, the
51 // value in 'iter' would be adjusted pointing to the next element in
52 // traversal. If there is no next element, 'iter' value would be 0.
53 // This function returns the address of the 'data entry' pointed by the
54 // 'iter'. If there are no more elements in the set, a 0 value is returned
55 // indicating the end of traversal.
56 //
57 static NV_REF NvNext(NV_REF* iter, // IN/OUT: the list iterator
58                     TPM_HANDLE* handle // OUT: the handle of the next item.
59 )
60 {
61     NV_REF currentAddr;
62     NV_ENTRY_HEADER header;
63     //
64     // If iterator is at the beginning of list
65     if(*iter == NV_REF_INIT)
66     {
67         // Initialize iterator
68         *iter = NV_USER_DYNAMIC;
69     }
70     // Step over the size field and point to the handle
71     currentAddr = *iter + sizeof(UINT32);
72
73     // read the header of the next entry
74     NvRead(&header, *iter, sizeof(NV_ENTRY_HEADER));
75
76     // if the size field is zero, then we have hit the end of the list
77     if(header.size == 0)
78     {
79         // leave the *iter pointing at the end of the list
80         return 0;
81     }
82     // advance the header by the size of the entry
83     *iter += header.size;
84
85     if(handle != NULL)
86         *handle = header.handle;
87     return currentAddr;
88 }
89
90 /*** NvNextByType()
91 // This function returns a reference to the next NV entry of the desired type
92 // Return Type: NV_REF
93 // 0 end of list
94 // != 0 the next entry of the indicated type
95 static NV_REF NvNextByType(
96     TPM_HANDLE* handle, // OUT: the handle of the found type or 0
97     NV_REF* iter, // IN: the iterator
98     TPM_HT type // IN: the handle type to look for
99 )

```



```

98 {
99     NV_REF    addr;
100     TPM_HANDLE nvHandle = 0;
101     //
102     while((addr = NvNext(iter, &nvHandle)) != 0)
103     {
104         // addr: the address of the location containing the handle of the value
105         // iter: the next location.
106         if(HandleGetType(nvHandle) == type)
107             break;
108     }
109     if(handle != NULL)
110         *handle = nvHandle;
111     return addr;
112 }
113
114 /*** NvNextIndex()
115 // This function returns the reference to the next NV Index entry. A value
116 // of 0 indicates the end of the list.
117 // Return Type: NV_REF
118 // 0          end of list
119 // != 0       the next reference
120 #define NvNextIndex(handle, iter) NvNextByType(handle, iter, TPM_HT_NV_INDEX)
121
122 /*** NvNextEvict()
123 // This function returns the offset in NV of the next evict object entry. A value
124 // of 0 indicates the end of the list.
125 #define NvNextEvict(handle, iter) NvNextByType(handle, iter, TPM_HT_PERSISTENT)
126
127 /*** NvGetEnd()
128 // Function to find the end of the NV dynamic data list
129 static NV_REF NvGetEnd(void)
130 {
131     NV_REF iter = NV_REF_INIT;
132     NV_REF currentAddr;
133     //
134     // Scan until the next address is 0
135     while((currentAddr = NvNext(&iter, NULL)) != 0)
136         ;
137     return iter;
138 }
139
140 /*** NvGetFreeBytes
141 // This function returns the number of free octets in NV space.
142 static UINT32 NvGetFreeBytes(void)
143 {
144     // This does not have an overflow issue because NvGetEnd() cannot return a value
145     // that is larger than s_evictNvEnd. This is because there is always a 'stop'
146     // word in the NV memory that terminates the search for the end before the
147     // value can go past s_evictNvEnd.
148     return s_evictNvEnd - NvGetEnd();
149 }
150
151 /*** NvTestSpace()
152 // This function will test if there is enough space to add a new entity.
153 // Return Type: BOOL
154 // TRUE(1)      space available
155 // FALSE(0)     no enough space
156 static BOOL NvTestSpace(UINT32 size,      // IN: size of the entity to be added
157                        BOOL isIndex,     // IN: TRUE if the entity is an index
158                        BOOL isCounter    // IN: TRUE if the index is a counter
159 )
160 {
161     UINT32 remainBytes = NvGetFreeBytes();
162     UINT32 reserved    = sizeof(UINT32) // size of the forward pointer
163                        + sizeof(NV_LIST_TERMINATOR);

```

```

164 //
165 // Do a compile time sanity check on the setting for NV_MEMORY_SIZE
166 #if NV_MEMORY_SIZE < 1024
167 # error "NV_MEMORY_SIZE probably isn't large enough"
168 #endif
169
170 // For NV Index, need to make sure that we do not allocate an Index if this
171 // would mean that the TPM cannot allocate the minimum number of evict
172 // objects.
173 if(isIndex)
174 {
175     // Get the number of persistent objects allocated
176     UINT32 persistentNum = NvCapGetPersistentNumber();
177
178     // If we have not allocated the requisite number of evict objects, then we
179     // need to reserve space for them.
180     // NOTE: some of this is not written as simply as it might seem because
181     // the values are all unsigned and subtracting needs to be done carefully
182     // so that an underflow doesn't cause problems.
183     if(persistentNum < MIN_EVICT_OBJECTS)
184         reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
185 }
186 // If this is not an index or is not a counter, reserve space for the
187 // required number of counter indexes
188 if(!isIndex || !isCounter)
189 {
190     // Get the number of counters
191     UINT32 counterNum = NvCapGetCounterNumber();
192
193     // If the required number of counters have not been allocated, reserved
194     // space for the extra needed counters
195     if(counterNum < MIN_COUNTER_INDICES)
196         reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
197 }
198 // Check that the requested allocation will fit after making sure that there
199 // will be no chance of overflow
200 return ((reserved < remainBytes) && (size <= remainBytes)
201         && (size + reserved <= remainBytes));
202 }
203
204 /*** NvWriteNvListEnd()
205 // Function to write the list terminator.
206 NV_REF
207 NvWriteNvListEnd(NV_REF end)
208 {
209     // Marker is initialized with zeros
210     BYTE listEndMarker[sizeof(NV_LIST_TERMINATOR)] = {0};
211     UINT64 maxCount = NvReadMaxCount();
212     //
213     // This is a constant check that can be resolved at compile time.
214     MUST_BE(sizeof(UINT64) <= sizeof(NV_LIST_TERMINATOR) - sizeof(UINT32));
215
216     // Copy the maxCount value to the marker buffer
217     MemoryCopy(&listEndMarker[sizeof(UINT32)], &maxCount, sizeof(UINT64));
218     pAssert(end + sizeof(NV_LIST_TERMINATOR) <= s_evictNvEnd);
219
220     // Write it to memory
221     NvWrite(end, sizeof(NV_LIST_TERMINATOR), &listEndMarker);
222     return end + sizeof(NV_LIST_TERMINATOR);
223 }
224
225 /*** NvAdd()
226 // This function adds a new entity to NV.
227 //
228 // This function requires that there is enough space to add a new entity (i.e.,
229 // that NvTestSpace() has been called and the available space is at least as

```

```

230 // large as the required space).
231 //
232 // The 'totalSize' will be the size of 'entity'. If a handle is added, this
233 // function will increase the size accordingly.
234 static TPM_RC NvAdd(UINT32 totalSize, // IN: total size needed for this entity For
235                     // evict object, totalSize is the same as
236                     // bufferSize. For NV Index, totalSize is
237                     // bufferSize plus index data size
238                     UINT32 bufferSize, // IN: size of initial buffer
239                     TPM_HANDLE handle, // IN: optional handle
240                     BYTE* entity // IN: initial buffer
241 )
242 {
243     NV_REF newAddr; // IN: where the new entity will start
244     NV_REF nextAddr;
245     //
246     RETURN_IF_NV_IS_NOT_AVAILABLE;
247
248     // Get the end of data list
249     newAddr = NvGetEnd();
250
251     // Step over the forward pointer
252     nextAddr = newAddr + sizeof(UINT32);
253
254     // Optionally write the handle. For indexes, the handle is TPM_RH_UNASSIGNED
255     // so that the handle in the nvIndex is used instead of writing this value
256     if(handle != TPM_RH_UNASSIGNED)
257     {
258         NvWrite((UINT32)newAddr, sizeof(TPM_HANDLE), &handle);
259         nextAddr += sizeof(TPM_HANDLE);
260     }
261     // Write entity data
262     NvWrite((UINT32)newAddr, bufferSize, entity);
263
264     // Advance the pointer by the amount of the total
265     nextAddr += totalSize;
266
267     // Finish by writing the link value
268
269     // Write the next offset (relative addressing)
270     totalSize = nextAddr - newAddr;
271
272     // Write link value
273     NvWrite((UINT32)newAddr, sizeof(UINT32), &totalSize);
274
275     // Write the list terminator
276     NvWriteNvListEnd(nextAddr);
277
278     return TPM_RC_SUCCESS;
279 }
280
281 /*** NvDelete()
282 // This function is used to delete an NV Index or persistent object from NV memory.
283 static TPM_RC NvDelete(NV_REF entityRef // IN: reference to entity to be deleted
284 )
285 {
286     UINT32 entrySize;
287     // adjust entityAddr to back up and point to the forward pointer
288     NV_REF entryRef = entityRef - sizeof(UINT32);
289     NV_REF endRef = NvGetEnd();
290     NV_REF nextAddr; // address of the next entry
291     //
292     RETURN_IF_NV_IS_NOT_AVAILABLE;
293
294     // Get the offset of the next entry. That is, back up and point to the size
295     // field of the entry

```

```

296     NvRead(&entrySize, entryRef, sizeof(UINT32));
297
298     // The next entry after the one being deleted is at a relative offset
299     // from the current entry
300     nextAddr = entryRef + entrySize;
301
302     // If this is not the last entry, move everything up
303     if(nextAddr < endRef)
304     {
305         pAssert(nextAddr > entryRef);
306         _plat__NvMemoryMove(nextAddr, entryRef, (endRef - nextAddr));
307     }
308     // The end of the used space is now moved up by the amount of space we just
309     // reclaimed
310     endRef -= entrySize;
311
312     // Write the end marker, and make the new end equal to the first byte after
313     // the just added end value. This will automatically update the NV value for
314     // maxCounter.
315     // NOTE: This is the call that sets flag to cause NV to be updated
316     endRef = NvWriteNvListEnd(endRef);
317
318     // Clear the reclaimed memory
319     _plat__NvMemoryClear(endRef, entrySize);
320
321     return TPM_RC_SUCCESS;
322 }
323
324 /*******
325 /*** RAM-based NV Index Data Access Functions
326 /*******
327 /*** Introduction
328 /** The data layout in ram buffer is {size of(NV_handle + attributes + data
329 /** NV_handle, attributes, data}
330 /** for each NV Index data stored in RAM.
331 /**
332 /** NV storage associated with orderly data is updated when a NV Index is added
333 /** but NOT when the data or attributes are changed. Orderly data is only updated
334 /** to NV on an orderly shutdown (TPM2_Shutdown())
335
336 /*** NvRamNext()
337 /** This function is used to iterate through the list of Ram Index values. *iter needs
338 /** to be initialized by calling
339 static NV_RAM_REF NvRamNext(NV_RAM_REF* iter, // IN/OUT: the list iterator
340                             TPM_HANDLE* handle // OUT: the handle of the next item.
341 )
342 {
343     NV_RAM_REF currentAddr;
344     NV_RAM_HEADER header;
345     //
346     // If iterator is at the beginning of list
347     if(*iter == NV_RAM_REF_INIT)
348     {
349         // Initialize iterator
350         *iter = &s_indexOrderlyRam[0];
351     }
352     // if we are going to return what the iter is currently pointing to...
353     currentAddr = *iter;
354
355     // If iterator reaches the end of NV space, then don't advance and return
356     // that we are at the end of the list. The end of the list occurs when
357     // we don't have space for a size and a handle
358     if(currentAddr + sizeof(NV_RAM_HEADER) > RAM_ORDERLY_END)
359         return NULL;
360     // read the header of the next entry
361     MemoryCopy(&header, currentAddr, sizeof(NV_RAM_HEADER));

```

```

362
363 // if the size field is zero, then we have hit the end of the list
364 if(header.size == 0)
365     // leave the *iter pointing at the end of the list
366     return NULL;
367 // advance the header by the size of the entry
368 *iter = currentAddr + header.size;
369
370 // pAssert(*iter <= RAM_ORDERLY_END);
371 if(handle != NULL)
372     *handle = header.handle;
373 return currentAddr;
374 }
375
376 /*** NvRamGetEnd()
377 // This routine performs the same function as NvGetEnd() but for the RAM data.
378 static NV_RAM_REF NvRamGetEnd(void)
379 {
380     NV_RAM_REF iter = NV_RAM_REF_INIT;
381     NV_RAM_REF currentAddr;
382     //
383     // Scan until the next address is 0
384     while((currentAddr = NvRamNext(&iter, NULL)) != 0)
385         ;
386     return iter;
387 }
388
389 /*** NvRamTestSpaceIndex()
390 // This function indicates if there is enough RAM space to add a data for a
391 // new NV Index.
392 // Return Type: BOOL
393 //     TRUE(1)      space available
394 //     FALSE(0)     no enough space
395 static BOOL NvRamTestSpaceIndex(
396     UINT32 size // IN: size of the data to be added to RAM
397 )
398 {
399     UINT32 remaining = (UINT32)(RAM_ORDERLY_END - NvRamGetEnd());
400     UINT32 needed    = sizeof(NV_RAM_HEADER) + size;
401     //
402     // NvRamGetEnd points to the next available byte.
403     return remaining >= needed;
404 }
405
406 /*** NvRamGetIndex()
407 // This function returns the offset of NV data in the RAM buffer
408 //
409 // This function requires that NV Index is in RAM. That is, the
410 // index must be known to exist.
411 static NV_RAM_REF NvRamGetIndex(TPMI_RH_NV_INDEX handle // IN: NV handle
412 )
413 {
414     NV_RAM_REF iter = NV_RAM_REF_INIT;
415     NV_RAM_REF currentAddr;
416     TPM_HANDLE foundHandle;
417     //
418     while((currentAddr = NvRamNext(&iter, &foundHandle)) != 0)
419     {
420         if(handle == foundHandle)
421             break;
422     }
423     return currentAddr;
424 }
425
426 /*** NvUpdateIndexOrderlyData()
427 // This function is used to cause an update of the orderly data to the NV backing

```

```

428 // store.
429 void NvUpdateIndexOrderlyData(void)
430 {
431     // Write reserved RAM space to NV
432     NvWrite(NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam), s_indexOrderlyRam);
433 }
434
435 /*** NvAddRAM()
436 // This function adds a new data area to RAM.
437 //
438 // This function requires that enough free RAM space is available to add
439 // the new data.
440 //
441 // This function should be called after the NV Index space has been updated
442 // and the index removed. This insures that NV is available so that checking
443 // for NV availability is not required during this function.
444 static void NvAddRAM(TPMS_NV_PUBLIC* index // IN: the index descriptor
445 )
446 {
447     NV_RAM_HEADER header;
448     NV_RAM_REF end = NvRamGetEnd();
449     //
450     header.size = sizeof(NV_RAM_HEADER) + index->dataSize;
451     header.handle = index->nvIndex;
452     MemoryCopy(&header.attributes, &index->attributes, sizeof(TPMA_NV));
453
454     pAssert(ORDERLY_RAM_ADDRESS_OK(end, header.size));
455
456     // Copy the header to the memory
457     MemoryCopy(end, &header, sizeof(NV_RAM_HEADER));
458
459     // Clear the data area (just in case)
460     MemorySet(end + sizeof(NV_RAM_HEADER), 0, index->dataSize);
461
462     // Step over this new entry
463     end += header.size;
464
465     // If the end marker will fit, add it
466     if(end + sizeof(UINT32) < RAM_ORDERLY_END)
467         MemorySet(end, 0, sizeof(UINT32));
468     // Write reserved RAM space to NV to reflect the newly added NV Index
469     SET_NV_UPDATE(UT_ORDERLY);
470
471     return;
472 }
473
474 /*** NvDeleteRAM()
475 // This function is used to delete a RAM-backed NV Index data area.
476 // The space used by the entry are overwritten by the contents of the
477 // Index data that comes after (the data is moved up to fill the hole left
478 // by removing this index. The reclaimed space is cleared to zeros.
479 // This function assumes the data of NV Index exists in RAM.
480 //
481 // This function should be called after the NV Index space has been updated
482 // and the index removed. This insures that NV is available so that checking
483 // for NV availability is not required during this function.
484 static void NvDeleteRAM(TPMI_RH_NV_INDEX handle // IN: NV handle
485 )
486 {
487     NV_RAM_REF nodeAddress;
488     NV_RAM_REF nextNode;
489     UINT32 size;
490     NV_RAM_REF lastUsed = NvRamGetEnd();
491     //
492     nodeAddress = NvRamGetIndex(handle);
493

```



```

494     pAssert(nodeAddress != 0);
495
496     // Get node size
497     MemoryCopy(&size, nodeAddress, sizeof(size));
498
499     // Get the offset of next node
500     nextNode = nodeAddress + size;
501
502     // Copy the data
503     MemoryCopy(nodeAddress, nextNode, (int)(lastUsed - nextNode));
504
505     // Clear out the reclaimed space
506     MemorySet(lastUsed - size, 0, size);
507
508     // Write reserved RAM space to NV to reflect the newly delete NV Index
509     SET_NV_UPDATE(UT_ORDERLY);
510
511     return;
512 }
513
514 /*** NvReadIndex()
515 // This function is used to read the NV Index NV_INDEX. This is used so that the
516 // index information can be compressed and only this function would be needed
517 // to decompress it. Mostly, compression would only be able to save the space
518 // needed by the policy.
519 void NvReadNvIndexInfo(NV_REF ref, // IN: points to NV where index is located
520                       NV_INDEX* nvIndex // OUT: place to receive index data
521 )
522 {
523     pAssert(nvIndex != NULL);
524     NvRead(nvIndex, ref, sizeof(NV_INDEX));
525     return;
526 }
527
528 /*** NvReadObject()
529 // This function is used to read a persistent object. This is used so that the
530 // object information can be compressed and only this function would be needed
531 // to uncompress it.
532 void NvReadObject(NV_REF ref, // IN: points to NV where index is located
533                  OBJECT* object // OUT: place to receive the object data
534 )
535 {
536     NvRead(object, (ref + sizeof(TPM_HANDLE)), sizeof(OBJECT));
537     return;
538 }
539
540 /*** NvFindEvict()
541 // This function will return the NV offset of an evict object
542 // Return Type: UINT32
543 // 0 evict object not found
544 // != 0 offset of evict object
545 static NV_REF NvFindEvict(TPM_HANDLE nvHandle, OBJECT* object)
546 {
547     NV_REF found = NvFindHandle(nvHandle);
548     //
549     // If we found the handle and the request included an object pointer, fill it in
550     if(found != 0 && object != NULL)
551         NvReadObject(found, object);
552     return found;
553 }
554
555 /*** NvIndexIsDefined()
556 // See if an index is already defined
557 BOOL NvIndexIsDefined(TPM_HANDLE nvHandle // IN: Index to look for
558 )
559 {

```

```

560     return (NvFindHandle(nvHandle) != 0);
561 }
562
563 /*** NvConditionallyWrite()
564 // Function to check if the data to be written has changed
565 // and write it if it has
566 // Return Type: TPM_RC
567 //     TPM_RC_NV_RATE          NV is unavailable because of rate limit
568 //     TPM_RC_NV_UNAVAILABLE   NV is inaccessible
569 static TPM_RC NvConditionallyWrite(NV_REF entryAddr, // IN: stating address
570                                  UINT32 size,      // IN: size of the data to write
571                                  void* data        // IN: the data to write
572 )
573 {
574     // If the index data is actually changed, then a write to NV is required
575     int isDifferent = _plat_NvGetChangedStatus(entryAddr, size, data);
576     if(isDifferent == NV_INVALID_LOCATION)
577     {
578         // invalid request, we should be in failure mode by now.
579         return TPM_RC_FAILURE;
580     }
581     else if(isDifferent == NV_HAS_CHANGED)
582     {
583         // Write the data if NV is available
584         if(g_NvStatus == TPM_RC_SUCCESS)
585         {
586             NvWrite(entryAddr, size, data);
587         }
588         return g_NvStatus;
589     }
590     else if(isDifferent == NV_IS_SAME)
591     {
592         return TPM_RC_SUCCESS;
593     }
594     // the platform gave us an invalid response.
595     FAIL_RC(FATAL_ERROR_PLATFORM);
596 }
597
598 /*** NvReadNvIndexAttributes()
599 // This function returns the attributes of an NV Index.
600 static TPMA_NV NvReadNvIndexAttributes(NV_REF locator // IN: reference to an NV index
601 )
602 {
603     TPMA_NV attributes;
604     //
605     NvRead(&attributes,
606           locator + offsetof(NV_INDEX, publicArea.attributes),
607           sizeof(TPMA_NV));
608     return attributes;
609 }
610
611 /*** NvReadRamIndexAttributes()
612 // This function returns the attributes from the RAM header structure. This function
613 // is used to deal with the fact that the header structure is only byte aligned.
614 static TPMA_NV NvReadRamIndexAttributes(
615     NV_RAM_REF ref // IN: pointer to a NV_RAM_HEADER
616 )
617 {
618     TPMA_NV attributes;
619     //
620     MemoryCopy(
621         &attributes, ref + offsetof(NV_RAM_HEADER, attributes), sizeof(TPMA_NV));
622     return attributes;
623 }
624
625 /*** NvWriteNvIndexAttributes()

```

```

626 // This function is used to write just the attributes of an index to NV.
627 // Return type: TPM_RC
628 //     TPM_RC_NV_RATE           NV is rate limiting so retry
629 //     TPM_RC_NV_UNAVAILABLE    NV is not available
630 static TPM_RC NvWriteNvIndexAttributes(NV_REF locator, // IN: location of the index
631                                       TPMA_NV attributes // IN: attributes to write
632 )
633 {
634     return NvConditionallyWrite(locator + offsetof(NV_INDEX, publicArea.attributes),
635                                sizeof(TPMA_NV),
636                                &attributes);
637 }
638
639 /*** NvWriteRamIndexAttributes()
640 // This function is used to write the index attributes into an unaligned structure
641 static void NvWriteRamIndexAttributes(
642     NV_RAM_REF ref,           // IN: address of the header
643     TPMA_NV attributes // IN: the attributes to write
644 )
645 {
646     MemoryCopy(
647         ref + offsetof(NV_RAM_HEADER, attributes), &attributes, sizeof(TPMA_NV));
648     return;
649 }
650
651 //*****
652 /*** Externally Accessible Functions
653 //*****
654
655 /*** NvIsPlatformPersistentHandle()
656 // This function indicates if a handle references a persistent object in the
657 // range belonging to the platform.
658 // Return Type: BOOL
659 //     TRUE(1)           handle references a platform persistent object
660 //     FALSE(0)          handle does not reference platform persistent object
661 BOOL NvIsPlatformPersistentHandle(TPM_HANDLE handle // IN: handle
662 )
663 {
664     return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
665 }
666
667 /*** NvIsOwnerPersistentHandle()
668 // This function indicates if a handle references a persistent object in the
669 // range belonging to the owner.
670 // Return Type: BOOL
671 //     TRUE(1)           handle is owner persistent handle
672 //     FALSE(0)          handle is not owner persistent handle and may not be
673 //                       a persistent handle at all
674 BOOL NvIsOwnerPersistentHandle(TPM_HANDLE handle // IN: handle
675 )
676 {
677     return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
678 }
679
680 /*** NvIndexIsAccessible()
681 //
682 // This function validates that a handle references a defined NV Index and
683 // that the Index is currently accessible.
684 // Return Type: TPM_RC
685 //     TPM_RC_HANDLE          the handle points to an undefined NV Index
686 //                             If shEnable is CLEAR, this would include an index
687 //                             created using ownerAuth. If phEnableNV is CLEAR,
688 //                             this would include an index created using
689 //                             platformAuth
690 //     TPM_RC_NV_READLOCKED   Index is present but locked for reading and command
691 //                             does not write to the index

```

```

692 //      TPM_RC_NV_WRITELOCKED      Index is present but locked for writing and command
693 //                                  writes to the index
694 TPM_RC
695 NvIndexIsAccessible(TPMI_RH_NV_INDEX handle // IN: handle
696 )
697 {
698     NV_INDEX* nvIndex = NvGetIndexInfo(handle, NULL);
699     //
700     if(nvIndex == NULL)
701         // If index is not found, return TPM_RC_HANDLE
702         return TPM_RC_HANDLE;
703     if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
704     {
705         // if shEnable is CLEAR, an ownerCreate NV Index should not be
706         // indicated as present
707         if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
708         {
709             if(gc.shEnable == FALSE)
710                 return TPM_RC_HANDLE;
711         }
712         // if phEnableNV is CLEAR, a platform created Index should not
713         // be visible
714         else if(gc.phEnableNV == FALSE)
715             return TPM_RC_HANDLE;
716     }
717     #if 0 // Writelock test for debug
718     // If the Index is write locked and this is an NV Write operation...
719     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITELOCKED)
720        && IsWriteOperation(commandIndex))
721     {
722         // then return a locked indication unless the command is TPM2_NV_WriteLock
723         if(GetCommandCode(commandIndex) != TPM_CC_NV_WriteLock)
724             return TPM_RC_NV_LOCKED;
725         return TPM_RC_SUCCESS;
726     }
727     #endif
728     #if 0 // Readlock Test for debug
729     // If the Index is read locked and this is an NV Read operation...
730     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, READLOCKED)
731        && IsReadOperation(commandIndex))
732     {
733         // then return a locked indication unless the command is TPM2_NV_ReadLock
734         if(GetCommandCode(commandIndex) != TPM_CC_NV_ReadLock)
735             return TPM_RC_NV_LOCKED;
736     }
737     #endif
738     // NV Index is accessible
739     return TPM_RC_SUCCESS;
740 }
741
742 /*** NvGetEvictObject()
743 // This function is used to dereference an evict object handle and get a pointer
744 // to the object.
745 // Return Type: TPM_RC
746 //      TPM_RC_HANDLE      the handle does not point to an existing
747 //                          persistent object
748 TPM_RC
749 NvGetEvictObject(TPM_HANDLE handle, // IN: handle
750                  OBJECT* object // OUT: object data
751 )
752 {
753     NV_REF entityAddr; // offset points to the entity
754                       //
755     // Find the address of evict object and copy to object
756     entityAddr = NvFindEvict(handle, object);
757

```

```

758     // whether there is an error or not, make sure that the evict
759     // status of the object is set so that the slot will get freed on exit
760     // Must do this after NvFindEvict loads the object
761     object->attributes.evict = SET;
762
763     // If handle is not found, return an error
764     if(entityAddr == 0)
765         return TPM_RC_HANDLE;
766     return TPM_RC_SUCCESS;
767 }
768
769 /*** NvIndexCacheInit()
770 // Function to initialize the Index cache
771 void NvIndexCacheInit(void)
772 {
773     s_cachedNvRef                = NV_REF_INIT;
774     s_cachedNvRamRef             = NV_RAM_REF_INIT;
775     s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
776     return;
777 }
778
779 /*** NvGetIndexData()
780 // This function is used to access the data in an NV Index. The data is returned
781 // as a byte sequence.
782 //
783 // This function requires that the NV Index be defined, and that the
784 // required data is within the data range. It also requires that TPMA_NV_WRITTEN
785 // of the Index is SET.
786 void NvGetIndexData(NV_INDEX* nvIndex, // IN: the in RAM index descriptor
787                    NV_REF locator, // IN: where the data is located
788                    UINT32 offset, // IN: offset of NV data
789                    UINT16 size, // IN: number of octets of NV data to read
790                    void* data // OUT: data buffer
791 )
792 {
793     TPMA_NV nvAttributes;
794     //
795     pAssert(nvIndex != NULL);
796
797     nvAttributes = nvIndex->publicArea.attributes;
798
799     pAssert(IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN));
800
801     if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, ORDERLY))
802     {
803         // Get data from RAM buffer
804         NV_RAM_REF ramAddr = NvRamGetIndex(nvIndex->publicArea.nvIndex);
805         pAssert(ramAddr != 0
806                && (size <= ((NV_RAM_HEADER*)ramAddr)->size - sizeof(NV_RAM_HEADER)
807                    - offset));
808         MemoryCopy(data, ramAddr + sizeof(NV_RAM_HEADER) + offset, size);
809     }
810     else
811     {
812         // Validate that read falls within range of the index
813         pAssert(offset <= nvIndex->publicArea.dataSize
814                && size <= (nvIndex->publicArea.dataSize - offset));
815         NvRead(data, locator + sizeof(NV_INDEX) + offset, size);
816     }
817     return;
818 }
819
820 /*** NvHashIndexData()
821 // This function adds Index data to a hash. It does this in parts to avoid large stack
822 // buffers.
823 void NvHashIndexData(HASH_STATE* hashState, // IN: Initialized hash state

```

```

824         NV_INDEX*   nvIndex,    // IN: Index
825         NV_REF      locator,    // IN: where the data is located
826         UINT32      offset,    // IN: starting offset
827         UINT16      size        // IN: amount to hash
828     )
829     {
830     #define BUFFER_SIZE 64
831         BYTE buffer[BUFFER_SIZE];
832         if(offset > nvIndex->publicArea.dataSize)
833             return;
834         // Make sure that we don't try to read off the end.
835         if((offset + size) > nvIndex->publicArea.dataSize)
836             size = nvIndex->publicArea.dataSize - (UINT16)offset;
837     #if BUFFER_SIZE >= MAX_NV_INDEX_SIZE
838         NvGetIndexData(nvIndex, locator, offset, size, buffer);
839         CryptDigestUpdate(hashState, size, buffer);
840     #else
841     {
842         INT16 i;
843         UINT16 readSize;
844         //
845         for(i = size; i > 0; offset += readSize, i -= readSize)
846         {
847             readSize = (i < BUFFER_SIZE) ? i : BUFFER_SIZE;
848             NvGetIndexData(nvIndex, locator, offset, readSize, buffer);
849             CryptDigestUpdate(hashState, readSize, buffer);
850         }
851     }
852     #endif // BUFFER_SIZE >= MAX_NV_INDEX_SIZE
853     #undef BUFFER_SIZE
854     }
855
856     /*** NvGetUINT64Data()
857     // Get data in integer format of a bit or counter NV Index.
858     //
859     // This function requires that the NV Index is defined and that the NV Index
860     // previously has been written.
861     UINT64
862     NvGetUINT64Data(NV_INDEX* nvIndex, // IN: the in RAM index descriptor
863                    NV_REF locator // IN: where index exists in NV
864     )
865     {
866         UINT64 intVal;
867         //
868         // Read the value and convert it to internal format
869         NvGetIndexData(nvIndex, locator, 0, 8, &intVal);
870         return BYTE_ARRAY_TO_UINT64((BYTE*)&intVal);
871     }
872
873     /*** NvWriteIndexAttributes()
874     // This function is used to write just the attributes of an index.
875     // Return type: TPM_RC
876     //     TPM_RC_NV_RATE      NV is rate limiting so retry
877     //     TPM_RC_NV_UNAVAILABLE NV is not available
878     TPM_RC
879     NvWriteIndexAttributes(TPM_HANDLE handle,
880                           NV_REF locator, // IN: location of the index
881                           TPMA_NV attributes // IN: attributes to write
882     )
883     {
884         TPM_RC result;
885         //
886         if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
887         {
888             NV_RAM_REF ram = NvRamGetIndex(handle);
889             NvWriteRamIndexAttributes(ram, attributes);

```



```

890     result = TPM_RC_SUCCESS;
891 }
892 else
893 {
894     result = NvWriteNvIndexAttributes(locator, attributes);
895 }
896 return result;
897 }
898
899 /*** NvWriteIndexAuth()
900 // This function is used to write the authValue of an index. It is used by
901 // TPM2_NV_ChangeAuth()
902 // Return type: TPM_RC
903 //     TPM_RC_NV_RATE          NV is rate limiting so retry
904 //     TPM_RC_NV_UNAVAILABLE   NV is not available
905 TPM_RC
906 NvWriteIndexAuth(NV_REF      locator,    // IN: location of the index
907                 TPM2B_AUTH* authValue    // IN: the authValue to write
908 )
909 {
910     TPM_RC result;
911     //
912     // If the locator is pointing to the cached index value...
913     if(locator == s_cachedNvRef)
914     {
915         // copy the authValue to the cached index so it will be there if we
916         // look for it. This is a safety thing.
917         MemoryCopy2B(&s_cachedNvIndex.authValue.b,
918                     &authValue->b,
919                     sizeof(s_cachedNvIndex.authValue.t.buffer));
920     }
921     result = NvConditionallyWrite(locator + offsetof(NV_INDEX, authValue),
922                                 sizeof(UINT16) + authValue->t.size,
923                                 authValue);
924     return result;
925 }
926
927 /*** NvGetIndexInfo()
928 // This function loads the nvIndex Info into the NV cache and returns a pointer
929 // to the NV_INDEX. If the returned value is zero, the index was not found.
930 // The 'locator' parameter, if not NULL, will be set to the offset in NV of the
931 // Index (the location of the handle of the Index).
932 //
933 // This function will set the index cache. If the index is orderly, the attributes
934 // from RAM are substituted for the attributes in the cached index
935 NV_INDEX* NvGetIndexInfo(TPM_HANDLE nvHandle, // IN: the index handle
936                         NV_REF* locator    // OUT: location of the index
937 )
938 {
939     if(s_cachedNvIndex.publicArea.nvIndex != nvHandle)
940     {
941         s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
942         s_cachedNvRamRef                    = 0;
943         s_cachedNvRef                      = NvFindHandle(nvHandle);
944         if(s_cachedNvRef == 0)
945             return NULL;
946         NvReadNvIndexInfo(s_cachedNvRef, &s_cachedNvIndex);
947         if(IS_ATTRIBUTE(s_cachedNvIndex.publicArea.attributes, TPMA_NV, ORDERLY))
948         {
949             s_cachedNvRamRef = NvRamGetIndex(nvHandle);
950             s_cachedNvIndex.publicArea.attributes =
951                 NvReadRamIndexAttributes(s_cachedNvRamRef);
952         }
953     }
954     if(locator != NULL)
955         *locator = s_cachedNvRef;

```

```

956     return &s_cachedNvIndex;
957 }
958
959 /*** NvWriteIndexData()
960 // This function is used to write NV index data. It is intended to be used to
961 // update the data associated with the default index.
962 //
963 // This function requires that the NV Index is defined, and the data is
964 // within the defined data range for the index.
965 //
966 // Index data is only written due to a command that modifies the data in a single
967 // index. There is no case where changes are made to multiple indexes data at the
968 // same time. Multiple attributes may be change but not multiple index data. This
969 // is important because we will normally be handling the index for which we have
970 // the cached pointer values.
971 // Return type: TPM_RC
972 //     TPM_RC_NV_RATE      NV is rate limiting so retry
973 //     TPM_RC_NV_UNAVAILABLE NV is not available
974 TPM_RC
975 NvWriteIndexData(NV_INDEX* nvIndex, // IN: the description of the index
976                 UINT32   offset,    // IN: offset of NV data
977                 UINT32   size,      // IN: size of NV data
978                 void*    data       // IN: data buffer
979 )
980 {
981     TPM_RC result = TPM_RC_SUCCESS;
982     //
983     pAssert(nvIndex != NULL);
984     // Make sure that this is dealing with the 'default' index.
985     // Note: it is tempting to change the calling sequence so that the 'default' is
986     // presumed.
987     pAssert(nvIndex->publicArea.nvIndex == s_cachedNvIndex.publicArea.nvIndex);
988
989     // Validate that write falls within range of the index
990     pAssert(offset <= nvIndex->publicArea.dataSize
991             && size <= (nvIndex->publicArea.dataSize - offset));
992
993     // Update TPMA_NV_WRITTEN bit if necessary
994     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
995     {
996         // Update the in memory version of the attributes
997         SET_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN);
998
999         // If this is not orderly, then update the NV version of
1000         // the attributes
1001         if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
1002         {
1003             result = NvWriteNvIndexAttributes(s_cachedNvRef,
1004                                               nvIndex->publicArea.attributes);
1005             if(result != TPM_RC_SUCCESS)
1006                 return result;
1007             // If this is a partial write of an ordinary index, clear the whole
1008             // index.
1009             if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes)
1010                && (nvIndex->publicArea.dataSize > size))
1011                 _plat_NvMemoryClear(s_cachedNvRef + sizeof(NV_INDEX),
1012                                     nvIndex->publicArea.dataSize);
1013         }
1014         else
1015         {
1016             // This is orderly so update the RAM version
1017             MemoryCopy(s_cachedNvRamRef + offsetof(NV_RAM_HEADER, attributes),
1018                       &nvIndex->publicArea.attributes,
1019                       sizeof(TPMA_NV));
1020             // If setting WRITTEN for an orderly counter, make sure that the
1021             // state saved version of the counter is saved

```

```

1022         if(IsNvCounterIndex(nvIndex->publicArea.attributes))
1023             SET_NV_UPDATE(UT_ORDERLY);
1024         // If setting the written attribute on an ordinary index, make sure that
1025         // the data is all cleared out in case there is a partial write. This
1026         // is only necessary for ordinary indexes because all of the other types
1027         // are always written in total.
1028         else if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes))
1029             MemorySet(s_cachedNvRamRef + sizeof(NV_RAM_HEADER),
1030                     0,
1031                     nvIndex->publicArea.dataSize);
1032     }
1033 }
1034 // If this is orderly data, write it to RAM
1035 if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
1036 {
1037     // Note: if this is the first write to a counter, the code above will queue
1038     // the write to NV of the RAM data in order to update TPMA_NV_WRITTEN. In
1039     // process of doing that write, it will also write the initial counter value
1040
1041     // Update RAM
1042     MemoryCopy(s_cachedNvRamRef + sizeof(NV_RAM_HEADER) + offset, data, size);
1043
1044     // And indicate that the TPM is no longer orderly
1045     g_clearOrderly = TRUE;
1046 }
1047 else
1048 {
1049     // Offset into the index to the first byte of the data to be written to NV
1050     result = NvConditionallyWrite(
1051         s_cachedNvRef + sizeof(NV_INDEX) + offset, size, data);
1052 }
1053 return result;
1054 }
1055
1056 /*** NvWriteUINT64Data()
1057 // This function to write back a UINT64 value. The various UINT64 values (bits,
1058 // counters, and PINs) are kept in canonical format but manipulate in native
1059 // format. This takes a native format value converts it and saves it back as
1060 // in canonical format.
1061 //
1062 // This function will return the value from NV or RAM depending on the type of the
1063 // index (orderly or not)
1064 //
1065 TPM_RC
1066 NvWriteUINT64Data(NV_INDEX* nvIndex, // IN: the description of the index
1067                  UINT64    intValue // IN: the value to write
1068 )
1069 {
1070     BYTE bytes[8];
1071     UINT64_TO_BYTE_ARRAY(intValue, bytes);
1072     //
1073     return NvWriteIndexData(nvIndex, 0, 8, &bytes);
1074 }
1075
1076 /*** NvGetNameByIndexHandle()
1077 // This function is used to compute the Name of an NV Index referenced by handle.
1078 //
1079 // The 'name' buffer receives the bytes of the Name and the return value
1080 // is the number of octets in the Name.
1081 //
1082 // This function requires that the NV Index is defined.
1083 TPM2B_NAME* NvGetNameByIndexHandle(
1084     TPMI_RH_NV_INDEX handle, // IN: handle of the index
1085     TPM2B_NAME*      name    // OUT: name of the index
1086 )
1087 {

```

```

1088     NV_INDEX* nvIndex = NvGetIndexInfo(handle, NULL);
1089     //
1090     return NvGetIndexName(nvIndex, name);
1091 }
1092
1093 /*** NvDefineIndex()
1094 // This function is used to assign NV memory to an NV Index.
1095 //
1096 // Return Type: TPM_RC
1097 //     TPM_RC_NV_SPACE      insufficient NV space
1098 TPM_RC
1099 NvDefineIndex(TPMS_NV_PUBLIC* publicArea, // IN: A template for an area to create.
1100              TPM2B_AUTH* authValue // IN: The initial authorization value
1101 )
1102 {
1103     // The buffer to be written to NV memory
1104     NV_INDEX nvIndex; // the index data
1105     UINT16 entrySize; // size of entry
1106     TPM_RC result;
1107     //
1108     entrySize = sizeof(NV_INDEX);
1109
1110     // only allocate data space for indexes that are going to be written to NV.
1111     // Orderly indexes don't need space.
1112     if(!IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
1113         entrySize += publicArea->dataSize;
1114     // Check if we have enough space to create the NV Index
1115     // In this implementation, the only resource limitation is the available NV
1116     // space (and possibly RAM space.) Other implementation may have other
1117     // limitation on counter or on NV slots
1118     if(!NvTestSpace(entrySize, TRUE, IsNvCounterIndex(publicArea->attributes)))
1119         return TPM_RC_NV_SPACE;
1120
1121     // if the index to be defined is RAM backed, check RAM space availability
1122     // as well
1123     if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY)
1124        && !NvRamTestSpaceIndex(publicArea->dataSize))
1125         return TPM_RC_NV_SPACE;
1126     // Copy input value to nvBuffer
1127     nvIndex.publicArea = *publicArea;
1128
1129     // Copy the authValue
1130     nvIndex.authValue = *authValue;
1131
1132     // Add index to NV memory
1133     result = NvAdd(entrySize, sizeof(NV_INDEX), TPM_RH_UNASSIGNED, (BYTE*)&nvIndex);
1134     if(result == TPM_RC_SUCCESS)
1135     {
1136         // If the data of NV Index is RAM backed, add the data area in RAM as well
1137         if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
1138             NvAddRAM(publicArea);
1139     }
1140     return result;
1141 }
1142
1143 /*** NvAddEvictObject()
1144 // This function is used to assign NV memory to a persistent object.
1145 // Return Type: TPM_RC
1146 //     TPM_RC_NV_HANDLE      the requested handle is already in use
1147 //     TPM_RC_NV_SPACE      insufficient NV space
1148 TPM_RC
1149 NvAddEvictObject(TPMI_DH_OBJECT evictHandle, // IN: new evict handle
1150                 OBJECT* object // IN: object to be added
1151 )
1152 {
1153     TPM_HANDLE temp = object->evictHandle;

```

```

1154     TPM_RC     result;
1155     //
1156     // Check if we have enough space to add the evict object
1157     // An evict object needs 8 bytes in index table + sizeof OBJECT
1158     // In this implementation, the only resource limitation is the available NV
1159     // space. Other implementation may have other limitation on evict object
1160     // handle space
1161     if(!NvTestSpace(sizeof(OBJECT) + sizeof(TPM_HANDLE), FALSE, FALSE))
1162         return TPM_RC_NV_SPACE;
1163
1164     // Set evict attribute and handle
1165     object->attributes.evict = SET;
1166     object->evictHandle      = evictHandle;
1167
1168     // Now put this in NV
1169     result = NvAdd(sizeof(OBJECT), sizeof(OBJECT), evictHandle, (BYTE*)object);
1170
1171     // Put things back the way they were
1172     object->attributes.evict = CLEAR;
1173     object->evictHandle      = temp;
1174
1175     return result;
1176 }
1177
1178 /*** NvDeleteIndex()
1179 // This function is used to delete an NV Index.
1180 // Return Type: TPM_RC
1181 //     TPM_RC_NV_UNAVAILABLE   NV is not accessible
1182 //     TPM_RC_NV_RATE          NV is rate limiting
1183 TPM_RC
1184 NvDeleteIndex(NV_INDEX* nvIndex,    // IN: an in RAM index descriptor
1185               NV_REF   entityAddr  // IN: location in NV
1186 )
1187 {
1188     TPM_RC result;
1189     //
1190     if(nvIndex != NULL)
1191     {
1192         // Whenever a counter is deleted, make sure that the MaxCounter value is
1193         // updated to reflect the value
1194         if(IsNvCounterIndex(nvIndex->publicArea.attributes)
1195            && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
1196             NvUpdateMaxCount(NvGetUINT64Data(nvIndex, entityAddr));
1197         result = NvDelete(entityAddr);
1198         if(result != TPM_RC_SUCCESS)
1199             return result;
1200         // If the NV Index is RAM backed, delete the RAM data as well
1201         if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
1202             NvDeleteRAM(nvIndex->publicArea.nvIndex);
1203         NvIndexCacheInit();
1204     }
1205     return TPM_RC_SUCCESS;
1206 }
1207
1208 /*** NvDeleteEvict()
1209 // This function will delete a NV evict object.
1210 // Will return success if object deleted or if it does not exist
1211
1212 TPM_RC
1213 NvDeleteEvict(TPM_HANDLE handle  // IN: handle of entity to be deleted
1214 )
1215 {
1216     NV_REF entityAddr = NvFindEvict(handle, NULL); // pointer to entity
1217     TPM_RC result     = TPM_RC_SUCCESS;
1218     //
1219     if(entityAddr != 0)

```

```

1220         result = NvDelete(entityAddr);
1221     return result;
1222 }
1223
1224 /*** NvFlushHierarchy()
1225 // This function will delete persistent objects belonging to the indicated hierarchy.
1226 // If the storage hierarchy is selected, the function will also delete any
1227 // NV Index defined using ownerAuth.
1228 // Return Type: TPM_RC
1229 //     TPM_RC_NV_RATE           NV is unavailable because of rate limit
1230 //     TPM_RC_NV_UNAVAILABLE    NV is inaccessible
1231 TPM_RC
1232 NvFlushHierarchy(TPMI_RH_HIERARCHY hierarchy // IN: hierarchy to be flushed.
1233 )
1234 {
1235     NV_REF      iter = NV_REF_INIT;
1236     NV_REF      currentAddr;
1237     TPM_HANDLE  entityHandle;
1238     TPM_RC      result = TPM_RC_SUCCESS;
1239     //
1240     while((currentAddr = NvNext(&iter, &entityHandle)) != 0)
1241     {
1242         if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
1243         {
1244             NV_INDEX nvIndex;
1245             //
1246             // If flush endorsement or platform hierarchy, no NV Index would be
1247             // flushed
1248             if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
1249                 continue;
1250             // Get the index information
1251             NvReadNvIndexInfo(currentAddr, &nvIndex);
1252
1253             // For storage hierarchy, flush OwnerCreated index
1254             if(!IS_ATTRIBUTE(nvIndex.publicArea.attributes, TPMA_NV, PLATFORMCREATE))
1255             {
1256                 // Delete the index (including RAM for orderly)
1257                 result = NvDeleteIndex(&nvIndex, currentAddr);
1258                 if(result != TPM_RC_SUCCESS)
1259                     break;
1260                 // Re-iterate from beginning after a delete
1261                 iter = NV_REF_INIT;
1262             }
1263         }
1264         else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
1265         {
1266             OBJECT_ATTRIBUTES attributes;
1267             //
1268             NvRead(&attributes,
1269                 (UINT32)(currentAddr + sizeof(TPM_HANDLE)
1270                     + offsetof(OBJECT, attributes)),
1271                 sizeof(OBJECT_ATTRIBUTES));
1272             // If the evict object belongs to the hierarchy to be flushed...
1273             if((hierarchy == TPM_RH_PLATFORM && attributes.ppsHierarchy == SET)
1274                 || (hierarchy == TPM_RH_OWNER && attributes.spsHierarchy == SET)
1275                 || (hierarchy == TPM_RH_ENDORSEMENT && attributes.epsHierarchy == SET))
1276             {
1277                 // ...then delete the evict object
1278                 result = NvDelete(currentAddr);
1279                 if(result != TPM_RC_SUCCESS)
1280                     break;
1281                 // Re-iterate from beginning after a delete
1282                 iter = NV_REF_INIT;
1283             }
1284         }
1285     }

```



```

1286     {
1287         FAIL(FATAL_ERROR_INTERNAL);
1288     }
1289 }
1290 return result;
1291 }
1292
1293 /*** NvSetGlobalLock()
1294 // This function is used to SET the TPMA_NV_WRITELOCKED attribute for all
1295 // NV indexes that have TPMA_NV_GLOBALLOCK SET. This function is use by
1296 // TPM2_NV_GlobalWriteLock().
1297 // Return Type: TPM_RC
1298 //     TPM_RC_NV_RATE           NV is unavailable because of rate limit
1299 //     TPM_RC_NV_UNAVAILABLE    NV is inaccessible
1300 TPM_RC
1301 NvSetGlobalLock(void)
1302 {
1303     NV_REF      iter      = NV_REF_INIT;
1304     NV_RAM_REF  ramIter = NV_RAM_REF_INIT;
1305     NV_REF      currentAddr;
1306     NV_RAM_REF  currentRamAddr;
1307     TPM_RC      result = TPM_RC_SUCCESS;
1308     //
1309     // Check all normal indexes
1310     while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1311     {
1312         TPMA_NV attributes = NvReadNvIndexAttributes(currentAddr);
1313         //
1314         // See if it should be locked
1315         if(!IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1316             && IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1317         {
1318             SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1319             result = NvWriteNvIndexAttributes(currentAddr, attributes);
1320             if(result != TPM_RC_SUCCESS)
1321                 return result;
1322         }
1323     }
1324     // Now search all the orderly attributes
1325     while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1326     {
1327         // See if it should be locked
1328         TPMA_NV attributes = NvReadRamIndexAttributes(currentRamAddr);
1329         if(IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1330         {
1331             SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1332             NvWriteRamIndexAttributes(currentRamAddr, attributes);
1333         }
1334     }
1335     return result;
1336 }
1337
1338 /***InsertSort()
1339 // Sort a handle into handle list in ascending order. The total handle number in
1340 // the list should not exceed MAX_CAP_HANDLES
1341 static void InsertSort(TPML_HANDLE* handleList, // IN/OUT: sorted handle list
1342                      UINT32 count, // IN: maximum count in the handle list
1343                      TPM_HANDLE entityHandle // IN: handle to be inserted
1344 )
1345 {
1346     UINT32 i, j;
1347     UINT32 originalCount;
1348     //
1349     // For a corner case that the maximum count is 0, do nothing
1350     if(count == 0)
1351         return;

```

```

1352 // For empty list, add the handle at the beginning and return
1353 if(handleList->count == 0)
1354 {
1355     handleList->handle[0] = entityHandle;
1356     handleList->count++;
1357     return;
1358 }
1359 // Check if the maximum of the list has been reached
1360 originalCount = handleList->count;
1361 if(originalCount < count)
1362     handleList->count++;
1363 // Insert the handle to the list
1364 for(i = 0; i < originalCount; i++)
1365 {
1366     if(handleList->handle[i] > entityHandle)
1367     {
1368         for(j = handleList->count - 1; j > i; j--)
1369         {
1370             handleList->handle[j] = handleList->handle[j - 1];
1371         }
1372         break;
1373     }
1374 }
1375 // If a slot was found, insert the handle in this position
1376 if(i < originalCount || handleList->count > originalCount)
1377     handleList->handle[i] = entityHandle;
1378 return;
1379 }
1380
1381 /*** NvCapGetPersistent()
1382 // This function is used to get a list of handles of the persistent objects,
1383 // starting at 'handle'.
1384 //
1385 // 'Handle' must be in valid persistent object handle range, but does not
1386 // have to reference an existing persistent object.
1387 // Return Type: TPMI_YES_NO
1388 //     YES      if there are more handles available
1389 //     NO       all the available handles has been returned
1390 TPMI_YES_NO
1391 NvCapGetPersistent(TPMI_DH_OBJECT handle, // IN: start handle
1392                   UINT32 count, // IN: maximum number of returned handles
1393                   TPML_HANDLE* handleList // OUT: list of handle
1394 )
1395 {
1396     TPMI_YES_NO more = NO;
1397     NV_REF iter = NV_REF_INIT;
1398     NV_REF currentAddr;
1399     TPM_HANDLE entityHandle;
1400     //
1401     pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1402
1403     // Initialize output handle list
1404     handleList->count = 0;
1405
1406     // The maximum count of handles we may return is MAX_CAP_HANDLES
1407     if(count > MAX_CAP_HANDLES)
1408         count = MAX_CAP_HANDLES;
1409
1410     while((currentAddr = NvNextEvict(&entityHandle, &iter)) != 0)
1411     {
1412         // Ignore persistent handles that have values less than the input handle
1413         if(entityHandle < handle)
1414             continue;
1415         // if the handles in the list have reached the requested count, and there
1416         // are still handles need to be inserted, indicate that there are more.
1417         if(handleList->count == count)

```

```

1418         more = YES;
1419         // A handle with a value larger than start handle is a candidate
1420         // for return. Insert sort it to the return list. Insert sort algorithm
1421         // is chosen here for simplicity based on the assumption that the total
1422         // number of NV indexes is small. For an implementation that may allow
1423         // large number of NV indexes, a more efficient sorting algorithm may be
1424         // used here.
1425         InsertSort(handleList, count, entityHandle);
1426     }
1427     return more;
1428 }
1429
1430 /*** NvCapGetOnePersistent()
1431 // This function returns whether a given persistent handle exists.
1432 //
1433 // 'Handle' must be in valid persistent object handle range.
1434 BOOL NvCapGetOnePersistent(TPMI_DH_OBJECT handle) // IN: handle
1435 {
1436     NV_REF      iter = NV_REF_INIT;
1437     NV_REF      currentAddr;
1438     TPM_HANDLE  entityHandle;
1439
1440     pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1441
1442     while((currentAddr = NvNextEvict(&entityHandle, &iter)) != 0)
1443     {
1444         if(entityHandle == handle)
1445         {
1446             return TRUE;
1447         }
1448     }
1449     return FALSE;
1450 }
1451
1452 /*** NvCapGetIndex()
1453 // This function returns a list of handles of NV indexes, starting from 'handle'.
1454 // 'Handle' must be in the range of NV indexes, but does not have to reference
1455 // an existing NV Index.
1456 // Return Type: TPMI_YES_NO
1457 // YES         if there are more handles to report
1458 // NO          all the available handles has been reported
1459 TPMI_YES_NO
1460 NvCapGetIndex(TPMI_DH_OBJECT handle, // IN: start handle
1461               UINT32      count, // IN: max number of returned handles
1462               TPML_HANDLE* handleList // OUT: list of handle
1463 )
1464 {
1465     TPMI_YES_NO more = NO;
1466     NV_REF      iter = NV_REF_INIT;
1467     NV_REF      currentAddr;
1468     TPM_HANDLE  nvHandle;
1469     //
1470     pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1471
1472     // Initialize output handle list
1473     handleList->count = 0;
1474
1475     // The maximum count of handles we may return is MAX_CAP_HANDLES
1476     if(count > MAX_CAP_HANDLES)
1477         count = MAX_CAP_HANDLES;
1478
1479     while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1480     {
1481         // Ignore index handles that have values less than the 'handle'
1482         if(nvHandle < handle)
1483             continue;

```

```

1484     // if the count of handles in the list has reached the requested count,
1485     // and there are still handles to report, set more.
1486     if(handleList->count == count)
1487         more = YES;
1488     // A handle with a value larger than start handle is a candidate
1489     // for return. Insert sort it to the return list. Insert sort algorithm
1490     // is chosen here for simplicity based on the assumption that the total
1491     // number of NV indexes is small. For an implementation that may allow
1492     // large number of NV indexes, a more efficient sorting algorithm may be
1493     // used here.
1494     InsertSort(handleList, count, nvHandle);
1495 }
1496 return more;
1497 }
1498
1499 /*** NvCapGetOneIndex()
1500 // This function whether an NV index exists.
1501 BOOL NvCapGetOneIndex(TPMI_DH_OBJECT handle) // IN: handle
1502 {
1503     NV_REF iter = NV_REF_INIT;
1504     NV_REF currentAddr;
1505     TPM_HANDLE nvHandle;
1506
1507     pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1508
1509     while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1510     {
1511         if(nvHandle == handle)
1512         {
1513             return TRUE;
1514         }
1515     }
1516     return FALSE;
1517 }
1518
1519 /*** NvCapGetIndexNumber()
1520 // This function returns the count of NV Indexes currently defined.
1521 UINT32
1522 NvCapGetIndexNumber(void)
1523 {
1524     UINT32 num = 0;
1525     NV_REF iter = NV_REF_INIT;
1526     //
1527     while(NvNextIndex(NULL, &iter) != 0)
1528         num++;
1529     return num;
1530 }
1531
1532 /*** NvCapGetPersistentNumber()
1533 // Function returns the count of persistent objects currently in NV memory.
1534 UINT32
1535 NvCapGetPersistentNumber(void)
1536 {
1537     UINT32 num = 0;
1538     NV_REF iter = NV_REF_INIT;
1539     TPM_HANDLE handle;
1540     //
1541     while(NvNextEvict(&handle, &iter) != 0)
1542         num++;
1543     return num;
1544 }
1545
1546 /*** NvCapGetPersistentAvail()
1547 // This function returns an estimate of the number of additional persistent
1548 // objects that could be loaded into NV memory.
1549 UINT32

```

```

1550 NvCapGetPersistentAvail(void)
1551 {
1552     UINT32 availNVSpace;
1553     UINT32 counterNum = NvCapGetCounterNumber();
1554     UINT32 reserved = sizeof(NV_LIST_TERMINATOR);
1555     //
1556     // Get the available space in NV storage
1557     availNVSpace = NvGetFreeBytes();
1558
1559     if(counterNum < MIN_COUNTER_INDICES)
1560     {
1561         // Some space has to be reserved for counter objects.
1562         reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
1563         if(reserved > availNVSpace)
1564             availNVSpace = 0;
1565         else
1566             availNVSpace -= reserved;
1567     }
1568     return availNVSpace / NV_EVICT_OBJECT_SIZE;
1569 }
1570
1571 /*** NvCapGetCounterNumber()
1572 // Get the number of defined NV Indexes that are counter indexes.
1573 UINT32
1574 NvCapGetCounterNumber(void)
1575 {
1576     NV_REF iter = NV_REF_INIT;
1577     NV_REF currentAddr;
1578     UINT32 num = 0;
1579     //
1580     while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1581     {
1582         TPMA_NV attributes = NvReadNvIndexAttributes(currentAddr);
1583         if(IsNvCounterIndex(attributes))
1584             num++;
1585     }
1586     return num;
1587 }
1588
1589 /*** NvSetStartupAttributes()
1590 // Local function to set the attributes of an Index at TPM Reset and TPM Restart.
1591 static TPMA_NV NvSetStartupAttributes(TPMA_NV attributes, // IN: attributes to change
1592                                     STARTUP_TYPE type // IN: start up type
1593 )
1594 {
1595     // Clear read lock
1596     CLEAR_ATTRIBUTE(attributes, TPMA_NV, READLOCKED);
1597
1598     // Will change a non counter index to the unwritten state if:
1599     // a) TPMA_NV_CLEAR_STCLEAR is SET
1600     // b) orderly and TPM Reset
1601     if(!IsNvCounterIndex(attributes))
1602     {
1603         if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
1604            || (IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY) && (type == SU_RESET)))
1605             CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITTEN);
1606     }
1607     // Unlock any index that is not written or that does not have
1608     // TPMA_NV_WRITEDEFINE SET.
1609     if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
1610        || !IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
1611         CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1612     return attributes;
1613 }
1614
1615 /*** NvEntityStartup()

```

```

1616 // This function is called at TPM Startup(). If the startup completes
1617 // a TPM Resume cycle, no action is taken. If the startup is a TPM Reset
1618 // or a TPM Restart, then this function will:
1619 // a) clear read/write lock;
1620 // b) reset NV Index data that has TPMA_NV_CLEAR_STCLEAR SET; and
1621 // c) set the lower bits in orderly counters to 1 for a non-orderly startup
1622 //
1623 // It is a prerequisite that NV be available for writing before this
1624 // function is called.
1625 BOOL NvEntityStartup(STARTUP_TYPE type // IN: start up type
1626 )
1627 {
1628     NV_REF iter = NV_REF_INIT;
1629     NV_RAM_REF ramIter = NV_RAM_REF_INIT;
1630     NV_REF currentAddr; // offset points to the current entity
1631     NV_RAM_REF currentRamAddr;
1632     TPM_HANDLE nvHandle;
1633     TPMA_NV attributes;
1634     //
1635     // Restore RAM index data
1636     NvRead(s_indexOrderlyRam, NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam));
1637
1638     // Initialize the max NV counter value
1639     NvSetMaxCount(NvGetMaxCount());
1640
1641     // If recovering from state save, do nothing else
1642     if(type == SU_RESUME)
1643         return TRUE;
1644     // Iterate all the NV Index to clear the locks
1645     while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1646     {
1647         attributes = NvReadNvIndexAttributes(currentAddr);
1648
1649         // If this is an orderly index, defer processing until loop below
1650         if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
1651             continue;
1652         // Set the attributes appropriate for this startup type
1653         attributes = NvSetStartupAttributes(attributes, type);
1654         NvWriteNvIndexAttributes(currentAddr, attributes);
1655     }
1656     // Iterate all the orderly indexes to clear the locks and initialize counters
1657     while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1658     {
1659         attributes = NvReadRamIndexAttributes(currentRamAddr);
1660
1661         attributes = NvSetStartupAttributes(attributes, type);
1662
1663         // update attributes in RAM
1664         NvWriteRamIndexAttributes(currentRamAddr, attributes);
1665
1666         // Set the lower bits in an orderly counter to 1 for a non-orderly startup
1667         if(IsNvCounterIndex(attributes) && (g_prevOrderlyState == SU_NONE_VALUE))
1668         {
1669             UINT64 counter;
1670             //
1671             // Read the counter value last saved to NV.
1672             counter = BYTE_ARRAY_TO_UINT64(currentRamAddr + sizeof(NV_RAM_HEADER));
1673
1674             // Set the lower bits of counter to 1's
1675             counter |= MAX_ORDERLY_COUNT;
1676
1677             // Write back to RAM
1678             // NOTE: Do not want to force a write to NV here. The counter value will
1679             // stay in RAM until the next shutdown or rollover.
1680             UINT64_TO_BYTE_ARRAY(counter, currentRamAddr + sizeof(NV_RAM_HEADER));
1681         }
1682     }

```



```

1682     }
1683     return TRUE;
1684 }
1685
1686 /*** NvCapGetCounterAvail()
1687 // This function returns an estimate of the number of additional counter type
1688 // NV indexes that can be defined.
1689 UINT32
1690 NvCapGetCounterAvail(void)
1691 {
1692     UINT32 availNVSpace;
1693     UINT32 availRAMSpace;
1694     UINT32 persistentNum = NvCapGetPersistentNumber();
1695     UINT32 reserved      = sizeof(NV_LIST_TERMINATOR);
1696     //
1697     // Get the available space in NV storage
1698     availNVSpace = NvGetFreeBytes();
1699
1700     if(persistentNum < MIN_EVICT_OBJECTS)
1701     {
1702         // Some space has to be reserved for evict object. Adjust availNVSpace.
1703         reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
1704         if(reserved > availNVSpace)
1705             availNVSpace = 0;
1706         else
1707             availNVSpace -= reserved;
1708     }
1709     // Compute the available space in RAM
1710     availRAMSpace = (int)(RAM_ORDERLY_END - NvRamGetEnd());
1711
1712     // Return the min of counter number in NV and in RAM
1713     if(availNVSpace / NV_INDEX_COUNTER_SIZE
1714        > availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE)
1715         return availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE;
1716     else
1717         return availNVSpace / NV_INDEX_COUNTER_SIZE;
1718 }
1719
1720 /*** NvFindHandle()
1721 // this function returns the offset in NV memory of the entity associated
1722 // with the input handle. A value of zero indicates that handle does not
1723 // exist reference an existing persistent object or defined NV Index.
1724 NV_REF
1725 NvFindHandle(TPM_HANDLE handle)
1726 {
1727     NV_REF    addr;
1728     NV_REF    iter = NV_REF_INIT;
1729     TPM_HANDLE nextHandle;
1730     //
1731     while((addr = NvNext(&iter, &nextHandle)) != 0)
1732     {
1733         if(nextHandle == handle)
1734             break;
1735     }
1736     return addr;
1737 }
1738
1739 /*** NV Max Counter
1740 /*** Introduction
1741 // The TPM keeps track of the highest value of a deleted counter index. When an
1742 // index is deleted, this value is updated if the deleted counter index is greater
1743 // than the previous value. When a new index is created and first incremented, it
1744 // will get a value that is at least one greater than any other index than any
1745 // previously deleted index. This insures that it is not possible to roll back an
1746 // index.
1747 //

```

```

1748 // The highest counter value is kept in NV in a special end-of-list marker. This
1749 // marker is only updated when an index is deleted. Otherwise it just moves.
1750 //
1751 // When the TPM starts up, it searches NV for the end of list marker and initializes
1752 // an in memory value (s_maxCounter).
1753
1754 /*** NvReadMaxCount()
1755 // This function returns the max NV counter value.
1756 //
1757 UINT64
1758 NvReadMaxCount(void)
1759 {
1760     return s_maxCounter;
1761 }
1762
1763 /*** NvUpdateMaxCount()
1764 // This function updates the max counter value to NV memory. This is just staging
1765 // for the actual write that will occur when the NV index memory is modified.
1766 //
1767 void NvUpdateMaxCount(UINT64 count)
1768 {
1769     if(count > s_maxCounter)
1770         s_maxCounter = count;
1771 }
1772
1773 /*** NvSetMaxCount()
1774 // This function is used at NV initialization time to set the initial value of
1775 // the maximum counter.
1776 void NvSetMaxCount(UINT64 value)
1777 {
1778     s_maxCounter = value;
1779 }
1780
1781 /*** NvGetMaxCount()
1782 // Function to get the NV max counter value from the end-of-list marker
1783 UINT64
1784 NvGetMaxCount(void)
1785 {
1786     NV_REF iter = NV_REF_INIT;
1787     NV_REF currentAddr;
1788     UINT64 maxCount;
1789     //
1790     // Find the end of list marker and initialize the NV Max Counter value.
1791     while((currentAddr = NvNext(&iter, NULL)) != 0)
1792         ;
1793     // 'iter' should be pointing at the end of list marker so read in the current
1794     // value of the s_maxCounter.
1795     NvRead(&maxCount, iter + sizeof(UINT32), sizeof(maxCount));
1796
1797     return maxCount;
1798 }

```

7.174 /tpm/src/subsystem/NvReserved.c

```

1  /*** Introduction
2
3  // The NV memory is divided into two areas: dynamic space for user defined NV
4  // Indices and evict objects, and reserved space for TPM persistent and state save
5  // data.
6  //
7  // The entries in dynamic space are a linked list of entries. Each entry has, as its
8  // first field, a size. If the size field is zero, it marks the end of the
9  // list.
10 //
11 // An allocation of an Index or evict object may use almost all of the remaining

```

```

12 // NV space such that the size field will not fit. The functions that search the
13 // list are aware of this and will terminate the search if they either find a zero
14 // size or recognize that there is insufficient space for the size field.
15 //
16 // An Index allocation will contain an NV_INDEX structure. If the Index does not
17 // have the orderly attribute, the NV_INDEX is followed immediately by the NV data.
18 //
19 // An evict object entry contains a handle followed by an OBJECT structure. This
20 // results in both the Index and Evict Object having an identifying handle as the
21 // first field following the size field.
22 //
23 // When an Index has the orderly attribute, the data is kept in RAM. This RAM is
24 // saved to backing store in NV memory on any orderly shutdown. The entries in
25 // orderly memory are also a linked list using a size field as the first entry. As
26 // with the NV memory, the list is terminated by a zero size field or when the last
27 // entry leaves insufficient space for the terminating size field.
28 //
29 // The attributes of an orderly index are maintained in RAM memory in order to
30 // reduce the number of NV writes needed for orderly data. When an orderly index
31 // is created, an entry is made in the dynamic NV memory space that holds the Index
32 // authorizations (authPolicy and authValue) and the size of the data. This entry is
33 // only modified if the authValue of the index is changed. The more volatile data
34 // of the index is kept in RAM. When an orderly Index is created or deleted, the
35 // RAM data is copied to NV backing store so that the image in the backing store
36 // matches the layout of RAM. In normal operation. The RAM data is also copied on
37 // any orderly shutdown. In normal operation, the only other reason for writing
38 // to the backing store for RAM is when a counter is first written (TPMA_NV_WRITTEN
39 // changes from CLEAR to SET) or when a counter "rolls over."
40 //
41 // Static space contains items that are individually modifiable. The values are in
42 // the 'gp' PERSISTENT_DATA structure in RAM and mapped to locations in NV.
43 //
44
45 /** Includes, Defines
46 #define NV_C
47 #include "Tpm.h"
48
49 /** Functions
50
51
52
53 /** NvInitStatic()
54 // This function initializes the static variables used in the NV subsystem.
55 static void NvInitStatic(void)
56 {
57     // In some implementations, the end of NV is variable and is set at boot time.
58     // This value will be the same for each boot, but is not necessarily known
59     // at compile time.
60     s_evictNvEnd = (NV_REF)NV_MEMORY_SIZE;
61     return;
62 }
63
64 /** NvCheckState()
65 // Function to check the NV state by accessing the platform-specific function
66 // to get the NV state. The result state is registered in s_NvIsAvailable
67 // that will be reported by NvIsAvailable.
68 //
69 // This function is called at the beginning of ExecuteCommand before any potential
70 // check of g_NvStatus.
71 void NvCheckState(void)
72 {
73     int func_return;
74     //
75     func_return = _plat_GetNvReadyState();
76     if(func_return == NV_READY)
77     {

```

```

78     g_NvStatus = TPM_RC_SUCCESS;
79 }
80 else if(func_return == NV_WRITEFAILURE)
81 {
82     g_NvStatus = TPM_RC_NV_UNAVAILABLE;
83 }
84 else
85 {
86     // if(func_return == NV_RATE_LIMIT) or anything else
87     // assume retry later might work
88     g_NvStatus = TPM_RC_NV_RATE;
89 }
90
91 return;
92 }
93
94 /*** NvCommit
95 // This is a wrapper for the platform function to commit pending NV writes.
96 BOOL NvCommit(void)
97 {
98     return (_plat__NvCommit() == 0);
99 }
100
101 /*** NvPowerOn()
102 // This function is called at _TPM_Init to initialize the NV environment.
103 // Return Type: BOOL
104 //     TRUE(1)         all NV was initialized
105 //     FALSE(0)        the NV containing saved state had an error and
106 //                     TPM2_Startup(CLEAR) is required
107 BOOL NvPowerOn(void)
108 {
109     int nvError = 0;
110     // If power was lost, need to re-establish the RAM data that is loaded from
111     // NV and initialize the static variables
112     if(g_powerWasLost)
113     {
114         if((nvError = _plat__NvEnable(NULL, 0)) < 0)
115             FAIL(FATAL_ERROR_NV_UNRECOVERABLE);
116         NvInitStatic();
117     }
118     return nvError == 0;
119 }
120
121 /*** NvManufacture()
122 // This function initializes the NV system at pre-install time.
123 //
124 // This function should only be called in a manufacturing environment or in a
125 // simulation.
126 //
127 // The layout of NV memory space is an implementation choice.
128 void NvManufacture(void)
129 {
130     #if SIMULATION
131         // Simulate the NV memory being in the erased state.
132         _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
133     #endif
134     // Initialize static variables
135     NvInitStatic();
136     // Clear the RAM used for Orderly Index data
137     MemorySet(s_indexOrderlyRam, 0, RAM_INDEX_SPACE);
138     // Write that Orderly Index data to NV
139     NvUpdateIndexOrderlyData();
140     // Initialize the next offset of the first entry in evict/index list to 0 (the
141     // end of list marker) and the initial s_maxCounterValue;
142     NvSetMaxCount(0);
143     // Put the end of list marker at the end of memory. This contains the MaxCount

```

```

144     // value as well as the end marker.
145     NvWriteNvListEnd(NV_USER_DYNAMIC);
146     return;
147 }
148
149 /*** NvRead()
150 // This function is used to move reserved data from NV memory to RAM.
151 void NvRead(void* outBuffer, // OUT: buffer to receive data
152             UINT32 nvOffset, // IN: offset in NV of value
153             UINT32 size      // IN: size of the value to read
154 )
155 {
156     // Input type should be valid
157     pAssert(nvOffset + size < NV_MEMORY_SIZE);
158     _plat_NvMemoryRead(nvOffset, size, outBuffer);
159     return;
160 }
161
162 /*** NvWrite()
163 // This function is used to post reserved data for writing to NV memory. Before
164 // the TPM completes the operation, the value will be written.
165 BOOL NvWrite(UINT32 nvOffset, // IN: location in NV to receive data
166             UINT32 size,      // IN: size of the data to move
167             void* inBuffer    // IN: location containing data to write
168 )
169 {
170     // Input type should be valid
171     if(nvOffset + size <= NV_MEMORY_SIZE)
172     {
173         // Set the flag that a NV write happened
174         SET_NV_UPDATE(UT_NV);
175         return _plat_NvMemoryWrite(nvOffset, size, inBuffer);
176     }
177     return FALSE;
178 }
179
180 /*** NvUpdatePersistent()
181 // This function is used to update a value in the PERSISTENT_DATA structure and
182 // commits the value to NV.
183 void NvUpdatePersistent(
184     UINT32 offset, // IN: location in PERMANENT_DATA to be updated
185     UINT32 size,   // IN: size of the value
186     void* buffer   // IN: the new data
187 )
188 {
189     pAssert(offset + size <= sizeof(gp));
190     MemoryCopy(&gp + offset, buffer, size);
191     NvWrite(offset, size, buffer);
192 }
193
194 /*** NvClearPersistent()
195 // This function is used to clear a persistent data entry and commit it to NV
196 void NvClearPersistent(UINT32 offset, // IN: the offset in the PERMANENT_DATA
197                       // structure to be cleared (zeroed)
198                       UINT32 size     // IN: number of bytes to clear
199 )
200 {
201     pAssert(offset + size <= sizeof(gp));
202     MemorySet(&gp + offset, 0, size);
203     NvWrite(offset, size, &gp + offset);
204 }
205
206 /*** NvReadPersistent()
207 // This function reads persistent data to the RAM copy of the 'gp' structure.
208 void NvReadPersistent(void)
209 {

```

```

210     NvRead(&gp, NV_PERSISTENT_DATA, sizeof(gp));
211     return;
212 }

```

7.175 /tpm/src/subsystem/Object.c

```

1  /** Introduction
2  // This file contains the functions that manage the object store of the TPM.
3
4  /** Includes and Data Definitions
5  #define OBJECT_C
6
7  #include "Tpm.h"
8  #include "Marshal.h"
9
10 /** Functions
11
12 /*** ObjectFlush()
13 // This function marks an object slot as available.
14 // Since there is no checking of the input parameters, it should be used
15 // judiciously.
16 // Note: This could be converted to a macro.
17 void ObjectFlush(OBJECT* object)
18 {
19     object->attributes.occupied = CLEAR;
20 }
21
22 /*** ObjectSetInUse()
23 // This access function sets the occupied attribute of an object slot.
24 void ObjectSetInUse(OBJECT* object)
25 {
26     object->attributes.occupied = SET;
27 }
28
29 /*** ObjectStartup()
30 // This function is called at TPM2_Startup() to initialize the object subsystem.
31 BOOL ObjectStartup(void)
32 {
33     UINT32 i;
34     //
35     // object slots initialization
36     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
37     {
38         //Set the slot to not occupied
39         ObjectFlush(&s_objects[i]);
40     }
41     return TRUE;
42 }
43
44 /*** ObjectCleanupEvict()
45 //
46 // In this implementation, a persistent object is moved from NV into an object slot
47 // for processing. It is flushed after command execution. This function is called
48 // from ExecuteCommand().
49 void ObjectCleanupEvict(void)
50 {
51     UINT32 i;
52     //
53     // This has to be iterated because a command may have two handles
54     // and they may both be persistent.
55     // This could be made to be more efficient so that a search is not needed.
56     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
57     {
58         // If an object is a temporary evict object, flush it from slot
59         OBJECT* object = &s_objects[i];

```



```

60         if(object->attributes.evict == SET)
61             ObjectFlush(object);
62     }
63     return;
64 }
65
66 /*** IsObjectPresent()
67 // This function checks to see if a transient handle references a loaded
68 // object. This routine should not be called if the handle is not a
69 // transient handle. The function validates that the handle is in the
70 // implementation-dependent allowed in range for loaded transient objects.
71 // Return Type: BOOL
72 //     TRUE(1)         handle references a loaded object
73 //     FALSE(0)        handle is not an object handle, or it does not
74 //                     reference to a loaded object
75 BOOL IsObjectPresent(TPMI_DH_OBJECT handle // IN: handle to be checked
76 )
77 {
78     UINT32 slotIndex = handle - TRANSIENT_FIRST;
79     // Since the handle is just an index into the array that is zero based, any
80     // handle value outside of the range of:
81     //     TRANSIENT_FIRST -- (TRANSIENT_FIRST + MAX_LOADED_OBJECT - 1)
82     // will now be greater than or equal to MAX_LOADED_OBJECTS
83     if(slotIndex >= MAX_LOADED_OBJECTS)
84         return FALSE;
85     // Indicate if the slot is occupied
86     return (s_objects[slotIndex].attributes.occupied == TRUE);
87 }
88
89 /*** ObjectIsSequence()
90 // This function is used to check if the object is a sequence object. This function
91 // should not be called if the handle does not reference a loaded object.
92 // Return Type: BOOL
93 //     TRUE(1)         object is an HMAC, hash, or event sequence object
94 //     FALSE(0)        object is not an HMAC, hash, or event sequence object
95 BOOL ObjectIsSequence(OBJECT* object // IN: handle to be checked
96 )
97 {
98     pAssert(object != NULL);
99     return (object->attributes.hmacSeq == SET || object->attributes.hashSeq == SET
100            || object->attributes.eventSeq == SET);
101 }
102
103 /*** HandleToObject()
104 // This function is used to find the object structure associated with a handle.
105 //
106 // This function requires that 'handle' references a loaded object or a permanent
107 // handle.
108 OBJECT* HandleToObject(TPMI_DH_OBJECT handle // IN: handle of the object
109 )
110 {
111     UINT32 index;
112     //
113     // Return NULL if the handle references a permanent handle because there is no
114     // associated OBJECT.
115     if(HandleGetType(handle) == TPM_HT_PERMANENT)
116         return NULL;
117     // In this implementation, the handle is determined by the slot occupied by the
118     // object.
119     index = handle - TRANSIENT_FIRST;
120     pAssert(index < MAX_LOADED_OBJECTS);
121     pAssert(s_objects[index].attributes.occupied);
122     return &s_objects[index];
123 }
124
125 /*** GetQualifiedName()

```

```

126 // This function returns the Qualified Name of the object. In this implementation,
127 // the Qualified Name is computed when the object is loaded and is saved in the
128 // internal representation of the object. The alternative would be to retain the
129 // Name of the parent and compute the QN when needed. This would take the same
130 // amount of space so it is not recommended that the alternate be used.
131 //
132 // This function requires that 'handle' references a loaded object.
133 void GetQualifiedName(TPMI_DH_OBJECT handle,      // IN: handle of the object
134                      TPM2B_NAME* qualifiedName    // OUT: qualified name of the object
135 )
136 {
137     OBJECT* object;
138     //
139     switch(HandleGetType(handle))
140     {
141         case TPM_HT_PERMANENT:
142             qualifiedName->t.size = sizeof(TPM_HANDLE);
143             UINT32_TO_BYTE_ARRAY(handle, qualifiedName->t.name);
144             break;
145         case TPM_HT_TRANSIENT:
146             object = HandleToObject(handle);
147             if(object == NULL || object->publicArea.nameAlg == TPM_ALG_NULL)
148                 qualifiedName->t.size = 0;
149             else
150                 // Copy the name
151                 *qualifiedName = object->qualifiedName;
152             break;
153         default:
154             FAIL(FATAL_ERROR_INTERNAL);
155     }
156     return;
157 }
158
159 /*** GetHierarchy()
160 // This function returns the handle of the hierarchy to which a handle belongs.
161 //
162 // This function requires that 'handle' references a loaded object.
163 TPMI_RH_HIERARCHY
164 GetHierarchy(TPMI_DH_OBJECT handle // IN :object handle
165 )
166 {
167     return HandleToObject(handle)->hierarchy;
168 }
169
170 /*** FindEmptyObjectSlot()
171 // This function finds an open object slot, if any. It will clear the attributes
172 // but will not set the occupied attribute. This is so that a slot may be used
173 // and discarded if everything does not go as planned.
174 // Return Type: OBJECT *
175 //     NULL          no open slot found
176 //     != NULL       pointer to available slot
177 OBJECT* FindEmptyObjectSlot(TPMI_DH_OBJECT* handle // OUT: (optional)
178 )
179 {
180     UINT32 i;
181     OBJECT* object;
182     //
183     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
184     {
185         object = &s_objects[i];
186         if(object->attributes.occupied == CLEAR)
187         {
188             if(handle)
189                 *handle = i + TRANSIENT_FIRST;
190             // Initialize the object attributes
191             MemorySet(&object->attributes, 0, sizeof(OBJECT_ATTRIBUTES));

```

```

192         object->hierarchy = TPM_RH_NULL;
193         return object;
194     }
195 }
196 return NULL;
197 }
198
199 /*** ObjectAllocateSlot()
200 // This function is used to allocate a slot in internal object array.
201 OBJECT* ObjectAllocateSlot(TPMI_DH_OBJECT* handle // OUT: handle of allocated object
202 )
203 {
204     OBJECT* object = FindEmptyObjectSlot(handle);
205     //
206     if(object != NULL)
207     {
208         // if found, mark as occupied
209         ObjectSetInUse(object);
210     }
211     return object;
212 }
213
214 /*** ObjectSetLoadedAttributes()
215 // This function sets the internal attributes for a loaded object. It is called to
216 // finalize the OBJECT attributes (not the TPMA_OBJECT attributes) for a loaded
217 // object.
218 void ObjectSetLoadedAttributes(OBJECT* object, // IN: object attributes to finalize
219                               TPM_HANDLE parentHandle // IN: the parent handle
220 )
221 {
222     OBJECT* parent = HandleToObject(parentHandle);
223     TPMA_OBJECT objectAttributes = object->publicArea.objectAttributes;
224     //
225     // Copy the stClear attribute from the public area. This could be overwritten
226     // if the parent has stClear SET
227     object->attributes.stClear = IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear);
228     // If parent handle is a permanent handle, it is a primary (unless it is NULL
229     if(parent == NULL)
230     {
231         object->hierarchy = parentHandle;
232         object->attributes.primary = SET;
233         switch(HierarchyNormalizeHandle(object->hierarchy))
234         {
235             case TPM_RH_ENDORSEMENT:
236                 object->attributes.epsHierarchy = SET;
237                 break;
238             case TPM_RH_OWNER:
239                 object->attributes.spsHierarchy = SET;
240                 break;
241             case TPM_RH_PLATFORM:
242                 object->attributes.ppsHierarchy = SET;
243                 break;
244             default:
245                 // Treat the temporary attribute as a hierarchy
246                 object->attributes.temporary = SET;
247                 object->attributes.primary = CLEAR;
248                 break;
249         }
250     }
251     else
252     {
253         // is this a stClear object
254         object->attributes.stClear =
255             (IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear)
256              || (parent->attributes.stClear == SET));
257         object->attributes.epsHierarchy = parent->attributes.epsHierarchy;

```

```

258     object->attributes.spsHierarchy = parent->attributes.spsHierarchy;
259     object->attributes.ppsHierarchy = parent->attributes.ppsHierarchy;
260     // An object is temporary if its parent is temporary or if the object
261     // is external
262     object->attributes.temporary = parent->attributes.temporary
263                               || object->attributes.external;
264     object->hierarchy = parent->hierarchy;
265 }
266 // If this is an external object, set the QN == name but don't SET other
267 // key properties ('parent' or 'derived')
268 if(object->attributes.external)
269     object->qualifiedName = object->name;
270 else
271 {
272     // check attributes for different types of parents
273     if(IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, restricted)
274       && !object->attributes.publicOnly
275       && IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, decrypt)
276       && object->publicArea.nameAlg != TPM_ALG_NULL)
277     {
278         // This is a parent. If it is not a KEYEDHASH, it is an ordinary parent.
279         // Otherwise, it is a derivation parent.
280         if(object->publicArea.type == TPM_ALG_KEYEDHASH)
281             object->attributes.derivation = SET;
282         else
283             object->attributes.isParent = SET;
284     }
285     ComputeQualifiedName(parentHandle,
286                          object->publicArea.nameAlg,
287                          &object->name,
288                          &object->qualifiedName);
289 }
290 // Set slot occupied
291 ObjectSetInUse(object);
292 return;
293 }
294
295 /*** ObjectLoad()
296 // Common function to load a non-primary object (i.e., either an Ordinary Object,
297 // or an External Object). A loaded object has its public area validated
298 // (unless its 'nameAlg' is TPM_ALG_NULL). If a sensitive part is loaded, it is
299 // verified to be correct and if both public and sensitive parts are loaded, then
300 // the cryptographic binding between the objects is validated. This function does
301 // not cause the allocated slot to be marked as in use.
302 TPM_RC
303 ObjectLoad(OBJECT* object,           // IN: pointer to object slot
304           // object
305           OBJECT* parent,           // IN: (optional) the parent object
306           TPMT_PUBLIC* publicArea,  // IN: public area to be installed in the object
307           TPMT_SENSITIVE* sensitive, // IN: (optional) sensitive area to be
308           // installed in the object
309           TPM_RC blamePublic,       // IN: parameter number to associate with the
310           // publicArea errors
311           TPM_RC blameSensitive,    // IN: parameter number to associate with the
312           // sensitive area errors
313           TPM2B_NAME* name          // IN: (optional)
314 )
315 {
316     TPM_RC result = TPM_RC_SUCCESS;
317     //
318     // Do validations of public area object descriptions
319     pAssert(publicArea != NULL);
320
321     // Is this public only or a no-name object?
322     if(sensitive == NULL || publicArea->nameAlg == TPM_ALG_NULL)
323     {

```

```

324     // Need to have schemes checked so that we do the right thing with the
325     // public key.
326     result = SchemeChecks(NULL, publicArea);
327 }
328 else
329 {
330     // For any sensitive area, make sure that the seedSize is no larger than the
331     // digest size of nameAlg
332     if(sensitive->seedValue.t.size > CryptHashGetDigestSize(publicArea->nameAlg))
333         return TPM_RCS_KEY_SIZE + blameSensitive;
334     // Check attributes and schemes for consistency
335     // For the purposes of attributes validation on this non-primary object,
336     // either:
337     // - parent is not NULL and therefore its attributes are checked for
338     //   consistency with the parent, OR
339     // - parent is NULL but the object is not a primary object, either
340     result =
341         PublicAttributesValidation(parent, /*primaryHierarchy = */ 0, publicArea);
342 }
343 if(result != TPM_RC_SUCCESS)
344     return RcSafeAddToResult(result, blamePublic);
345
346 // Sensitive area and binding checks
347
348 // On load, check nothing if the parent is fixedTPM.
349 // If the parent is fixedTPM, then this TPM produced this key blob (either
350 // by import, or creation). If the parent is not fixedTPM, then an external
351 // copy of the parent's protection seed might have been used to create the
352 // blob, and we have to validate it.
353 // NOTE: By the time a TPMT_SENSITIVE has been decrypted and passed to this
354 // function, it has been validated against the corresponding TPMT_PUBLIC.
355 // For more information about this check, see PrivateToSensitive.
356 if((parent == NULL)
357    || ((parent != NULL)
358        && !IS_ATTRIBUTE(
359            parent->publicArea.objectAttributes, TPMA_OBJECT, fixedTPM)))
360 {
361     // Do the cryptographic key validation
362     result =
363         CryptValidateKeys(publicArea, sensitive, blamePublic, blameSensitive);
364     if(result != TPM_RC_SUCCESS)
365         return result;
366 }
367 #if ALG_RSA
368 // If this is an RSA key, then expand the private exponent.
369 // Note: ObjectLoad() is only called by TPM2_Import() if the parent is fixedTPM.
370 // For any key that does not have a fixedTPM parent, the exponent is computed
371 // whenever it is loaded
372 if((publicArea->type == TPM_ALG_RSA) && (sensitive != NULL))
373 {
374     result = CryptRsaLoadPrivateExponent(publicArea, sensitive);
375     if(result != TPM_RC_SUCCESS)
376         return result;
377 }
378 #endif // ALG_RSA
379 // See if there is an object to populate
380 if((result == TPM_RC_SUCCESS) && (object != NULL))
381 {
382     // Initialize public
383     object->publicArea = *publicArea;
384     // Copy sensitive if there is one
385     if(sensitive == NULL)
386         object->attributes.publicOnly = SET;
387     else
388         object->sensitive = *sensitive;
389     // Set the name, if one was provided

```

```

390     if(name != NULL)
391         object->name = *name;
392     else
393         object->name.t.size = 0;
394 }
395 return result;
396 }
397
398 /*** AllocateSequenceSlot()
399 // This function allocates a sequence slot and initializes the parts that
400 // are used by the normal objects so that a sequence object is not inadvertently
401 // used for an operation that is not appropriate for a sequence.
402 //
403 static HASH_OBJECT* AllocateSequenceSlot(
404     TPM_HANDLE* newHandle, // OUT: receives the allocated handle
405     TPM2B_AUTH* auth       // IN: the authValue for the slot
406 )
407 {
408     HASH_OBJECT* object = (HASH_OBJECT*)ObjectAllocateSlot(newHandle);
409     //
410     // Validate that the proper location of the hash state data relative to the
411     // object state data. It would be good if this could have been done at compile
412     // time but it can't so do it in something that can be removed after debug.
413     MUST_BE(offsetof(HASH_OBJECT, auth) == offsetof(OBJECT, publicArea.authPolicy));
414
415     if(object != NULL)
416     {
417         // Set the common values that a sequence object shares with an ordinary object
418         // First, clear all attributes
419         MemorySet(&object->objectAttributes, 0, sizeof(TPMA_OBJECT));
420
421         // The type is TPM_ALG_NULL
422         object->type = TPM_ALG_NULL;
423
424         // This has no name algorithm and the name is the Empty Buffer
425         object->nameAlg = TPM_ALG_NULL;
426
427         // A sequence object is considered to be in the NULL hierarchy so it should
428         // be marked as temporary so that it can't be persisted
429         object->attributes.temporary = SET;
430
431         // A sequence object is DA exempt.
432         SET_ATTRIBUTE(object->objectAttributes, TPMA_OBJECT, noDA);
433
434         // Copy the authorization value
435         if(auth != NULL)
436             object->auth = *auth;
437         else
438             object->auth.t.size = 0;
439     }
440     return object;
441 }
442 }
443
444 #if CC_HMAC_Start || CC_MAC_Start
445 /*** ObjectCreateHMACSequence()
446 // This function creates an internal HMAC sequence object.
447 // Return Type: TPM_RC
448 //     TPM_RC_OBJECT_MEMORY      if there is no free slot for an object
449 TPM_RC
450 ObjectCreateHMACSequence(
451     TPMI_ALG_HASH hashAlg, // IN: hash algorithm
452     OBJECT* keyObject,     // IN: the object containing the HMAC key
453     TPM2B_AUTH* auth,      // IN: authValue
454     TPMI_DH_OBJECT* newHandle // OUT: HMAC sequence object handle
455 )

```



```

456 {
457     HASH_OBJECT* hmacObject;
458     //
459     // Try to allocate a slot for new object
460     hmacObject = AllocateSequenceSlot(newHandle, auth);
461
462     if(hmacObject == NULL)
463         return TPM_RC_OBJECT_MEMORY;
464     // Set HMAC sequence bit
465     hmacObject->attributes.hmacSeq = SET;
466
467 # if !SMAC_IMPLEMENTED
468     if(CryptHmacStart(&hmacObject->state.hmacState,
469                     hashAlg,
470                     keyObject->sensitive.sensitive.bits.b.size,
471                     keyObject->sensitive.sensitive.bits.b.buffer)
472        == 0)
473 # else
474     if(CryptMacStart(&hmacObject->state.hmacState,
475                    &keyObject->publicArea.parameters,
476                    hashAlg,
477                    &keyObject->sensitive.sensitive.any.b)
478        == 0)
479 # endif // SMAC_IMPLEMENTED
480     return TPM_RC_FAILURE;
481     return TPM_RC_SUCCESS;
482 }
483 #endif
484
485 /*** ObjectCreateHashSequence()
486 // This function creates a hash sequence object.
487 // Return Type: TPM_RC
488 //     TPM_RC_OBJECT_MEMORY    if there is no free slot for an object
489 TPM_RC
490 ObjectCreateHashSequence(TPMI_ALG_HASH hashAlg, // IN: hash algorithm
491                         TPM2B_AUTH* auth, // IN: authValue
492                         TPMI_DH_OBJECT* newHandle // OUT: sequence object handle
493 )
494 {
495     HASH_OBJECT* hashObject = AllocateSequenceSlot(newHandle, auth);
496     //
497     // See if slot allocated
498     if(hashObject == NULL)
499         return TPM_RC_OBJECT_MEMORY;
500     // Set hash sequence bit
501     hashObject->attributes.hashSeq = SET;
502
503     // Start hash for hash sequence
504     CryptHashStart(&hashObject->state.hashState[0], hashAlg);
505
506     return TPM_RC_SUCCESS;
507 }
508
509 /*** ObjectCreateEventSequence()
510 // This function creates an event sequence object.
511 // Return Type: TPM_RC
512 //     TPM_RC_OBJECT_MEMORY    if there is no free slot for an object
513 TPM_RC
514 ObjectCreateEventSequence(TPM2B_AUTH* auth, // IN: authValue
515                          TPMI_DH_OBJECT* newHandle // OUT: sequence object handle
516 )
517 {
518     HASH_OBJECT* hashObject = AllocateSequenceSlot(newHandle, auth);
519     UINT32 count;
520     TPM_ALG_ID hash;
521     //

```

```

522     // See if slot allocated
523     if(hashObject == NULL)
524         return TPM_RC_OBJECT_MEMORY;
525     // Set the event sequence attribute
526     hashObject->attributes.eventSeq = SET;
527
528     // Initialize hash states for each implemented PCR algorithms
529     for(count = 0; (hash = CryptHashGetAlgByIndex(count)) != TPM_ALG_NULL; count++)
530         CryptHashStart(&hashObject->state.hashState[count], hash);
531     return TPM_RC_SUCCESS;
532 }
533
534 /*** ObjectTerminateEvent()
535 // This function is called to close out the event sequence and clean up the hash
536 // context states.
537 void ObjectTerminateEvent(void)
538 {
539     HASH_OBJECT* hashObject;
540     int          count;
541     BYTE         buffer[MAX_DIGEST_SIZE];
542     //
543     hashObject = (HASH_OBJECT*)HandleToObject(g_DRTMHandle);
544
545     // Don't assume that this is a proper sequence object
546     if(hashObject->attributes.eventSeq)
547     {
548         // If it is, close any open hash contexts. This is done in case
549         // the cryptographic implementation has some context values that need to be
550         // cleaned up (hygiene).
551         //
552         for(count = 0; CryptHashGetAlgByIndex(count) != TPM_ALG_NULL; count++)
553         {
554             CryptHashEnd(&hashObject->state.hashState[count], 0, buffer);
555         }
556         // Flush sequence object
557         FlushObject(g_DRTMHandle);
558     }
559     g_DRTMHandle = TPM_RH_UNASSIGNED;
560 }
561
562 /*** ObjectContextLoad()
563 // This function loads an object from a saved object context.
564 // Return Type: OBJECT *
565 //     NULL      if there is no free slot for an object
566 //     != NULL    points to the loaded object
567 OBJECT* ObjectContextLoad(
568     ANY_OBJECT_BUFFER* object, // IN: pointer to object structure in saved
569                             // context
570     TPMI_DH_OBJECT* handle    // OUT: object handle
571 )
572 {
573     OBJECT* newObject = ObjectAllocateSlot(handle);
574     //
575     // Try to allocate a slot for new object
576     if(newObject != NULL)
577     {
578         // Copy the first part of the object
579         MemoryCopy(newObject, object, offsetof(HASH_OBJECT, state));
580         // See if this is a sequence object
581         if(ObjectIsSequence(newObject))
582         {
583             // If this is a sequence object, import the data
584             SequenceDataImport((HASH_OBJECT*)newObject, (HASH_OBJECT_BUFFER*)object);
585         }
586         else
587         {

```

```

588         // Copy input object data to internal structure
589         MemoryCopy(newObject, object, sizeof(OBJECT));
590     }
591 }
592 return newObject;
593 }
594
595 /*** FlushObject()
596 // This function frees an object slot.
597 //
598 // This function requires that the object is loaded.
599 void FlushObject(TPMI_DH_OBJECT handle // IN: handle to be freed
600 )
601 {
602     UINT32 index = handle - TRANSIENT_FIRST;
603     //
604     pAssert(index < MAX_LOADED_OBJECTS);
605     // Clear all the object attributes
606     MemorySet((BYTE*)&(s_objects[index].attributes), 0, sizeof(OBJECT_ATTRIBUTES));
607     return;
608 }
609
610 /*** ObjectFlushHierarchy()
611 // This function is called to flush all the loaded transient objects associated
612 // with a hierarchy when the hierarchy is disabled.
613 void ObjectFlushHierarchy(TPMI_RH_HIERARCHY hierarchy // IN: hierarchy to be flush
614 )
615 {
616     UINT16 i;
617     //
618     // iterate object slots
619     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
620     {
621         if(s_objects[i].attributes.occupied) // If found an occupied slot
622         {
623             switch(hierarchy)
624             {
625                 case TPM_RH_PLATFORM:
626                     if(s_objects[i].attributes.ppsHierarchy == SET)
627                         s_objects[i].attributes.occupied = FALSE;
628                     break;
629                 case TPM_RH_OWNER:
630                     if(s_objects[i].attributes.spsHierarchy == SET)
631                         s_objects[i].attributes.occupied = FALSE;
632                     break;
633                 case TPM_RH_ENDORSEMENT:
634                     if(s_objects[i].attributes.epsHierarchy == SET)
635                         s_objects[i].attributes.occupied = FALSE;
636                     break;
637                 default:
638                     FAIL(FATAL_ERROR_INTERNAL);
639                     break;
640             }
641         }
642     }
643
644     return;
645 }
646
647 /*** ObjectLoadEvict()
648 // This function loads a persistent object into a transient object slot.
649 //
650 // This function requires that 'handle' is associated with a persistent object.
651 // Return Type: TPM_RC
652 //     TPM_RC_HANDLE           the persistent object does not exist
653 //                               or the associated hierarchy is disabled.

```

```

654 //      TPM_RC_OBJECT_MEMORY      no object slot
655 TPM_RC
656 ObjectLoadEvict(TPM_HANDLE* handle, // IN:OUT: evict object handle. If success, it
657 // will be replace by the loaded object handle
658          COMMAND_INDEX commandIndex // IN: the command being processed
659 )
660 {
661     TPM_RC      result;
662     TPM_HANDLE  evictHandle = *handle; // Save the evict handle
663     OBJECT*     object;
664     //
665     // If this is an index that references a persistent object created by
666     // the platform, then return TPM_RH_HANDLE if the phEnable is FALSE
667     if(*handle >= PLATFORM_PERSISTENT)
668     {
669         // belongs to platform
670         if(g_phEnable == CLEAR)
671             return TPM_RC_HANDLE;
672     }
673     // belongs to owner
674     else if(gc.shEnable == CLEAR)
675         return TPM_RC_HANDLE;
676     // Try to allocate a slot for an object
677     object = ObjectAllocateSlot(handle);
678     if(object == NULL)
679         return TPM_RC_OBJECT_MEMORY;
680     // Copy persistent object to transient object slot. A TPM_RC_HANDLE
681     // may be returned at this point. This will mark the slot as containing
682     // a transient object so that it will be flushed at the end of the
683     // command
684     result = NvGetEvictObject(evictHandle, object);
685
686     // Bail out if this failed
687     if(result != TPM_RC_SUCCESS)
688         return result;
689     // check the object to see if it is in the endorsement hierarchy
690     // if it is and this is not a TPM2_EvictControl() command, indicate
691     // that the hierarchy is disabled.
692     // If the associated hierarchy is disabled, make it look like the
693     // handle is not defined
694     if(HierarchyNormalizeHandle(object->hierarchy) == TPM_RH_ENDORSEMENT
695        && gc.ehEnable == CLEAR && GetCommandCode(commandIndex) != TPM_CC_EvictControl)
696         return TPM_RC_HANDLE;
697
698     return result;
699 }
700
701 /*** ObjectComputeName()
702 // This does the name computation from a public area (can be marshaled or not).
703 TPM2B_NAME* ObjectComputeName(UINT32      size, // IN: the size of the area to digest
704                               BYTE*      publicArea, // IN: the public area to digest
705                               TPM_ALG_ID nameAlg, // IN: the hash algorithm to use
706                               TPM2B_NAME* name // OUT: Computed name
707 )
708 {
709     // Hash the publicArea into the name buffer leaving room for the nameAlg
710     name->t.size = CryptHashBlock(
711         nameAlg, size, publicArea, sizeof(name->t.name) - 2, &name->t.name[2]);
712     // set the nameAlg
713     UINT16_TO_BYTE_ARRAY(nameAlg, name->t.name);
714     name->t.size += 2;
715     return name;
716 }
717
718 /*** PublicMarshalAndComputeName()
719 // This function computes the Name of an object from its public area.

```

```

720 TPM2B_NAME* PublicMarshalAndComputeName(
721     TPMT_PUBLIC* publicArea, // IN: public area of an object
722     TPM2B_NAME* name         // OUT: name of the object
723 )
724 {
725     // Will marshal a public area into a template. This is because the internal
726     // format for a TPM2B_PUBLIC is a structure and not a simple BYTE buffer.
727     TPM2B_TEMPLATE marshaled; // this is big enough to hold a
728                               // marshaled TPMT_PUBLIC
729     BYTE* buffer = (BYTE*)&marshaled.t.buffer;
730     //
731     // if the nameAlg is NULL then there is no name.
732     if(publicArea->nameAlg == TPM_ALG_NULL)
733         name->t.size = 0;
734     else
735     {
736         // Marshal the public area into its canonical form
737         marshaled.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
738         // and compute the name
739         ObjectComputeName(
740             marshaled.t.size, marshaled.t.buffer, publicArea->nameAlg, name);
741     }
742     return name;
743 }
744
745 /*** ComputeQualifiedName()
746 // This function computes the qualified name of an object.
747 void ComputeQualifiedName(
748     TPM_HANDLE parentHandle, // IN: parent's handle
749     TPM_ALG_ID nameAlg,      // IN: name hash
750     TPM2B_NAME* name,        // IN: name of the object
751     TPM2B_NAME* qualifiedName // OUT: qualified name of the object
752 )
753 {
754     HASH_STATE hashState; // hash state
755     TPM2B_NAME parentName;
756     //
757     if(parentHandle == TPM_RH_UNASSIGNED)
758     {
759         MemoryCopy2B(&qualifiedName->b, &name->b, sizeof(qualifiedName->t.name));
760         *qualifiedName = *name;
761     }
762     else
763     {
764         GetQualifiedName(parentHandle, &parentName);
765
766         // QN_A = hash_A (QN of parent || NAME_A)
767
768         // Start hash
769         qualifiedName->t.size = CryptHashStart(&hashState, nameAlg);
770
771         // Add parent's qualified name
772         CryptDigestUpdate2B(&hashState, &parentName.b);
773
774         // Add self name
775         CryptDigestUpdate2B(&hashState, &name->b);
776
777         // Complete hash leaving room for the name algorithm
778         CryptHashEnd(&hashState, qualifiedName->t.size, &qualifiedName->t.name[2]);
779         UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
780         qualifiedName->t.size += 2;
781     }
782     return;
783 }
784
785 /*** ObjectIsStorage()

```

```

786 // This function determines if an object has the attributes associated
787 // with a parent. A parent is an asymmetric or symmetric block cipher key
788 // that has its 'restricted' and 'decrypt' attributes SET, and 'sign' CLEAR.
789 // Return Type: BOOL
790 //     TRUE(1)         object is a storage key
791 //     FALSE(0)        object is not a storage key
792 BOOL ObjectIsStorage(TPMI_DH_OBJECT handle // IN: object handle
793 )
794 {
795     OBJECT*      object      = HandleToObject(handle);
796     TPMT_PUBLIC* publicArea = ((object != NULL) ? &object->publicArea : NULL);
797     //
798     return (publicArea != NULL
799         && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
800         && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
801         && !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
802         && (object->publicArea.type == TPM_ALG_RSA
803             || object->publicArea.type == TPM_ALG_ECC));
804 }
805
806 /*** ObjectCapGetLoaded()
807 // This function returns a list of handles of loaded object, starting from
808 // 'handle'. 'Handle' must be in the range of valid transient object handles,
809 // but does not have to be the handle of a loaded transient object.
810 // Return Type: TPMI_YES_NO
811 //     YES         if there are more handles available
812 //     NO          all the available handles has been returned
813 TPMI_YES_NO
814 ObjectCapGetLoaded(TPMI_DH_OBJECT handle, // IN: start handle
815     UINT32 count, // IN: count of returned handles
816     TPML_HANDLE* handleList // OUT: list of handle
817 )
818 {
819     TPMI_YES_NO more = NO;
820     UINT32 i;
821     //
822     pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
823
824     // Initialize output handle list
825     handleList->count = 0;
826
827     // The maximum count of handles we may return is MAX_CAP_HANDLES
828     if(count > MAX_CAP_HANDLES)
829         count = MAX_CAP_HANDLES;
830
831     // Iterate object slots to get loaded object handles
832     for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
833     {
834         if(s_objects[i].attributes.occupied == TRUE)
835         {
836             // A valid transient object can not be the copy of a persistent object
837             pAssert(s_objects[i].attributes.evict == CLEAR);
838
839             if(handleList->count < count)
840             {
841                 // If we have not filled up the return list, add this object
842                 // handle to it
843                 handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
844                 handleList->count++;
845             }
846             else
847             {
848                 // If the return list is full but we still have loaded object
849                 // available, report this and stop iterating
850                 more = YES;
851                 break;

```



```

852     }
853 }
854 }
855
856 return more;
857 }
858
859 /*** ObjectCapGetOneLoaded()
860 // This function returns whether a handle is loaded.
861 BOOL ObjectCapGetOneLoaded(TPMI_DH_OBJECT handle) // IN: handle
862 {
863     UINT32 i;
864
865     pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
866
867     // Iterate object slots to get loaded object handles
868     for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
869     {
870         if(s_objects[i].attributes.occupied == TRUE)
871         {
872             // A valid transient object can not be the copy of a persistent object
873             pAssert(s_objects[i].attributes.evict == CLEAR);
874
875             return TRUE;
876         }
877     }
878
879     return FALSE;
880 }
881
882 /*** ObjectCapGetTransientAvail()
883 // This function returns an estimate of the number of additional transient
884 // objects that could be loaded into the TPM.
885 UINT32
886 ObjectCapGetTransientAvail(void)
887 {
888     UINT32 i;
889     UINT32 num = 0;
890     //
891     // Iterate object slot to get the number of unoccupied slots
892     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
893     {
894         if(s_objects[i].attributes.occupied == FALSE)
895             num++;
896     }
897
898     return num;
899 }
900
901 /*** ObjectGetPublicAttributes()
902 // Returns the attributes associated with an object handles.
903 TPMA_OBJECT
904 ObjectGetPublicAttributes(TPM_HANDLE handle)
905 {
906     return HandleToObject(handle)->publicArea.objectAttributes;
907 }
908
909 OBJECT_ATTRIBUTES
910 ObjectGetProperties(TPM_HANDLE handle)
911 {
912     return HandleToObject(handle)->attributes;
913 }

```

7.176 /tpm/src/subsystem/PCR.c

```

1  /** Introduction
2  //
3  // This function contains the functions needed for PCR access and manipulation.
4  //
5  // This implementation uses a static allocation for the PCR. The amount of
6  // memory is allocated based on the number of PCR in the implementation and
7  // the number of implemented hash algorithms. This is not the expected
8  // implementation. PCR SPACE DEFINITIONS.
9  //
10 // In the definitions below, the g_hashPcrMap is a bit array that indicates
11 // which of the PCR are implemented. The g_hashPcr array is an array of digests.
12 // In this implementation, the space is allocated whether the PCR is implemented
13 // or not.
14
15 /** Includes, Defines, and Data Definitions
16 #define PCR_C
17 #include "Tpm.h"
18
19 // verify values from pcrstruct.h. not <= because group #0 is reserved
20 // indicating no auth/policy support
21 TPM_STATIC_ASSERT(NUM_AUTHVALUE_PCR_GROUP < (1 << MAX_PCR_GROUP_BITS));
22 TPM_STATIC_ASSERT(NUM_POLICY_PCR_GROUP < (1 << MAX_PCR_GROUP_BITS));
23
24 /** Functions
25
26 /** PCRBelongsAuthGroup()
27 // This function indicates if a PCR belongs to a group that requires an authValue
28 // in order to modify the PCR. If it does, 'groupIndex' is set to value of
29 // the group index. This feature of PCR is decided by the platform specification.
30 //
31 // Return Type: BOOL
32 //     TRUE(1)      PCR belongs an authorization group
33 //     FALSE(0)     PCR does not belong an authorization group
34 BOOL PCRBelongsAuthGroup(TPMI_DH_PCR handle, // IN: handle of PCR
35                          UINT32* groupIndex // OUT: group array index if PCR
36                          // belongs to a group that allows authValue. If PCR
37                          // does not belong to an authorization
38                          // group, the value in this parameter is zero
39 )
40 {
41     *groupIndex = 0;
42
43 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
44     // Platform specification determines to which authorization group a PCR belongs
45     // (if any). In this implementation, we assume there is only
46     // one authorization group which contains PCR[20-22]. If the platform
47     // specification requires differently, the implementation should be changed
48     // accordingly
49     UINT32 pcr = handle - PCR_FIRST;
50     PCR_Attributes currentPcrAttributes =
51         _platPcr__GetPcrInitializationAttributes(pcr);
52
53     if(currentPcrAttributes.authValuesGroup != 0)
54     {
55         // turn 1-based group number into actual array index expected by callers
56         *groupIndex = currentPcrAttributes.authValuesGroup - 1;
57         pAssert_BOOL(*groupIndex < NUM_AUTHVALUE_PCR_GROUP);
58         return TRUE;
59     }
60
61 #endif
62     return FALSE;
63 }
64

```

```

65  /*** PCRBelongsPolicyGroup()
66  // This function indicates if a PCR belongs to a group that requires a policy
67  // authorization in order to modify the PCR. If it does, 'groupIndex' is set
68  // to value of the group index. This feature of PCR is decided by the platform
69  // specification.
70  // return type: BOOL
71  //     TRUE:          PCR belongs a policy group
72  //     FALSE:         PCR does not belong a policy group
73  BOOL PCRBelongsPolicyGroup(
74      TPMI_DH_PCR handle, // IN: handle of PCR
75      UINT32* groupIndex // OUT: group index if PCR belongs a group that
76                          // allows policy. If PCR does not belong to
77                          // a policy group, the value in this
78                          // parameter is zero
79  )
80  {
81      *groupIndex = 0;
82
83      #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
84          // Platform specification decides if a PCR belongs to a policy group and
85          // belongs to which group.
86          UINT32 pcr = handle - PCR_FIRST;
87          PCR_Attributes currentPcrAttributes =
88              _platPcr_GetPcrInitializationAttributes(pcr);
89          if(currentPcrAttributes.policyAuthGroup != 0)
90          {
91              // turn 1-based group number into actual array index expected by callers
92              *groupIndex = currentPcrAttributes.policyAuthGroup - 1;
93              pAssert_BOOL(*groupIndex < NUM_POLICY_PCR_GROUP);
94              return TRUE;
95          }
96      #endif
97      return FALSE;
98  }
99
100 /*** PCRBelongsTCBGroup()
101 // This function indicates if a PCR belongs to the TCB group.
102 // return type: BOOL
103 //     TRUE:          PCR belongs to TCB group
104 //     FALSE:         PCR does not belong to TCB group
105 static BOOL PCRBelongsTCBGroup(TPMI_DH_PCR handle // IN: handle of PCR
106 )
107 {
108     #if ENABLE_PCR_NO_INCREMENT == YES
109         // Platform specification decides if a PCR belongs to a TCB group.
110         UINT32 pcr = handle - PCR_FIRST;
111         PCR_Attributes currentPcrAttributes =
112             _platPcr_GetPcrInitializationAttributes(pcr);
113         return currentPcrAttributes.doNotIncrementPcrCounter;
114     #else
115         return FALSE;
116     #endif
117 }
118
119 /*** PCRPolicyIsAvailable()
120 // This function indicates if a policy is available for a PCR.
121 // return type: BOOL
122 //     TRUE          the PCR may be authorized by policy
123 //     FALSE         the PCR does not allow policy
124 BOOL PCRPolicyIsAvailable(TPMI_DH_PCR handle // IN: PCR handle
125 )
126 {
127     UINT32 groupIndex;
128
129     return PCRBelongsPolicyGroup(handle, &groupIndex);
130 }

```

```

131
132 /*** PCRGetAuthValue()
133 // This function is used to access the authValue of a PCR. If PCR does not
134 // belong to an authValue group, an EmptyAuth will be returned.
135 TPM2B_AUTH* PCRGetAuthValue(TPMI_DH_PCR handle // IN: PCR handle
136 )
137 {
138     UINT32 groupIndex;
139
140     if(PCRBelongsAuthGroup(handle, &groupIndex))
141     {
142         return &gc.pcrAuthValues.auth[groupIndex];
143     }
144     else
145     {
146         return NULL;
147     }
148 }
149
150 /*** PCRGetAuthPolicy()
151 // This function is used to access the authorization policy of a PCR. It sets
152 // 'policy' to the authorization policy and returns the hash algorithm for policy
153 // If the PCR does not allow a policy, TPM_ALG_NULL is returned.
154 TPMI_ALG_HASH
155 PCRGetAuthPolicy(TPMI_DH_PCR handle, // IN: PCR handle
156                 TPM2B_DIGEST* policy // OUT: policy of PCR
157 )
158 {
159     UINT32 groupIndex;
160
161     if(PCRBelongsPolicyGroup(handle, &groupIndex))
162     {
163         *policy = gp.pcrPolicies.policy[groupIndex];
164         return gp.pcrPolicies.hashAlg[groupIndex];
165     }
166     else
167     {
168         policy->t.size = 0;
169         return TPM_ALG_NULL;
170     }
171 }
172
173 /*** PCRManufacture()
174 // This function is used to initialize the policies when a TPM is manufactured.
175 // This function would only be called in a manufacturing environment or in
176 // a TPM simulator.
177 void PCRManufacture(void)
178 {
179     UINT32 i;
180 #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
181     for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
182     {
183         gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
184         gp.pcrPolicies.policy[i].t.size = 0;
185     }
186 #endif
187 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
188     for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
189     {
190         gc.pcrAuthValues.auth[i].t.size = 0;
191     }
192 #endif
193 // We need to give an initial configuration on allocated PCR before
194 // receiving any TPM2_PCR_Allocate command to change this configuration
195 // When the simulation environment starts, we allocate all the PCRs
196 for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;

```

```

197     gp.pcrAllocated.count++)
198 {
199     TPM_ALG_ID currentBank = CryptHashGetAlgByIndex(gp.pcrAllocated.count);
200     BOOL        isBankActive = _platPcr_IsPcrBankDefaultActive(currentBank);
201
202     gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash = currentBank;
203
204     gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect =
205         PCR_SELECT_MAX;
206     for(i = 0; i < PCR_SELECT_MAX; i++)
207     {
208         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i] =
209             isBankActive ? 0xFF : 0;
210     }
211 }
212
213 // Store the initial configuration to NV
214 NV_SYNC_PERSISTENT(pcrPolicies);
215 NV_SYNC_PERSISTENT(pcrAllocated);
216
217 return;
218 }
219
220 /*** GetSavedPcrPointer()
221 // This function returns the address of an array of state saved PCR based
222 // on the hash algorithm.
223 //
224 // Return Type: BYTE *
225 //     NULL          no such algorithm
226 //     != NULL       pointer to the 0th byte of the 0th PCR
227 static BYTE* GetSavedPcrPointer(TPM_ALG_ID alg,      // IN: algorithm for bank
228                                UINT32    pcrIndex  // IN: PCR index in PCR_SAVE
229 )
230 {
231     BYTE* retVal = NULL;
232     switch(alg)
233     {
234 #define HASH_CASE(HASH, Hash) \
235     case TPM_ALG_ ##HASH: \
236         retVal = gc.pcrSave.Hash[pcrIndex]; \
237         break;
238
239     FOR_EACH_HASH(HASH_CASE)
240 #undef HASH_CASE
241
242     default:
243         FAIL_NULL(FATAL_ERROR_INTERNAL);
244     }
245     return retVal;
246 }
247
248 /*** PcrIsAllocated()
249 // This function indicates if a PCR number for the particular hash algorithm
250 // is allocated.
251 // Return Type: BOOL
252 //     TRUE(1)      PCR is allocated
253 //     FALSE(0)     PCR is not allocated
254 BOOL PcrIsAllocated(UINT32    pcr,      // IN: The number of the PCR
255                     TPMI_ALG_HASH hashAlg // IN: The PCR algorithm
256 )
257 {
258     UINT32 i;
259     BOOL    allocated = FALSE;
260
261     if(pcr < IMPLEMENTATION_PCR)
262     {

```

```

263     for(i = 0; i < gp.pcrAllocated.count; i++)
264     {
265         if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
266         {
267             if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr / 8])
268                 & (1 << (pcr % 8)))
269                 != 0)
270                 allocated = TRUE;
271             else
272                 allocated = FALSE;
273             break;
274         }
275     }
276 }
277 return allocated;
278 }
279
280 // Get pointer to particular PCR from bank (array)
281 // CAUTION: This function does not validate the pcrNumber
282 // vs the size of the array.
283 // See Also: GetPcrPointerIfAllocated
284 BYTE* GetPcrPointerFromPcrArray(PCR* pPcrArray,
285                                 TPM_ALG_ID alg, // IN: algorithm for bank
286                                 UINT32 pcrNumber // IN: PCR number
287 )
288 {
289     switch(alg)
290     {
291     #if ALG_SHA1
292         case TPM_ALG_SHA1:
293             return pPcrArray[pcrNumber].Sha1Pcr;
294     #endif
295     #if ALG_SHA256
296         case TPM_ALG_SHA256:
297             return pPcrArray[pcrNumber].Sha256Pcr;
298     #endif
299     #if ALG_SHA384
300         case TPM_ALG_SHA384:
301             return pPcrArray[pcrNumber].Sha384;
302     #endif
303     #if ALG_SHA512
304         case TPM_ALG_SHA512:
305             return pPcrArray[pcrNumber].Sha512;
306     #endif
307     #if ALG_SM3_256
308         case TPM_ALG_SM3_256:
309             return pPcrArray[pcrNumber].Sm3_256;
310     #endif
311     #if ALG_SHA3_256
312         case TPM_ALG_SHA3_256:
313             return pPcrArray[pcrNumber].Sha3_256;
314     #endif
315     #if ALG_SHA3_384
316         case TPM_ALG_SHA3_384:
317             return pPcrArray[pcrNumber].Sha3_384;
318     #endif
319     #if ALG_SHA3_512
320         case TPM_ALG_SHA3_512:
321             return pPcrArray[pcrNumber].Sha3_512;
322     #endif
323         default:
324             FAIL(FATAL_ERROR_INTERNAL);
325             break;
326     }
327     return NULL;
328 }

```



```

329
330 BYTE* GetPcrPointerIfAllocated(PCR*      pPcrArray,
331                               TPM_ALG_ID alg,      // IN: algorithm for bank
332                               UINT32      pcrNumber // IN: PCR number
333 )
334 {
335     //
336     if(!PcrIsAllocated(pcrNumber, alg))
337         return NULL;
338
339     return GetPcrPointerFromPcrArray(pPcrArray,
340                                     alg,      // IN: algorithm for bank
341                                     pcrNumber // IN: PCR number
342     );
343 }
344
345 /*** GetPcrPointer()
346 // This function returns the address of an array of PCR based on the
347 // hash algorithm.
348 //
349 // Return Type: BYTE *
350 //     NULL      no such algorithm
351 //     != NULL    pointer to the 0th byte of the requested PCR
352 BYTE* GetPcrPointer(TPM_ALG_ID alg,      // IN: algorithm for bank
353                    UINT32      pcrNumber // IN: PCR number
354 )
355 {
356     return GetPcrPointerIfAllocated(s_pcrs, alg, pcrNumber);
357 }
358
359 /*** IsPcrSelected()
360 // This function indicates if an indicated PCR number is selected by the bit map in
361 // 'selection'.
362 //
363 // Return Type: BOOL
364 //     TRUE(1)    PCR is selected
365 //     FALSE(0)   PCR is not selected
366 static BOOL IsPcrSelected(
367     UINT32      pcr,      // IN: The number of the PCR
368     TPMS_PCR_SELECTION* selection // IN: The selection structure
369 )
370 {
371     BOOL selected;
372     selected = (pcr < IMPLEMENTATION_PCR
373                && ((selection->pcrSelect[pcr / 8]) & (1 << (pcr % 8))) != 0);
374     return selected;
375 }
376
377 /*** FilterPcr()
378 // This function modifies a PCR selection array based on the implemented
379 // PCR.
380 static void FilterPcr(TPMS_PCR_SELECTION* selection // IN: input PCR selection
381 )
382 {
383     UINT32      i;
384     TPMS_PCR_SELECTION* allocated = NULL;
385
386     // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
387     for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
388         selection->pcrSelect[i] = 0;
389
390     // Find the internal configuration for the bank
391     for(i = 0; i < gp.pcrAllocated.count; i++)
392     {
393         if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
394             {

```

```

395         allocated = &gp.pcrAllocated.pcrSelections[i];
396         break;
397     }
398 }
399
400 for(i = 0; i < selection->sizeofSelect; i++)
401 {
402     if(allocated == NULL)
403     {
404         // If the required bank does not exist, clear input selection
405         selection->pcrSelect[i] = 0;
406     }
407     else
408         selection->pcrSelect[i] &= allocated->pcrSelect[i];
409 }
410
411 return;
412 }
413
414 /*** PcrDrtm()
415 // This function does the DRTM and H-CRTM processing it is called from
416 // _TPM_Hash_End.
417 void PcrDrtm(const TPMI_DH_PCR pcrHandle, // IN: the index of the PCR to be
418             // modified
419             const TPMI_ALG_HASH hash,    // IN: the bank identifier
420             const TPM2B_DIGEST* digest  // IN: the digest to modify the PCR
421 )
422 {
423     BYTE* pcrData = GetPcrPointer(hash, pcrHandle);
424
425     if(pcrData != NULL)
426     {
427         // Rest the PCR to zeros
428         MemorySet(pcrData, 0, digest->t.size);
429
430         // if the TPM has not started, then set the PCR to 0...04 and then extend
431         if(!TPMIsStarted())
432         {
433             pcrData[digest->t.size - 1] = 4;
434         }
435         // Now, extend the value
436         PCRExtend(pcrHandle, hash, digest->t.size, (BYTE*)digest->t.buffer);
437     }
438 }
439
440 /*** PCR_ClearAuth()
441 // This function is used to reset the PCR authorization values. It is called
442 // on TPM2_Startup(CLEAR) and TPM2_Clear().
443 void PCR_ClearAuth(void)
444 {
445     #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
446     int j;
447     for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
448     {
449         gc.pcrAuthValues.auth[j].t.size = 0;
450     }
451     #endif
452 }
453
454 /*** PCRStartup()
455 // This function initializes the PCR subsystem at TPM2_Startup().
456 BOOL PCRStartup(STARTUP_TYPE type, // IN: startup type
457                BYTE locality // IN: startup locality
458 )
459 {
460     UINT32 pcr, j;

```

```

461     UINT32 saveIndex = 0;
462
463     g_pcrReConfig      = FALSE;
464
465     // Don't test for SU_RESET because that should be the default when nothing
466     // else is selected
467     if(type != SU_RESUME && type != SU_RESTART)
468     {
469         // PCR generation counter is cleared at TPM_RESET
470         gr.pcrCounter = 0;
471     }
472
473     // check the TPM library and platform are properly paired.
474     // if this fails the platform and library are compiled with different
475     // definitions of the number of PCRs - immediately enter FAILURE mode and
476     // return FALSE
477     pAssert_BOOL(_platPcr__NumberOfPcrs() == IMPLEMENTATION_PCR);
478
479     // Initialize/Restore PCR values
480     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
481     {
482         // On resume, need to know if this PCR had its state saved or not
483         UINT32 stateSaved;
484         // note structure is a bitfield and returned by value.
485         PCR_Attributes currentPcrAttributes =
486             _platPcr__GetPcrInitializationAttributes(pcr);
487
488         if(type == SU_RESUME && currentPcrAttributes.stateSave == SET)
489         {
490             stateSaved = 1;
491         }
492         else
493         {
494             stateSaved = 0;
495             PCRChanged(pcr);
496         }
497
498         // If this is the H-CRTM PCR and we are not doing a resume and we
499         // had an H-CRTM event, then we don't change this PCR
500         if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
501             continue;
502
503         // Iterate each hash algorithm bank
504         for(j = 0; j < gp.pcrAllocated.count; j++)
505         {
506             TPMI_ALG_HASH hash      = gp.pcrAllocated.pcrSelections[j].hash;
507             BYTE*          pcrData = GetPcrPointer(hash, pcr);
508             UINT16          pcrSize = CryptHashGetDigestSize(hash);
509
510             if(pcrData != NULL)
511             {
512                 // if state was saved
513                 if(stateSaved == 1)
514                 {
515                     // Restore saved PCR value
516                     BYTE* pcrSavedData;
517                     pcrSavedData = GetSavedPcrPointer(hash, saveIndex);
518                     if(pcrSavedData == NULL)
519                         return FALSE;
520                     MemoryCopy(pcrData, pcrSavedData, pcrSize);
521                 }
522                 else // PCR was not restored by state save
523                 {
524                     // give platform opportunity to provide the PCR initialization
525                     // value and it's length. this provides a platform specification
526                     // the ability to change the default values without affecting the

```

```

527         // core library. if the platform doesn't have a value, then the
528         // result is expected to be TPM_RC_PCR and the size to be 0 and we
529         // provide the original defaults.
530         uint16_t pcrLength = 0;
531         TPM_RC pcrInitialResult = _platPcr_GetInitialValueForPcr(
532             pcr, hash, locality, pcrData, pcrSize, &pcrLength);
533
534         // any other result is a fatal error
535         pAssert_BOOL(pcrInitialResult == TPM_RC_SUCCESS
536             || pcrInitialResult == TPM_RC_PCR);
537         if(pcrInitialResult == TPM_RC_SUCCESS && pcrLength == pcrSize)
538         {
539             // just use the PCR initialized by platform
540         }
541         else
542         {
543             // If the reset locality contains locality 4, then this
544             // indicates a DRTM PCR where the reset value is all ones,
545             // otherwise it is all zero. Don't check with equal because
546             // resetLocality is a bitfield of multiple values and does
547             // not support extended localities.
548             BYTE defaultValue = 0;
549             if((currentPcrAttributes.resetLocality & 0x10) != 0)
550             {
551                 defaultValue = 0xFF;
552             }
553             MemorySet(pcrData, defaultValue, pcrSize);
554             if(pcr == HCRTP_PCR)
555             {
556                 pcrData[pcrSize - 1] = locality;
557             }
558         }
559     }
560 }
561 }
562     saveIndex += stateSaved;
563 }
564 // Reset authValues on TPM2_Startup(CLEAR)
565 if(type != SU_RESUME)
566     PCR_ClearAuth();
567 return TRUE;
568 }
569
570 /*** PCRStateSave()
571 // This function is used to save the PCR values that will be restored on TPM Resume.
572 void PCRStateSave(TPM_SU type // IN: startup type
573 )
574 {
575     UINT32 pcr, j;
576     UINT32 saveIndex = 0;
577
578     // if state save CLEAR, nothing to be done. Return here
579     if(type == TPM_SU_CLEAR)
580         return;
581
582     // Copy PCR values to the structure that should be saved to NV
583     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
584     {
585         PCR_Attributes currentPcrAttributes =
586             _platPcr_GetPcrInitializationAttributes(pcr);
587
588         UINT32 stateSaved = (currentPcrAttributes.stateSave == SET) ? 1 : 0;
589
590         // Iterate each hash algorithm bank
591         for(j = 0; j < gp.pcrAllocated.count; j++)
592         {

```

```

593     BYTE* pcrData;
594     UINT32 pcrSize;
595
596     pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
597
598     if(pcrData != NULL)
599     {
600         pcrSize =
601             CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
602
603         if(stateSaved == 1)
604         {
605             // Restore saved PCR value
606             BYTE* pcrSavedData;
607             pcrSavedData = GetSavedPcrPointer(
608                 gp.pcrAllocated.pcrSelections[j].hash, saveIndex);
609             MemoryCopy(pcrSavedData, pcrData, pcrSize);
610         }
611     }
612     }
613     saveIndex += stateSaved;
614 }
615
616 return;
617 }
618
619 /*** PCRIsStateSaved()
620 // This function indicates if the selected PCR is a PCR that is state saved
621 // on TPM2_Shutdown(STATE). The return value is based on PCR attributes.
622 // Return Type: BOOL
623 // TRUE(1) PCR is state saved
624 // FALSE(0) PCR is not state saved
625 BOOL PCRIsStateSaved(TPMI_DH_PCR handle // IN: PCR handle to be extended
626 )
627 {
628     UINT32 pcr = handle - PCR_FIRST;
629     PCR_Attributes currentPcrAttributes =
630         _platPcr__GetPcrInitializationAttributes(pcr);
631
632     if(currentPcrAttributes.stateSave == SET)
633         return TRUE;
634     else
635         return FALSE;
636 }
637
638 /*** PCRIsResetAllowed()
639 // This function indicates if a PCR may be reset by the current command locality.
640 // The return value is based on PCR attributes, and not the PCR allocation.
641 // Return Type: BOOL
642 // TRUE(1) TPM2_PCR_Reset is allowed
643 // FALSE(0) TPM2_PCR_Reset is not allowed
644 BOOL PCRIsResetAllowed(TPMI_DH_PCR handle // IN: PCR handle to be extended
645 )
646 {
647     UINT8 commandLocality;
648     UINT8 localityBits = 1;
649     UINT32 pcr = handle - PCR_FIRST;
650     PCR_Attributes currentPcrAttributes =
651         _platPcr__GetPcrInitializationAttributes(pcr);
652
653     // Check for the locality
654     commandLocality = _plat__LocalityGet();
655
656 #ifdef DRTM_PCR
657     // For a TPM that does DRTM, Reset is not allowed at locality 4
658     if(commandLocality == 4)

```

```

659         return FALSE;
660 #endif
661
662     localityBits = localityBits << commandLocality;
663     if((localityBits & currentPcrAttributes.resetLocality) == 0)
664         return FALSE;
665     else
666         return TRUE;
667 }
668
669 /*** PCRChanged()
670 // This function checks a PCR handle to see if the attributes for the PCR are set
671 // so that any change to the PCR causes an increment of the pcrCounter. If it does,
672 // then the function increments the counter. Will also bump the counter if the
673 // handle is zero which means that PCR 0 can not be in the TCB group. Bump on zero
674 // is used by TPM2_Clear().
675 void PCRChanged(TPM_HANDLE pcrHandle // IN: the handle of the PCR that changed.
676 )
677 {
678     // For the reference implementation, the only change that does not cause
679     // increment is a change to a PCR in the TCB group.
680     if((pcrHandle == 0) || !PCRBelongsTCBGroup(pcrHandle))
681     {
682         gr.pcrCounter++;
683         if(gr.pcrCounter == 0)
684             FAIL(FATAL_ERROR_COUNTER_OVERFLOW);
685     }
686 }
687
688 /*** PCRIExtendAllowed()
689 // This function indicates a PCR may be extended at the current command locality.
690 // The return value is based on PCR attributes, and not the PCR allocation.
691 // Return Type: BOOL
692 //     TRUE(1)         extend is allowed
693 //     FALSE(0)        extend is not allowed
694 BOOL PCRIExtendAllowed(TPMI_DH_PCR handle // IN: PCR handle to be extended
695 )
696 {
697     UINT8      commandLocality;
698     UINT8      localityBits = 1;
699     UINT32     pcr          = handle - PCR_FIRST;
700     PCR_Attributes currentPcrAttributes =
701         _platPcr_GetPcrInitializationAttributes(pcr);
702
703     // Check for the locality
704     commandLocality = _plat_LocalityGet();
705     localityBits    = localityBits << commandLocality;
706     if((localityBits & currentPcrAttributes.extendLocality) == 0)
707         return FALSE;
708     else
709         return TRUE;
710 }
711
712 /*** PCRExtend()
713 // This function is used to extend a PCR in a specific bank.
714 void PCRExtend(TPMI_DH_PCR handle, // IN: PCR handle to be extended
715               TPMI_ALG_HASH hash, // IN: hash algorithm of PCR
716               UINT32 size, // IN: size of data to be extended
717               BYTE* data // IN: data to be extended
718 )
719 {
720     BYTE* pcrData;
721     HASH_STATE hashState;
722     UINT16 pcrSize;
723
724     pcrData = GetPcrPointer(hash, handle - PCR_FIRST);

```



```

725
726 // Extend PCR if it is allocated
727 if(pcrData != NULL)
728 {
729     pcrSize = CryptHashGetDigestSize(hash);
730     CryptHashStart(&hashState, hash);
731     CryptDigestUpdate(&hashState, pcrSize, pcrData);
732     CryptDigestUpdate(&hashState, size, data);
733     CryptHashEnd(&hashState, pcrSize, pcrData);
734
735     // PCR has changed so update the pcrCounter if necessary
736     PCRChanged(handle);
737 }
738
739 return;
740 }
741
742 /*** PCRComputeCurrentDigest()
743 // This function computes the digest of the selected PCR.
744 //
745 // As a side-effect, 'selection' is modified so that only the implemented PCR
746 // will have their bits still set.
747 void PCRComputeCurrentDigest(
748     TPMI_ALG_HASH hashAlg, // IN: hash algorithm to compute digest
749     TPML_PCR_SELECTION* selection, // IN/OUT: PCR selection (filtered on
750                                     // output)
751     TPM2B_DIGEST* digest // OUT: digest
752 )
753 {
754     HASH_STATE hashState;
755     TPMS_PCR_SELECTION* select;
756     BYTE* pcrData; // will point to a digest
757     UINT32 pcrSize;
758     UINT32 pcr;
759     UINT32 i;
760
761     // Initialize the hash
762     digest->t.size = CryptHashStart(&hashState, hashAlg);
763     pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
764
765     // Iterate through the list of PCR selection structures
766     for(i = 0; i < selection->count; i++)
767     {
768         // Point to the current selection
769         select = &selection->pcrSelections[i]; // Point to the current selection
770         FilterPcr(select); // Clear out the bits for unimplemented PCR
771
772         // Need the size of each digest
773         pcrSize = CryptHashGetDigestSize(selection->pcrSelections[i].hash);
774
775         // Iterate through the selection
776         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
777         {
778             if(IsPcrSelected(pcr, select)) // Is this PCR selected
779             {
780                 // Get pointer to the digest data for the bank
781                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
782                 pAssert(pcrData != NULL);
783                 CryptDigestUpdate(&hashState, pcrSize, pcrData); // add to digest
784             }
785         }
786     }
787     // Complete hash stack
788     CryptHashEnd2B(&hashState, &digest->b);
789
790     return;

```

```

791 }
792
793 /*** PCRRead()
794 // This function is used to read a list of selected PCR. If the requested PCR
795 // number exceeds the maximum number that can be output, the 'selection' is
796 // adjusted to reflect the actual output PCR.
797 void PCRRead(TPML_PCR_SELECTION* selection, // IN/OUT: PCR selection (filtered on
798 // output)
799             TPML_DIGEST* digest,           // OUT: digest
800             UINT32*      pcrCounter        // OUT: the current value of PCR generation
801             // number
802 )
803 {
804     TPMS_PCR_SELECTION* select;
805     BYTE*                pcrData; // will point to a digest
806     UINT32                pcr;
807     UINT32                i;
808
809     digest->count = 0;
810
811     // Iterate through the list of PCR selection structures
812     for(i = 0; i < selection->count; i++)
813     {
814         // Point to the current selection
815         select = &selection->pcrSelections[i]; // Point to the current selection
816         FilterPcr(select); // Clear out the bits for unimplemented PCR
817
818         // Iterate through the selection
819         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
820         {
821             if(IsPcrSelected(pcr, select)) // Is this PCR selected
822             {
823                 // Check if number of digest exceed upper bound
824                 if(digest->count > 7)
825                 {
826                     // Clear rest of the current select bitmap
827                     while(pcr < IMPLEMENTATION_PCR
828                         // do not round up!
829                         && (pcr / 8) < select->sizeofSelect)
830                     {
831                         // do not round up!
832                         select->pcrSelect[pcr / 8] &= (BYTE) ~(1 << (pcr % 8));
833                         pcr++;
834                     }
835                     // Exit inner loop
836                     break;
837                 }
838                 // Need the size of each digest
839                 digest->digests[digest->count].t.size =
840                     CryptHashGetDigestSize(select->pcrSelections[i].hash);
841
842                 // Get pointer to the digest data for the bank
843                 pcrData = GetPcrPointer(select->pcrSelections[i].hash, pcr);
844                 pAssert(pcrData != NULL);
845                 // Add to the data to digest
846                 MemoryCopy(digest->digests[digest->count].t.buffer,
847                     pcrData,
848                     digest->digests[digest->count].t.size);
849                 digest->count++;
850             }
851         }
852         // If we exit inner loop because we have exceed the output upper bound
853         if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
854         {
855             // Clear rest of the selection
856             while(i < selection->count)

```

```

857     {
858         MemorySet(selection->pcrSelections[i].pcrSelect,
859                 0,
860                 selection->pcrSelections[i].sizeofSelect);
861         i++;
862     }
863     // exit outer loop
864     break;
865 }
866 }
867
868 *pcrCounter = gr.pcrCounter;
869
870 return;
871 }
872
873 /** PCRAAllocate()
874 // This function is used to change the PCR allocation.
875 // Return Type: TPM_RC
876 //     TPM_RC_NO_RESULT      allocate failed
877 //     TPM_RC_PCR            improper allocation
878 TPM_RC
879 PCRAAllocate(TPML_PCR_SELECTION* allocate, // IN: required allocation
880             UINT32* maxPCR, // OUT: Maximum number of PCR
881             UINT32* sizeNeeded, // OUT: required space
882             UINT32* sizeAvailable // OUT: available space
883 )
884 {
885     UINT32 i, j, k;
886     TPML_PCR_SELECTION newAllocate;
887     // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
888     BOOL pcrHcrtm = FALSE;
889     BOOL pcrDrtm = FALSE;
890
891     // Create the expected new PCR allocation based on the existing allocation
892     // and the new input:
893     // 1. if a PCR bank does not appear in the new allocation, the existing
894     // allocation of this PCR bank will be preserved.
895     // 2. if a PCR bank appears multiple times in the new allocation, only the
896     // last one will be in effect.
897     newAllocate = gp.pcrAllocated;
898     for(i = 0; i < allocate->count; i++)
899     {
900         for(j = 0; j < newAllocate.count; j++)
901         {
902             // If hash matches, the new allocation covers the old allocation
903             // for this particular bank.
904             // The assumption is the initial PCR allocation (from manufacture)
905             // has all the supported hash algorithms with an assigned bank
906             // (possibly empty). So there must be a match for any new bank
907             // allocation from the input.
908             if(newAllocate.pcrSelections[j].hash == allocate->pcrSelections[i].hash)
909             {
910                 newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
911                 break;
912             }
913         }
914         // The j loop must exit with a match.
915         pAssert(j < newAllocate.count);
916     }
917
918     // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
919     *maxPCR = _platPcr_NumberOfPcrs();
920     if(*maxPCR > IMPLEMENTATION_PCR)
921         *maxPCR = IMPLEMENTATION_PCR;
922 }

```

```

923     // Compute required size for allocation
924     *sizeNeeded = 0;
925     for(i = 0; i < newAllocate.count; i++)
926     {
927         UINT32 digestSize = CryptHashGetDigestSize(newAllocate.pcrSelections[i].hash);
928     #if defined(DRTM_PCR)
929         // Make sure that we end up with at least one DRTM PCR
930         pcrDrtm = pcrDrtm
931             || TestBit(DRTM_PCR,
932                 newAllocate.pcrSelections[i].pcrSelect,
933                 newAllocate.pcrSelections[i].sizeofSelect);
934     #else // if DRTM PCR is not required, indicate that the allocation is OK
935         pcrDrtm = TRUE;
936     #endif
937
938     #if defined(HCRTM_PCR)
939         // and one HCRTM PCR (since this is usually PCR 0...)
940         pcrHcrtm = pcrHcrtm
941             || TestBit(HCRTM_PCR,
942                 newAllocate.pcrSelections[i].pcrSelect,
943                 newAllocate.pcrSelections[i].sizeofSelect);
944     #else
945         pcrHcrtm = TRUE;
946     #endif
947     for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
948     {
949         BYTE mask = 1;
950         for(k = 0; k < 8; k++)
951         {
952             if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
953                 *sizeNeeded += digestSize;
954             mask = mask << 1;
955         }
956     }
957 }
958
959 if(!pcrDrtm || !pcrHcrtm)
960     return TPM_RC_PCR;
961
962 // In this particular implementation, we always have enough space to
963 // allocate PCR. Different implementation may return a sizeAvailable less
964 // than the sizeNeed.
965 *sizeAvailable = sizeof(s_pcrs);
966
967 // Save the required allocation to NV. Note that after NV is written, the
968 // PCR allocation in NV is no longer consistent with the RAM data
969 // gp.pcrAllocated. The NV version reflect the allocate after next
970 // TPM_RESET, while the RAM version reflects the current allocation
971 NV_WRITE_PERSISTENT(pcrAllocated, newAllocate);
972
973 return TPM_RC_SUCCESS;
974 }
975
976
977 /*** PCRSetValue()
978 // This function is used to set the designated PCR in all banks to an initial value.
979 // The initial value is signed and will be sign extended into the entire PCR.
980 //
981 void PCRSetValue(TPM_HANDLE handle,          // IN: the handle of the PCR to set
982                 INT8      initialValue      // IN: the value to set
983 )
984 {
985     int          i;
986     UINT32       pcr = handle - PCR_FIRST;
987     TPMI_ALG_HASH hash;
988     UINT16       digestSize;

```

```

989     BYTE*          pcrData;
990
991     // Iterate supported PCR bank algorithms to reset
992     for(i = 0; i < HASH_COUNT; i++)
993     {
994         hash = CryptHashGetAlgByIndex(i);
995         // Prevent runaway
996         if(hash == TPM_ALG_NULL)
997             break;
998
999         // Get a pointer to the data
1000        pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
1001
1002        // If the PCR is allocated
1003        if(pcrData != NULL)
1004        {
1005            // And the size of the digest
1006            digestSize = CryptHashGetDigestSize(hash);
1007
1008            // Set the LSO to the input value
1009            pcrData[digestSize - 1] = initialValue;
1010
1011            // Sign extend
1012            if(initialValue >= 0)
1013                MemorySet(pcrData, 0, digestSize - 1);
1014            else
1015                MemorySet(pcrData, -1, digestSize - 1);
1016        }
1017    }
1018 }
1019
1020 /*** PCRResetDynamics
1021 // This function is used to reset a dynamic PCR to 0. This function is used in
1022 // DRTM sequence.
1023 void PCRResetDynamics(void)
1024 {
1025     UINT32 pcr, i;
1026
1027     // Initialize PCR values
1028     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
1029     {
1030         // Iterate each hash algorithm bank
1031         for(i = 0; i < gp.pcrAllocated.count; i++)
1032         {
1033             BYTE*          pcrData;
1034             UINT32          pcrSize;
1035             PCR_Attributes currentPcrAttributes =
1036                 _platPcr_GetPcrInitializationAttributes(pcr);
1037
1038             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
1039
1040             if(pcrData != NULL)
1041             {
1042                 pcrSize =
1043                     CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[i].hash);
1044
1045                 // Reset PCR
1046                 // Any PCR can be reset by locality 4 should be reset to 0
1047                 if((currentPcrAttributes.resetLocality & 0x10) != 0)
1048                     MemorySet(pcrData, 0, pcrSize);
1049             }
1050         }
1051     }
1052     return;
1053 }
1054

```

```

1055  /*** PCRGetAllocation()
1056  // This function is used to get the current allocation of PCR banks.
1057  // Return Type: TPMI_YES_NO
1058  //     YES      if the return count is 0
1059  //     NO       if the return count is not 0
1060  TPMI_YES_NO
1061  PCRGetAllocation(UINT32 count, // IN: count of return
1062                  TPML_PCR_SELECTION* pcrSelection // OUT: PCR allocation list
1063  )
1064  {
1065      if(count == 0)
1066      {
1067          pcrSelection->count = 0;
1068          return YES;
1069      }
1070      else
1071      {
1072          *pcrSelection = gp.pcrAllocated;
1073          return NO;
1074      }
1075  }
1076
1077  /*** PCRSetSelectBit()
1078  // This function sets a bit in a bitmap array.
1079  static void PCRSetSelectBit(UINT32 pcr, // IN: PCR number
1080                             BYTE* bitmap // OUT: bit map to be set
1081  )
1082  {
1083      bitmap[pcr / 8] |= (1 << (pcr % 8));
1084      return;
1085  }
1086
1087  /*** PCRGetProperty()
1088  // This function returns the selected PCR property.
1089  // Return Type: BOOL
1090  //     TRUE(1)   the property type is implemented
1091  //     FALSE(0)  the property type is not implemented
1092  BOOL PCRGetProperty(TPM_PT_PCR property, TPMS_TAGGED_PCR_SELECT* select)
1093  {
1094      UINT32 pcr;
1095      UINT32 groupIndex;
1096
1097      select->tag = property;
1098      // Always set the bitmap to be the size of all PCR
1099      select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
1100
1101      // Initialize bitmap
1102      MemorySet(select->pcrSelect, 0, select->sizeofSelect);
1103
1104      // Collecting properties
1105      for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
1106      {
1107          PCR_Attributes currentPcrAttributes =
1108              _platPcr_GetPcrInitializationAttributes(pcr);
1109
1110          switch(property)
1111          {
1112              case TPM_PT_PCR_SAVE:
1113                  if(currentPcrAttributes.stateSave == SET)
1114                      PCRSetSelectBit(pcr, select->pcrSelect);
1115                  break;
1116              case TPM_PT_PCR_EXTEND_L0:
1117                  if((currentPcrAttributes.extendLocality & 0x01) != 0)
1118                      PCRSetSelectBit(pcr, select->pcrSelect);
1119                  break;
1120              case TPM_PT_PCR_RESET_L0:

```



```

1121         if((currentPcrAttributes.resetLocality & 0x01) != 0)
1122             PCRSetSelectBit(pcr, select->pcrSelect);
1123         break;
1124     case TPM_PT_PCR_EXTEND_L1:
1125         if((currentPcrAttributes.extendLocality & 0x02) != 0)
1126             PCRSetSelectBit(pcr, select->pcrSelect);
1127         break;
1128     case TPM_PT_PCR_RESET_L1:
1129         if((currentPcrAttributes.resetLocality & 0x02) != 0)
1130             PCRSetSelectBit(pcr, select->pcrSelect);
1131         break;
1132     case TPM_PT_PCR_EXTEND_L2:
1133         if((currentPcrAttributes.extendLocality & 0x04) != 0)
1134             PCRSetSelectBit(pcr, select->pcrSelect);
1135         break;
1136     case TPM_PT_PCR_RESET_L2:
1137         if((currentPcrAttributes.resetLocality & 0x04) != 0)
1138             PCRSetSelectBit(pcr, select->pcrSelect);
1139         break;
1140     case TPM_PT_PCR_EXTEND_L3:
1141         if((currentPcrAttributes.extendLocality & 0x08) != 0)
1142             PCRSetSelectBit(pcr, select->pcrSelect);
1143         break;
1144     case TPM_PT_PCR_RESET_L3:
1145         if((currentPcrAttributes.resetLocality & 0x08) != 0)
1146             PCRSetSelectBit(pcr, select->pcrSelect);
1147         break;
1148     case TPM_PT_PCR_EXTEND_L4:
1149         if((currentPcrAttributes.extendLocality & 0x10) != 0)
1150             PCRSetSelectBit(pcr, select->pcrSelect);
1151         break;
1152     case TPM_PT_PCR_RESET_L4:
1153         if((currentPcrAttributes.resetLocality & 0x10) != 0)
1154             PCRSetSelectBit(pcr, select->pcrSelect);
1155         break;
1156     case TPM_PT_PCR_DRTM_RESET:
1157         // DRTM reset PCRs are the PCR reset by locality 4
1158         if((currentPcrAttributes.resetLocality & 0x10) != 0)
1159             PCRSetSelectBit(pcr, select->pcrSelect);
1160         break;
1161     #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
1162     case TPM_PT_PCR_POLICY:
1163         if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
1164             PCRSetSelectBit(pcr, select->pcrSelect);
1165         break;
1166     #endif
1167     #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
1168     case TPM_PT_PCR_AUTH:
1169         if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
1170             PCRSetSelectBit(pcr, select->pcrSelect);
1171         break;
1172     #endif
1173     #if ENABLE_PCR_NO_INCREMENT == YES
1174     case TPM_PT_PCR_NO_INCREMENT:
1175         if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
1176             PCRSetSelectBit(pcr, select->pcrSelect);
1177         break;
1178     #endif
1179     default:
1180         // If property is not supported, stop scanning PCR attributes
1181         // and return.
1182         return FALSE;
1183         break;
1184     }
1185 }
1186 return TRUE;

```

```

1187 }
1188
1189 /*** PCRCapGetProperties()
1190 // This function returns a list of PCR properties starting at 'property'.
1191 // Return Type: TPMI_YES_NO
1192 //     YES      if no more property is available
1193 //     NO       if there are more properties not reported
1194 TPMI_YES_NO
1195 PCRCapGetProperties(TPM_PT_PCR property, // IN: the starting PCR property
1196                   UINT32 count,        // IN: count of returned properties
1197                   TPML_TAGGED_PCR_PROPERTY* select // OUT: PCR select
1198 )
1199 {
1200     TPMI_YES_NO more = NO;
1201     UINT32 i;
1202
1203     // Initialize output property list
1204     select->count = 0;
1205
1206     // The maximum count of properties we may return is MAX_PCR_PROPERTIES
1207     if(count > MAX_PCR_PROPERTIES)
1208         count = MAX_PCR_PROPERTIES;
1209
1210     // TPM_PT_PCR_FIRST is defined as 0 in spec. It ensures that property
1211     // value would never be less than TPM_PT_PCR_FIRST
1212     MUST_BE(TPM_PT_PCR_FIRST == 0);
1213
1214     // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
1215     // implemented on the TPM.
1216     for(i = property; i <= TPM_PT_PCR_LAST; i++)
1217     {
1218         if(select->count < count)
1219         {
1220             // If we have not filled up the return list, add more properties to it
1221             if(PCRCapGetProperty(i, &select->pcrProperty[select->count]))
1222                 // only increment if the property is implemented
1223                 select->count++;
1224         }
1225         else
1226         {
1227             // If the return list is full but we still have properties
1228             // available, report this and stop iterating.
1229             more = YES;
1230             break;
1231         }
1232     }
1233     return more;
1234 }
1235
1236 /*** PCRCapGetHandles()
1237 // This function is used to get a list of handles of PCR, started from 'handle'.
1238 // If 'handle' exceeds the maximum PCR handle range, an empty list will be
1239 // returned and the return value will be NO.
1240 // Return Type: TPMI_YES_NO
1241 //     YES      if there are more handles available
1242 //     NO       all the available handles has been returned
1243 TPMI_YES_NO
1244 PCRCapGetHandles(TPMI_DH_PCR handle, // IN: start handle
1245                 UINT32 count,        // IN: count of returned handles
1246                 TPML_HANDLE* handleList // OUT: list of handle
1247 )
1248 {
1249     TPMI_YES_NO more = NO;
1250     UINT32 i;
1251
1252     pAssert(HandleGetType(handle) == TPM_HT_PCR);

```

```

1253
1254 // Initialize output handle list
1255 handleList->count = 0;
1256
1257 // The maximum count of handles we may return is MAX_CAP_HANDLES
1258 if(count > MAX_CAP_HANDLES)
1259     count = MAX_CAP_HANDLES;
1260
1261 // Iterate PCR handle range
1262 for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1263 {
1264     if(handleList->count < count)
1265     {
1266         // If we have not filled up the return list, add this PCR
1267         // handle to it
1268         handleList->handle[handleList->count] = i + PCR_FIRST;
1269         handleList->count++;
1270     }
1271     else
1272     {
1273         // If the return list is full but we still have PCR handle
1274         // available, report this and stop iterating
1275         more = YES;
1276         break;
1277     }
1278 }
1279 return more;
1280 }
1281
1282 /*** PCRCapGetOneHandle()
1283 // This function is used to check whether a PCR handle exists.
1284 BOOL PCRCapGetOneHandle(TPMI_DH_PCR handle) // IN: handle
1285 {
1286     pAssert(HandleGetType(handle) == TPM_HT_PCR);
1287
1288     if((handle & HR_HANDLE_MASK) <= PCR_LAST)
1289     {
1290         return TRUE;
1291     }
1292     return FALSE;
1293 }

```

7.177 /tpm/src/subsystem/PP.c

```

1  /*** Introduction
2  // This file contains the functions that support the physical presence operations
3  // of the TPM.
4
5  /*** Includes
6
7  #include "Tpm.h"
8
9  /*** Functions
10
11  /*** PhysicalPresencePreInstall_Init()
12  // This function is used to initialize the array of commands that always require
13  // confirmation with physical presence. The array is an array of bits that
14  // has a correspondence with the command code.
15  //
16  // This command should only ever be executable in a manufacturing setting or in
17  // a simulation.
18  //
19  // When set, these cannot be cleared.
20  //
21  void PhysicalPresencePreInstall_Init(void)

```

```

22 {
23     COMMAND_INDEX commandIndex;
24     // Clear all the PP commands
25     MemorySet(&gp.ppList, 0, sizeof(gp.ppList));
26
27     // Any command that is PP_REQUIRED should be SET
28     for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
29     {
30         if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED
31            && s_commandAttributes[commandIndex] & PP_REQUIRED)
32             SET_BIT(commandIndex, gp.ppList);
33     }
34     // Write PP list to NV
35     NV_SYNC_PERSISTENT(ppList);
36     return;
37 }
38
39 /*** PhysicalPresenceCommandSet()
40 // This function is used to set the indicator that a command requires
41 // PP confirmation.
42 void PhysicalPresenceCommandSet(TPM_CC commandCode // IN: command code
43 )
44 {
45     COMMAND_INDEX commandIndex = CommandCodeToCommandIndex(commandCode);
46
47     // if the command isn't implemented, the do nothing
48     if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
49         return;
50
51     // only set the bit if this is a command for which PP is allowed
52     if(s_commandAttributes[commandIndex] & PP_COMMAND)
53         SET_BIT(commandIndex, gp.ppList);
54     return;
55 }
56
57 /*** PhysicalPresenceCommandClear()
58 // This function is used to clear the indicator that a command requires PP
59 // confirmation.
60 void PhysicalPresenceCommandClear(TPM_CC commandCode // IN: command code
61 )
62 {
63     COMMAND_INDEX commandIndex = CommandCodeToCommandIndex(commandCode);
64
65     // If the command isn't implemented, then don't do anything
66     if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
67         return;
68
69     // Only clear the bit if the command does not require PP
70     if((s_commandAttributes[commandIndex] & PP_REQUIRED) == 0)
71         CLEAR_BIT(commandIndex, gp.ppList);
72
73     return;
74 }
75
76 /*** PhysicalPresenceIsRequired()
77 // This function indicates if PP confirmation is required for a command.
78 // Return Type: BOOL
79 //     TRUE(1)         physical presence is required
80 //     FALSE(0)        physical presence is not required
81 BOOL PhysicalPresenceIsRequired(COMMAND_INDEX commandIndex // IN: command index
82 )
83 {
84     // Check the bit map. If the bit is SET, PP authorization is required
85     return (TEST_BIT(commandIndex, gp.ppList));
86 }
87

```

```

88  /*** PhysicalPresenceCapGetCCList()
89  // This function returns a list of commands that require PP confirmation. The
90  // list starts from the first implemented command that has a command code that
91  // the same or greater than 'commandCode'.
92  // Return Type: TPMI_YES_NO
93  //     YES      if there are more command codes available
94  //     NO       all the available command codes have been returned
95  TPMI_YES_NO
96  PhysicalPresenceCapGetCCList(TPM_CC  commandCode, // IN: start command code
97                             UINT32  count,       // IN: count of returned TPM_CC
98                             TPML_CC* commandList // OUT: list of TPM_CC
99  )
100 {
101     TPMI_YES_NO  more = NO;
102     COMMAND_INDEX  commandIndex;
103
104     // Initialize output handle list
105     commandList->count = 0;
106
107     // The maximum count of command we may return is MAX_CAP_CC
108     if(count > MAX_CAP_CC)
109         count = MAX_CAP_CC;
110
111     // Collect PP commands
112     for(commandIndex = GetClosestCommandIndex(commandCode);
113         commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
114         commandIndex = GetNextCommandIndex(commandIndex))
115     {
116         if(PhysicalPresenceIsRequired(commandIndex))
117         {
118             if(commandList->count < count)
119             {
120                 // If we have not filled up the return list, add this command
121                 // code to it
122                 commandList->commandCodes[commandList->count] =
123                     GetCommandCode(commandIndex);
124                 commandList->count++;
125             }
126             else
127             {
128                 // If the return list is full but we still have PP command
129                 // available, report this and stop iterating
130                 more = YES;
131                 break;
132             }
133         }
134     }
135     return more;
136 }
137
138 /*** PhysicalPresenceCapGetOneCC()
139 // This function returns true if the command requires Physical Presence.
140 BOOL PhysicalPresenceCapGetOneCC(TPM_CC commandCode) // IN: command code
141 {
142     COMMAND_INDEX  commandIndex = CommandCodeToCommandIndex(commandCode);
143     if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
144     {
145         return PhysicalPresenceIsRequired(commandIndex);
146     }
147     return FALSE;
148 }

```

7.178 /tpm/src/subsystem/Session.c

```
1  /***Introduction
```

```
2  /*
3   The code in this file is used to manage the session context counter.
4   The scheme implemented here is a "truncated counter".
5   This scheme allows the TPM to not need TPM_SU_CLEAR for a
6   very long period of time and still not have the context
7   count for a session repeated.
8
9   The counter (contextCounter) in this implementation is a UINT64 but
10  can be smaller. The "tracking array" (contextArray) only
11  has 16-bits per context. The tracking array is the data
12  that needs to be saved and restored across TPM_SU_STATE so that
13  sessions are not lost when the system enters the sleep state.
14  Also, when the TPM is active, the tracking array is kept in
15  RAM making it important that the number of bytes for each
16  entry be kept as small as possible.
17
18  The TPM prevents "collisions" of these truncated values by
19  not allowing a contextID to be assigned if it would be the
20  same as an existing value. Since the array holds 16 bits,
21  after a context has been saved, an additional 2^16-1 contexts
22  may be saved before the count would again match. The normal
23  expectation is that the context will be flushed before its count
24  value is needed again but it is always possible to have long-lived
25  sessions.
26
27  The contextID is assigned when the context is saved (TPM2_ContextSave()).
28  At that time, the TPM will compare the low-order 16 bits of
29  contextCounter to the existing values in contextArray and if one
30  matches, the TPM will return TPM_RC_CONTEXT_GAP (by construction,
31  the entry that contains the matching value is the oldest
32  context).
33
34  The expected remediation by the TRM is to load the oldest saved
35  session context (the one found by the TPM), and save it. Since loading
36  the oldest session also eliminates its contextID value from
37  contextArray, there TPM will always be able to load and save the oldest
38  existing context.
39
40  In the worst case, software may have to load and save several contexts
41  in order to save an additional one. This should happen very infrequently.
42
43  When the TPM searches contextArray and finds that none of the contextIDs
44  match the low-order 16-bits of contextCount, the TPM can copy the low bits
45  to the contextArray associated with the session, and increment contextCount.
46
47  There is one entry in contextArray for each of the active sessions
48  allowed by the TPM implementation. This array contains either a
49  context count, an index, or a value indicating the slot is available (0).
50
51  The index into the contextArray is the handle for the session with the region
52  selector byte of the session set to zero. If an entry in contextArray contains
53  0, then the corresponding handle may be assigned to a session. If the entry
54  contains a value that is less than or equal to the number of loaded sessions
55  for the TPM, then the array entry is the slot in which the context is loaded.
56
57  EXAMPLE: If the TPM allows 8 loaded sessions, then the slot numbers would
58  be 1-8 and a contextArray value in that range would represent the loaded
59  session.
60
61  NOTE: When the TPM firmware determines that the array entry is for a loaded
62  session, it will subtract 1 to create the zero-based slot number.
63
64  There is one significant corner case in this scheme. When the contextCount
65  is equal to a value in the contextArray, the oldest session needs to be
66  recycled or flushed. In order to recycle the session, it must be loaded.
67  To be loaded, there must be an available slot. Rather than require that a
```



```

68     spare slot be available all the time, the TPM will check to see if the
69     contextCount is equal to some value in the contextArray when a session is
70     created. This prevents the last session slot from being used when it
71     is likely that a session will need to be recycled.
72
73     If a TPM with both 1.2 and 2.0 functionality uses this scheme for both
74     1.2 and 2.0 sessions, and the list of active contexts is read with
75     TPM_GetCapabiltiy(), the TPM will create 32-bit representations of the
76     list that contains 16-bit values (the TPM2_GetCapability() returns a list
77     of handles for active sessions rather than a list of contextID). The full
78     contextID has high-order bits that are either the same as the current
79     contextCount or one less. It is one less if the 16-bits
80     of the contextArray has a value that is larger than the low-order 16 bits
81     of contextCount.
82 */
83
84 /** Includes, Defines, and Local Variables
85 #define SESSION_C
86 #include "Tpm.h"
87
88 /** File Scope Function -- ContextIdSetOldest()
89 /*
90     This function is called when the oldest contextID is being loaded or deleted.
91     Once a saved context becomes the oldest, it stays the oldest until it is
92     deleted.
93
94     Finding the oldest is a bit tricky. It is not just the numeric comparison of
95     values but is dependent on the value of contextCounter.
96
97     Assume we have a small contextArray with 8, 4-bit values with values 1 and 2
98     used to indicate the loaded context slot number. Also assume that the array
99     contains hex values of (0 0 1 0 3 0 9 F) and that the contextCounter is an
100    8-bit counter with a value of 0x37. Since the low nibble is 7, that means
101    that values above 7 are older than values below it and, in this example,
102    9 is the oldest value.
103
104    Note if we subtract the counter value, from each slot that contains a saved
105    contextID we get (- - - - B - 2 - 8) and the oldest entry is now easy to find.
106 */
107 static void ContextIdSetOldest(void)
108 {
109     CONTEXT_SLOT lowBits;
110     CONTEXT_SLOT entry;
111     CONTEXT_SLOT smallest = ((CONTEXT_SLOT)~0);
112     UINT32 i;
113
114     // Set oldestSaveContext to a value indicating none assigned
115     s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
116
117     lowBits = (CONTEXT_SLOT)gr.contextCounter;
118     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
119     {
120         entry = gr.contextArray[i];
121
122         // only look at entries that are saved contexts
123         if(entry > MAX_LOADED_SESSIONS)
124         {
125             // Use a less than or equal in case the oldest
126             // is brand new (= lowBits-1) and equal to our initial
127             // value for smallest.
128             if(((CONTEXT_SLOT)(entry - lowBits)) <= smallest)
129             {
130                 smallest = (entry - lowBits);
131                 s_oldestSavedSession = i;
132             }
133         }
134     }

```

```

134     }
135     // When we finish, either the s_oldestSavedSession still has its initial
136     // value, or it has the index of the oldest saved context.
137 }
138
139 /** Startup Function -- SessionStartup()
140 // This function initializes the session subsystem on TPM2_Startup().
141 BOOL SessionStartup(STARTUP_TYPE type)
142 {
143     UINT32 i;
144
145     // Initialize session slots. At startup, all the in-memory session slots
146     // are cleared and marked as not occupied
147     for(i = 0; i < MAX_LOADED_SESSIONS; i++)
148         s_sessions[i].occupied = FALSE; // session slot is not occupied
149
150     // The free session slots the number of maximum allowed loaded sessions
151     s_freeSessionSlots = MAX_LOADED_SESSIONS;
152
153     // Initialize context ID data. On a ST_SAVE or hibernate sequence, it will
154     // scan the saved array of session context counts, and clear any entry that
155     // references a session that was in memory during the state save since that
156     // memory was not preserved over the ST_SAVE.
157     if(type == SU_RESUME || type == SU_RESTART)
158     {
159         // On ST_SAVE we preserve the contexts that were saved but not the ones
160         // in memory
161         for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
162         {
163             // If the array value is unused or references a loaded session then
164             // that loaded session context is lost and the array entry is
165             // reclaimed.
166             if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
167                 gr.contextArray[i] = 0;
168         }
169         // Find the oldest session in context ID data and set it in
170         // s_oldestSavedSession
171         ContextIdSetOldest();
172     }
173     else
174     {
175         // For STARTUP_CLEAR, clear out the contextArray
176         for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
177             gr.contextArray[i] = 0;
178
179         // reset the context counter
180         gr.contextCounter = MAX_LOADED_SESSIONS + 1;
181
182         // Initialize oldest saved session
183         s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
184     }
185     return TRUE;
186 }
187
188 /*******
189 /** Access Functions
190 /*******
191
192 /** SessionIsLoaded()
193 // This function test a session handle references a loaded session. The handle
194 // must have previously been checked to make sure that it is a valid handle for
195 // an authorization session.
196 // NOTE: A PWAP authorization does not have a session.
197 //
198 // Return Type: BOOL
199 // TRUE(1) session is loaded

```

```

200 // FALSE(0) session is not loaded
201 //
202 BOOL SessionIsLoaded(TPM_HANDLE handle // IN: session handle
203 )
204 {
205     pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
206             || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
207
208     handle = handle & HR_HANDLE_MASK;
209
210     // if out of range of possible active session, or not assigned to a loaded
211     // session return false
212     if(handle >= MAX_ACTIVE_SESSIONS || gr.contextArray[handle] == 0
213        || gr.contextArray[handle] > MAX_LOADED_SESSIONS)
214         return FALSE;
215
216     return TRUE;
217 }
218
219 /*** SessionIsSaved()
220 // This function test a session handle references a saved session. The handle
221 // must have previously been checked to make sure that it is a valid handle for
222 // an authorization session.
223 // NOTE: An password authorization does not have a session.
224 //
225 // This function requires that the handle be a valid session handle.
226 //
227 // Return Type: BOOL
228 // TRUE(1) session is saved
229 // FALSE(0) session is not saved
230 //
231 BOOL SessionIsSaved(TPM_HANDLE handle // IN: session handle
232 )
233 {
234     pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
235             || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
236
237     handle = handle & HR_HANDLE_MASK;
238     // if out of range of possible active session, or not assigned, or
239     // assigned to a loaded session, return false
240     if(handle >= MAX_ACTIVE_SESSIONS || gr.contextArray[handle] == 0
241        || gr.contextArray[handle] <= MAX_LOADED_SESSIONS)
242         return FALSE;
243
244     return TRUE;
245 }
246
247 /*** SequenceNumberForSavedContextIsValid()
248 // This function validates that the sequence number and handle value within a
249 // saved context are valid.
250 BOOL SequenceNumberForSavedContextIsValid(
251     TPMS_CONTEXT* context // IN: pointer to a context structure to be
252                          // validated
253 )
254 {
255     #define MAX_CONTEXT_GAP ((UINT64)((CONTEXT_SLOT)~0) + 1)
256
257     TPM_HANDLE handle = context->savedHandle & HR_HANDLE_MASK;
258
259     if( // Handle must be with the range of active sessions
260        handle >= MAX_ACTIVE_SESSIONS
261        // the array entry must be for a saved context
262        || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
263        // the array entry must agree with the sequence number
264        || gr.contextArray[handle] != (CONTEXT_SLOT)context->sequence
265        // the provided sequence number has to be less than the current counter

```

```

266     || context->sequence > gr.contextCounter
267     // but not so much that it could not be a valid sequence number
268     || gr.contextCounter - context->sequence > MAX_CONTEXT_GAP)
269     return FALSE;
270
271     return TRUE;
272 }
273
274 /** SessionPCRValueIsCurrent()
275 //
276 // This function is used to check if PCR values have been updated since the
277 // last time they were checked in a policy session.
278 //
279 // This function requires the session is loaded.
280 // Return Type: BOOL
281 //     TRUE(1)         PCR value is current
282 //     FALSE(0)        PCR value is not current
283 BOOL SessionPCRValueIsCurrent(SESSION* session // IN: session structure
284 )
285 {
286     if(session->pcrCounter != 0 && session->pcrCounter != gr.pcrCounter)
287         return FALSE;
288     else
289         return TRUE;
290 }
291
292 /** SessionGet()
293 // This function returns a pointer to the session object associated with a
294 // session handle.
295 //
296 // The function requires that the session is loaded.
297 SESSION* SessionGet(TPM_HANDLE handle // IN: session handle
298 )
299 {
300     size_t    slotIndex;
301     CONTEXT_SLOT sessionIndex;
302
303     pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
304             || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
305
306     slotIndex = handle & HR_HANDLE_MASK;
307
308     pAssert(slotIndex < MAX_ACTIVE_SESSIONS);
309
310     // get the contents of the session array. Because session is loaded, we
311     // should always get a valid sessionIndex
312     sessionIndex = gr.contextArray[slotIndex] - 1;
313
314     pAssert(sessionIndex < MAX_LOADED_SESSIONS);
315
316     return &s_sessions[sessionIndex].session;
317 }
318
319 /** *****
320 /** Utility Functions
321 /** *****
322
323 /** ContextIdSessionCreate()
324 //
325 // This function is called when a session is created. It will check
326 // to see if the current gap would prevent a context from being saved. If
327 // so it will return TPM_RC_CONTEXT_GAP. Otherwise, it will try to find
328 // an open slot in contextArray, set contextArray to the slot.
329 //
330 // This routine requires that the caller has determined the session array
331 // index for the session.

```

```

332 //
333 // Return Type: TPM_RC
334 //     TPM_RC_CONTEXT_GAP        can't assign a new contextID until the oldest
335 //                               saved session context is recycled
336 //     TPM_RC_SESSION_HANDLE     there is no slot available in the context array
337 //                               for tracking of this session context
338 static TPM_RC ContextIdSessionCreate(
339     TPM_HANDLE* handle, // OUT: receives the assigned handle. This will
340                        // be an index that must be adjusted by the
341                        // caller according to the type of the
342                        // session created
343     UINT32 sessionIndex // IN: The session context array entry that will
344                        // be occupied by the created session
345 )
346 {
347     pAssert(sessionIndex < MAX_LOADED_SESSIONS);
348
349     // check to see if creating the context is safe
350     // Is this going to be an assignment for the last session context
351     // array entry? If so, then there will be no room to recycle the
352     // oldest context if needed. If the gap is not at maximum, then
353     // it will be possible to save a context if it becomes necessary.
354     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS && s_freeSessionSlots == 1)
355     {
356         // See if the gap is at maximum
357         // The current value of the contextCounter will be assigned to the next
358         // saved context. If the value to be assigned would make the same as an
359         // existing context, then we can't use it because of the ambiguity it would
360         // create.
361         if((CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession])
362             return TPM_RC_CONTEXT_GAP;
363     }
364
365     // Find an unoccupied entry in the contextArray
366     for(*handle = 0; *handle < MAX_ACTIVE_SESSIONS; (*handle)++)
367     {
368         if(gr.contextArray[*handle] == 0)
369         {
370             // indicate that the session associated with this handle
371             // references a loaded session
372             gr.contextArray[*handle] = (CONTEXT_SLOT)(sessionIndex + 1);
373             return TPM_RC_SUCCESS;
374         }
375     }
376     return TPM_RC_SESSION_HANDLES;
377 }
378
379 /*** SessionCreate()
380 //
381 // This function does the detailed work for starting an authorization session.
382 // This is done in a support routine rather than in the action code because
383 // the session management may differ in implementations. This implementation
384 // uses a fixed memory allocation to hold sessions and a fixed allocation
385 // to hold the contextID for the saved contexts.
386 //
387 // Return Type: TPM_RC
388 //     TPM_RC_CONTEXT_GAP        need to recycle sessions
389 //     TPM_RC_SESSION_HANDLE     active session space is full
390 //     TPM_RC_SESSION_MEMORY     loaded session space is full
391 TPM_RC
392 SessionCreate(TPM_SE sessionType, // IN: the session type
393               TPMI_ALG_HASH authHash, // IN: the hash algorithm
394               TPM2B_NONCE* nonceCaller, // IN: initial nonceCaller
395               TPMT_SYM_DEF* symmetric, // IN: the symmetric algorithm
396               TPMI_DH_ENTITY bind, // IN: the bind object
397               TPM2B_DATA* seed, // IN: seed data

```

```

398         TPM_HANDLE*    sessionHandle, // OUT: the session handle
399         TPM2B_NONCE*    nonceTpm      // OUT: the session nonce
400     )
401     {
402         TPM_RC    result = TPM_RC_SUCCESS;
403         CONTEXT_SLOT slotIndex;
404         SESSION*  session = NULL;
405
406         pAssert(sessionType == TPM_SE_HMAC || sessionType == TPM_SE_POLICY
407             || sessionType == TPM_SE_TRIAL);
408
409         // If there are no open spots in the session array, then no point in searching
410         if(s_freeSessionSlots == 0)
411             return TPM_RC_SESSION_MEMORY;
412
413         // Find a space for loading a session
414         for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
415         {
416             // Is this available?
417             if(s_sessions[slotIndex].occupied == FALSE)
418             {
419                 session = &s_sessions[slotIndex].session;
420                 break;
421             }
422         }
423         // if no spot found, then this is an internal error
424         if(slotIndex >= MAX_LOADED_SESSIONS)
425             FAIL(FATAL_ERROR_INTERNAL);
426
427         // Call context ID function to get a handle.  TPM_RC_SESSION_HANDLE may be
428         // returned from ContextIdHandleAssign()
429         result = ContextIdSessionCreate(sessionHandle, slotIndex);
430         if(result != TPM_RC_SUCCESS)
431             return result;
432
433         /*** Only return from this point on is TPM_RC_SUCCESS
434
435         // Can now indicate that the session array entry is occupied.
436         s_freeSessionSlots--;
437         s_sessions[slotIndex].occupied = TRUE;
438
439         // Initialize the session data
440         MemorySet(session, 0, sizeof(SESSION));
441
442         // Initialize internal session data
443         session->authHashAlg = authHash;
444         // Initialize session type
445         if(sessionType == TPM_SE_HMAC)
446         {
447             *sessionHandle += HMAC_SESSION_FIRST;
448         }
449         else
450         {
451             *sessionHandle += POLICY_SESSION_FIRST;
452
453             // For TPM_SE_POLICY or TPM_SE_TRIAL
454             session->attributes.isPolicy = SET;
455             if(sessionType == TPM_SE_TRIAL)
456                 session->attributes.isTrialPolicy = SET;
457
458             SessionSetStartTime(session);
459
460             // Initialize policyDigest.  policyDigest is initialized with a string of 0
461             // of session algorithm digest size. Since the session is already clear.
462             // Just need to set the size
463             session->u2.policyDigest.t.size =

```



```

464         CryptHashGetDigestSize(session->authHashAlg);
465     }
466     // Create initial session nonce
467     session->nonceTPM.t.size = nonceCaller->t.size;
468     CryptRandomGenerate(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
469     MemoryCopy2B(&nonceTpm->b, &session->nonceTPM.b, sizeof(nonceTpm->t.buffer));
470
471     // Set up session parameter encryption algorithm
472     session->symmetric = *symmetric;
473
474     // If there is a bind object or a session secret, then need to compute
475     // a sessionKey.
476     if(bind != TPM_RH_NULL || seed->t.size != 0)
477     {
478         // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
479         //                      nonceCaller, bits)
480         // The HMAC key for generating the sessionSecret can be the concatenation
481         // of an authorization value and a seed value
482         TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
483         TPM2B_KEY key;
484
485         // Get hash size, which is also the length of sessionKey
486         session->sessionKey.t.size = CryptHashGetDigestSize(session->authHashAlg);
487
488         // Get authValue of associated entity
489         EntityGetAuthValue(bind, (TPM2B_AUTH*)&key);
490         pAssert(key.t.size + seed->t.size <= sizeof(key.t.buffer));
491
492         // Concatenate authValue and seed
493         MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));
494
495         // Compute the session key
496         CryptKDFa(session->authHashAlg,
497                 &key.b,
498                 SESSION_KEY,
499                 &session->nonceTPM.b,
500                 &nonceCaller->b,
501                 session->sessionKey.t.size * 8,
502                 session->sessionKey.t.buffer,
503                 NULL,
504                 FALSE);
505     }
506
507     // Copy the name of the entity that the HMAC session is bound to
508     // Policy session is not bound to an entity
509     if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
510     {
511         session->attributes.isBound = SET;
512         SessionComputeBoundEntity(bind, &session->ul.boundEntity);
513     }
514     // If there is a bind object and it is subject to DA, then use of this session
515     // is subject to DA regardless of how it is used.
516     session->attributes.isDaBound = (bind != TPM_RH_NULL)
517                                     && (IsDAExempted(bind) == FALSE);
518
519     // If the session is bound, then check to see if it is bound to lockoutAuth
520     session->attributes.isLockoutBound = (session->attributes.isDaBound == SET)
521                                         && (bind == TPM_RH_LOCKOUT);
522     return TPM_RC_SUCCESS;
523 }
524
525 /*** SessionContextSave()
526 // This function is called when a session context is to be saved. The
527 // contextID of the saved session is returned. If no contextID can be
528 // assigned, then the routine returns TPM_RC_CONTEXT_GAP.
529 // If the function completes normally, the session slot will be freed.

```

```

530 //
531 // This function requires that 'handle' references a loaded session.
532 // Otherwise, it should not be called at the first place.
533 //
534 // Return Type: TPM_RC
535 //     TPM_RC_CONTEXT_GAP          a contextID could not be assigned
536 //     TPM_RC_TOO_MANY_CONTEXTS   the counter maxed out
537 //
538 TPM_RC
539 SessionContextSave(TPM_HANDLE handle, // IN: session handle
540                   CONTEXT_COUNTER* contextID // OUT: assigned contextID
541 )
542 {
543     UINT32 contextIndex;
544     CONTEXT_SLOT slotIndex;
545
546     pAssert(SessionIsLoaded(handle));
547
548     // check to see if the gap is already maxed out
549     // Need to have a saved session
550     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
551        // if the oldest saved session has the same value as the low bits
552        // of the contextCounter, then the GAP is maxed out.
553        && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
554         return TPM_RC_CONTEXT_GAP;
555
556     // if the caller wants the context counter, set it
557     if(contextID != NULL)
558         *contextID = gr.contextCounter;
559
560     contextIndex = handle & HR_HANDLE_MASK;
561     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
562
563     // Extract the session slot number referenced by the contextArray
564     // because we are going to overwrite this with the low order
565     // contextID value.
566     slotIndex = gr.contextArray[contextIndex] - 1;
567
568     // Set the contextID for the contextArray
569     gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
570
571     // Increment the counter
572     gr.contextCounter++;
573
574     // In the unlikely event that the 64-bit context counter rolls over...
575     if(gr.contextCounter == 0)
576     {
577         // back it up
578         gr.contextCounter--;
579         // return an error
580         return TPM_RC_TOO_MANY_CONTEXTS;
581     }
582     // if the low-order bits wrapped, need to advance the value to skip over
583     // the values used to indicate that a session is loaded
584     if(((CONTEXT_SLOT)gr.contextCounter) == 0)
585         gr.contextCounter += MAX_LOADED_SESSIONS + 1;
586
587     // If no other sessions are saved, this is now the oldest.
588     if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
589         s_oldestSavedSession = contextIndex;
590
591     // Mark the session slot as unoccupied
592     s_sessions[slotIndex].occupied = FALSE;
593
594     // and indicate that there is an additional open slot
595     s_freeSessionSlots++;

```

```

596
597     return TPM_RC_SUCCESS;
598 }
599
600 /*** SessionContextLoad()
601 // This function is used to load a session from saved context. The session
602 // handle must be for a saved context.
603 //
604 // If the gap is at a maximum, then the only session that can be loaded is
605 // the oldest session, otherwise TPM_RC_CONTEXT_GAP is returned.
606 //
607 // This function requires that 'handle' references a valid saved session.
608 //
609 // Return Type: TPM_RC
610 //     TPM_RC_SESSION_MEMORY    no free session slots
611 //     TPM_RC_CONTEXT_GAP       the gap count is maximum and this
612 //                               is not the oldest saved context
613 //
614 TPM_RC
615 SessionContextLoad(SESSION_BUF* session, // IN: session structure from saved context
616                   TPM_HANDLE* handle    // IN/OUT: session handle
617 )
618 {
619     UINT32    contextIndex;
620     CONTEXT_SLOT slotIndex;
621
622     pAssert(HandleGetType(*handle) == TPM_HT_POLICY_SESSION
623             || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);
624
625     // Don't bother looking if no openings
626     if(s_freeSessionSlots == 0)
627         return TPM_RC_SESSION_MEMORY;
628
629     // Find a free session slot to load the session
630     for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
631         if(s_sessions[slotIndex].occupied == FALSE)
632             break;
633
634     // if no spot found, then this is an internal error
635     pAssert(slotIndex < MAX_LOADED_SESSIONS);
636
637     contextIndex = *handle & HR_HANDLE_MASK; // extract the index
638
639     // If there is only one slot left, and the gap is at maximum, the only session
640     // context that we can safely load is the oldest one.
641     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS && s_freeSessionSlots == 1
642        && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
643        && contextIndex != s_oldestSavedSession)
644         return TPM_RC_CONTEXT_GAP;
645
646     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
647
648     // set the contextArray value to point to the session slot where
649     // the context is loaded
650     gr.contextArray[contextIndex] = slotIndex + 1;
651
652     // if this was the oldest context, find the new oldest
653     if(contextIndex == s_oldestSavedSession)
654         ContextIdSetOldest();
655
656     // Copy session data to session slot
657     MemoryCopy(&s_sessions[slotIndex].session, session, sizeof(SESSION));
658
659     // Set session slot as occupied
660     s_sessions[slotIndex].occupied = TRUE;
661

```

```

662     // Reduce the number of open spots
663     s_freeSessionSlots--;
664
665     return TPM_RC_SUCCESS;
666 }
667
668 /*** SessionFlush()
669 // This function is used to flush a session referenced by its handle. If the
670 // session associated with 'handle' is loaded, the session array entry is
671 // marked as available.
672 //
673 // This function requires that 'handle' be a valid active session.
674 //
675 void SessionFlush(TPM_HANDLE handle // IN: loaded or saved session handle
676 )
677 {
678     CONTEXT_SLOT slotIndex;
679     UINT32 contextIndex; // Index into contextArray
680
681     pAssert((HandleGetType(handle) == TPM_HT_POLICY_SESSION
682             || HandleGetType(handle) == TPM_HT_HMAC_SESSION)
683             && (SessionIsLoaded(handle) || SessionIsSaved(handle)));
684
685     // Flush context ID of this session
686     // Convert handle to an index into the contextArray
687     contextIndex = handle & HR_HANDLE_MASK;
688
689     pAssert(contextIndex < sizeof(gr.contextArray) / sizeof(gr.contextArray[0]));
690
691     // Get the current contents of the array
692     slotIndex = gr.contextArray[contextIndex];
693
694     // Mark context array entry as available
695     gr.contextArray[contextIndex] = 0;
696
697     // Is this a saved session being flushed
698     if(slotIndex > MAX_LOADED_SESSIONS)
699     {
700         // Flushing the oldest session?
701         if(contextIndex == s_oldestSavedSession)
702             // If so, find a new value for oldest.
703             ContextIdSetOldest();
704     }
705     else
706     {
707         // Adjust slot index to point to session array index
708         slotIndex -= 1;
709
710         // Free session array index
711         s_sessions[slotIndex].occupied = FALSE;
712         s_freeSessionSlots++;
713     }
714
715     return;
716 }
717
718 /*** SessionComputeBoundEntity()
719 // This function computes the binding value for a session. The binding value
720 // for a reserved handle is the handle itself. For all the other entities,
721 // the authValue at the time of binding is included to prevent squatting.
722 // For those values, the Name and the authValue are concatenated
723 // into the bind buffer. If they will not both fit, the will be overlapped
724 // by XORing bytes. If XOR is required, the bind value will be full.
725 void SessionComputeBoundEntity(TPMI_DH_ENTITY entityHandle, // IN: handle of entity
726                               TPM2B_NAME* bind // OUT: binding value
727 )

```

```

728 {
729     TPM2B_AUTH auth;
730     BYTE*      pAuth = auth.t.buffer;
731     UINT16      i;
732
733     // Get name
734     EntityGetName(entityHandle, bind);
735
736     // // The bound value of a reserved handle is the handle itself
737     // if(bind->t.size == sizeof(TPM_HANDLE)) return;
738
739     // For all the other entities, concatenate the authorization value to the name.
740     // Get a local copy of the authorization value because some overlapping
741     // may be necessary.
742     EntityGetAuthValue(entityHandle, &auth);
743
744     // Make sure that the extra space is zeroed
745     MemorySet(&bind->t.name[bind->t.size], 0, sizeof(bind->t.name) - bind->t.size);
746     // XOR the authValue at the end of the name
747     for(i = sizeof(bind->t.name) - auth.t.size; i < sizeof(bind->t.name); i++)
748         bind->t.name[i] ^= *pAuth++;
749
750     // Set the bind value to the maximum size
751     bind->t.size = sizeof(bind->t.name);
752
753     return;
754 }
755
756 /*** SessionSetStartTime()
757 // This function is used to initialize the session timing
758 void SessionSetStartTime(SESSION* session // IN: the session to update
759 )
760 {
761     session->startTime = g_time;
762     session->epoch      = g_timeEpoch;
763     session->timeout     = 0;
764 }
765
766 /*** SessionResetPolicyData()
767 // This function is used to reset the policy data without changing the nonce
768 // or the start time of the session.
769 void SessionResetPolicyData(SESSION* session // IN: the session to reset
770 )
771 {
772     SESSION_ATTRIBUTES oldAttributes;
773     pAssert(session != NULL);
774
775     // Will need later
776     oldAttributes = session->attributes;
777
778     // No command
779     session->commandCode = 0;
780
781     // No locality selected
782     MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
783
784     // The cpHash size to zero
785     session->ul.cpHash.b.size = 0;
786
787     // No timeout
788     session->timeout = 0;
789
790     // Reset the pcrCounter
791     session->pcrCounter = 0;
792
793     // Reset the policy hash

```

```

794     MemorySet(&session->u2.policyDigest.t.buffer, 0, session->u2.policyDigest.t.size);
795
796     // Reset the session attributes
797     MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
798
799     // Restore the policy attributes
800     session->attributes.isPolicy = SET;
801     session->attributes.isTrialPolicy = oldAttributes.isTrialPolicy;
802
803     // Restore the bind attributes
804     session->attributes.isDaBound = oldAttributes.isDaBound;
805     session->attributes.isLockoutBound = oldAttributes.isLockoutBound;
806 }
807
808 /*** SessionCapGetLoaded()
809 // This function returns a list of handles of loaded session, started
810 // from input 'handle'
811 //
812 // 'Handle' must be in valid loaded session handle range, but does not
813 // have to point to a loaded session.
814 // Return Type: TPMI_YES_NO
815 //     YES      if there are more handles available
816 //     NO      all the available handles has been returned
817 TPMI_YES_NO
818 SessionCapGetLoaded(TPMI_SH_POLICY handle, // IN: start handle
819                    UINT32 count, // IN: count of returned handles
820                    TPML_HANDLE* handleList // OUT: list of handle
821 )
822 {
823     TPMI_YES_NO more = NO;
824     UINT32 i;
825
826     pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
827
828     // Initialize output handle list
829     handleList->count = 0;
830
831     // The maximum count of handles we may return is MAX_CAP_HANDLES
832     if(count > MAX_CAP_HANDLES)
833         count = MAX_CAP_HANDLES;
834
835     // Iterate session context ID slots to get loaded session handles
836     for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
837     {
838         // If session is active
839         if(gr.contextArray[i] != 0)
840         {
841             // If session is loaded
842             if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
843             {
844                 if(handleList->count < count)
845                 {
846                     SESSION* session;
847
848                     // If we have not filled up the return list, add this
849                     // session handle to it
850                     // assume that this is going to be an HMAC session
851                     handle = i + HMAC_SESSION_FIRST;
852                     session = SessionGet(handle);
853                     if(session->attributes.isPolicy)
854                         handle = i + POLICY_SESSION_FIRST;
855                     handleList->handle[handleList->count] = handle;
856                     handleList->count++;
857                 }
858                 else
859                 {

```



```

860         // If the return list is full but we still have loaded object
861         // available, report this and stop iterating
862         more = YES;
863         break;
864     }
865 }
866 }
867 }
868
869 return more;
870 }
871
872 /*** SessionCapGetOneLoaded()
873 // This function returns whether a session handle exists and is loaded.
874 BOOL SessionCapGetOneLoaded(TPMI_SH_POLICY handle) // IN: handle
875 {
876     pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
877
878     if((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS
879        && gr.contextArray[(handle & HR_HANDLE_MASK)])
880     {
881         return TRUE;
882     }
883
884     return FALSE;
885 }
886
887 /*** SessionCapGetSaved()
888 // This function returns a list of handles for saved session, starting at
889 // 'handle'.
890 //
891 // 'Handle' must be in a valid handle range, but does not have to point to a
892 // saved session
893 //
894 // Return Type: TPMI_YES_NO
895 //     YES      if there are more handles available
896 //     NO      all the available handles has been returned
897 TPMI_YES_NO
898 SessionCapGetSaved(TPMI_SH_HMAC handle, // IN: start handle
899                   UINT32 count, // IN: count of returned handles
900                   TPML_HANDLE* handleList // OUT: list of handle
901 )
902 {
903     TPMI_YES_NO more = NO;
904     UINT32 i;
905
906     pAssert(HandleGetType(handle) == TPM_HT_SAVED_SESSION);
907
908     // Initialize output handle list
909     handleList->count = 0;
910
911     // The maximum count of handles we may return is MAX_CAP_HANDLES
912     if(count > MAX_CAP_HANDLES)
913         count = MAX_CAP_HANDLES;
914
915     // Iterate session context ID slots to get loaded session handles
916     for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
917     {
918         // If session is active
919         if(gr.contextArray[i] != 0)
920         {
921             // If session is saved
922             if(gr.contextArray[i] > MAX_LOADED_SESSIONS)
923             {
924                 if(handleList->count < count)
925                 {

```

```

926         // If we have not filled up the return list, add this
927         // session handle to it
928         handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
929         handleList->count++;
930     }
931     else
932     {
933         // If the return list is full but we still have loaded object
934         // available, report this and stop iterating
935         more = YES;
936         break;
937     }
938 }
939 }
940 }
941
942 return more;
943 }
944
945 /*** SessionCapGetOneSaved()
946 // This function returns whether a session handle exists and is saved.
947 BOOL SessionCapGetOneSaved(TPMI_SH_HMAC handle) // IN: handle
948 {
949     pAssert(HandleGetType(handle) == TPM_HT_SAVED_SESSION);
950
951     if((handle & HR_HANDLE_MASK) < MAX_ACTIVE_SESSIONS
952        && gr.contextArray[(handle & HR_HANDLE_MASK)])
953     {
954         return TRUE;
955     }
956
957     return FALSE;
958 }
959
960 /*** SessionCapGetLoadedNumber()
961 // This function return the number of authorization sessions currently
962 // loaded into TPM RAM.
963 UINT32
964 SessionCapGetLoadedNumber(void)
965 {
966     return MAX_LOADED_SESSIONS - s_freeSessionSlots;
967 }
968
969 /*** SessionCapGetLoadedAvail()
970 // This function returns the number of additional authorization sessions, of
971 // any type, that could be loaded into TPM RAM.
972 // NOTE: In other implementations, this number may just be an estimate. The only
973 // requirement for the estimate is, if it is one or more, then at least one
974 // session must be loadable.
975 UINT32
976 SessionCapGetLoadedAvail(void)
977 {
978     return s_freeSessionSlots;
979 }
980
981 /*** SessionCapGetActiveNumber()
982 // This function returns the number of active authorization sessions currently
983 // being tracked by the TPM.
984 UINT32
985 SessionCapGetActiveNumber(void)
986 {
987     UINT32 i;
988     UINT32 num = 0;
989
990     // Iterate the context array to find the number of non-zero slots
991     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)

```

```

992     {
993         if(gr.contextArray[i] != 0)
994             num++;
995     }
996
997     return num;
998 }
999
1000 /*** SessionCapGetActiveAvail()
1001 // This function returns the number of additional authorization sessions, of any
1002 // type, that could be created. This not the number of slots for sessions, but
1003 // the number of additional sessions that the TPM is capable of tracking.
1004 UINT32
1005 SessionCapGetActiveAvail(void)
1006 {
1007     UINT32 i;
1008     UINT32 num = 0;
1009
1010     // Iterate the context array to find the number of zero slots
1011     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
1012     {
1013         if(gr.contextArray[i] == 0)
1014             num++;
1015     }
1016
1017     return num;
1018 }
1019
1020 /*** IsCpHashUnionOccupied()
1021 // This function indicates whether the session attributes indicate that one of
1022 // the members of the union containing `cpHash` are set.
1023 BOOL IsCpHashUnionOccupied(SESSION_ATTRIBUTES attrs)
1024 {
1025     return attrs.isBound || attrs.isCpHashDefined || attrs.isNameHashDefined
1026         || attrs.isParametersHashDefined || attrs.isTemplateHashDefined;
1027 }

```

7.179 /tpm/src/subsystem/Time.c

```

1  /*** Introduction
2  // This file contains the functions relating to the TPM's time functions including
3  // the interface to the implementation-specific time functions.
4  //
5  /*** Includes
6  #include "Tpm.h"
7  #include "Marshal.h"
8
9  /*** Functions
10
11  /*** TimePowerOn()
12  // This function initialize time info at _TPM_Init().
13  //
14  // This function is called at _TPM_Init() so that the TPM time can start counting
15  // as soon as the TPM comes out of reset and doesn't have to wait until
16  // TPM2_Startup() in order to begin the new time epoch. This could be significant
17  // for systems that could get powered up but not run any TPM commands for some
18  // period of time.
19  //
20  void TimePowerOn(void)
21  {
22      g_time = _plat__TimerRead();
23  }
24
25  /*** TimeNewEpoch()
26  // This function does the processing to generate a new time epoch nonce and

```

```

27 // set NV for update. This function is only called when NV is known to be available
28 // and the clock is running. The epoch is updated to persistent data.
29 static void TimeNewEpoch(void)
30 {
31 #if CLOCK_STOPS
32     CryptRandomGenerate(sizeof(CLOCK_NONCE), (BYTE*)&g_timeEpoch);
33 #else
34     // if the epoch is kept in NV, update it.
35     gp.timeEpoch++;
36     NV_SYNC_PERSISTENT(timeEpoch);
37 #endif
38     // Clean out any lingering state
39     _plat__TimerWasStopped();
40 }
41
42 /** TimeStartup()
43 // This function updates the resetCount and restartCount components of
44 // TPMS_CLOCK_INFO structure at TPM2_Startup().
45 //
46 // This function will deal with the deferred creation of a new epoch.
47 // TimeUpdateToCurrent() will not start a new epoch even if one is due when
48 // TPM_Startup() has not been run. This is because the state of NV is not known
49 // until startup completes. When Startup is done, then it will create the epoch
50 // nonce to complete the initializations by calling this function.
51 BOOL TimeStartup(STARTUP_TYPE type // IN: start up type
52 )
53 {
54     NOT_REFERENCED(type);
55     // If the previous cycle is orderly shut down, the value of the safe bit
56     // the same as previously saved. Otherwise, it is not safe.
57     if(!NV_IS_ORDERLY)
58         go.clockSafe = NO;
59     return TRUE;
60 }
61
62 /** TimeClockUpdate()
63 // This function updates go.clock. If 'newTime' requires an update of NV, then
64 // NV is checked for availability. If it is not available or is rate limiting, then
65 // go.clock is not updated and the function returns an error. If 'newTime' would
66 // not cause an NV write, then go.clock is updated. If an NV write occurs, then
67 // go.safe is SET.
68 void TimeClockUpdate(UINT64 newTime // IN: New time value in mS.
69 )
70 {
71 #define CLOCK_UPDATE_MASK ((1ULL << NV_CLOCK_UPDATE_INTERVAL) - 1)
72
73     // Check to see if the update will cause a need for an nvClock update
74     if((newTime | CLOCK_UPDATE_MASK) > (go.clock | CLOCK_UPDATE_MASK))
75     {
76         pAssert(g_NvStatus == TPM_RC_SUCCESS);
77
78         // Going to update the NV time state so SET the safe flag
79         go.clockSafe = YES;
80
81         // update the time
82         go.clock = newTime;
83
84         NvWrite(NV_ORDERLY_DATA, sizeof(go), &go);
85     }
86     else
87         // No NV update needed so just update
88         go.clock = newTime;
89 }
90
91 /** TimeUpdate()
92 // This function is used to update the time and clock values. If the TPM

```

```

93 // has run TPM2_Startup(), this function is called at the start of each command.
94 // If the TPM has not run TPM2_Startup(), this is called from TPM2_Startup() to
95 // get the clock values initialized. It is not called on command entry because, in
96 // this implementation, the go structure is not read from NV until TPM2_Startup().
97 // The reason for this is that the initialization code (_TPM_Init()) may run before
98 // NV is accessible.
99 void TimeUpdate(void)
100 {
101     UINT64 elapsed;
102     //
103     // Make sure that we consume the current _plat_TimerWasStopped() state.
104     if(_plat_TimerWasStopped())
105     {
106         TimeNewEpoch();
107     }
108     // Get the difference between this call and the last time we updated the tick
109     // timer.
110     elapsed = _plat_TimerRead() - g_time;
111     // Don't read +
112     g_time += elapsed;
113
114     // Don't need to check the result because it has to be success because have
115     // already checked that NV is available.
116     TimeClockUpdate(go.clock + elapsed);
117
118     // Call self healing logic for dictionary attack parameters
119     DASelfHeal();
120 }
121
122 /*** TimeUpdateToCurrent()
123 // This function updates the 'Time' and 'Clock' in the global
124 // TPMS_TIME_INFO structure.
125 //
126 // In this implementation, 'Time' and 'Clock' are updated at the beginning
127 // of each command and the values are unchanged for the duration of the
128 // command.
129 //
130 // Because 'Clock' updates may require a write to NV memory, 'Time' and 'Clock'
131 // are not allowed to advance if NV is not available. When clock is not advancing,
132 // any function that uses 'Clock' will fail and return TPM_RC_NV_UNAVAILABLE or
133 // TPM_RC_NV_RATE.
134 //
135 // This implementation does not do rate limiting. If the implementation does do
136 // rate limiting, then the 'Clock' update should not be inhibited even when doing
137 // rate limiting.
138 void TimeUpdateToCurrent(void)
139 {
140     // Can't update time during the dark interval or when rate limiting so don't
141     // make any modifications to the internal clock value. Also, defer any clock
142     // processing until TPM has run TPM2_Startup()
143     if(!NV_IS_AVAILABLE || !TPMIsStarted())
144         return;
145
146     TimeUpdate();
147 }
148
149 /*** TimeSetAdjustRate()
150 // This function is used to perform rate adjustment on 'Time' and 'Clock'.
151 void TimeSetAdjustRate(TPM_CLOCK_ADJUST adjust // IN: adjust constant
152 )
153 {
154     switch(adjust)
155     {
156     case TPM_CLOCK_COARSE_SLOWER:
157         _plat_ClockRateAdjust(PLAT_TPM_CLOCK_ADJUST_COARSE_SLOWER);
158         break;

```

```

159     case TPM_CLOCK_COARSE_FASTER:
160         _plat__ClockRateAdjust(PLAT_TPM_CLOCK_ADJUST_COARSE_FASTER);
161         break;
162     case TPM_CLOCK_MEDIUM_SLOWER:
163         _plat__ClockRateAdjust(PLAT_TPM_CLOCK_ADJUST_MEDIUM_SLOWER);
164         break;
165     case TPM_CLOCK_MEDIUM_FASTER:
166         _plat__ClockRateAdjust(PLAT_TPM_CLOCK_ADJUST_MEDIUM_FASTER);
167         break;
168     case TPM_CLOCK_FINE_SLOWER:
169         _plat__ClockRateAdjust(PLAT_TPM_CLOCK_ADJUST_FINE_SLOWER);
170         break;
171     case TPM_CLOCK_FINE_FASTER:
172         _plat__ClockRateAdjust(PLAT_TPM_CLOCK_ADJUST_FINE_FASTER);
173         break;
174     case TPM_CLOCK_NO_CHANGE:
175         break;
176     default:
177         // should have been blocked sooner
178         FAIL(FATAL_ERROR_INTERNAL);
179         break;
180 }
181
182 return;
183 }
184
185 /*** TimeGetMarshaled()
186 // This function is used to access TPMS_TIME_INFO in canonical form.
187 // The function collects the time information and marshals it into 'dataBuffer'
188 // and returns the marshaled size
189 UINT16
190 TimeGetMarshaled(TIME_INFO* dataBuffer // OUT: result buffer
191 )
192 {
193     TPMS_TIME_INFO timeInfo;
194
195     // Fill TPMS_TIME_INFO structure
196     timeInfo.time = g_time;
197     TimeFillInfo(&timeInfo.clockInfo);
198
199     // Marshal TPMS_TIME_INFO to canonical form
200     return TPMS_TIME_INFO_Marshal(&timeInfo, (BYTE**)&dataBuffer, NULL);
201 }
202
203 /*** TimeFillInfo
204 // This function gathers information to fill in a TPMS_CLOCK_INFO structure.
205 void TimeFillInfo(TPMS_CLOCK_INFO* clockInfo)
206 {
207     clockInfo->clock = go.clock;
208     clockInfo->resetCount = gp.resetCount;
209     clockInfo->restartCount = gr.restartCount;
210
211     // If NV is not available, clock stopped advancing and the value reported is
212     // not "safe".
213     if(NV_IS_AVAILABLE)
214         clockInfo->safe = go.clockSafe;
215     else
216         clockInfo->safe = NO;
217
218     return;
219 }

```

7.180 /tpm/src/support/AlgorithmCap.c

```
1 /*** Description
```



```

2  // This file contains the algorithm property definitions for the algorithms and the
3  // code for the TPM2_GetCapability() to return the algorithm properties.
4
5  /** Includes and Defines
6
7  #include "Tpm.h"
8
9  typedef struct
10 {
11     TPM_ALG_ID      algID;
12     TPMA_ALGORITHM attributes;
13 } ALGORITHM;
14
15 static const ALGORITHM s_algorithms[] = {
16 // The entries in this table need to be in ascending order but the table doesn't
17 // need to be full (gaps are allowed). One day, a tool might exist to fill in the
18 // table from the TPM_ALG description
19 #if ALG_RSA
20     {TPM_ALG_RSA, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
21 #endif
22 #if ALG_SHA1
23     {TPM_ALG_SHA1, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
24 #endif
25
26     {TPM_ALG_HMAC, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 1, 0, 0, 0)},
27
28 #if ALG_AES
29     {TPM_ALG_AES, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
30 #endif
31 #if ALG_MGF1
32     {TPM_ALG_MGF1, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
33 #endif
34
35     {TPM_ALG_KEYEDHASH, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 1, 0, 1, 1, 0, 0)},
36
37 #if ALG_XOR
38     {TPM_ALG_XOR, TPMA_ALGORITHM_INITIALIZER(0, 1, 1, 0, 0, 0, 0, 0, 0)},
39 #endif
40
41 #if ALG_SHA256
42     {TPM_ALG_SHA256, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
43 #endif
44 #if ALG_SHA384
45     {TPM_ALG_SHA384, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
46 #endif
47 #if ALG_SHA512
48     {TPM_ALG_SHA512, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
49 #endif
50 #if ALG_SM3_256
51     {TPM_ALG_SM3_256, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
52 #endif
53 #if ALG_SM4
54     {TPM_ALG_SM4, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
55 #endif
56 #if ALG_RSASSA
57     {TPM_ALG_RSASSA, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
58 #endif
59 #if ALG_RSAES
60     {TPM_ALG_RSAES, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
61 #endif
62 #if ALG_RSAPSS
63     {TPM_ALG_RSAPSS, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
64 #endif
65 #if ALG_OAEP
66     {TPM_ALG_OAEP, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
67 #endif

```

```

68  #if ALG_ECDSA
69      {TPM_ALG_ECDSA, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
70  #endif
71  #if ALG_ECDH
72      {TPM_ALG_ECDH, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
73  #endif
74  #if ALG_ECDA
75      {TPM_ALG_ECDA, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
76  #endif
77  #if ALG_SM2
78      {TPM_ALG_SM2, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 1, 0)},
79  #endif
80  #if ALG_ECSCNORR
81      {TPM_ALG_ECSCNORR, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
82  #endif
83  #if ALG_ECMQV
84      {TPM_ALG_ECMQV, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
85  #endif
86  #if ALG_KDF1_SP800_56A
87      {TPM_ALG_KDF1_SP800_56A, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
88  #endif
89  #if ALG_KDF2
90      {TPM_ALG_KDF2, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
91  #endif
92  #if ALG_KDF1_SP800_108
93      {TPM_ALG_KDF1_SP800_108, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
94  #endif
95  #if ALG_ECC
96      {TPM_ALG_ECC, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
97  #endif
98
99      {TPM_ALG_SYMCIPHER, TPMA_ALGORITHM_INITIALIZER(0, 0, 0, 1, 0, 0, 0, 0, 0)},
100
101  #if ALG_CAMELLIA
102      {TPM_ALG_CAMELLIA, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
103  #endif
104  #if ALG_CMAC
105      {TPM_ALG_CMAC, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 1, 0, 0, 0)},
106  #endif
107  #if ALG_CTR
108      {TPM_ALG_CTR, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
109  #endif
110  #if ALG_OFB
111      {TPM_ALG_OFB, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
112  #endif
113  #if ALG_CBC
114      {TPM_ALG_CBC, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
115  #endif
116  #if ALG_CFB
117      {TPM_ALG_CFB, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
118  #endif
119  #if ALG_ECB
120      {TPM_ALG_ECB, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
121  #endif
122  };
123
124  /** AlgorithmCapGetImplemented()
125   * This function is used by TPM2_GetCapability() to return a list of the
126   * implemented algorithms.
127   *
128   * Return Type: TPMI_YES_NO
129   * YES         more algorithms to report
130   * NO          no more algorithms to report
131   *
132   * TPMI_YES_NO
133   * AlgorithmCapGetImplemented(TPM_ALG_ID algID, // IN: the starting algorithm ID
134                               UINT32 count, // IN: count of returned algorithms

```

```

134         TPML_ALG_PROPERTY* algList  // OUT: algorithm list
135     )
136     {
137         TPMI_YES_NO more = NO;
138         UINT32      i;
139         UINT32      algNum;
140
141         // initialize output algorithm list
142         algList->count = 0;
143
144         // The maximum count of algorithms we may return is MAX_CAP_ALGS.
145         if(count > MAX_CAP_ALGS)
146             count = MAX_CAP_ALGS;
147
148         // Compute how many algorithms are defined in s_algorithms array.
149         algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
150
151         // Scan the implemented algorithm list to see if there is a match to 'algID'.
152         for(i = 0; i < algNum; i++)
153         {
154             // If algID is less than the starting algorithm ID, skip it
155             if(s_algorithms[i].algID < algID)
156                 continue;
157             if(algList->count < count)
158             {
159                 // If we have not filled up the return list, add more algorithms
160                 // to it
161                 algList->algProperties[algList->count].alg = s_algorithms[i].algID;
162                 algList->algProperties[algList->count].algProperties =
163                     s_algorithms[i].attributes;
164                 algList->count++;
165             }
166             else
167             {
168                 // If the return list is full but we still have algorithms
169                 // available, report this and stop scanning.
170                 more = YES;
171                 break;
172             }
173         }
174         return more;
175     }
176 }
177
178 /** AlgorithmCapGetOneImplemented()
179 // This function returns whether a single algorithm was implemented, along
180 // with its properties (if implemented).
181 BOOL AlgorithmCapGetOneImplemented(
182     TPM_ALG_ID      algID,          // IN: the algorithm ID
183     TPMS_ALG_PROPERTY* algProperty  // OUT: algorithm properties
184 )
185 {
186     UINT32 i;
187     UINT32 algNum;
188
189     // Compute how many algorithms are defined in s_algorithms array.
190     algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
191
192     // Scan the implemented algorithm list to see if there is a match to 'algID'.
193     for(i = 0; i < algNum; i++)
194     {
195         // If algID is less than the starting algorithm ID, skip it
196         if(s_algorithms[i].algID == algID)
197         {
198             algProperty->alg = algID;
199             algProperty->algProperties = s_algorithms[i].attributes;

```

```

200         return TRUE;
201     }
202 }
203
204 return FALSE;
205 }
206
207 /** AlgorithmGetImplementedVector()
208 // This function returns the bit vector of the implemented algorithms.
209 LIB_EXPORT
210 void AlgorithmGetImplementedVector(
211     ALGORITHM_VECTOR* implemented // OUT: the implemented bits are SET
212 )
213 {
214     int index;
215
216     // Nothing implemented until we say it is
217     MemorySet(implemented, 0, sizeof(ALGORITHM_VECTOR));
218     // Go through the list of implemented algorithms and SET the corresponding bit in
219     // in the implemented vector
220     for(index = (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1; index >= 0;
221         index--)
222         SET_BIT(s_algorithms[index].algID, *implemented);
223     return;
224 }

```

7.181 /tpm/src/support/Bits.c

```

1  /** Introduction
2  // This file contains bit manipulation routines. They operate on bit arrays.
3  //
4  // The 0th bit in the array is the right-most bit in the 0th octet in
5  // the array.
6  //
7  // NOTE: If pAssert() is defined, the functions will assert if the indicated bit
8  // number is outside of the range of 'bArray'. How the assert is handled is
9  // implementation dependent.
10
11 /** Includes
12
13 #include "Tpm.h"
14
15 /** Functions
16
17 /** TestBit()
18 // This function is used to check the setting of a bit in an array of bits.
19 // Return Type: BOOL
20 //     TRUE(1)      bit is set
21 //     FALSE(0)     bit is not set
22 BOOL TestBit(unsigned int bitNum, // IN: number of the bit in 'bArray'
23             BYTE* bArray, // IN: array containing the bits
24             unsigned int bytesInArray // IN: size in bytes of 'bArray'
25 )
26 {
27     pAssert(bytesInArray > (bitNum >> 3));
28     return ((bArray[bitNum >> 3] & (1 << (bitNum & 7))) != 0);
29 }
30
31 /** SetBit()
32 // This function will set the indicated bit in 'bArray'.
33 void SetBit(unsigned int bitNum, // IN: number of the bit in 'bArray'
34            BYTE* bArray, // IN: array containing the bits
35            unsigned int bytesInArray // IN: size in bytes of 'bArray'
36 )
37 {

```

```

38     pAssert(bytesInArray > (bitNum >> 3));
39     bArray[bitNum >> 3] |= (1 << (bitNum & 7));
40 }
41
42 /*** ClearBit()
43 // This function will clear the indicated bit in 'bArray'.
44 void ClearBit(unsigned int bitNum,      // IN: number of the bit in 'bArray'.
45              BYTE*      bArray,      // IN: array containing the bits
46              unsigned int bytesInArray // IN: size in bytes of 'bArray'
47 )
48 {
49     pAssert(bytesInArray > (bitNum >> 3));
50     bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
51 }

```

7.182 /tpm/src/support/CommandCodeAttributes.c

```

1  /*** Introduction
2  // This file contains the functions for testing various command properties.
3
4  /*** Includes and Defines
5
6  #include "Tpm.h"
7  #include "CommandCodeAttributes_fp.h"
8
9  // Set the default value for CC_VEND if not already set
10 #ifndef CC_VEND
11 # define CC_VEND (TPM_CC) (0x20000000)
12 #endif
13
14 typedef UINT16 ATTRIBUTE_TYPE;
15
16 // The following file is produced from the command tables in part 3 of the
17 // specification. It defines the attributes for each of the commands.
18 // NOTE: This file is currently produced by an automated process. Files
19 // produced from Part 2 or Part 3 tables through automated processes are not
20 // included in the specification so that their is no ambiguity about the
21 // table containing the information being the normative definition.
22 #define _COMMAND_CODE_ATTRIBUTES_
23 #include "CommandAttributeData.h"
24
25 /*** Command Attribute Functions
26
27 /*** NextImplementedIndex()
28 // This function is used when the lists are not compressed. In a compressed list,
29 // only the implemented commands are present. So, a search might find a value
30 // but that value may not be implemented. This function checks to see if the input
31 // commandIndex points to an implemented command and, if not, it searches upwards
32 // until it finds one. When the list is compressed, this function gets defined
33 // as a no-op.
34 // Return Type: COMMAND_INDEX
35 // UNIMPLEMENTED_COMMAND_INDEX    command is not implemented
36 // other                          index of the command
37 #if !COMPRESSED_LISTS
38 static COMMAND_INDEX NextImplementedIndex(COMMAND_INDEX commandIndex)
39 {
40     for(; commandIndex < COMMAND_COUNT; commandIndex++)
41     {
42         if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
43             return commandIndex;
44     }
45     return UNIMPLEMENTED_COMMAND_INDEX;
46 }
47 #else
48 # define NextImplementedIndex(x) (x)

```

```

49  #endif
50
51  /*** GetClosestCommandIndex()
52  // This function returns the command index for the command with a value that is
53  // equal to or greater than the input value
54  // Return Type: COMMAND_INDEX
55  // UNIMPLEMENTED_COMMAND_INDEX    command is not implemented
56  // other                          index of a command
57  COMMAND_INDEX
58  GetClosestCommandIndex(TPM_CC commandCode // IN: the command code to start at
59  )
60  {
61      BOOL vendor = (commandCode & CC_VEND) != 0;
62      COMMAND_INDEX searchIndex = (COMMAND_INDEX)commandCode;
63
64      // The commandCode is a UINT32 and the search index is UINT16. We are going to
65      // search for a match but need to make sure that the commandCode value is not
66      // out of range. To do this, need to clear the vendor bit of the commandCode
67      // (if set) and compare the result to the 16-bit searchIndex value. If it is
68      // out of range, indicate that the command is not implemented
69      if((commandCode & ~CC_VEND) != searchIndex)
70          return UNIMPLEMENTED_COMMAND_INDEX;
71
72      // if there is at least one vendor command, the last entry in the array will
73      // have the v bit set. If the input commandCode is larger than the last
74      // vendor-command, then it is out of range.
75      if(vendor)
76      {
77          #if VENDOR_COMMAND_ARRAY_SIZE > 0
78              COMMAND_INDEX commandIndex;
79              COMMAND_INDEX min;
80              COMMAND_INDEX max;
81              int diff;
82              # if LIBRARY_COMMAND_ARRAY_SIZE == COMMAND_COUNT
83              #   error "Constants are not consistent."
84              # endif
85              // Check to see if the value is equal to or below the minimum
86              // entry.
87              // Note: Put this check first so that the typical case of only one vendor-
88              // specific command doesn't waste any more time.
89              if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE], TPMA_CC, commandIndex)
90                 >= searchIndex)
91              {
92                  // the vendor array is always assumed to be packed so there is
93                  // no need to check to see if the command is implemented
94                  return LIBRARY_COMMAND_ARRAY_SIZE;
95              }
96              // See if this is out of range on the top
97              if(GET_ATTRIBUTE(s_ccAttr[COMMAND_COUNT - 1], TPMA_CC, commandIndex)
98                 < searchIndex)
99              {
100                  return UNIMPLEMENTED_COMMAND_INDEX;
101              }
102              commandIndex = UNIMPLEMENTED_COMMAND_INDEX; // Needs initialization to keep
103                                                             // compiler happy
104              min = LIBRARY_COMMAND_ARRAY_SIZE;           // first vendor command
105              max = COMMAND_COUNT - 1;                    // last vendor command
106              diff = 1;                                    // needs initialization to keep
107                                                             // compiler happy
108              while(min <= max)
109              {
110                  commandIndex = (min + max + 1) / 2;
111                  diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
112                         - searchIndex;
113                  if(diff == 0)
114                      return commandIndex;

```



```

115         if(diff > 0)
116             max = commandIndex - 1;
117         else
118             min = commandIndex + 1;
119     }
120     // didn't find an exact match. commandIndex will be pointing at the last
121     // item tested. If 'diff' is positive, then the last item tested was
122     // larger index of the command code so it is the smallest value
123     // larger than the requested value.
124     if(diff > 0)
125         return commandIndex;
126     // if 'diff' is negative, then the value tested was smaller than
127     // the commandCode index and the next higher value is the correct one.
128     // Note: this will necessarily be in range because of the earlier check
129     // that the index was within range.
130     return commandIndex + 1;
131 #else
132     // If there are no vendor commands so anything with the vendor bit set is out
133     // of range
134     return UNIMPLEMENTED_COMMAND_INDEX;
135 #endif
136 }
137 // Get here if the V-Bit was not set in 'commandCode'
138
139 if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE - 1], TPMA_CC, commandIndex)
140    < searchIndex)
141 {
142     // requested index is out of the range to the top
143 #if VENDOR_COMMAND_ARRAY_SIZE > 0
144     // If there are vendor commands, then the first vendor command
145     // is the next value greater than the commandCode.
146     // NOTE: we got here if the starting index did not have the V bit but we
147     // reached the end of the array of library commands (non-vendor). Since
148     // there is at least one vendor command, and vendor commands are always
149     // in a compressed list that starts after the library list, the next
150     // index value contains a valid vendor command.
151     return LIBRARY_COMMAND_ARRAY_SIZE;
152 #else
153     // if there are no vendor commands, then this is out of range
154     return UNIMPLEMENTED_COMMAND_INDEX;
155 #endif
156 }
157 // If the request is lower than any value in the array, then return
158 // the lowest value (needs to be an index for an implemented command
159 if(GET_ATTRIBUTE(s_ccAttr[0], TPMA_CC, commandIndex) >= searchIndex)
160 {
161     return NextImplementedIndex(0);
162 }
163 else
164 {
165 #if COMPRESSED_LISTS
166     COMMAND_INDEX commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
167     COMMAND_INDEX min           = 0;
168     COMMAND_INDEX max           = LIBRARY_COMMAND_ARRAY_SIZE - 1;
169     int diff                    = 1;
170 # if LIBRARY_COMMAND_ARRAY_SIZE == 0
171 #   error "Something is terribly wrong"
172 # endif
173     // The s_ccAttr array contains an extra entry at the end (a zero value).
174     // Don't count this as an array entry. This means that max should start
175     // out pointing to the last valid entry in the array which is - 2
176     pAssert(
177         max
178         == (sizeof(s_ccAttr) / sizeof(TPMA_CC) - VENDOR_COMMAND_ARRAY_SIZE - 2));
179     while(min <= max)
180     {

```

```

181         commandIndex = (min + max + 1) / 2;
182         diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
183             - searchIndex;
184         if(diff == 0)
185             return commandIndex;
186         if(diff > 0)
187             max = commandIndex - 1;
188         else
189             min = commandIndex + 1;
190     }
191     // didn't find an exact match. commandIndex will be pointing at the
192     // last item tested. If diff is positive, then the last item tested was
193     // larger index of the command code so it is the smallest value
194     // larger than the requested value.
195     if(diff > 0)
196         return commandIndex;
197     // if diff is negative, then the value tested was smaller than
198     // the commandCode index and the next higher value is the correct one.
199     // Note: this will necessarily be in range because of the earlier check
200     // that the index was within range.
201     return commandIndex + 1;
202 #else
203     // The list is not compressed so offset into the array by the command
204     // code value of the first entry in the list. Then go find the first
205     // implemented command.
206     return NextImplementedIndex(
207         searchIndex - (COMMAND_INDEX)s_ccAttr[0].commandIndex);
208 #endif
209     }
210 }
211
212 /*** CommandCodeToCommandIndex()
213 // This function returns the index in the various attributes arrays of the
214 // command.
215 // Return Type: COMMAND_INDEX
216 // UNIMPLEMENTED_COMMAND_INDEX    command is not implemented
217 // other                          index of the command
218 COMMAND_INDEX
219 CommandCodeToCommandIndex(TPM_CC commandCode // IN: the command code to look up
220 )
221 {
222     // Extract the low 16-bits of the command code to get the starting search index
223     COMMAND_INDEX searchIndex = (COMMAND_INDEX)commandCode;
224     BOOL vendor = (commandCode & CC_VEND) != 0;
225     COMMAND_INDEX commandIndex;
226 #if !COMPRESSED_LISTS
227     if(!vendor)
228     {
229         commandIndex = searchIndex - (COMMAND_INDEX)s_ccAttr[0].commandIndex;
230         // Check for out of range or unimplemented.
231         // Note, since a COMMAND_INDEX is unsigned, if searchIndex is smaller than
232         // the lowest value of command, it will become a 'negative' number making
233         // it look like a large unsigned number, this will cause it to fail
234         // the unsigned check below.
235         if(commandIndex >= LIBRARY_COMMAND_ARRAY_SIZE
236            || (s_commandAttributes[commandIndex] & IS_IMPLEMENTED) == 0)
237             return UNIMPLEMENTED_COMMAND_INDEX;
238         return commandIndex;
239     }
240 #endif
241     // Need this code for any vendor code lookup or for compressed lists
242     commandIndex = GetClosestCommandIndex(commandCode);
243
244     // Look at the returned value from get closest. If it isn't the one that was
245     // requested, then the command is not implemented.
246     if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)

```

```

247     {
248         if((GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
249             != searchIndex)
250             || (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V)) != vendor)
251             commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
252     }
253     return commandIndex;
254 }
255
256 /*** GetNextCommandIndex()
257 // This function returns the index of the next implemented command.
258 // Return Type: COMMAND_INDEX
259 // UNIMPLEMENTED_COMMAND_INDEX    no more implemented commands
260 // other                          the index of the next implemented command
261 COMMAND_INDEX
262 GetNextCommandIndex(COMMAND_INDEX commandIndex // IN: the starting index
263 )
264 {
265     while(++commandIndex < COMMAND_COUNT)
266     {
267         #if !COMPRESSED_LISTS
268             if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
269             #endif
270                 return commandIndex;
271     }
272     return UNIMPLEMENTED_COMMAND_INDEX;
273 }
274
275 /*** GetCommandCode()
276 // This function returns the commandCode associated with the command index
277 TPM_CC
278 GetCommandCode(COMMAND_INDEX commandIndex // IN: the command index
279 )
280 {
281     TPM_CC commandCode = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex);
282     if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
283         commandCode += CC_VEND;
284     return commandCode;
285 }
286
287 /*** CommandAuthRole()
288 //
289 // This function returns the authorization role required of a handle.
290 //
291 // Return Type: AUTH_ROLE
292 // AUTH_NONE    no authorization is required
293 // AUTH_USER    user role authorization is required
294 // AUTH_ADMIN   admin role authorization is required
295 // AUTH_DUP     duplication role authorization is required
296 AUTH_ROLE
297 CommandAuthRole(COMMAND_INDEX commandIndex, // IN: command index
298                 UINT32 handleIndex // IN: handle index (zero based)
299 )
300 {
301     if(0 == handleIndex)
302     {
303         // Any authorization role set?
304         COMMAND_ATTRIBUTES properties = s_commandAttributes[commandIndex];
305
306         if(properties & HANDLE_1_USER)
307             return AUTH_USER;
308         if(properties & HANDLE_1_ADMIN)
309             return AUTH_ADMIN;
310         if(properties & HANDLE_1_DUP)
311             return AUTH_DUP;
312     }

```

```

313     else if(1 == handleIndex)
314     {
315         if(s_commandAttributes[commandIndex] & HANDLE_2_USER)
316             return AUTH_USER;
317     }
318     return AUTH_NONE;
319 }
320
321 /*** EncryptSize()
322 // This function returns the size of the decrypt size field. This function returns
323 // 0 if encryption is not allowed
324 // Return Type: int
325 // 0      encryption not allowed
326 // 2      size field is two bytes
327 // 4      size field is four bytes
328 int EncryptSize(COMMAND_INDEX commandIndex // IN: command index
329 )
330 {
331     return ((s_commandAttributes[commandIndex] & ENCRYPT_2) ? 2
332            : (s_commandAttributes[commandIndex] & ENCRYPT_4) ? 4
333            : 0);
334 }
335
336 /*** DecryptSize()
337 // This function returns the size of the decrypt size field. This function returns
338 // 0 if decryption is not allowed
339 // Return Type: int
340 // 0      encryption not allowed
341 // 2      size field is two bytes
342 // 4      size field is four bytes
343 int DecryptSize(COMMAND_INDEX commandIndex // IN: command index
344 )
345 {
346     return ((s_commandAttributes[commandIndex] & DECRYPT_2) ? 2
347            : (s_commandAttributes[commandIndex] & DECRYPT_4) ? 4
348            : 0);
349 }
350
351 /*** IsSessionAllowed()
352 //
353 // This function indicates if the command is allowed to have sessions.
354 //
355 // This function must not be called if the command is not known to be implemented.
356 //
357 // Return Type: BOOL
358 // TRUE(1)      session is allowed with this command
359 // FALSE(0)     session is not allowed with this command
360 BOOL IsSessionAllowed(COMMAND_INDEX commandIndex // IN: the command to be checked
361 )
362 {
363     return ((s_commandAttributes[commandIndex] & NO_SESSIONS) == 0);
364 }
365
366 /*** IsHandleInResponse()
367 // This function determines if a command has a handle in the response
368 BOOL IsHandleInResponse(COMMAND_INDEX commandIndex)
369 {
370     return ((s_commandAttributes[commandIndex] & R_HANDLE) != 0);
371 }
372
373 /*** IsWriteOperation()
374 // Checks to see if an operation will write to an NV Index and is subject to being
375 // blocked by read-lock
376 BOOL IsWriteOperation(COMMAND_INDEX commandIndex // IN: Command to check
377 )
378 {

```

```

379 #ifndef WRITE_LOCK
380     return ((s_commandAttributes[commandIndex] & WRITE_LOCK) != 0);
381 #else
382     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
383     {
384         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
385         {
386             case TPM_CC_NV_Write:
387 # if CC_NV_Increment
388                 case TPM_CC_NV_Increment:
389 # endif
390 # if CC_NV_SetBits
391                 case TPM_CC_NV_SetBits:
392 # endif
393 # if CC_NV_Extend
394                 case TPM_CC_NV_Extend:
395 # endif
396 # if CC_AC_Send
397                 case TPM_CC_AC_Send:
398 # endif
399                     // NV write lock counts as a write operation for authorization purposes.
400                     // We check to see if the NV is write locked before we do the
401                     // authorization. If it is locked, we fail the command early.
402                     case TPM_CC_NV_WriteLock:
403                         return TRUE;
404                     default:
405                         break;
406                 }
407             }
408         return FALSE;
409 #endif
410 }
411
412 /*** IsReadOperation()
413 // Checks to see if an operation will write to an NV Index and is
414 // subject to being blocked by write-lock.
415 BOOL IsReadOperation(COMMAND_INDEX commandIndex // IN: Command to check
416 )
417 {
418 #ifndef READ_LOCK
419     return ((s_commandAttributes[commandIndex] & READ_LOCK) != 0);
420 #else
421     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
422     {
423         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
424         {
425             case TPM_CC_NV_Read:
426             case TPM_CC_PolicyNV:
427             case TPM_CC_NV_Certify:
428                 // NV read lock counts as a read operation for authorization purposes.
429                 // We check to see if the NV is read locked before we do the
430                 // authorization. If it is locked, we fail the command early.
431                 case TPM_CC_NV_ReadLock:
432                     return TRUE;
433                 default:
434                     break;
435             }
436         }
437     }
438     return FALSE;
439 #endif
440 }
441
442 /*** CommandCapGetCCList()
443 // This function returns a list of implemented commands and command attributes
444 // starting from the command in 'commandCode'.

```

```

445 // Return Type: TPMI_YES_NO
446 //     YES      more command attributes are available
447 //     NO       no more command attributes are available
448 TPMI_YES_NO
449 CommandCapGetCCList(TPM_CC commandCode, // IN: start command code
450                    UINT32 count,        // IN: maximum count for number of entries in
451                                       // 'commandList'
452                    TPML_CCA* commandList // OUT: list of TPMA_CC
453 )
454 {
455     TPMI_YES_NO more = NO;
456     COMMAND_INDEX commandIndex;
457
458     // initialize output handle list count
459     commandList->count = 0;
460
461     for(commandIndex = GetClosestCommandIndex(commandCode);
462        commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
463        commandIndex = GetNextCommandIndex(commandIndex))
464     {
465         #if !COMPRESSED_LISTS
466             // this check isn't needed for compressed lists.
467             if(!(s_commandAttributes[commandIndex] & IS_IMPLEMENTED))
468                 continue;
469         #endif
470         if(commandList->count < count)
471         {
472             // If the list is not full, add the attributes for this command.
473             commandList->commandAttributes[commandList->count] =
474                 s_ccAttr[commandIndex];
475             commandList->count++;
476         }
477         else
478         {
479             // If the list is full but there are more commands to report,
480             // indicate this and return.
481             more = YES;
482             break;
483         }
484     }
485     return more;
486 }
487
488 /*** CommandCapGetOneCC()
489 // This function checks whether a command is implemented, and returns its
490 // attributes if so.
491 BOOL CommandCapGetOneCC(TPM_CC commandCode, // IN: command code
492                        TPMA_CC* commandAttributes // OUT: command attributes
493 )
494 {
495     COMMAND_INDEX commandIndex = CommandCodeToCommandIndex(commandCode);
496     if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
497     {
498         *commandAttributes = s_ccAttr[commandIndex];
499         return TRUE;
500     }
501     return FALSE;
502 }
503
504 /*** IsVendorCommand()
505 // Function indicates if a command index references a vendor command.
506 // Return Type: BOOL
507 //     TRUE(1)      command is a vendor command
508 //     FALSE(0)     command is not a vendor command
509 BOOL IsVendorCommand(COMMAND_INDEX commandIndex // IN: command index to check
510 )

```



```

511 {
512     return (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V));
513 }

```

7.183 /tpm/src/support/Entity.c

```

1  /** Description
2  // The functions in this file are used for accessing properties for handles of
3  // various types. Functions in other files require handles of a specific
4  // type but the functions in this file allow use of any handle type.
5
6  /** Includes
7
8  #include "Tpm.h"
9
10 /** Functions
11 /** EntityGetLoadStatus()
12 // This function will check that all the handles access loaded entities.
13 // Return Type: TPM_RC
14 //     TPM_RC_HANDLE           handle type does not match
15 //     TPM_RC_REFERENCE_Hx     entity is not present
16 //     TPM_RC_HIERARCHY       entity belongs to a disabled hierarchy
17 //     TPM_RC_OBJECT_MEMORY   handle is an evict object but there is no
18 //                             space to load it to RAM
19 TPM_RC
20 EntityGetLoadStatus(COMMAND* command // IN/OUT: command parsing structure
21 )
22 {
23     UINT32 i;
24     TPM_RC result = TPM_RC_SUCCESS;
25     //
26     for(i = 0; i < command->handleNum; i++)
27     {
28         TPM_HANDLE handle = command->handles[i];
29         switch(HandleGetType(handle))
30         {
31             // For handles associated with hierarchies, the entity is present
32             // only if the associated enable is SET.
33             case TPM_HT_PERMANENT:
34                 switch(handle)
35                 {
36                     // First handle non-hierarchy cases
37                     #if VENDOR_PERMANENT_AUTH_ENABLED == YES
38                     case VENDOR_PERMANENT_AUTH_HANDLE:
39                         if(!gc.ehEnable)
40                             result = TPM_RC_HIERARCHY;
41                     break;
42                     #endif
43                     // PW session handle and lockout handle are always available
44                     case TPM_RS_PW:
45                         // Need to be careful for lockout. Lockout is always available
46                         // for policy checks but not always available when authValue
47                         // is being checked.
48                     case TPM_RH_LOCKOUT:
49                         // Rather than have #ifdefs all over the code,
50                         // CASE_ACT_HANDLE is defined in ACT.h. It is 'case
51                         TPM_RH_ACT_x: '
52                         // FOR_EACH_ACT(CASE_ACT_HANDLE) creates a simple
53                         // case TPM_RH_ACT_x: // for each of the implemented ACT.
54                         FOR_EACH_ACT(CASE_ACT_HANDLE)
55                         break;
56                     default:
57                         // If the implementation has a manufacturer-specific value
58                         // then test for it here. Since this implementation does
59                         // not have any, this implementation returns the same failure

```

```

59         // that unmarshaling of a bad handle would produce.
60         if(((TPM_RH)handle >= TPM_RH_AUTH_00)
61            && ((TPM_RH)handle <= TPM_RH_AUTH_FF))
62             // if the implementation has a manufacturer-specific value
63             result = TPM_RC_VALUE;
64         else
65             // The handle either refers to a hierarchy or is invalid.
66             result = ValidateHierarchy(handle);
67         break;
68     }
69     break;
70     case TPM_HT_TRANSIENT:
71         // For a transient object, check if the handle is associated
72         // with a loaded object.
73         if(!IsObjectPresent(handle))
74             result = TPM_RC_REFERENCE_H0;
75         break;
76     case TPM_HT_PERSISTENT:
77         // Persistent object
78         // Copy the persistent object to RAM and replace the handle with the
79         // handle of the assigned slot. A TPM_RC_OBJECT_MEMORY,
80         // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
81         // ObjectLoadEvict()
82         result = ObjectLoadEvict(&command->handles[i], command->index);
83         break;
84     case TPM_HT_HMAC_SESSION:
85         // For an HMAC session, see if the session is loaded
86         // and if the session in the session slot is actually
87         // an HMAC session.
88         if(SessionIsLoaded(handle))
89         {
90             SESSION* session;
91             session = SessionGet(handle);
92             // Check if the session is a HMAC session
93             if(session->attributes.isPolicy == SET)
94                 result = TPM_RC_HANDLE;
95         }
96         else
97             result = TPM_RC_REFERENCE_H0;
98         break;
99     case TPM_HT_POLICY_SESSION:
100         // For a policy session, see if the session is loaded
101         // and if the session in the session slot is actually
102         // a policy session.
103         if(SessionIsLoaded(handle))
104         {
105             SESSION* session;
106             session = SessionGet(handle);
107             // Check if the session is a policy session
108             if(session->attributes.isPolicy == CLEAR)
109                 result = TPM_RC_HANDLE;
110         }
111         else
112             result = TPM_RC_REFERENCE_H0;
113         break;
114     case TPM_HT_NV_INDEX:
115         // For an NV Index, use the TPM-specific routine
116         // to search the IN Index space.
117         result = NvIndexIsAccessible(handle);
118         break;
119     case TPM_HT_PCR:
120         // Any PCR handle that is unmarshaled successfully referenced
121         // a PCR that is defined.
122         break;
123 #if CC_AC_Send
124     case TPM_HT_AC:

```

```

125         // Use the TPM-specific routine to search for the AC
126         result = AcIsAccessible(handle);
127         break;
128     #endif
129     case TPM_HT_EXTERNAL_NV:
130     case TPM_HT_PERMANENT_NV:
131         // Not yet supported.
132         result = TPM_RC_VALUE;
133         break;
134     default:
135         // Any other handle type is a defect in the unmarshaling code.
136         FAIL(FATAL_ERROR_INTERNAL);
137         break;
138     }
139     if(result != TPM_RC_SUCCESS)
140     {
141         if(result == TPM_RC_REFERENCE_H0)
142             result = result + i;
143         else
144             result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
145         break;
146     }
147 }
148 return result;
149 }
150
151 /*** EntityGetAuthValue()
152 // This function is used to access the 'authValue' associated with a handle.
153 // This function assumes that the handle references an entity that is accessible
154 // and the handle is not for a persistent objects. That is EntityGetLoadStatus()
155 // should have been called. Also, the accessibility of the authValue should have
156 // been verified by IsAuthValueAvailable().
157 //
158 // This function copies the authorization value of the entity to 'auth'.
159 // Return Type: UINT16
160 // count          number of bytes in the authValue with 0's stripped
161
162 EntityGetAuthValue(TPMI_DH_ENTITY handle, // IN: handle of entity
163                   TPM2B_AUTH* auth      // OUT: authValue of the entity
164 )
165 {
166     TPM2B_AUTH* pAuth = NULL;
167
168     auth->t.size = 0;
169
170     switch(HandleGetType(handle))
171     {
172     case TPM_HT_PERMANENT:
173     {
174         switch(HierarchyNormalizeHandle(handle))
175         {
176         case TPM_RH_OWNER:
177             // ownerAuth for TPM_RH_OWNER
178             pAuth = &gp.ownerAuth;
179             break;
180         case TPM_RH_ENDORSEMENT:
181             // endorsementAuth for TPM_RH_ENDORSEMENT
182             pAuth = &gp.endorsementAuth;
183             break;
184
185             // The ACT use platformAuth for auth
186             FOR_EACH_ACT(CASE_ACT_HANDLE)
187
188         case TPM_RH_PLATFORM:
189             // platformAuth for TPM_RH_PLATFORM
190             pAuth = &gc.platformAuth;

```

```

191         break;
192     case TPM_RH_LOCKOUT:
193         // lockoutAuth for TPM_RH_LOCKOUT
194         pAuth = &gp.lockoutAuth;
195         break;
196     case TPM_RH_NULL:
197         // nullAuth for TPM_RH_NULL. Return 0 directly here
198         return 0;
199         break;
200 #if VENDOR_PERMANENT_AUTH_ENABLED == YES
201     case VENDOR_PERMANENT_AUTH_HANDLE:
202         // vendor authorization value
203         pAuth = &g_platformUniqueAuth;
204 #endif
205     default:
206         // If any other permanent handle is present it is
207         // a code defect.
208         FAIL(FATAL_ERROR_INTERNAL);
209         break;
210 }
211 break;
212 }
213 case TPM_HT_TRANSIENT:
214     // authValue for an object
215     // A persistent object would have been copied into RAM
216     // and would have an transient object handle here.
217     {
218         OBJECT* object;
219
220         object = HandleToObject(handle);
221         // special handling if this is a sequence object
222         if(ObjectIsSequence(object))
223         {
224             pAuth = &((HASH_OBJECT*)object)->auth;
225         }
226         else
227         {
228             // Authorization is available only when the private portion of
229             // the object is loaded. The check should be made before
230             // this function is called
231             pAssert(object->attributes.publicOnly == CLEAR);
232             pAuth = &object->sensitive.authValue;
233         }
234     }
235     break;
236 case TPM_HT_NV_INDEX:
237     // authValue for an NV index
238     {
239         NV_INDEX* nvIndex = NvGetIndexInfo(handle, NULL);
240         pAssert(nvIndex != NULL);
241         pAuth = &nvIndex->authValue;
242     }
243     break;
244 case TPM_HT_PCR:
245     // authValue for PCR
246     pAuth = PCRGetAuthValue(handle);
247     break;
248 default:
249     // If any other handle type is present here, then there is a defect
250     // in the unmarshaling code.
251     FAIL(FATAL_ERROR_INTERNAL);
252     break;
253 }
254 // Copy the authValue
255 MemoryCopy2B((TPM2B*)auth, (TPM2B*)pAuth, sizeof(auth->t.buffer));
256 MemoryRemoveTrailingZeros(auth);

```

```

257     return auth->t.size;
258 }
259
260 /*** EntityGetAuthPolicy()
261 // This function is used to access the 'authPolicy' associated with a handle.
262 // This function assumes that the handle references an entity that is accessible
263 // and the handle is not for a persistent objects. That is EntityGetLoadStatus()
264 // should have been called. Also, the accessibility of the authPolicy should have
265 // been verified by IsAuthPolicyAvailable().
266 //
267 // This function copies the authorization policy of the entity to 'authPolicy'.
268 //
269 // The return value is the hash algorithm for the policy.
270 TPMI_ALG_HASH
271 EntityGetAuthPolicy(TPMI_DH_ENTITY handle,      // IN: handle of entity
272                    TPM2B_DIGEST* authPolicy    // OUT: authPolicy of the entity
273 )
274 {
275     TPMI_ALG_HASH hashAlg = TPM_ALG_NULL;
276     authPolicy->t.size = 0;
277
278     switch(HandleGetType(handle))
279     {
280     case TPM_HT_PERMANENT:
281         switch(HierarchyNormalizeHandle(handle))
282         {
283         case TPM_RH_OWNER:
284             // ownerPolicy for TPM_RH_OWNER
285             *authPolicy = gp.ownerPolicy;
286             hashAlg = gp.ownerAlg;
287             break;
288         case TPM_RH_ENDORSEMENT:
289             // endorsementPolicy for TPM_RH_ENDORSEMENT
290             *authPolicy = gp.endorsementPolicy;
291             hashAlg = gp.endorsementAlg;
292             break;
293         case TPM_RH_PLATFORM:
294             // platformPolicy for TPM_RH_PLATFORM
295             *authPolicy = gc.platformPolicy;
296             hashAlg = gc.platformAlg;
297             break;
298         case TPM_RH_LOCKOUT:
299             // lockoutPolicy for TPM_RH_LOCKOUT
300             *authPolicy = gp.lockoutPolicy;
301             hashAlg = gp.lockoutAlg;
302             break;
303         #define ACT_GET_POLICY(N) \
304         case TPM_RH_ACT_##N: \
305             *authPolicy = go.ACT_##N.authPolicy; \
306             hashAlg = go.ACT_##N.hashAlg; \
307             break;
308             // Get the policy for each implemented ACT
309             FOR_EACH_ACT(ACT_GET_POLICY)
310         default:
311             hashAlg = TPM_ALG_ERROR;
312             break;
313         }
314         break;
315     case TPM_HT_TRANSIENT:
316         // authPolicy for an object
317         {
318             OBJECT* object = HandleToObject(handle);
319             *authPolicy = object->publicArea.authPolicy;
320             hashAlg = object->publicArea.nameAlg;
321         }
322         break;

```

```

323     case TPM_HT_NV_INDEX:
324         // authPolicy for a NV index
325         {
326             NV_INDEX* nvIndex = NvGetIndexInfo(handle, NULL);
327             pAssert(nvIndex != 0);
328             *authPolicy = nvIndex->publicArea.authPolicy;
329             hashAlg      = nvIndex->publicArea.nameAlg;
330         }
331         break;
332     case TPM_HT_PCR:
333         // authPolicy for a PCR
334         hashAlg = PCRGetAuthPolicy(handle, authPolicy);
335         break;
336     default:
337         // If any other handle type is present it is a code defect.
338         FAIL(FATAL_ERROR_INTERNAL);
339         break;
340 }
341 return hashAlg;
342 }
343
344 /*** EntityGetName()
345 // This function returns the Name associated with a handle.
346 TPM2B_NAME* EntityGetName(TPMI_DH_ENTITY handle, // IN: handle of entity
347                           TPM2B_NAME* name      // OUT: name of entity
348 )
349 {
350     switch(HandleGetType(handle))
351     {
352     case TPM_HT_TRANSIENT:
353     {
354         // Name for an object
355         OBJECT* object = HandleToObject(handle);
356         // an object with no nameAlg has no name
357         if(object->publicArea.nameAlg == TPM_ALG_NULL)
358             name->b.size = 0;
359         else
360             *name = object->name;
361         break;
362     }
363     case TPM_HT_NV_INDEX:
364         // Name for a NV index
365         NvGetNameByIndexHandle(handle, name);
366         break;
367     default:
368         // For all other types, the handle is the Name
369         name->t.size = sizeof(TPM_HANDLE);
370         UINT32_TO_BYTE_ARRAY(handle, name->t.name);
371         break;
372     }
373     return name;
374 }
375
376 /*** EntityGetHierarchy()
377 // This function returns the hierarchy handle associated with an entity.
378 // a) A handle that is a hierarchy handle is associated with itself.
379 // b) An NV index belongs to TPM_RH_PLATFORM if TPMA_NV_PLATFORMCREATE,
380 //    is SET, otherwise it belongs to TPM_RH_OWNER
381 // c) An object handle belongs to its hierarchy.
382 TPMI_RH_HIERARCHY
383 EntityGetHierarchy(TPMI_DH_ENTITY handle // IN :handle of entity
384 )
385 {
386     TPMI_RH_HIERARCHY hierarchy = TPM_RH_NULL;
387
388     switch(HandleGetType(handle))

```



```

389 {
390     case TPM_HT_PERMANENT:
391         // hierarchy for a permanent handle
392
393         if(HierarchyIsFirmwareLimited(handle) || HierarchyIsSvnLimited(handle))
394         {
395             hierarchy = handle;
396             break;
397         }
398
399         switch(handle)
400         {
401             case TPM_RH_PLATFORM:
402             case TPM_RH_ENDORSEMENT:
403             case TPM_RH_NULL:
404                 hierarchy = handle;
405                 break;
406             // all other permanent handles are associated with the owner
407             // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
408             default:
409                 hierarchy = TPM_RH_OWNER;
410                 break;
411         }
412         break;
413     case TPM_HT_NV_INDEX:
414         // hierarchy for NV index
415         {
416             NV_INDEX* nvIndex = NvGetIndexInfo(handle, NULL);
417             pAssert(nvIndex != NULL);
418
419             // If only the platform can delete the index, then it is
420             // considered to be in the platform hierarchy, otherwise it
421             // is in the owner hierarchy.
422             if(IS_ATTRIBUTE(
423                 nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
424                 hierarchy = TPM_RH_PLATFORM;
425             else
426                 hierarchy = TPM_RH_OWNER;
427         }
428         break;
429     case TPM_HT_TRANSIENT:
430         // hierarchy for an object
431         {
432             OBJECT* object;
433             object = HandleToObject(handle);
434             if(object->attributes.ppsHierarchy)
435             {
436                 hierarchy = TPM_RH_PLATFORM;
437             }
438             else if(object->attributes.epsHierarchy)
439             {
440                 hierarchy = TPM_RH_ENDORSEMENT;
441             }
442             else if(object->attributes.spsHierarchy)
443             {
444                 hierarchy = TPM_RH_OWNER;
445             }
446         }
447         break;
448     case TPM_HT_PCR:
449         hierarchy = TPM_RH_OWNER;
450         break;
451     default:
452         FAIL(FATAL_ERROR_INTERNAL);
453         break;
454 }

```

```

455     // this is unreachable but it provides a return value for the default
456     // case which makes the compiler happy
457     return hierarchy;
458 }

```

7.184 /tpm/src/support/Global.c

```

1  /** Description
2  // This file will instance the TPM variables that are not stack allocated.
3
4  // Descriptions of global variables are in Global.h. There macro definitions
5  // that allows a variable to be instanced or simply defined as an external variable.
6  // When global.h is included from this .c file, GLOBAL_C is defined and values are
7  // instanced (and possibly initialized), but when global.h is included by any other
8  // file, they are simply defined as external values. DO NOT DEFINE GLOBAL_C IN ANY
9  // OTHER FILE.
10 //
11 // NOTE: This is a change from previous implementations where Global.h just contained
12 // the extern declaration and values were instanced in this file. This change keeps
13 // the definition and instance in one file making maintenance easier. The instanced
14 // data will still be in the global.obj file.
15 //
16 // The OIDs.h file works in a way that is similar to the Global.h with the definition
17 // of the values in OIDs.h such that they are instanced in global.obj. The macros
18 // that are defined in Global.h are used in OIDs.h in the same way as they are in
19 // Global.h.
20
21 /** Defines and Includes
22 #define GLOBAL_C
23 #include "Tpm.h"
24 #include "OIDs.h"
25
26 #if CC_CertifyX509
27 # include "X509.h"
28 #endif // CC_CertifyX509
29
30 // Global string constants for consistency in KDF function calls.
31 // These string constants are shared across functions to make sure that they
32 // are all using consistent string values.
33
34 // each instance must define a different struct since the buffer sizes vary.
35 #define TPM2B_STRING(name, value)
36     typedef union name##_
37     {
38         struct
39         {
40             UINT16 size;
41             BYTE  buffer[sizeof(value)];
42         } t;
43         TPM2B b;
44     } TPM2B_##name##_;
45     const TPM2B_##name##_ name##_data = {{sizeof(value), {value}}}; \
46     const TPM2B* name = &name##_data.b
47
48 TPM2B_STRING(PRIMARY_OBJECT_CREATION, "Primary Object Creation");
49 TPM2B_STRING(CFB_KEY, "CFB");
50 TPM2B_STRING(CONTEXT_KEY, "CONTEXT");
51 TPM2B_STRING(INTEGRITY_KEY, "INTEGRITY");
52 TPM2B_STRING(SECRET_KEY, "SECRET");
53 TPM2B_STRING(HIERARCHY_PROOF_SECRET_LABEL, "H_PROOF_SECRET");
54 TPM2B_STRING(HIERARCHY_SEED_SECRET_LABEL, "H_SEED_SECRET");
55 TPM2B_STRING(HIERARCHY_FW_SECRET_LABEL, "H_FW_SECRET");
56 TPM2B_STRING(HIERARCHY_SVN_SECRET_LABEL, "H_SVN_SECRET");
57 TPM2B_STRING(SESSION_KEY, "ATH");
58 TPM2B_STRING(STORAGE_KEY, "STORAGE");

```

```

59 TPM2B_STRING(XOR_KEY, "XOR");
60 TPM2B_STRING(COMMIT_STRING, "ECDAA Commit");
61 TPM2B_STRING(DUPLICATE_STRING, "DUPLICATE");
62 TPM2B_STRING(IDENTITY_STRING, "IDENTITY");
63 TPM2B_STRING(OBFUSCATE_STRING, "OBFUSCATE");
64 #if SELF_TEST
65 TPM2B_STRING(OAEP_TEST_STRING, "OAEP Test Value");
66 #endif // SELF_TEST
67
68 /*** g_rcIndex[]
69 const UINT16 g_rcIndex[15] = {TPM_RC_1,
70                                TPM_RC_2,
71                                TPM_RC_3,
72                                TPM_RC_4,
73                                TPM_RC_5,
74                                TPM_RC_6,
75                                TPM_RC_7,
76                                TPM_RC_8,
77                                TPM_RC_9,
78                                TPM_RC_A,
79                                TPM_RC_B,
80                                TPM_RC_C,
81                                TPM_RC_D,
82                                TPM_RC_E,
83                                TPM_RC_F};
84
85 BOOL          g_manufactured = FALSE;

```

7.185 /tpm/src/support/Handle.c

```

1  /*** Description
2  // This file contains the functions that return the type of a handle.
3
4  /*** Includes
5  #include "Tpm.h"
6
7  /*** Functions
8
9  /*** HandleGetType()
10 // This function returns the type of a handle which is the MSO of the handle.
11 TPM_HT
12 HandleGetType(TPM_HANDLE handle // IN: a handle to be checked
13 )
14 {
15     // return the upper bytes of input data
16     return (TPM_HT)((handle & HR_RANGE_MASK) >> HR_SHIFT);
17 }
18
19 /*** NextPermanentHandle()
20 // This function returns the permanent handle that is equal to the input value or
21 // is the next higher value. If there is no handle with the input value and there
22 // is no next higher value, it returns 0:
23 TPM_HANDLE
24 NextPermanentHandle(TPM_HANDLE inHandle // IN: the handle to check
25 )
26 {
27     // If inHandle is below the start of the range of permanent handles
28     // set it to the start and scan from there
29     if(inHandle < TPM_RH_FIRST)
30         inHandle = TPM_RH_FIRST;
31     // scan from input value until we find an implemented permanent handle
32     // or go out of range
33     for(; inHandle <= TPM_RH_LAST; inHandle++)
34     {
35         // Skip over gaps in the reserved handle space.

```

```

36     if(inHandle > TPM_RH_FW_NULL && inHandle < SVN_OWNER_FIRST)
37         inHandle = SVN_OWNER_FIRST;
38     if(inHandle > SVN_OWNER_FIRST && inHandle <= SVN_OWNER_LAST)
39         inHandle = SVN_ENDORSEMENT_FIRST;
40     if(inHandle > SVN_ENDORSEMENT_FIRST && inHandle <= SVN_ENDORSEMENT_LAST)
41         inHandle = SVN_PLATFORM_FIRST;
42     if(inHandle > SVN_PLATFORM_FIRST && inHandle <= SVN_PLATFORM_LAST)
43         inHandle = SVN_NULL_FIRST;
44     if(inHandle > SVN_NULL_FIRST)
45         inHandle = TPM_RH_LAST;
46
47     switch(inHandle)
48     {
49         case TPM_RH_OWNER:
50         case TPM_RH_NULL:
51         case TPM_RS_PW:
52         case TPM_RH_LOCKOUT:
53         case TPM_RH_ENDORSEMENT:
54         case TPM_RH_PLATFORM:
55         case TPM_RH_PLATFORM_NV:
56 #if FW_LIMITED_SUPPORT
57         case TPM_RH_FW_OWNER:
58         case TPM_RH_FW_ENDORSEMENT:
59         case TPM_RH_FW_PLATFORM:
60         case TPM_RH_FW_NULL:
61 #endif
62 #if SVN_LIMITED_SUPPORT
63         case TPM_RH_SVN_OWNER_BASE:
64         case TPM_RH_SVN_ENDORSEMENT_BASE:
65         case TPM_RH_SVN_PLATFORM_BASE:
66         case TPM_RH_SVN_NULL_BASE:
67 #endif
68 #if VENDOR_PERMANENT_AUTH_ENABLED == YES
69         case VENDOR_PERMANENT_AUTH_HANDLE:
70 #endif
71 // Each of the implemented ACT
72 #define ACT_IMPLEMENTED_CASE(N) case TPM_RH_ACT_##N:
73
74         FOR_EACH_ACT(ACT_IMPLEMENTED_CASE)
75
76         return inHandle;
77         break;
78     default:
79         break;
80     }
81 }
82 // Out of range on the top
83 return 0;
84 }
85
86 /*** PermanentCapGetHandles()
87 // This function returns a list of the permanent handles of PCR, started from
88 // 'handle'. If 'handle' is larger than the largest permanent handle, an empty list
89 // will be returned with 'more' set to NO.
90 // Return Type: TPMI_YES_NO
91 //     YES     if there are more handles available
92 //     NO      all the available handles has been returned
93 TPMI_YES_NO
94 PermanentCapGetHandles(TPM_HANDLE handle, // IN: start handle
95                        UINT32 count, // IN: count of returned handles
96                        TPML_HANDLE* handleList // OUT: list of handle
97 )
98 {
99     TPMI_YES_NO more = NO;
100     UINT32 i;
101

```

```

102     pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
103
104     // Initialize output handle list
105     handleList->count = 0;
106
107     // The maximum count of handles we may return is MAX_CAP_HANDLES
108     if(count > MAX_CAP_HANDLES)
109         count = MAX_CAP_HANDLES;
110
111     // Iterate permanent handle range
112     for(i = NextPermanentHandle(handle); i != 0; i = NextPermanentHandle(i + 1))
113     {
114         if(handleList->count < count)
115         {
116             // If we have not filled up the return list, add this permanent
117             // handle to it
118             handleList->handle[handleList->count] = i;
119             handleList->count++;
120         }
121         else
122         {
123             // If the return list is full but we still have permanent handle
124             // available, report this and stop iterating
125             more = YES;
126             break;
127         }
128     }
129     return more;
130 }
131
132 /*** PermanentCapGetOneHandle()
133 // This function returns whether a permanent handle exists.
134 BOOL PermanentCapGetOneHandle(TPM_HANDLE handle) // IN: handle
135 {
136     UINT32 i;
137
138     pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
139
140     // Iterate permanent handle range
141     for(i = NextPermanentHandle(handle); i != 0; i = NextPermanentHandle(i + 1))
142     {
143         if(i == handle)
144         {
145             return TRUE;
146         }
147     }
148     return FALSE;
149 }
150
151 /*** PermanentHandleGetPolicy()
152 // This function returns a list of the permanent handles of PCR, started from
153 // 'handle'. If 'handle' is larger than the largest permanent handle, an empty list
154 // will be returned with 'more' set to NO.
155 // Return Type: TPMI_YES_NO
156 //     YES      if there are more handles available
157 //     NO      all the available handles has been returned
158 TPMI_YES_NO
159 PermanentHandleGetPolicy(TPM_HANDLE handle, // IN: start handle
160                          UINT32 count, // IN: max count of returned handles
161                          TPML_TAGGED_POLICY* policyList // OUT: list of handle
162 )
163 {
164     TPMI_YES_NO more = NO;
165
166     pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
167

```

```

168 // Initialize output handle list
169 policyList->count = 0;
170
171 // The maximum count of policies we may return is MAX_TAGGED_POLICIES
172 if(count > MAX_TAGGED_POLICIES)
173     count = MAX_TAGGED_POLICIES;
174
175 // Iterate permanent handle range
176 for(handle = NextPermanentHandle(handle); handle != 0;
177     handle = NextPermanentHandle(handle + 1))
178 {
179     TPM2B_DIGEST policyDigest;
180     TPM_ALG_ID policyAlg;
181     // Check to see if this permanent handle has a policy
182     policyAlg = EntityGetAuthPolicy(handle, &policyDigest);
183     if(policyAlg == TPM_ALG_ERROR)
184         continue;
185     if(policyList->count < count)
186     {
187         // If we have not filled up the return list, add this
188         // policy to the list;
189         policyList->policies[policyList->count].handle = handle;
190         policyList->policies[policyList->count].policyHash.hashAlg = policyAlg;
191         MemoryCopy(&policyList->policies[policyList->count].policyHash.digest,
192                 policyDigest.t.buffer,
193                 policyDigest.t.size);
194         policyList->count++;
195     }
196     else
197     {
198         // If the return list is full but we still have permanent handle
199         // available, report this and stop iterating
200         more = YES;
201         break;
202     }
203 }
204 return more;
205 }
206
207 /** PermanentHandleGetOnePolicy()
208  * This function returns a permanent handle's policy, if present.
209  * BOOL PermanentHandleGetOnePolicy(TPM_HANDLE handle, // IN: handle
210  * TPMS_TAGGED_POLICY* policy // OUT: tagged policy
211  *)
212 {
213     pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
214
215     if(NextPermanentHandle(handle) == handle)
216     {
217         TPM2B_DIGEST policyDigest;
218         TPM_ALG_ID policyAlg;
219         // Check to see if this permanent handle has a policy
220         policyAlg = EntityGetAuthPolicy(handle, &policyDigest);
221         if(policyAlg == TPM_ALG_ERROR)
222         {
223             return FALSE;
224         }
225         policy->handle = handle;
226         policy->policyHash.hashAlg = policyAlg;
227         MemoryCopy(
228             &policy->policyHash.digest, policyDigest.t.buffer, policyDigest.t.size);
229         return TRUE;
230     }
231     return FALSE;
232 }

```


7.186 /tpm/src/support/IOBuffers.c

```

1  /** Includes and Data Definitions
2
3  // This definition allows this module to "see" the values that are private
4  // to this module but kept in Global.c for ease of state migration.
5  #define IO_BUFFER_C
6  #include "Tpm.h"
7  #include "IOBuffers_fp.h"
8
9  /** Buffers and Functions
10
11 // These buffers are set aside to hold command and response values. In this
12 // implementation, it is not guaranteed that the code will stop accessing
13 // the s_actionInputBuffer before starting to put values in the
14 // s_actionOutputBuffer so different buffers are required.
15 //
16
17 /** MemoryIoBufferAllocationReset()
18 // This function is used to reset the allocation of buffers.
19 void MemoryIoBufferAllocationReset(void)
20 {
21     s_actionIoAllocation = 0;
22 }
23
24 /** MemoryIoBufferZero()
25 // Function zeros the action I/O buffer at the end of a command. Calling this is
26 // not mandatory for proper functionality.
27 void MemoryIoBufferZero(void)
28 {
29     memset(s_actionIoBuffer, 0, s_actionIoAllocation);
30 }
31
32 /** MemoryGetInBuffer()
33 // This function returns the address of the buffer into which the
34 // command parameters will be unmarshaled in preparation for calling
35 // the command actions.
36 BYTE* MemoryGetInBuffer(UINT32 size // Size, in bytes, required for the input
37                               // unmarshaling
38 )
39 {
40     pAssert(size <= sizeof(s_actionIoBuffer));
41 // In this implementation, a static buffer is set aside for the command action
42 // buffers. The buffer is shared between input and output. This is because
43 // there is no need to allocate for the worst case input and worst case output
44 // at the same time.
45 // Round size up
46 #define UoM (sizeof(s_actionIoBuffer[0]))
47     size = (size + (UoM - 1)) & (UINT32_MAX - (UoM - 1));
48     memset(s_actionIoBuffer, 0, size);
49     s_actionIoAllocation = size;
50     return (BYTE*)&s_actionIoBuffer[0];
51 }
52
53 /** MemoryGetOutBuffer()
54 // This function returns the address of the buffer into which the command
55 // action code places its output values.
56 BYTE* MemoryGetOutBuffer(UINT32 size // required size of the buffer
57 )
58 {
59     BYTE* retVal = (BYTE*)&s_actionIoBuffer[s_actionIoAllocation / UoM];
60     pAssert((size + s_actionIoAllocation) < (sizeof(s_actionIoBuffer)));
61 // In this implementation, a static buffer is set aside for the command action
62 // output buffer.
63     memset(retVal, 0, size);
64     s_actionIoAllocation += size;

```

```

65     return retVal;
66 }
67
68 /*** IsLabelProperlyFormatted()
69 // This function checks that a label is a null-terminated string.
70 // NOTE: this function is here because there was no better place for it.
71 // Return Type: BOOL
72 //     TRUE(1)         string is null terminated
73 //     FALSE(0)        string is not null terminated
74 BOOL IsLabelProperlyFormatted(TPM2B* x)
75 {
76     return ((x->size == 0) || ((x->buffer[(x->size) - 1] == 0));
77 }

```

7.187 /tpm/src/support/Locality.c

```

1  /*** Includes
2  #include "Tpm.h"
3
4  /*** LocalityGetAttributes()
5  // This function will convert a locality expressed as an integer into
6  // TPMA_LOCALITY form.
7  //
8  // The function returns the locality attribute.
9  TPMA_LOCALITY
10 LocalityGetAttributes(UINT8 locality // IN: locality value
11 )
12 {
13     TPMA_LOCALITY locality_attributes;
14     BYTE* localityAsByte = (BYTE*)&locality_attributes;
15
16     MemorySet(&locality_attributes, 0, sizeof(TPMA_LOCALITY));
17     switch(locality)
18     {
19     case 0:
20         SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ZERO);
21         break;
22     case 1:
23         SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ONE);
24         break;
25     case 2:
26         SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_TWO);
27         break;
28     case 3:
29         SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_THREE);
30         break;
31     case 4:
32         SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_FOUR);
33         break;
34     default:
35         pAssert(locality > 31);
36         *localityAsByte = locality;
37         break;
38     }
39     return locality_attributes;
40 }

```

7.188 /tpm/src/support/Manufacture.c

```

1  /*** Description
2  // This file contains the function that performs the "manufacturing" of the TPM
3  // in a simulated environment. These functions should not be used outside of
4  // a manufacturing or simulation environment.
5

```

```

6  /** Includes and Data Definitions
7  #define MANUFACTURE_C
8  #include "Tpm.h"
9  #include "TpmSizeChecks_fp.h"
10
11 /** Functions
12
13 /*** TPM_Manufacture()
14 // This function initializes the TPM values in preparation for the TPM's first
15 // use. This function will fail if previously called. The TPM can be re-manufactured
16 // by calling TPM_Teardown() first and then calling this function again.
17 // NV must be enabled first (typically with NvPowerOn() via _TPM_Init)
18 //
19 // return type: int
20 //      -2      NV System not available
21 //      -1      FAILURE - System is incorrectly compiled.
22 //      0       success
23 //      1       manufacturing process previously performed
24 LIB_EXPORT int TPM_Manufacture(
25     int firstTime // IN: indicates if this is the first call from
26                 //      main()
27 )
28 {
29     TPM_SU orderlyShutdown;
30
31 #if RUNTIME_SIZE_CHECKS
32     // Call the function to verify the sizes of values that result from different
33     // compile options.
34     if(!TpmSizeChecks())
35         return MANUF_INVALID_CONFIG;
36 #endif
37 #if LIBRARY_COMPATIBILITY_CHECK
38     // Make sure that the attached library performs as expected.
39     if(!ExtMath_Debug_CompatibilityCheck())
40         return MANUF_INVALID_CONFIG;
41 #endif
42
43     // If TPM has been manufactured, return indication.
44     if(!firstTime && g_manufactured)
45         return MANUF_ALREADY_DONE;
46
47     // trigger failure mode if called in error.
48     int nvReadyState = _plat_GetNvReadyState();
49     pAssert(nvReadyState == NV_READY); // else failure mode
50     if(nvReadyState != NV_READY)
51     {
52         return MANUF_NV_NOT_READY;
53     }
54
55     // Do power on initializations of the cryptographic libraries.
56     CryptInit();
57
58     s_DAPendingOnNV = FALSE;
59
60     // initialize NV
61     NvManufacture();
62
63     // Clear the magic value in the DRBG state
64     go.drbgState.magic = 0;
65
66     CryptStartup(SU_RESET);
67
68     // default configuration for PCR
69     PCRManufacture();
70
71     // initialize pre-installed hierarchy data

```

```

72 // This should happen after NV is initialized because hierarchy data is
73 // stored in NV.
74 HierarchyPreInstall_Init();
75
76 // initialize dictionary attack parameters
77 DAPreInstall_Init();
78
79 // initialize PP list
80 PhysicalPresencePreInstall_Init();
81
82 // initialize command audit list
83 CommandAuditPreInstall_Init();
84
85 // first start up is required to be Startup(CLEAR)
86 orderlyShutdown = TPM_SU_CLEAR;
87 NV_WRITE_PERSISTENT(orderlyState, orderlyShutdown);
88
89 // initialize the firmware version
90 gp.firmwareV1 = _plat__GetTpmFirmwareVersionHigh();
91 gp.firmwareV2 = _plat__GetTpmFirmwareVersionLow();
92
93 _plat__GetPlatformManufactureData(gp.platformReserved,
94                                 sizeof(gp.platformReserved));
95 NV_SYNC_PERSISTENT(platformReserved);
96
97 NV_SYNC_PERSISTENT(firmwareV1);
98 NV_SYNC_PERSISTENT(firmwareV2);
99
100 // initialize the total reset counter to 0
101 gp.totalResetCount = 0;
102 NV_SYNC_PERSISTENT(totalResetCount);
103
104 // initialize the clock stuff
105 go.clock = 0;
106 go.clockSafe = YES;
107
108 NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
109
110 // Commit NV writes. Manufacture process is an artificial process existing
111 // only in simulator environment and it is not defined in the specification
112 // that what should be the expected behavior if the NV write fails at this
113 // point. Therefore, it is assumed the NV write here is always success and
114 // no return code of this function is checked.
115 NvCommit();
116
117 g_manufactured = TRUE;
118
119 return MANUF_OK;
120 }
121
122 /*** TPM_TearDown()
123 // This function prepares the TPM for re-manufacture. It should not be implemented
124 // in anything other than a simulated TPM.
125 //
126 // In this implementation, all that is needs is to stop the cryptographic units
127 // and set a flag to indicate that the TPM can be re-manufactured. This should
128 // be all that is necessary to start the manufacturing process again.
129 // Return Type: int
130 //      0      success
131 //      1      TPM not previously manufactured
132 LIB_EXPORT int TPM_TearDown(void)
133 {
134     g_manufactured = FALSE;
135     _plat__TearDown();
136     return TEARDOWN_OK;
137 }

```

```

138
139 /** TpmEndSimulation()
140 // This function is called at the end of the simulation run. It is used to provoke
141 // printing of any statistics that might be needed.
142 LIB_EXPORT void TpmEndSimulation(void)
143 {
144 #if SIMULATION
145     HashLibSimulationEnd();
146     SymLibSimulationEnd();
147     MathLibSimulationEnd();
148 # if ALG_RSA
149     RsaSimulationEnd();
150 # endif
151 # if ALG_ECC
152     EccSimulationEnd();
153 # endif
154 #endif // SIMULATION
155 }

```

7.189 /tpm/src/support/Marshal.c

```

1 // FILE GENERATED BY TpmExtractCode: DO NOT EDIT
2
3 #include "Tpm.h"
4 #if !TABLE_DRIVEN_MARSHAL
5 # include "Marshal_fp.h"
6
7 // Table "Definition of Base Types" (Part 2: Structures)
8 // UINT8 definition
9 TPM_RC
10 UINT8_Unmarshal(UINT8* target, BYTE** buffer, INT32* size)
11 {
12     if((*size -= 1) < 0)
13         return TPM_RC_INSUFFICIENT;
14     *target = BYTE_ARRAY_TO_UINT8(*buffer);
15     *buffer += 1;
16     return TPM_RC_SUCCESS;
17 }
18
19 UINT16
20 UINT8_Marshal(UINT8* source, BYTE** buffer, INT32* size)
21 {
22     if(buffer != 0)
23     {
24         if((size == 0) || ((*size -= 1) >= 0))
25         {
26             UINT8_TO_BYTE_ARRAY(*source, *buffer);
27             *buffer += 1;
28         }
29         assert(size == 0 || (*size >= 0));
30     }
31     return (1);
32 }
33
34 // BYTE definition
35 # if !USE_MARSHALING_DEFINES
36 TPM_RC
37 BYTE_Unmarshal(BYTE* target, BYTE** buffer, INT32* size)
38 {
39     return UINT8_Unmarshal((UINT8*)target, buffer, size);
40 }
41
42 UINT16
43 BYTE_Marshal(BYTE* source, BYTE** buffer, INT32* size)
44 {
45     return UINT8_Marshal((UINT8*)source, buffer, size);
46 }

```

```

45 # endif // !USE_MARSHALING_DEFINES
46
47 // INT8 definition
48 # if !USE_MARSHALING_DEFINES
49 TPM_RC
50 INT8_Unmarshal(INT8* target, BYTE** buffer, INT32* size)
51 {
52     return UINT8_Unmarshal((UINT8*)target, buffer, size);
53 }
54 UINT16
55 INT8_Marshal(INT8* source, BYTE** buffer, INT32* size)
56 {
57     return UINT8_Marshal((UINT8*)source, buffer, size);
58 }
59 # endif // !USE_MARSHALING_DEFINES
60
61 // UINT16 definition
62 TPM_RC
63 UINT16_Unmarshal(UINT16* target, BYTE** buffer, INT32* size)
64 {
65     if((*size -= 2) < 0)
66         return TPM_RC_INSUFFICIENT;
67     *target = BYTE_ARRAY_TO_UINT16(*buffer);
68     *buffer += 2;
69     return TPM_RC_SUCCESS;
70 }
71 UINT16
72 UINT16_Marshal(UINT16* source, BYTE** buffer, INT32* size)
73 {
74     if(buffer != 0)
75     {
76         if((size == 0) || ((*size -= 2) >= 0))
77         {
78             UINT16_TO_BYTE_ARRAY(*source, *buffer);
79             *buffer += 2;
80         }
81         pAssert(size == 0 || (*size >= 0));
82     }
83     return (2);
84 }
85
86 // INT16 definition
87 # if !USE_MARSHALING_DEFINES
88 TPM_RC
89 INT16_Unmarshal(INT16* target, BYTE** buffer, INT32* size)
90 {
91     return UINT16_Unmarshal((UINT16*)target, buffer, size);
92 }
93 UINT16
94 INT16_Marshal(INT16* source, BYTE** buffer, INT32* size)
95 {
96     return UINT16_Marshal((UINT16*)source, buffer, size);
97 }
98 # endif // !USE_MARSHALING_DEFINES
99
100 // UINT32 definition
101 TPM_RC
102 UINT32_Unmarshal(UINT32* target, BYTE** buffer, INT32* size)
103 {
104     if((*size -= 4) < 0)
105         return TPM_RC_INSUFFICIENT;
106     *target = BYTE_ARRAY_TO_UINT32(*buffer);
107     *buffer += 4;
108     return TPM_RC_SUCCESS;
109 }
110 UINT16

```



```

111 UINT32_Marshal(UINT32* source, BYTE** buffer, INT32* size)
112 {
113     if(buffer != 0)
114     {
115         if((size == 0) || ((*size -= 4) >= 0))
116         {
117             UINT32_TO_BYTE_ARRAY(*source, *buffer);
118             *buffer += 4;
119         }
120         pAssert(size == 0 || (*size >= 0));
121     }
122     return (4);
123 }
124
125 // INT32 definition
126 # if !USE_MARSHALING_DEFINES
127 TPM_RC
128 INT32_Unmarshal(INT32* target, BYTE** buffer, INT32* size)
129 {
130     return UINT32_Unmarshal((UINT32*)target, buffer, size);
131 }
132
133 UINT16
134 INT32_Marshal(UINT32* source, BYTE** buffer, INT32* size)
135 {
136     return UINT32_Marshal((UINT32*)source, buffer, size);
137 }
138 # endif // !USE_MARSHALING_DEFINES
139
140 // UINT64 definition
141 TPM_RC
142 UINT64_Unmarshal(UINT64* target, BYTE** buffer, INT32* size)
143 {
144     if((*size -= 8) < 0)
145         return TPM_RC_INSUFFICIENT;
146     *target = BYTE_ARRAY_TO_UINT64(*buffer);
147     *buffer += 8;
148     return TPM_RC_SUCCESS;
149 }
150
151 UINT16
152 UINT64_Marshal(UINT64* source, BYTE** buffer, INT32* size)
153 {
154     if(buffer != 0)
155     {
156         if((size == 0) || ((*size -= 8) >= 0))
157         {
158             UINT64_TO_BYTE_ARRAY(*source, *buffer);
159             *buffer += 8;
160         }
161         pAssert(size == 0 || (*size >= 0));
162     }
163     return (8);
164 }
165
166 // INT64 definition
167 # if !USE_MARSHALING_DEFINES
168 TPM_RC
169 INT64_Unmarshal(INT64* target, BYTE** buffer, INT32* size)
170 {
171     return UINT64_Unmarshal((UINT64*)target, buffer, size);
172 }
173
174 UINT16
175 INT64_Marshal(INT64* source, BYTE** buffer, INT32* size)
176 {
177     return UINT64_Marshal((UINT64*)source, buffer, size);
178 }
179 # endif // !USE_MARSHALING_DEFINES

```

```

177
178 // Table "Definition of Types for Documentation Clarity" (Part 2: Structures)
179 # if !USE_MARSHALING_DEFINES
180 TPM_RC
181 TPM_ALGORITHM_ID_Unmarshal(TPM_ALGORITHM_ID* target, BYTE** buffer, INT32* size)
182 {
183     return UINT32_Unmarshal((UINT32*)target, buffer, size);
184 }
185 UINT16
186 TPM_ALGORITHM_ID_Marshal(TPM_ALGORITHM_ID* source, BYTE** buffer, INT32* size)
187 {
188     return UINT32_Marshal((UINT32*)source, buffer, size);
189 }
190 TPM_RC
191 TPM_AUTHORIZATION_SIZE_Unmarshal(
192     TPM_AUTHORIZATION_SIZE* target, BYTE** buffer, INT32* size)
193 {
194     return UINT32_Unmarshal((UINT32*)target, buffer, size);
195 }
196 UINT16
197 TPM_AUTHORIZATION_SIZE_Marshal(
198     TPM_AUTHORIZATION_SIZE* source, BYTE** buffer, INT32* size)
199 {
200     return UINT32_Marshal((UINT32*)source, buffer, size);
201 }
202 TPM_RC
203 TPM_KEY_BITS_Unmarshal(TPM_KEY_BITS* target, BYTE** buffer, INT32* size)
204 {
205     return UINT16_Unmarshal((UINT16*)target, buffer, size);
206 }
207 UINT16
208 TPM_KEY_BITS_Marshal(TPM_KEY_BITS* source, BYTE** buffer, INT32* size)
209 {
210     return UINT16_Marshal((UINT16*)source, buffer, size);
211 }
212 TPM_RC
213 TPM_KEY_SIZE_Unmarshal(TPM_KEY_SIZE* target, BYTE** buffer, INT32* size)
214 {
215     return UINT16_Unmarshal((UINT16*)target, buffer, size);
216 }
217 UINT16
218 TPM_KEY_SIZE_Marshal(TPM_KEY_SIZE* source, BYTE** buffer, INT32* size)
219 {
220     return UINT16_Marshal((UINT16*)source, buffer, size);
221 }
222 TPM_RC
223 TPM_MODIFIER_INDICATOR_Unmarshal(
224     TPM_MODIFIER_INDICATOR* target, BYTE** buffer, INT32* size)
225 {
226     return UINT32_Unmarshal((UINT32*)target, buffer, size);
227 }
228 UINT16
229 TPM_MODIFIER_INDICATOR_Marshal(
230     TPM_MODIFIER_INDICATOR* source, BYTE** buffer, INT32* size)
231 {
232     return UINT32_Marshal((UINT32*)source, buffer, size);
233 }
234 TPM_RC
235 TPM_PARAMETER_SIZE_Unmarshal(TPM_PARAMETER_SIZE* target, BYTE** buffer, INT32* size)
236 {
237     return UINT32_Unmarshal((UINT32*)target, buffer, size);
238 }
239 UINT16
240 TPM_PARAMETER_SIZE_Marshal(TPM_PARAMETER_SIZE* source, BYTE** buffer, INT32* size)
241 {
242     return UINT32_Marshal((UINT32*)source, buffer, size);

```

```

243 }
244 # endif // !USE_MARSHALING_DEFINES
245
246 // Table "Definition of TPM_CONSTANTS32 Constants" (Part 2: Structures)
247 # if !USE_MARSHALING_DEFINES
248 UINT16
249 TPM_CONSTANTS32_Marshal(TPM_CONSTANTS32* source, BYTE** buffer, INT32* size)
250 {
251     return UINT32_Marshal((UINT32*)source, buffer, size);
252 }
253 # endif // !USE_MARSHALING_DEFINES
254
255 // Table "Definition of TPM_ALG_ID Constants" (Part 2: Structures)
256 # if !USE_MARSHALING_DEFINES
257 TPM_RC
258 TPM_ALG_ID_Unmarshal(TPM_ALG_ID* target, BYTE** buffer, INT32* size)
259 {
260     return UINT16_Unmarshal((UINT16*)target, buffer, size);
261 }
262 UINT16
263 TPM_ALG_ID_Marshal(TPM_ALG_ID* source, BYTE** buffer, INT32* size)
264 {
265     return UINT16_Marshal((UINT16*)source, buffer, size);
266 }
267 # endif // !USE_MARSHALING_DEFINES
268
269 // Table "Definition of TPM_ECC_CURVE Constants" (Part 2: Structures)
270 TPM_RC
271 TPM_ECC_CURVE_Unmarshal(TPM_ECC_CURVE* target, BYTE** buffer, INT32* size)
272 {
273     TPM_RC result;
274     result = UINT16_Unmarshal((UINT16*)target, buffer, size);
275     if(result == TPM_RC_SUCCESS)
276     {
277         switch(*target)
278         {
279             # if ECC_NIST_P192
280                 case TPM_ECC_NIST_P192:
281             # endif // ECC_NIST_P192
282             # if ECC_NIST_P224
283                 case TPM_ECC_NIST_P224:
284             # endif // ECC_NIST_P224
285             # if ECC_NIST_P256
286                 case TPM_ECC_NIST_P256:
287             # endif // ECC_NIST_P256
288             # if ECC_NIST_P384
289                 case TPM_ECC_NIST_P384:
290             # endif // ECC_NIST_P384
291             # if ECC_NIST_P521
292                 case TPM_ECC_NIST_P521:
293             # endif // ECC_NIST_P521
294             # if ECC_BN_P256
295                 case TPM_ECC_BN_P256:
296             # endif // ECC_BN_P256
297             # if ECC_BN_P638
298                 case TPM_ECC_BN_P638:
299             # endif // ECC_BN_P638
300             # if ECC_SM2_P256
301                 case TPM_ECC_SM2_P256:
302             # endif // ECC_SM2_P256
303             # if ECC_BP_P256_R1
304                 case TPM_ECC_BP_P256_R1:
305             # endif // ECC_BP_P256_R1
306             # if ECC_BP_P384_R1
307                 case TPM_ECC_BP_P384_R1:
308             # endif // ECC_BP_P384_R1

```

```

309 # if ECC_BP_P512_R1
310     case TPM_ECC_BP_P512_R1:
311 # endif // ECC_BP_P512_R1
312 # if ECC_CURVE_25519
313     case TPM_ECC_CURVE_25519:
314 # endif // ECC_CURVE_25519
315 # if ECC_CURVE_448
316     case TPM_ECC_CURVE_448:
317 # endif // ECC_CURVE_448
318         break;
319     default:
320         result = TPM_RC_CURVE;
321         break;
322     }
323 }
324 return result;
325 }
326 # if !USE_MARSHALING_DEFINES
327 UINT16
328 TPM_ECC_CURVE_Marshal(TPM_ECC_CURVE* source, BYTE** buffer, INT32* size)
329 {
330     return UINT16_Marshal((UINT16*)source, buffer, size);
331 }
332 # endif // !USE_MARSHALING_DEFINES
333 // Table "Definition of TPM_CC Constants" (Part 2: Structures)
334 # if !USE_MARSHALING_DEFINES
335 TPM_RC
336 TPM_CC_Unmarshal(TPM_CC* target, BYTE** buffer, INT32* size)
337 {
338     return UINT32_Unmarshal((UINT32*)target, buffer, size);
339 }
340
341 UINT16
342 TPM_CC_Marshal(TPM_CC* source, BYTE** buffer, INT32* size)
343 {
344     return UINT32_Marshal((UINT32*)source, buffer, size);
345 }
346 # endif // !USE_MARSHALING_DEFINES
347 // Table "Definition of TPM_RC Constants" (Part 2: Structures)
348 # if !USE_MARSHALING_DEFINES
349 UINT16
350 TPM_RC_Marshal(TPM_RC* source, BYTE** buffer, INT32* size)
351 {
352     return UINT32_Marshal((UINT32*)source, buffer, size);
353 }
354 # endif // !USE_MARSHALING_DEFINES
355 // Table "Definition of TPM_CLOCK_ADJUST Constants" (Part 2: Structures)
356 TPM_RC
357 TPM_CLOCK_ADJUST_Unmarshal(TPM_CLOCK_ADJUST* target, BYTE** buffer, INT32* size)
358 {
359     TPM_RC result;
360     result = INT8_Unmarshal((INT8*)target, buffer, size);
361     if(result == TPM_RC_SUCCESS)
362     {
363         switch(*target)
364         {
365             case TPM_CLOCK_COARSE_SLOWER:
366             case TPM_CLOCK_MEDIUM_SLOWER:
367             case TPM_CLOCK_FINE_SLOWER:
368             case TPM_CLOCK_NO_CHANGE:
369             case TPM_CLOCK_FINE_FASTER:
370             case TPM_CLOCK_MEDIUM_FASTER:
371             case TPM_CLOCK_COARSE_FASTER:
372             break;

```

```

375         default:
376             result = TPM_RC_VALUE;
377             break;
378     }
379 }
380 return result;
381 }
382
383 // Table "Definition of TPM_EO Constants" (Part 2: Structures)
384 TPM_RC
385 TPM_EO_Unmarshal(TPM_EO* target, BYTE** buffer, INT32* size)
386 {
387     TPM_RC result;
388     result = UINT16_Unmarshal((UINT16*)target, buffer, size);
389     if(result == TPM_RC_SUCCESS)
390     {
391         switch(*target)
392         {
393             case TPM_EO_EQ:
394             case TPM_EO_NEQ:
395             case TPM_EO_SIGNED_GT:
396             case TPM_EO_UNSIGNED_GT:
397             case TPM_EO_SIGNED_LT:
398             case TPM_EO_UNSIGNED_LT:
399             case TPM_EO_SIGNED_GE:
400             case TPM_EO_UNSIGNED_GE:
401             case TPM_EO_SIGNED_LE:
402             case TPM_EO_UNSIGNED_LE:
403             case TPM_EO_BITSET:
404             case TPM_EO_BITCLEAR:
405                 break;
406             default:
407                 result = TPM_RC_VALUE;
408                 break;
409         }
410     }
411     return result;
412 }
413 # if !USE_MARSHALING_DEFINES
414 UINT16
415 TPM_EO_Marshal(TPM_EO* source, BYTE** buffer, INT32* size)
416 {
417     return UINT16_Marshal((UINT16*)source, buffer, size);
418 }
419 # endif // !USE_MARSHALING_DEFINES
420
421 // Table "Definition of TPM_ST Constants" (Part 2: Structures)
422 # if !USE_MARSHALING_DEFINES
423 TPM_RC
424 TPM_ST_Unmarshal(TPM_ST* target, BYTE** buffer, INT32* size)
425 {
426     return UINT16_Unmarshal((UINT16*)target, buffer, size);
427 }
428 UINT16
429 TPM_ST_Marshal(TPM_ST* source, BYTE** buffer, INT32* size)
430 {
431     return UINT16_Marshal((UINT16*)source, buffer, size);
432 }
433 # endif // !USE_MARSHALING_DEFINES
434
435 // Table "Definition of TPM_SU Constants" (Part 2: Structures)
436 TPM_RC
437 TPM_SU_Unmarshal(TPM_SU* target, BYTE** buffer, INT32* size)
438 {
439     TPM_RC result;
440     result = UINT16_Unmarshal((UINT16*)target, buffer, size);

```

```

441     if(result == TPM_RC_SUCCESS)
442     {
443         switch(*target)
444         {
445             case TPM_SU_CLEAR:
446             case TPM_SU_STATE:
447                 break;
448             default:
449                 result = TPM_RC_VALUE;
450                 break;
451         }
452     }
453     return result;
454 }
455
456 // Table "Definition of TPM_SE Constants" (Part 2: Structures)
457 TPM_RC
458 TPM_SE_Unmarshal(TPM_SE* target, BYTE** buffer, INT32* size)
459 {
460     TPM_RC result;
461     result = UINT8_Unmarshal((UINT8*)target, buffer, size);
462     if(result == TPM_RC_SUCCESS)
463     {
464         switch(*target)
465         {
466             case TPM_SE_HMAC:
467             case TPM_SE_POLICY:
468             case TPM_SE_TRIAL:
469                 break;
470             default:
471                 result = TPM_RC_VALUE;
472                 break;
473         }
474     }
475     return result;
476 }
477
478 // Table "Definition of TPM_CAP Constants" (Part 2: Structures)
479 TPM_RC
480 TPM_CAP_Unmarshal(TPM_CAP* target, BYTE** buffer, INT32* size)
481 {
482     TPM_RC result;
483     result = UINT32_Unmarshal((UINT32*)target, buffer, size);
484     if(result == TPM_RC_SUCCESS)
485     {
486         switch(*target)
487         {
488             case TPM_CAP_ALGS:
489             case TPM_CAP_HANDLES:
490             case TPM_CAP_COMMANDS:
491             case TPM_CAP_PP_COMMANDS:
492             case TPM_CAP_AUDIT_COMMANDS:
493             case TPM_CAP_PCERS:
494             case TPM_CAP_TPM_PROPERTIES:
495             case TPM_CAP_PCR_PROPERTIES:
496             # if ALG_ECC
497                 case TPM_CAP_ECC_CURVES:
498             # endif // ALG_ECC
499             case TPM_CAP_AUTH_POLICIES:
500             case TPM_CAP_ACT:
501             case TPM_CAP_VENDOR_PROPERTY:
502                 break;
503             default:
504                 result = TPM_RC_VALUE;
505                 break;
506         }

```



```

507     }
508     return result;
509 }
510 # if !USE_MARSHALING_DEFINES
511 UINT16
512 TPM_CAP_Marshal(TPM_CAP* source, BYTE** buffer, INT32* size)
513 {
514     return UINT32_Marshal((UINT32*)source, buffer, size);
515 }
516 # endif // !USE_MARSHALING_DEFINES
517
518 // Table "Definition of TPM_PT Constants" (Part 2: Structures)
519 # if !USE_MARSHALING_DEFINES
520 TPM_RC
521 TPM_PT_Unmarshal(TPM_PT* target, BYTE** buffer, INT32* size)
522 {
523     return UINT32_Unmarshal((UINT32*)target, buffer, size);
524 }
525 UINT16
526 TPM_PT_Marshal(TPM_PT* source, BYTE** buffer, INT32* size)
527 {
528     return UINT32_Marshal((UINT32*)source, buffer, size);
529 }
530 # endif // !USE_MARSHALING_DEFINES
531
532 // Table "Definition of TPM_PT_PCR Constants" (Part 2: Structures)
533 # if !USE_MARSHALING_DEFINES
534 TPM_RC
535 TPM_PT_PCR_Unmarshal(TPM_PT_PCR* target, BYTE** buffer, INT32* size)
536 {
537     return UINT32_Unmarshal((UINT32*)target, buffer, size);
538 }
539 UINT16
540 TPM_PT_PCR_Marshal(TPM_PT_PCR* source, BYTE** buffer, INT32* size)
541 {
542     return UINT32_Marshal((UINT32*)source, buffer, size);
543 }
544 # endif // !USE_MARSHALING_DEFINES
545
546 // Table "Definition of TPM_PS Constants" (Part 2: Structures)
547 # if !USE_MARSHALING_DEFINES
548 UINT16
549 TPM_PS_Marshal(TPM_PS* source, BYTE** buffer, INT32* size)
550 {
551     return UINT32_Marshal((UINT32*)source, buffer, size);
552 }
553 # endif // !USE_MARSHALING_DEFINES
554
555 // Table "Definition of Types for Handles" (Part 2: Structures)
556 # if !USE_MARSHALING_DEFINES
557 TPM_RC
558 TPM_HANDLE_Unmarshal(TPM_HANDLE* target, BYTE** buffer, INT32* size)
559 {
560     return UINT32_Unmarshal((UINT32*)target, buffer, size);
561 }
562 UINT16
563 TPM_HANDLE_Marshal(TPM_HANDLE* source, BYTE** buffer, INT32* size)
564 {
565     return UINT32_Marshal((UINT32*)source, buffer, size);
566 }
567 # endif // !USE_MARSHALING_DEFINES
568
569 // Table "Definition of TPM_HT Constants" (Part 2: Structures)
570 # if !USE_MARSHALING_DEFINES
571 TPM_RC
572 TPM_HT_Unmarshal(TPM_HT* target, BYTE** buffer, INT32* size)

```

```

573 {
574     return UINT8_Unmarshal((UINT8*)target, buffer, size);
575 }
576 UINT16
577 TPM_HT_Marshal(TPM_HT* source, BYTE** buffer, INT32* size)
578 {
579     return UINT8_Marshal((UINT8*)source, buffer, size);
580 }
581 # endif // !USE_MARSHALING_DEFINES
582
583 // Table "Definition of TPM_RH Constants" (Part 2: Structures)
584 # if !USE_MARSHALING_DEFINES
585 TPM_RC
586 TPM_RH_Unmarshal(TPM_RH* target, BYTE** buffer, INT32* size)
587 {
588     return TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
589 }
590 UINT16
591 TPM_RH_Marshal(TPM_RH* source, BYTE** buffer, INT32* size)
592 {
593     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
594 }
595 # endif // !USE_MARSHALING_DEFINES
596
597 // Table "Definition of TPM_HC Constants" (Part 2: Structures)
598 # if !USE_MARSHALING_DEFINES
599 TPM_RC
600 TPM_HC_Unmarshal(TPM_HC* target, BYTE** buffer, INT32* size)
601 {
602     return TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
603 }
604 UINT16
605 TPM_HC_Marshal(TPM_HC* source, BYTE** buffer, INT32* size)
606 {
607     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
608 }
609 # endif // !USE_MARSHALING_DEFINES
610
611 // Table "Definition of TPMA_ALGORITHM Bits" (Part 2: Structures)
612 TPM_RC
613 TPMA_ALGORITHM_Unmarshal(TPMA_ALGORITHM* target, BYTE** buffer, INT32* size)
614 {
615     TPM_RC result;
616     result = UINT32_Unmarshal((UINT32*)target, buffer, size);
617     if(result == TPM_RC_SUCCESS)
618     {
619         // check that no reserved bits are set
620         if(*(UINT32*)target & (UINT32)0xffff8f0)
621             result = TPM_RC_RESERVED_BITS;
622     }
623     return result;
624 }
625 # if !USE_MARSHALING_DEFINES
626 UINT16
627 TPMA_ALGORITHM_Marshal(TPMA_ALGORITHM* source, BYTE** buffer, INT32* size)
628 {
629     return UINT32_Marshal((UINT32*)source, buffer, size);
630 }
631 # endif // !USE_MARSHALING_DEFINES
632
633 // Table "Definition of TPMA_OBJECT Bits" (Part 2: Structures)
634 TPM_RC
635 TPMA_OBJECT_Unmarshal(TPMA_OBJECT* target, BYTE** buffer, INT32* size)
636 {
637     TPM_RC result;
638     result = UINT32_Unmarshal((UINT32*)target, buffer, size);

```

```

639     if(result == TPM_RC_SUCCESS)
640     {
641         // check that no reserved bits are set
642         if(*(UINT32*)target) & (UINT32)0xfff0f001)
643             result = TPM_RC_RESERVED_BITS;
644     }
645     return result;
646 }
647 # if !USE_MARSHALING_DEFINES
648 UINT16
649 TPMA_OBJECT_Marshal(TPMA_OBJECT* source, BYTE** buffer, INT32* size)
650 {
651     return UINT32_Marshal((UINT32*)source, buffer, size);
652 }
653 # endif // !USE_MARSHALING_DEFINES
654
655 // Table "Definition of TPMA_SESSION Bits" (Part 2: Structures)
656 TPM_RC
657 TPMA_SESSION_Unmarshal(TPMA_SESSION* target, BYTE** buffer, INT32* size)
658 {
659     TPM_RC result;
660     result = UINT8_Unmarshal((UINT8*)target, buffer, size);
661     if(result == TPM_RC_SUCCESS)
662     {
663         // check that no reserved bits are set
664         if(*(UINT8*)target) & (UINT8)0x18)
665             result = TPM_RC_RESERVED_BITS;
666     }
667     return result;
668 }
669 # if !USE_MARSHALING_DEFINES
670 UINT16
671 TPMA_SESSION_Marshal(TPMA_SESSION* source, BYTE** buffer, INT32* size)
672 {
673     return UINT8_Marshal((UINT8*)source, buffer, size);
674 }
675 # endif // !USE_MARSHALING_DEFINES
676
677 // Table "Definition of TPMA_LOCALITY Bits" (Part 2: Structures)
678 # if !USE_MARSHALING_DEFINES
679 TPM_RC
680 TPMA_LOCALITY_Unmarshal(TPMA_LOCALITY* target, BYTE** buffer, INT32* size)
681 {
682     return UINT8_Unmarshal((UINT8*)target, buffer, size);
683 }
684
685 TPMA_LOCALITY_Marshal(TPMA_LOCALITY* source, BYTE** buffer, INT32* size)
686 {
687     return UINT8_Marshal((UINT8*)source, buffer, size);
688 }
689 # endif // !USE_MARSHALING_DEFINES
690
691 // Table "Definition of TPMA_PERMANENT Bits" (Part 2: Structures)
692 # if !USE_MARSHALING_DEFINES
693 UINT16
694 TPMA_PERMANENT_Marshal(TPMA_PERMANENT* source, BYTE** buffer, INT32* size)
695 {
696     return UINT32_Marshal((UINT32*)source, buffer, size);
697 }
698 # endif // !USE_MARSHALING_DEFINES
699
700 // Table "Definition of TPMA_STARTUP_CLEAR Bits" (Part 2: Structures)
701 # if !USE_MARSHALING_DEFINES
702 UINT16
703 TPMA_STARTUP_CLEAR_Marshal(TPMA_STARTUP_CLEAR* source, BYTE** buffer, INT32* size)
704 {

```

```

705     return UINT32_Marshal((UINT32*)source, buffer, size);
706 }
707 # endif // !USE_MARSHALING_DEFINES
708
709 // Table "Definition of TPMA_MEMORY Bits" (Part 2: Structures)
710 # if !USE_MARSHALING_DEFINES
711 UINT16
712 TPMA_MEMORY_Marshal(TPMA_MEMORY* source, BYTE** buffer, INT32* size)
713 {
714     return UINT32_Marshal((UINT32*)source, buffer, size);
715 }
716 # endif // !USE_MARSHALING_DEFINES
717
718 // Table "Definition of TPMA_CC Bits" (Part 2: Structures)
719 # if !USE_MARSHALING_DEFINES
720 UINT16
721 TPMA_CC_Marshal(TPMA_CC* source, BYTE** buffer, INT32* size)
722 {
723     return UINT32_Marshal((UINT32*)source, buffer, size);
724 }
725 # endif // !USE_MARSHALING_DEFINES
726
727 // Table "Definition of TPMA_MODES Bits" (Part 2: Structures)
728 # if !USE_MARSHALING_DEFINES
729 UINT16
730 TPMA_MODES_Marshal(TPMA_MODES* source, BYTE** buffer, INT32* size)
731 {
732     return UINT32_Marshal((UINT32*)source, buffer, size);
733 }
734 # endif // !USE_MARSHALING_DEFINES
735
736 // Table "Definition of TPMA_ACT Bits" (Part 2: Structures)
737 TPM_RC
738 TPMA_ACT_Unmarshal(TPMA_ACT* target, BYTE** buffer, INT32* size)
739 {
740     TPM_RC result;
741     result = UINT32_Unmarshal((UINT32*)target, buffer, size);
742     if(result == TPM_RC_SUCCESS)
743     {
744         // check that no reserved bits are set
745         if(*(UINT32*)target & (UINT32)0xffffffff)
746             result = TPM_RC_RESERVED_BITS;
747     }
748     return result;
749 }
750 # if !USE_MARSHALING_DEFINES
751 UINT16
752 TPMA_ACT_Marshal(TPMA_ACT* source, BYTE** buffer, INT32* size)
753 {
754     return UINT32_Marshal((UINT32*)source, buffer, size);
755 }
756 # endif // !USE_MARSHALING_DEFINES
757
758 // Table "Definition of TPMI_YES_NO Type" (Part 2: Structures)
759 TPM_RC
760 TPMI_YES_NO_Unmarshal(TPMI_YES_NO* target, BYTE** buffer, INT32* size)
761 {
762     TPM_RC result;
763     result = BYTE_Unmarshal((BYTE*)target, buffer, size);
764     if(result == TPM_RC_SUCCESS)
765     {
766         switch(*target)
767         {
768             case NO:
769             case YES:
770                 break;

```

```

771         default:
772             result = TPM_RC_VALUE;
773             break;
774     }
775 }
776 return result;
777 }
778 # if !USE_MARSHALING_DEFINES
779 UINT16
780 TPMI_YES_NO_Marshal(TPMI_YES_NO* source, BYTE** buffer, INT32* size)
781 {
782     return BYTE_Marshal((BYTE*)source, buffer, size);
783 }
784 # endif // !USE_MARSHALING_DEFINES
785
786 // Table "Definition of TPMI_DH_OBJECT Type" (Part 2: Structures)
787 TPM_RC
788 TPMI_DH_OBJECT_Unmarshal(
789     TPMI_DH_OBJECT* target, BYTE** buffer, INT32* size, BOOL flag)
790 {
791     TPM_RC result;
792     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
793     if((result == TPM_RC_SUCCESS) && ((*target != TPM_RH_NULL) || !flag)
794         && ((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
795         && ((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
796         result = TPM_RC_VALUE;
797     return result;
798 }
799 # if !USE_MARSHALING_DEFINES
800 UINT16
801 TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT* source, BYTE** buffer, INT32* size)
802 {
803     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
804 }
805 # endif // !USE_MARSHALING_DEFINES
806
807 // Table "Definition of TPMI_DH_PARENT Type" (Part 2: Structures)
808 TPM_RC
809 TPMI_DH_PARENT_Unmarshal(TPMI_DH_PARENT* target, BYTE** buffer, INT32* size)
810 {
811     TPM_RC result;
812     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
813     if(result == TPM_RC_SUCCESS)
814     {
815         switch(*target)
816         {
817             case TPM_RH_OWNER:
818             case TPM_RH_ENDORSEMENT:
819             case TPM_RH_PLATFORM:
820             case TPM_RH_FW_OWNER:
821             case TPM_RH_FW_ENDORSEMENT:
822             case TPM_RH_FW_PLATFORM:
823                 break;
824             default:
825                 if((*target != TPM_RH_NULL) && (*target != TPM_RH_FW_NULL)
826                     && ((*target < SVN_NULL_FIRST) || (*target > SVN_NULL_LAST))
827                     && ((*target < SVN_OWNER_FIRST) || (*target > SVN_OWNER_LAST))
828                     && ((*target < SVN_ENDORSEMENT_FIRST)
829                         || (*target > SVN_ENDORSEMENT_LAST))
830                     && ((*target < SVN_PLATFORM_FIRST)
831                         || (*target > SVN_PLATFORM_LAST))
832                     && ((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
833                     && ((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
834                     result = TPM_RC_VALUE;
835                 break;
836         }
837     }

```

```

837     }
838     return result;
839 }
840 # if !USE_MARSHALING_DEFINES
841 UINT16
842 TPMI_DH_PARENT_Marshal(TPMI_DH_PARENT* source, BYTE** buffer, INT32* size)
843 {
844     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
845 }
846 # endif // !USE_MARSHALING_DEFINES
847
848 // Table "Definition of TPMI_DH_PERSISTENT Type" (Part 2: Structures)
849 TPM_RC
850 TPMI_DH_PERSISTENT_Unmarshal(TPMI_DH_PERSISTENT* target, BYTE** buffer, INT32* size)
851 {
852     TPM_RC result;
853     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
854     if((result == TPM_RC_SUCCESS)
855         && ((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
856         result = TPM_RC_VALUE;
857     return result;
858 }
859 # if !USE_MARSHALING_DEFINES
860 UINT16
861 TPMI_DH_PERSISTENT_Marshal(TPMI_DH_PERSISTENT* source, BYTE** buffer, INT32* size)
862 {
863     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
864 }
865 # endif // !USE_MARSHALING_DEFINES
866
867 // Table "Definition of TPMI_DH_ENTITY Type" (Part 2: Structures)
868 TPM_RC
869 TPMI_DH_ENTITY_Unmarshal(
870     TPMI_DH_ENTITY* target, BYTE** buffer, INT32* size, BOOL flag)
871 {
872     TPM_RC result;
873     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
874     if(result == TPM_RC_SUCCESS)
875     {
876         switch(*target)
877         {
878             case TPM_RH_OWNER:
879             case TPM_RH_ENDORSEMENT:
880             case TPM_RH_PLATFORM:
881             case TPM_RH_LOCKOUT:
882                 break;
883             default:
884                 if((*target != TPM_RH_NULL) || !flag)
885                     && ((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
886                     && ((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST))
887                     && ((*target < NV_INDEX_FIRST) || (*target > NV_INDEX_LAST))
888                     && (*target > PCR_LAST)
889                     && ((*target < TPM_RH_AUTH_00) || (*target > TPM_RH_AUTH_FF)))
890                     result = TPM_RC_VALUE;
891                 break;
892         }
893     }
894     return result;
895 }
896
897 // Table "Definition of TPMI_DH_PCR Type" (Part 2: Structures)
898 TPM_RC
899 TPMI_DH_PCR_Unmarshal(TPMI_DH_PCR* target, BYTE** buffer, INT32* size, BOOL flag)
900 {
901     TPM_RC result;
902     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);

```



```

903     if((result == TPM_RC_SUCCESS) && ((*target != TPM_RH_NULL) || !flag)
904         && (*target > PCR_LAST))
905         result = TPM_RC_VALUE;
906     return result;
907 }
908
909 // Table "Definition of TPMT_SH_AUTH_SESSION Type" (Part 2: Structures)
910 TPM_RC
911 TPMT_SH_AUTH_SESSION_Unmarshal(
912     TPMT_SH_AUTH_SESSION* target, BYTE** buffer, INT32* size, BOOL flag)
913 {
914     TPM_RC result;
915     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
916     if((result == TPM_RC_SUCCESS) && ((*target != TPM_RS_PW) || !flag)
917         && ((*target < HMAC_SESSION_FIRST) || (*target > HMAC_SESSION_LAST))
918         && ((*target < POLICY_SESSION_FIRST) || (*target > POLICY_SESSION_LAST)))
919         result = TPM_RC_VALUE;
920     return result;
921 }
922 # if !USE_MARSHALING_DEFINES
923 UINT16
924 TPMT_SH_AUTH_SESSION_Marshal(TPMT_SH_AUTH_SESSION* source, BYTE** buffer, INT32* size)
925 {
926     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
927 }
928 # endif // !USE_MARSHALING_DEFINES
929
930 // Table "Definition of TPMT_SH_HMAC Type" (Part 2: Structures)
931 TPM_RC
932 TPMT_SH_HMAC_Unmarshal(TPMT_SH_HMAC* target, BYTE** buffer, INT32* size)
933 {
934     TPM_RC result;
935     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
936     if((result == TPM_RC_SUCCESS)
937         && ((*target < HMAC_SESSION_FIRST) || (*target > HMAC_SESSION_LAST)))
938         result = TPM_RC_VALUE;
939     return result;
940 }
941 # if !USE_MARSHALING_DEFINES
942 UINT16
943 TPMT_SH_HMAC_Marshal(TPMT_SH_HMAC* source, BYTE** buffer, INT32* size)
944 {
945     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
946 }
947 # endif // !USE_MARSHALING_DEFINES
948
949 // Table "Definition of TPMT_SH_POLICY Type" (Part 2: Structures)
950 TPM_RC
951 TPMT_SH_POLICY_Unmarshal(TPMT_SH_POLICY* target, BYTE** buffer, INT32* size)
952 {
953     TPM_RC result;
954     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
955     if((result == TPM_RC_SUCCESS)
956         && ((*target < POLICY_SESSION_FIRST) || (*target > POLICY_SESSION_LAST)))
957         result = TPM_RC_VALUE;
958     return result;
959 }
960 # if !USE_MARSHALING_DEFINES
961 UINT16
962 TPMT_SH_POLICY_Marshal(TPMT_SH_POLICY* source, BYTE** buffer, INT32* size)
963 {
964     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
965 }
966 # endif // !USE_MARSHALING_DEFINES
967
968 // Table "Definition of TPMT_DH_CONTEXT Type" (Part 2: Structures)

```

```

969 TPM_RC
970 TPMI_DH_CONTEXT_Unmarshal(TPMI_DH_CONTEXT* target, BYTE** buffer, INT32* size)
971 {
972     TPM_RC result;
973     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
974     if((result == TPM_RC_SUCCESS)
975         && ((*target < HMAC_SESSION_FIRST) || (*target > HMAC_SESSION_LAST))
976         && ((*target < POLICY_SESSION_FIRST) || (*target > POLICY_SESSION_LAST))
977         && ((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST)))
978         result = TPM_RC_VALUE;
979     return result;
980 }
981 # if !USE_MARSHALING_DEFINES
982 UINT16
983 TPMI_DH_CONTEXT_Marshal(TPMI_DH_CONTEXT* source, BYTE** buffer, INT32* size)
984 {
985     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
986 }
987 # endif // !USE_MARSHALING_DEFINES
988
989 // Table "Definition of TPMI_DH_SAVED Type" (Part 2: Structures)
990 TPM_RC
991 TPMI_DH_SAVED_Unmarshal(TPMI_DH_SAVED* target, BYTE** buffer, INT32* size)
992 {
993     TPM_RC result;
994     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
995     if(result == TPM_RC_SUCCESS)
996     {
997         switch(*target)
998         {
999             case 0x80000000:
1000             case 0x80000001:
1001             case 0x80000002:
1002                 break;
1003             default:
1004                 if(((*target < HMAC_SESSION_FIRST) || (*target > HMAC_SESSION_LAST))
1005                     && ((*target < POLICY_SESSION_FIRST)
1006                         || (*target > POLICY_SESSION_LAST)))
1007                     result = TPM_RC_VALUE;
1008                 break;
1009         }
1010     }
1011     return result;
1012 }
1013 # if !USE_MARSHALING_DEFINES
1014 UINT16
1015 TPMI_DH_SAVED_Marshal(TPMI_DH_SAVED* source, BYTE** buffer, INT32* size)
1016 {
1017     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
1018 }
1019 # endif // !USE_MARSHALING_DEFINES
1020
1021 // Table "Definition of TPMI_RH_HIERARCHY Type" (Part 2: Structures)
1022 TPM_RC
1023 TPMI_RH_HIERARCHY_Unmarshal(TPMI_RH_HIERARCHY* target, BYTE** buffer, INT32* size)
1024 {
1025     TPM_RC result;
1026     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1027     if(result == TPM_RC_SUCCESS)
1028     {
1029         switch(*target)
1030         {
1031             case TPM_RH_OWNER:
1032             case TPM_RH_ENDORSEMENT:
1033             case TPM_RH_PLATFORM:
1034             case TPM_RH_FW_OWNER:

```

```

1035         case TPM_RH_FW_ENDORSEMENT:
1036         case TPM_RH_FW_PLATFORM:
1037             break;
1038         default:
1039             if ((*target != TPM_RH_NULL) && (*target != TPM_RH_FW_NULL)
1040                 && (*target < SVN_NULL_FIRST) || (*target > SVN_NULL_LAST))
1041                 && (*target < SVN_OWNER_FIRST) || (*target > SVN_OWNER_LAST))
1042                 && (*target < SVN_ENDORSEMENT_FIRST)
1043                     || (*target > SVN_ENDORSEMENT_LAST))
1044                 && (*target < SVN_PLATFORM_FIRST)
1045                     || (*target > SVN_PLATFORM_LAST)))
1046                 result = TPM_RC_VALUE;
1047             break;
1048     }
1049 }
1050 return result;
1051 }
1052 # if !USE_MARSHALING_DEFINES
1053 UINT16
1054 TPMI_RH_HIERARCHY_Marshal(TPMI_RH_HIERARCHY* source, BYTE** buffer, INT32* size)
1055 {
1056     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
1057 }
1058 # endif // !USE_MARSHALING_DEFINES
1059 // Table "Definition of TPMI_RH_ENABLES Type" (Part 2: Structures)
1060 TPM_RC
1061 TPMI_RH_ENABLES_Unmarshal(
1062     TPMI_RH_ENABLES* target, BYTE** buffer, INT32* size, BOOL flag)
1063 {
1064     TPM_RC result;
1065     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1066     if(result == TPM_RC_SUCCESS)
1067     {
1068         switch(*target)
1069         {
1070             case TPM_RH_OWNER:
1071             case TPM_RH_ENDORSEMENT:
1072             case TPM_RH_PLATFORM:
1073             case TPM_RH_PLATFORM_NV:
1074                 break;
1075             default:
1076                 if ((*target != TPM_RH_NULL) || !flag)
1077                     result = TPM_RC_VALUE;
1078                 break;
1079         }
1080     }
1081 }
1082 return result;
1083 }
1084 # if !USE_MARSHALING_DEFINES
1085 UINT16
1086 TPMI_RH_ENABLES_Marshal(TPMI_RH_ENABLES* source, BYTE** buffer, INT32* size)
1087 {
1088     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
1089 }
1090 # endif // !USE_MARSHALING_DEFINES
1091 // Table "Definition of TPMI_RH_HIERARCHY_AUTH Type" (Part 2: Structures)
1092 TPM_RC
1093 TPMI_RH_HIERARCHY_AUTH_Unmarshal(
1094     TPMI_RH_HIERARCHY_AUTH* target, BYTE** buffer, INT32* size)
1095 {
1096     TPM_RC result;
1097     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1098     if(result == TPM_RC_SUCCESS)
1099     {
1100

```

```

1101     switch(*target)
1102     {
1103         case TPM_RH_OWNER:
1104         case TPM_RH_ENDORSEMENT:
1105         case TPM_RH_PLATFORM:
1106         case TPM_RH_LOCKOUT:
1107             break;
1108         default:
1109             result = TPM_RC_VALUE;
1110             break;
1111     }
1112 }
1113 return result;
1114 }
1115
1116 // Table "Definition of TPMI_RH_HIERARCHY_POLICY Type" (Part 2: Structures)
1117 TPM_RC
1118 TPMI_RH_HIERARCHY_POLICY_Unmarshal(
1119     TPMI_RH_HIERARCHY_POLICY* target, BYTE** buffer, INT32* size)
1120 {
1121     TPM_RC result;
1122     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1123     if(result == TPM_RC_SUCCESS)
1124     {
1125         switch(*target)
1126         {
1127             case TPM_RH_OWNER:
1128             case TPM_RH_ENDORSEMENT:
1129             case TPM_RH_PLATFORM:
1130             case TPM_RH_LOCKOUT:
1131                 break;
1132             default:
1133                 if((*target < TPM_RH_ACT_0) || (*target > TPM_RH_ACT_F))
1134                     result = TPM_RC_VALUE;
1135                 break;
1136         }
1137     }
1138     return result;
1139 }
1140
1141 // Table "Definition of TPMI_RH_BASE_HIERARCHY Type" (Part 2: Structures)
1142 TPM_RC
1143 TPMI_RH_BASE_HIERARCHY_Unmarshal(
1144     TPMI_RH_BASE_HIERARCHY* target, BYTE** buffer, INT32* size)
1145 {
1146     TPM_RC result;
1147     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1148     if(result == TPM_RC_SUCCESS)
1149     {
1150         switch(*target)
1151         {
1152             case TPM_RH_OWNER:
1153             case TPM_RH_ENDORSEMENT:
1154             case TPM_RH_PLATFORM:
1155                 break;
1156             default:
1157                 result = TPM_RC_VALUE;
1158                 break;
1159         }
1160     }
1161     return result;
1162 }
1163 # if !USE_MARSHALING_DEFINES
1164 UINT16
1165 TPMI_RH_BASE_HIERARCHY_Marshal(
1166     TPMI_RH_BASE_HIERARCHY* source, BYTE** buffer, INT32* size)

```

```

1167 {
1168     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
1169 }
1170 # endif // !USE_MARSHALING_DEFINES
1171
1172 // Table "Definition of TPMI_RH_PLATFORM Type" (Part 2: Structures)
1173 TPM_RC
1174 TPMI_RH_PLATFORM_Unmarshal(TPMI_RH_PLATFORM* target, BYTE** buffer, INT32* size)
1175 {
1176     TPM_RC result;
1177     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1178     if(result == TPM_RC_SUCCESS)
1179     {
1180         switch(*target)
1181         {
1182             case TPM_RH_PLATFORM:
1183                 break;
1184             default:
1185                 result = TPM_RC_VALUE;
1186                 break;
1187         }
1188     }
1189     return result;
1190 }
1191
1192 // Table "Definition of TPMI_RH_OWNER Type" (Part 2: Structures)
1193 TPM_RC
1194 TPMI_RH_OWNER_Unmarshal(TPMI_RH_OWNER* target, BYTE** buffer, INT32* size, BOOL flag)
1195 {
1196     TPM_RC result;
1197     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1198     if(result == TPM_RC_SUCCESS)
1199     {
1200         switch(*target)
1201         {
1202             case TPM_RH_OWNER:
1203                 break;
1204             default:
1205                 if((*target != TPM_RH_NULL) || !flag)
1206                     result = TPM_RC_VALUE;
1207                 break;
1208         }
1209     }
1210     return result;
1211 }
1212
1213 // Table "Definition of TPMI_RH_ENDORSEMENT Type" (Part 2: Structures)
1214 TPM_RC
1215 TPMI_RH_ENDORSEMENT_Unmarshal(
1216     TPMI_RH_ENDORSEMENT* target, BYTE** buffer, INT32* size, BOOL flag)
1217 {
1218     TPM_RC result;
1219     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1220     if(result == TPM_RC_SUCCESS)
1221     {
1222         switch(*target)
1223         {
1224             case TPM_RH_ENDORSEMENT:
1225                 break;
1226             default:
1227                 if((*target != TPM_RH_NULL) || !flag)
1228                     result = TPM_RC_VALUE;
1229                 break;
1230         }
1231     }
1232     return result;

```

```

1233 }
1234
1235 // Table "Definition of TPMI_RH_PROVISION Type" (Part 2: Structures)
1236 TPM_RC
1237 TPMI_RH_PROVISION_Unmarshal(TPMI_RH_PROVISION* target, BYTE** buffer, INT32* size)
1238 {
1239     TPM_RC result;
1240     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1241     if(result == TPM_RC_SUCCESS)
1242     {
1243         switch(*target)
1244         {
1245             case TPM_RH_OWNER:
1246             case TPM_RH_PLATFORM:
1247                 break;
1248             default:
1249                 result = TPM_RC_VALUE;
1250                 break;
1251         }
1252     }
1253     return result;
1254 }
1255
1256 // Table "Definition of TPMI_RH_CLEAR Type" (Part 2: Structures)
1257 TPM_RC
1258 TPMI_RH_CLEAR_Unmarshal(TPMI_RH_CLEAR* target, BYTE** buffer, INT32* size)
1259 {
1260     TPM_RC result;
1261     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1262     if(result == TPM_RC_SUCCESS)
1263     {
1264         switch(*target)
1265         {
1266             case TPM_RH_PLATFORM:
1267             case TPM_RH_LOCKOUT:
1268                 break;
1269             default:
1270                 result = TPM_RC_VALUE;
1271                 break;
1272         }
1273     }
1274     return result;
1275 }
1276
1277 // Table "Definition of TPMI_RH_NV_AUTH Type" (Part 2: Structures)
1278 TPM_RC
1279 TPMI_RH_NV_AUTH_Unmarshal(TPMI_RH_NV_AUTH* target, BYTE** buffer, INT32* size)
1280 {
1281     TPM_RC result;
1282     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1283     if(result == TPM_RC_SUCCESS)
1284     {
1285         switch(*target)
1286         {
1287             case TPM_RH_OWNER:
1288             case TPM_RH_PLATFORM:
1289                 break;
1290             default:
1291                 if((*target < NV_INDEX_FIRST) || (*target > NV_INDEX_LAST))
1292                     result = TPM_RC_VALUE;
1293                 break;
1294         }
1295     }
1296     return result;
1297 }
1298

```



```

1299 // Table "Definition of TPMI_RH_LOCKOUT Type" (Part 2: Structures)
1300 TPM_RC
1301 TPMI_RH_LOCKOUT_Unmarshal(TPMI_RH_LOCKOUT* target, BYTE** buffer, INT32* size)
1302 {
1303     TPM_RC result;
1304     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1305     if(result == TPM_RC_SUCCESS)
1306     {
1307         switch(*target)
1308         {
1309             case TPM_RH_LOCKOUT:
1310                 break;
1311             default:
1312                 result = TPM_RC_VALUE;
1313                 break;
1314         }
1315     }
1316     return result;
1317 }
1318
1319 // Table "Definition of TPMI_RH_NV_INDEX Type" (Part 2: Structures)
1320 TPM_RC
1321 TPMI_RH_NV_INDEX_Unmarshal(TPMI_RH_NV_INDEX* target, BYTE** buffer, INT32* size)
1322 {
1323     TPM_RC result;
1324     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1325     if((result == TPM_RC_SUCCESS)
1326         && ((*target < NV_INDEX_FIRST) || (*target > NV_INDEX_LAST))
1327         && ((*target < EXTERNAL_NV_FIRST) || (*target > EXTERNAL_NV_LAST))
1328         && ((*target < PERMANENT_NV_FIRST) || (*target > PERMANENT_NV_LAST)))
1329         result = TPM_RC_VALUE;
1330     return result;
1331 }
1332 # if !USE_MARSHALING_DEFINES
1333 UINT16
1334 TPMI_RH_NV_INDEX_Marshal(TPMI_RH_NV_INDEX* source, BYTE** buffer, INT32* size)
1335 {
1336     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
1337 }
1338 # endif // !USE_MARSHALING_DEFINES
1339
1340 // Table "Definition of TPMI_RH_NV_DEFINED_INDEX Type" (Part 2: Structures)
1341 TPM_RC
1342 TPMI_RH_NV_DEFINED_INDEX_Unmarshal(
1343     TPMI_RH_NV_DEFINED_INDEX* target, BYTE** buffer, INT32* size)
1344 {
1345     TPM_RC result;
1346     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1347     if((result == TPM_RC_SUCCESS)
1348         && ((*target < NV_INDEX_FIRST) || (*target > NV_INDEX_LAST))
1349         && ((*target < EXTERNAL_NV_FIRST) || (*target > EXTERNAL_NV_LAST)))
1350         result = TPM_RC_VALUE;
1351     return result;
1352 }
1353
1354 // Table "Definition of TPMI_RH_NV_LEGACY_INDEX Type" (Part 2: Structures)
1355 TPM_RC
1356 TPMI_RH_NV_LEGACY_INDEX_Unmarshal(
1357     TPMI_RH_NV_LEGACY_INDEX* target, BYTE** buffer, INT32* size)
1358 {
1359     TPM_RC result;
1360     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1361     if((result == TPM_RC_SUCCESS)
1362         && ((*target < NV_INDEX_FIRST) || (*target > NV_INDEX_LAST)))
1363         result = TPM_RC_VALUE;
1364     return result;

```

```

1365 }
1366 # if !USE_MARSHALING_DEFINES
1367 UINT16
1368 TPMI_RH_NV_LEGACY_INDEX_Marshal(
1369     TPMI_RH_NV_LEGACY_INDEX* source, BYTE** buffer, INT32* size)
1370 {
1371     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
1372 }
1373 # endif // !USE_MARSHALING_DEFINES
1374
1375 // Table "Definition of TPMI_RH_NV_EXP_INDEX Type" (Part 2: Structures)
1376 TPM_RC
1377 TPMI_RH_NV_EXP_INDEX_Unmarshal(
1378     TPMI_RH_NV_EXP_INDEX* target, BYTE** buffer, INT32* size)
1379 {
1380     TPM_RC result;
1381     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1382     if((result == TPM_RC_SUCCESS)
1383         && ((*target < EXTERNAL_NV_FIRST) || (*target > EXTERNAL_NV_LAST)))
1384         result = TPM_RC_VALUE;
1385     return result;
1386 }
1387 # if !USE_MARSHALING_DEFINES
1388 UINT16
1389 TPMI_RH_NV_EXP_INDEX_Marshal(TPMI_RH_NV_EXP_INDEX* source, BYTE** buffer, INT32* size)
1390 {
1391     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
1392 }
1393 # endif // !USE_MARSHALING_DEFINES
1394
1395 // Table "Definition of TPMI_RH_AC Type" (Part 2: Structures)
1396 TPM_RC
1397 TPMI_RH_AC_Unmarshal(TPMI_RH_AC* target, BYTE** buffer, INT32* size)
1398 {
1399     TPM_RC result;
1400     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1401     if((result == TPM_RC_SUCCESS) && ((*target < AC_FIRST) || (*target > AC_LAST)))
1402         result = TPM_RC_VALUE;
1403     return result;
1404 }
1405
1406 // Table "Definition of TPMI_RH_ACT Type" (Part 2: Structures)
1407 TPM_RC
1408 TPMI_RH_ACT_Unmarshal(TPMI_RH_ACT* target, BYTE** buffer, INT32* size)
1409 {
1410     TPM_RC result;
1411     result = TPM_HANDLE_Unmarshal((TPM_HANDLE*)target, buffer, size);
1412     if((result == TPM_RC_SUCCESS)
1413         && ((*target < TPM_RH_ACT_0) || (*target > TPM_RH_ACT_F)))
1414         result = TPM_RC_VALUE;
1415     return result;
1416 }
1417 # if !USE_MARSHALING_DEFINES
1418 UINT16
1419 TPMI_RH_ACT_Marshal(TPMI_RH_ACT* source, BYTE** buffer, INT32* size)
1420 {
1421     return TPM_HANDLE_Marshal((TPM_HANDLE*)source, buffer, size);
1422 }
1423 # endif // !USE_MARSHALING_DEFINES
1424
1425 // Table "Definition of TPMI_ALG_HASH Type" (Part 2: Structures)
1426 TPM_RC
1427 TPMI_ALG_HASH_Unmarshal(TPMI_ALG_HASH* target, BYTE** buffer, INT32* size, BOOL flag)
1428 {
1429     TPM_RC result;
1430     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);

```

```

1431     if(result == TPM_RC_SUCCESS)
1432     {
1433         switch(*target)
1434         {
1435 # if ALG_SHA1
1436             case TPM_ALG_SHA1:
1437 # endif // ALG_SHA1
1438 # if ALG_SHA256
1439             case TPM_ALG_SHA256:
1440 # endif // ALG_SHA256
1441 # if ALG_SHA384
1442             case TPM_ALG_SHA384:
1443 # endif // ALG_SHA384
1444 # if ALG_SHA512
1445             case TPM_ALG_SHA512:
1446 # endif // ALG_SHA512
1447 # if ALG_SHA256_192
1448             case TPM_ALG_SHA256_192:
1449 # endif // ALG_SHA256_192
1450 # if ALG_SM3_256
1451             case TPM_ALG_SM3_256:
1452 # endif // ALG_SM3_256
1453 # if ALG_SHA3_256
1454             case TPM_ALG_SHA3_256:
1455 # endif // ALG_SHA3_256
1456 # if ALG_SHA3_384
1457             case TPM_ALG_SHA3_384:
1458 # endif // ALG_SHA3_384
1459 # if ALG_SHA3_512
1460             case TPM_ALG_SHA3_512:
1461 # endif // ALG_SHA3_512
1462 # if ALG_SHAKE256_192
1463             case TPM_ALG_SHAKE256_192:
1464 # endif // ALG_SHAKE256_192
1465 # if ALG_SHAKE256_256
1466             case TPM_ALG_SHAKE256_256:
1467 # endif // ALG_SHAKE256_256
1468 # if ALG_SHAKE256_512
1469             case TPM_ALG_SHAKE256_512:
1470 # endif // ALG_SHAKE256_512
1471             break;
1472         default:
1473             if((*target != TPM_ALG_NULL) || !flag)
1474                 result = TPM_RC_HASH;
1475             break;
1476         }
1477     }
1478     return result;
1479 }
1480 # if !USE_MARSHALING_DEFINES
1481 UINT16
1482 TPMI_ALG_HASH_Marshal(TPMI_ALG_HASH* source, BYTE** buffer, INT32* size)
1483 {
1484     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1485 }
1486 # endif // !USE_MARSHALING_DEFINES
1487
1488 // Table "Definition of TPMI_ALG_ASYM Type" (Part 2: Structures)
1489 TPM_RC
1490 TPMI_ALG_ASYM_Unmarshal(TPMI_ALG_ASYM* target, BYTE** buffer, INT32* size, BOOL flag)
1491 {
1492     TPM_RC result;
1493     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
1494     if(result == TPM_RC_SUCCESS)
1495     {
1496         switch(*target)

```

```

1497     {
1498 #   if ALG_RSA
1499         case TPM_ALG_RSA:
1500 #   endif // ALG_RSA
1501 #   if ALG_ECC
1502         case TPM_ALG_ECC:
1503 #   endif // ALG_ECC
1504         break;
1505     default:
1506         if ((*target != TPM_ALG_NULL) || !flag)
1507             result = TPM_RC_ASYMMETRIC;
1508         break;
1509     }
1510 }
1511 return result;
1512 }
1513 # if !USE_MARSHALING_DEFINES
1514 UINT16
1515 TPMI_ALG_ASYM_Marshal(TPMI_ALG_ASYM* source, BYTE** buffer, INT32* size)
1516 {
1517     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1518 }
1519 # endif // !USE_MARSHALING_DEFINES
1520
1521 // Table "Definition of TPMI_ALG_SYM Type" (Part 2: Structures)
1522 TPM_RC
1523 TPMI_ALG_SYM_Unmarshal(TPMI_ALG_SYM* target, BYTE** buffer, INT32* size, BOOL flag)
1524 {
1525     TPM_RC result;
1526     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
1527     if (result == TPM_RC_SUCCESS)
1528     {
1529         switch(*target)
1530         {
1531 #   if ALG_AES
1532             case TPM_ALG_AES:
1533 #   endif // ALG_AES
1534 #   if ALG_XOR
1535             case TPM_ALG_XOR:
1536 #   endif // ALG_XOR
1537 #   if ALG_SM4
1538             case TPM_ALG_SM4:
1539 #   endif // ALG_SM4
1540 #   if ALG_CAMELLIA
1541             case TPM_ALG_CAMELLIA:
1542 #   endif // ALG_CAMELLIA
1543             break;
1544         default:
1545             if ((*target != TPM_ALG_NULL) || !flag)
1546                 result = TPM_RC_SYMMETRIC;
1547             break;
1548         }
1549     }
1550     return result;
1551 }
1552 # if !USE_MARSHALING_DEFINES
1553 UINT16
1554 TPMI_ALG_SYM_Marshal(TPMI_ALG_SYM* source, BYTE** buffer, INT32* size)
1555 {
1556     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1557 }
1558 # endif // !USE_MARSHALING_DEFINES
1559
1560 // Table "Definition of TPMI_ALG_SYM_OBJECT Type" (Part 2: Structures)
1561 TPM_RC
1562 TPMI_ALG_SYM_OBJECT_Unmarshal(

```

```

1563     TPMI_ALG_SYM_OBJECT* target, BYTE** buffer, INT32* size, BOOL flag)
1564 {
1565     TPM_RC result;
1566     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
1567     if(result == TPM_RC_SUCCESS)
1568     {
1569         switch(*target)
1570         {
1571             # if ALG_AES
1572             case TPM_ALG_AES:
1573             # endif // ALG_AES
1574             # if ALG_SM4
1575             case TPM_ALG_SM4:
1576             # endif // ALG_SM4
1577             # if ALG_CAMELLIA
1578             case TPM_ALG_CAMELLIA:
1579             # endif // ALG_CAMELLIA
1580                 break;
1581             default:
1582                 if((*target != TPM_ALG_NULL) || !flag)
1583                     result = TPM_RC_SYMMETRIC;
1584                 break;
1585         }
1586     }
1587     return result;
1588 }
1589 # if !USE_MARSHALING_DEFINES
1590 UINT16
1591 TPMI_ALG_SYM_OBJECT_Marshal(TPMI_ALG_SYM_OBJECT* source, BYTE** buffer, INT32* size)
1592 {
1593     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1594 }
1595 # endif // !USE_MARSHALING_DEFINES
1596
1597 // Table "Definition of TPMI_ALG_SYM_MODE Type" (Part 2: Structures)
1598 TPM_RC
1599 TPMI_ALG_SYM_MODE_Unmarshal(
1600     TPMI_ALG_SYM_MODE* target, BYTE** buffer, INT32* size, BOOL flag)
1601 {
1602     TPM_RC result;
1603     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
1604     if(result == TPM_RC_SUCCESS)
1605     {
1606         switch(*target)
1607         {
1608             # if ALG_CMAC
1609             case TPM_ALG_CMAC:
1610             # endif // ALG_CMAC
1611             # if ALG_CTR
1612             case TPM_ALG_CTR:
1613             # endif // ALG_CTR
1614             # if ALG_OFB
1615             case TPM_ALG_OFB:
1616             # endif // ALG_OFB
1617             # if ALG_CBC
1618             case TPM_ALG_CBC:
1619             # endif // ALG_CBC
1620             # if ALG_CFB
1621             case TPM_ALG_CFB:
1622             # endif // ALG_CFB
1623             # if ALG_ECB
1624             case TPM_ALG_ECB:
1625             # endif // ALG_ECB
1626                 break;
1627             default:
1628                 if((*target != TPM_ALG_NULL) || !flag)

```

```

1629         result = TPM_RC_MODE;
1630         break;
1631     }
1632 }
1633 return result;
1634 }
1635 # if !USE_MARSHALING_DEFINES
1636 UINT16
1637 TPMI_ALG_SYM_MODE_Marshal(TPMI_ALG_SYM_MODE* source, BYTE** buffer, INT32* size)
1638 {
1639     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1640 }
1641 # endif // !USE_MARSHALING_DEFINES
1642
1643 // Table "Definition of TPMI_ALG_KDF Type" (Part 2: Structures)
1644 TPM_RC
1645 TPMI_ALG_KDF_Unmarshal(TPMI_ALG_KDF* target, BYTE** buffer, INT32* size, BOOL flag)
1646 {
1647     TPM_RC result;
1648     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
1649     if(result == TPM_RC_SUCCESS)
1650     {
1651         switch(*target)
1652         {
1653             # if ALG_MGF1
1654             case TPM_ALG_MGF1:
1655             # endif // ALG_MGF1
1656             # if ALG_KDF1_SP800_56A
1657             case TPM_ALG_KDF1_SP800_56A:
1658             # endif // ALG_KDF1_SP800_56A
1659             # if ALG_KDF2
1660             case TPM_ALG_KDF2:
1661             # endif // ALG_KDF2
1662             # if ALG_KDF1_SP800_108
1663             case TPM_ALG_KDF1_SP800_108:
1664             # endif // ALG_KDF1_SP800_108
1665             break;
1666             default:
1667                 if((*target != TPM_ALG_NULL) || !flag)
1668                     result = TPM_RC_KDF;
1669                 break;
1670         }
1671     }
1672     return result;
1673 }
1674 # if !USE_MARSHALING_DEFINES
1675 UINT16
1676 TPMI_ALG_KDF_Marshal(TPMI_ALG_KDF* source, BYTE** buffer, INT32* size)
1677 {
1678     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1679 }
1680 # endif // !USE_MARSHALING_DEFINES
1681
1682 // Table "Definition of TPMI_ALG_SIG_SCHEME Type" (Part 2: Structures)
1683 TPM_RC
1684 TPMI_ALG_SIG_SCHEME_Unmarshal(
1685     TPMI_ALG_SIG_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
1686 {
1687     TPM_RC result;
1688     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
1689     if(result == TPM_RC_SUCCESS)
1690     {
1691         switch(*target)
1692         {
1693             # if ALG_HMAC
1694             case TPM_ALG_HMAC:

```



```

1695 # endif // ALG_HMAC
1696 # if ALG_RSASSA
1697     case TPM_ALG_RSASSA:
1698 # endif // ALG_RSASSA
1699 # if ALG_RSAPSS
1700     case TPM_ALG_RSAPSS:
1701 # endif // ALG_RSAPSS
1702 # if ALG_ECDSA
1703     case TPM_ALG_ECDSA:
1704 # endif // ALG_ECDSA
1705 # if ALG_ECDAA
1706     case TPM_ALG_ECDAA:
1707 # endif // ALG_ECDAA
1708 # if ALG_SM2
1709     case TPM_ALG_SM2:
1710 # endif // ALG_SM2
1711 # if ALG_ECSCNORR
1712     case TPM_ALG_ECSCNORR:
1713 # endif // ALG_ECSCNORR
1714 # if ALG_EDDSA
1715     case TPM_ALG_EDDSA:
1716 # endif // ALG_EDDSA
1717 # if ALG_EDDSA_PH
1718     case TPM_ALG_EDDSA_PH:
1719 # endif // ALG_EDDSA_PH
1720 # if ALG_LMS
1721     case TPM_ALG_LMS:
1722 # endif // ALG_LMS
1723 # if ALG_XMSS
1724     case TPM_ALG_XMSS:
1725 # endif // ALG_XMSS
1726     break;
1727     default:
1728         if ((*target != TPM_ALG_NULL) || !flag)
1729             result = TPM_RC_SCHEME;
1730         break;
1731     }
1732 }
1733 return result;
1734 }
1735 # if !USE_MARSHALING_DEFINES
1736 UINT16
1737 TPMI_ALG_SIG_SCHEME_Marshal(TPMI_ALG_SIG_SCHEME* source, BYTE** buffer, INT32* size)
1738 {
1739     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1740 }
1741 # endif // !USE_MARSHALING_DEFINES
1742
1743 // Table "Definition of TPMI_ECC_KEY_EXCHANGE Type" (Part 2: Structures)
1744 TPM_RC
1745 TPMI_ECC_KEY_EXCHANGE_Unmarshal(
1746     TPMI_ECC_KEY_EXCHANGE* target, BYTE** buffer, INT32* size, BOOL flag)
1747 {
1748     TPM_RC result;
1749     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
1750     if (result == TPM_RC_SUCCESS)
1751     {
1752         switch(*target)
1753         {
1754 # if ALG_ECDH
1755             case TPM_ALG_ECDH:
1756 # endif // ALG_ECDH
1757 # if ALG_SM2
1758             case TPM_ALG_SM2:
1759 # endif // ALG_SM2
1760 # if ALG_ECMQV

```

```

1761         case TPM_ALG_ECMQV:
1762 # endif // ALG_ECMQV
1763         break;
1764         default:
1765             if((*target != TPM_ALG_NULL) || !flag)
1766                 result = TPM_RC_SCHEME;
1767             break;
1768     }
1769 }
1770 return result;
1771 }
1772 # if !USE_MARSHALING_DEFINES
1773 UINT16
1774 TPMI_ECC_KEY_EXCHANGE_Marshal(
1775     TPMI_ECC_KEY_EXCHANGE* source, BYTE** buffer, INT32* size)
1776 {
1777     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1778 }
1779 # endif // !USE_MARSHALING_DEFINES
1780
1781 // Table "Definition of TPMI_ST_COMMAND_TAG Type" (Part 2: Structures)
1782 TPM_RC
1783 TPMI_ST_COMMAND_TAG_Unmarshal(TPMI_ST_COMMAND_TAG* target, BYTE** buffer, INT32* size)
1784 {
1785     TPM_RC result;
1786     result = TPM_ST_Unmarshal((TPM_ST*)target, buffer, size);
1787     if(result == TPM_RC_SUCCESS)
1788     {
1789         switch(*target)
1790         {
1791             case TPM_ST_NO_SESSIONS:
1792             case TPM_ST_SESSIONS:
1793                 break;
1794             default:
1795                 result = TPM_RC_BAD_TAG;
1796                 break;
1797         }
1798     }
1799     return result;
1800 }
1801 # if !USE_MARSHALING_DEFINES
1802 UINT16
1803 TPMI_ST_COMMAND_TAG_Marshal(TPMI_ST_COMMAND_TAG* source, BYTE** buffer, INT32* size)
1804 {
1805     return TPM_ST_Marshal((TPM_ST*)source, buffer, size);
1806 }
1807 # endif // !USE_MARSHALING_DEFINES
1808
1809 // Table "Definition of TPMI_ALG_MAC_SCHEME Type" (Part 2: Structures)
1810 TPM_RC
1811 TPMI_ALG_MAC_SCHEME_Unmarshal(
1812     TPMI_ALG_MAC_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
1813 {
1814     TPM_RC result;
1815     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
1816     if(result == TPM_RC_SUCCESS)
1817     {
1818         switch(*target)
1819         {
1820 # if ALG_SHA1
1821             case TPM_ALG_SHA1:
1822 # endif // ALG_SHA1
1823 # if ALG_SHA256
1824             case TPM_ALG_SHA256:
1825 # endif // ALG_SHA256
1826 # if ALG_SHA384

```

```

1827         case TPM_ALG_SHA384:
1828     # endif // ALG_SHA384
1829     # if ALG_SHA512
1830         case TPM_ALG_SHA512:
1831     # endif // ALG_SHA512
1832     # if ALG_SHA256_192
1833         case TPM_ALG_SHA256_192:
1834     # endif // ALG_SHA256_192
1835     # if ALG_SM3_256
1836         case TPM_ALG_SM3_256:
1837     # endif // ALG_SM3_256
1838     # if ALG_SHA3_256
1839         case TPM_ALG_SHA3_256:
1840     # endif // ALG_SHA3_256
1841     # if ALG_SHA3_384
1842         case TPM_ALG_SHA3_384:
1843     # endif // ALG_SHA3_384
1844     # if ALG_SHA3_512
1845         case TPM_ALG_SHA3_512:
1846     # endif // ALG_SHA3_512
1847     # if ALG_SHAKE256_192
1848         case TPM_ALG_SHAKE256_192:
1849     # endif // ALG_SHAKE256_192
1850     # if ALG_SHAKE256_256
1851         case TPM_ALG_SHAKE256_256:
1852     # endif // ALG_SHAKE256_256
1853     # if ALG_SHAKE256_512
1854         case TPM_ALG_SHAKE256_512:
1855     # endif // ALG_SHAKE256_512
1856     # if ALG_CMAC
1857         case TPM_ALG_CMAC:
1858     # endif // ALG_CMAC
1859         break;
1860     default:
1861         if ((*target != TPM_ALG_NULL) || !flag)
1862             result = TPM_RC_SYMMETRIC;
1863         break;
1864     }
1865 }
1866 return result;
1867 }
1868 # if !USE_MARSHALING_DEFINES
1869 UINT16
1870 TPMI_ALG_MAC_SCHEME_Marshal(TPMI_ALG_MAC_SCHEME* source, BYTE** buffer, INT32* size)
1871 {
1872     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1873 }
1874 # endif // !USE_MARSHALING_DEFINES
1875
1876 // Table "Definition of TPMI_ALG_CIPHER_MODE Type" (Part 2: Structures)
1877 TPM_RC
1878 TPMI_ALG_CIPHER_MODE_Unmarshal(
1879     TPMI_ALG_CIPHER_MODE* target, BYTE** buffer, INT32* size, BOOL flag)
1880 {
1881     TPM_RC result;
1882     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
1883     if (result == TPM_RC_SUCCESS)
1884     {
1885         switch(*target)
1886         {
1887     # if ALG_CTR
1888         case TPM_ALG_CTR:
1889     # endif // ALG_CTR
1890     # if ALG_OFB
1891         case TPM_ALG_OFB:
1892     # endif // ALG_OFB

```

```

1893 # if ALG_CBC
1894     case TPM_ALG_CBC:
1895 # endif // ALG_CBC
1896 # if ALG_CFB
1897     case TPM_ALG_CFB:
1898 # endif // ALG_CFB
1899 # if ALG_ECB
1900     case TPM_ALG_ECB:
1901 # endif // ALG_ECB
1902     break;
1903     default:
1904         if ((*target != TPM_ALG_NULL) || !flag)
1905             result = TPM_RC_MODE;
1906         break;
1907     }
1908 }
1909 return result;
1910 }
1911 # if !USE_MARSHALING_DEFINES
1912 UINT16
1913 TPMI_ALG_CIPHER_MODE_Marshal(TPMI_ALG_CIPHER_MODE* source, BYTE** buffer, INT32* size)
1914 {
1915     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
1916 }
1917 # endif // !USE_MARSHALING_DEFINES
1918
1919 // Table "Definition of TPMS_EMPTY Structure" (Part 2: Structures)
1920 TPM_RC
1921 TPMS_EMPTY_Unmarshal(TPMS_EMPTY* target, BYTE** buffer, INT32* size)
1922 {
1923     // to prevent the compiler from complaining
1924     NOT_REFERENCED(target);
1925     NOT_REFERENCED(buffer);
1926     NOT_REFERENCED(size);
1927     return TPM_RC_SUCCESS;
1928 }
1929
1930 UINT16
1931 TPMS_EMPTY_Marshal(TPMS_EMPTY* source, BYTE** buffer, INT32* size)
1932 {
1933     // to prevent the compiler from complaining
1934     NOT_REFERENCED(source);
1935     NOT_REFERENCED(buffer);
1936     NOT_REFERENCED(size);
1937     return 0;
1938 }
1939
1940 // Table "Definition of TPMS_ALGORITHM_DESCRIPTION Structure" (Part 2: Structures)
1941 UINT16
1942 TPMS_ALGORITHM_DESCRIPTION_Marshal(
1943     TPMS_ALGORITHM_DESCRIPTION* source, BYTE** buffer, INT32* size)
1944 {
1945     UINT16 result = 0;
1946     result =
1947         (UINT16) (result
1948             + TPM_ALG_ID_Marshal((TPM_ALG_ID*)&(source->alg), buffer, size));
1949     result = (UINT16) (result
1950         + TPMA_ALGORITHM_Marshal(
1951             (TPMA_ALGORITHM*)&(source->attributes), buffer, size));
1952     return result;
1953 }
1954
1955 // Table "Definition of TPMU_HA Union" (Part 2: Structures)
1956 TPM_RC
1957 TPMU_HA_Unmarshal(TPMU_HA* target, BYTE** buffer, INT32* size, UINT32 selector)
1958 {
1959     switch(selector)

```

```

1959     {
1960         case TPM_ALG_NULL:
1961             return TPM_RC_SUCCESS;
1962     # if ALG_SHA1
1963         case TPM_ALG_SHA1:
1964             return BYTE_Array_Unmarshal(
1965                 (BYTE*)&(target->sha1), buffer, size, (INT32)SHA1_DIGEST_SIZE);
1966     # endif // ALG_SHA1
1967     # if ALG_SHA256
1968         case TPM_ALG_SHA256:
1969             return BYTE_Array_Unmarshal(
1970                 (BYTE*)&(target->sha256), buffer, size, (INT32)SHA256_DIGEST_SIZE);
1971     # endif // ALG_SHA256
1972     # if ALG_SHA256_192
1973         case TPM_ALG_SHA256_192:
1974             return BYTE_Array_Unmarshal((BYTE*)&(target->sha256_192),
1975                 buffer,
1976                 size,
1977                 (INT32)SHA256_192_DIGEST_SIZE);
1978     # endif // ALG_SHA256_192
1979     # if ALG_SHA3_256
1980         case TPM_ALG_SHA3_256:
1981             return BYTE_Array_Unmarshal((BYTE*)&(target->sha3_256),
1982                 buffer,
1983                 size,
1984                 (INT32)SHA3_256_DIGEST_SIZE);
1985     # endif // ALG_SHA3_256
1986     # if ALG_SHA3_384
1987         case TPM_ALG_SHA3_384:
1988             return BYTE_Array_Unmarshal((BYTE*)&(target->sha3_384),
1989                 buffer,
1990                 size,
1991                 (INT32)SHA3_384_DIGEST_SIZE);
1992     # endif // ALG_SHA3_384
1993     # if ALG_SHA3_512
1994         case TPM_ALG_SHA3_512:
1995             return BYTE_Array_Unmarshal((BYTE*)&(target->sha3_512),
1996                 buffer,
1997                 size,
1998                 (INT32)SHA3_512_DIGEST_SIZE);
1999     # endif // ALG_SHA3_512
2000     # if ALG_SHA384
2001         case TPM_ALG_SHA384:
2002             return BYTE_Array_Unmarshal(
2003                 (BYTE*)&(target->sha384), buffer, size, (INT32)SHA384_DIGEST_SIZE);
2004     # endif // ALG_SHA384
2005     # if ALG_SHA512
2006         case TPM_ALG_SHA512:
2007             return BYTE_Array_Unmarshal(
2008                 (BYTE*)&(target->sha512), buffer, size, (INT32)SHA512_DIGEST_SIZE);
2009     # endif // ALG_SHA512
2010     # if ALG_SHAKE256_192
2011         case TPM_ALG_SHAKE256_192:
2012             return BYTE_Array_Unmarshal((BYTE*)&(target->shake256_192),
2013                 buffer,
2014                 size,
2015                 (INT32)SHAKE256_192_DIGEST_SIZE);
2016     # endif // ALG_SHAKE256_192
2017     # if ALG_SHAKE256_256
2018         case TPM_ALG_SHAKE256_256:
2019             return BYTE_Array_Unmarshal((BYTE*)&(target->shake256_256),
2020                 buffer,
2021                 size,
2022                 (INT32)SHAKE256_256_DIGEST_SIZE);
2023     # endif // ALG_SHAKE256_256
2024     # if ALG_SHAKE256_512

```

```

2025         case TPM_ALG_SHAKE256_512:
2026             return BYTE_Array_Unmarshal((BYTE*)&(target->shake256_512),
2027                                         buffer,
2028                                         size,
2029                                         (INT32)SHAKE256_512_DIGEST_SIZE);
2030 # endif // ALG_SHAKE256_512
2031 # if ALG_SM3_256
2032     case TPM_ALG_SM3_256:
2033         return BYTE_Array_Unmarshal(
2034             (BYTE*)&(target->sm3_256), buffer, size, (INT32)SM3_256_DIGEST_SIZE);
2035 # endif // ALG_SM3_256
2036     }
2037     return TPM_RC_SELECTOR;
2038 }
2039 UINT16
2040 TPMU_HA_Marshal(TPMU_HA* source, BYTE** buffer, INT32* size, UINT32 selector)
2041 {
2042     switch(selector)
2043     {
2044 # if ALG_SHA1
2045         case TPM_ALG_SHA1:
2046             return BYTE_Array_Marshal(
2047                 (BYTE*)&(source->sha1), buffer, size, (INT32)SHA1_DIGEST_SIZE);
2048 # endif // ALG_SHA1
2049 # if ALG_SHA256
2050         case TPM_ALG_SHA256:
2051             return BYTE_Array_Marshal(
2052                 (BYTE*)&(source->sha256), buffer, size, (INT32)SHA256_DIGEST_SIZE);
2053 # endif // ALG_SHA256
2054 # if ALG_SHA256_192
2055         case TPM_ALG_SHA256_192:
2056             return BYTE_Array_Marshal((BYTE*)&(source->sha256_192),
2057                                     buffer,
2058                                     size,
2059                                     (INT32)SHA256_192_DIGEST_SIZE);
2060 # endif // ALG_SHA256_192
2061 # if ALG_SHA3_256
2062         case TPM_ALG_SHA3_256:
2063             return BYTE_Array_Marshal((BYTE*)&(source->sha3_256),
2064                                     buffer,
2065                                     size,
2066                                     (INT32)SHA3_256_DIGEST_SIZE);
2067 # endif // ALG_SHA3_256
2068 # if ALG_SHA3_384
2069         case TPM_ALG_SHA3_384:
2070             return BYTE_Array_Marshal((BYTE*)&(source->sha3_384),
2071                                     buffer,
2072                                     size,
2073                                     (INT32)SHA3_384_DIGEST_SIZE);
2074 # endif // ALG_SHA3_384
2075 # if ALG_SHA3_512
2076         case TPM_ALG_SHA3_512:
2077             return BYTE_Array_Marshal((BYTE*)&(source->sha3_512),
2078                                     buffer,
2079                                     size,
2080                                     (INT32)SHA3_512_DIGEST_SIZE);
2081 # endif // ALG_SHA3_512
2082 # if ALG_SHA384
2083         case TPM_ALG_SHA384:
2084             return BYTE_Array_Marshal(
2085                 (BYTE*)&(source->sha384), buffer, size, (INT32)SHA384_DIGEST_SIZE);
2086 # endif // ALG_SHA384
2087 # if ALG_SHA512
2088         case TPM_ALG_SHA512:
2089             return BYTE_Array_Marshal(
2090                 (BYTE*)&(source->sha512), buffer, size, (INT32)SHA512_DIGEST_SIZE);

```



```

2091 # endif // ALG_SHA512
2092 # if ALG_SHAKE256_192
2093     case TPM_ALG_SHAKE256_192:
2094         return BYTE_Array_Marshal((BYTE*)&(source->shake256_192),
2095                                     buffer,
2096                                     size,
2097                                     (INT32)SHAKE256_192_DIGEST_SIZE);
2098 # endif // ALG_SHAKE256_192
2099 # if ALG_SHAKE256_256
2100     case TPM_ALG_SHAKE256_256:
2101         return BYTE_Array_Marshal((BYTE*)&(source->shake256_256),
2102                                     buffer,
2103                                     size,
2104                                     (INT32)SHAKE256_256_DIGEST_SIZE);
2105 # endif // ALG_SHAKE256_256
2106 # if ALG_SHAKE256_512
2107     case TPM_ALG_SHAKE256_512:
2108         return BYTE_Array_Marshal((BYTE*)&(source->shake256_512),
2109                                     buffer,
2110                                     size,
2111                                     (INT32)SHAKE256_512_DIGEST_SIZE);
2112 # endif // ALG_SHAKE256_512
2113 # if ALG_SM3_256
2114     case TPM_ALG_SM3_256:
2115         return BYTE_Array_Marshal(
2116             (BYTE*)&(source->sm3_256), buffer, size, (INT32)SM3_256_DIGEST_SIZE);
2117 # endif // ALG_SM3_256
2118     }
2119     return 0;
2120 }
2121
2122 // Table "Definition of TPMT_HA Structure" (Part 2: Structures)
2123 TPM_RC
2124 TPMT_HA_Unmarshal(TPMT_HA* target, BYTE** buffer, INT32* size, BOOL flag)
2125 {
2126     TPM_RC result;
2127     result = TPMI_ALG_HASH_Unmarshal(
2128         (TPMI_ALG_HASH*)&(target->hashAlg), buffer, size, flag);
2129     if(result == TPM_RC_SUCCESS)
2130         result = TPMU_HA_Unmarshal(
2131             (TPMU_HA*)&(target->digest), buffer, size, (UINT32)target->hashAlg);
2132     return result;
2133 }
2134 UINT16
2135 TPMT_HA_Marshal(TPMT_HA* source, BYTE** buffer, INT32* size)
2136 {
2137     UINT16 result = 0;
2138     result = (UINT16)(result
2139         + TPMI_ALG_HASH_Marshal(
2140             (TPMI_ALG_HASH*)&(source->hashAlg), buffer, size));
2141     result = (UINT16)(result
2142         + TPMU_HA_Marshal((TPMU_HA*)&(source->digest),
2143             buffer,
2144             size,
2145             (UINT32)source->hashAlg));
2146     return result;
2147 }
2148
2149 // Table "Definition of TPM2B_DIGEST Structure" (Part 2: Structures)
2150 TPM_RC
2151 TPM2B_DIGEST_Unmarshal(TPM2B_DIGEST* target, BYTE** buffer, INT32* size)
2152 {
2153     TPM_RC result;
2154     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
2155     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(TPMU_HA)))
2156         result = TPM_RC_SIZE;

```

```

2157     if(result == TPM_RC_SUCCESS)
2158         result = BYTE_Array_Unmarshal(
2159             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
2160     return result;
2161 }
2162 UINT16
2163 TPM2B_DIGEST_Marshal(TPM2B_DIGEST* source, BYTE** buffer, INT32* size)
2164 {
2165     UINT16 result = 0;
2166     result =
2167         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
2168     // if size equal to 0, the rest of the structure is a zero buffer
2169     if(source->t.size == 0)
2170         return result;
2171     result = (UINT16)(result
2172         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
2173             buffer,
2174             size,
2175             (INT32)source->t.size));
2176     return result;
2177 }
2178
2179 // Table "Definition of TPM2B_DATA Structure" (Part 2: Structures)
2180 TPM_RC
2181 TPM2B_DATA_Unmarshal(TPM2B_DATA* target, BYTE** buffer, INT32* size)
2182 {
2183     TPM_RC result;
2184     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
2185     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(TPMT_HA)))
2186         result = TPM_RC_SIZE;
2187     if(result == TPM_RC_SUCCESS)
2188         result = BYTE_Array_Unmarshal(
2189             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
2190     return result;
2191 }
2192 UINT16
2193 TPM2B_DATA_Marshal(TPM2B_DATA* source, BYTE** buffer, INT32* size)
2194 {
2195     UINT16 result = 0;
2196     result =
2197         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
2198     // if size equal to 0, the rest of the structure is a zero buffer
2199     if(source->t.size == 0)
2200         return result;
2201     result = (UINT16)(result
2202         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
2203             buffer,
2204             size,
2205             (INT32)source->t.size));
2206     return result;
2207 }
2208
2209 // Table "Definition of Types for TPM2B_NONCE" (Part 2: Structures)
2210 # if !USE_MARSHALING_DEFINES
2211 TPM_RC
2212 TPM2B_NONCE_Unmarshal(TPM2B_NONCE* target, BYTE** buffer, INT32* size)
2213 {
2214     return TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)target, buffer, size);
2215 }
2216 UINT16
2217 TPM2B_NONCE_Marshal(TPM2B_NONCE* source, BYTE** buffer, INT32* size)
2218 {
2219     return TPM2B_DIGEST_Marshal((TPM2B_DIGEST*)source, buffer, size);
2220 }
2221 # endif // !USE_MARSHALING_DEFINES
2222

```

```

2223 // Table "Definition of Types for TPM2B_AUTH" (Part 2: Structures)
2224 # if !USE_MARSHALING_DEFINES
2225 TPM_RC
2226 TPM2B_AUTH_Unmarshal(TPM2B_AUTH* target, BYTE** buffer, INT32* size)
2227 {
2228     return TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)target, buffer, size);
2229 }
2230 UINT16
2231 TPM2B_AUTH_Marshal(TPM2B_AUTH* source, BYTE** buffer, INT32* size)
2232 {
2233     return TPM2B_DIGEST_Marshal((TPM2B_DIGEST*)source, buffer, size);
2234 }
2235 # endif // !USE_MARSHALING_DEFINES
2236
2237 // Table "Definition of Types for TPM2B_OPERAND" (Part 2: Structures)
2238 # if !USE_MARSHALING_DEFINES
2239 TPM_RC
2240 TPM2B_OPERAND_Unmarshal(TPM2B_OPERAND* target, BYTE** buffer, INT32* size)
2241 {
2242     return TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)target, buffer, size);
2243 }
2244 UINT16
2245 TPM2B_OPERAND_Marshal(TPM2B_OPERAND* source, BYTE** buffer, INT32* size)
2246 {
2247     return TPM2B_DIGEST_Marshal((TPM2B_DIGEST*)source, buffer, size);
2248 }
2249 # endif // !USE_MARSHALING_DEFINES
2250
2251 // Table "Definition of TPM2B_EVENT Structure" (Part 2: Structures)
2252 TPM_RC
2253 TPM2B_EVENT_Unmarshal(TPM2B_EVENT* target, BYTE** buffer, INT32* size)
2254 {
2255     TPM_RC result;
2256     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
2257     if((result == TPM_RC_SUCCESS) && (target->t.size > 1024))
2258         result = TPM_RC_SIZE;
2259     if(result == TPM_RC_SUCCESS)
2260         result = BYTE_Array_Unmarshal(
2261             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
2262     return result;
2263 }
2264 UINT16
2265 TPM2B_EVENT_Marshal(TPM2B_EVENT* source, BYTE** buffer, INT32* size)
2266 {
2267     UINT16 result = 0;
2268     result =
2269         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
2270     // if size equal to 0, the rest of the structure is a zero buffer
2271     if(source->t.size == 0)
2272         return result;
2273     result = (UINT16)(result
2274         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
2275             buffer,
2276             size,
2277             (INT32)source->t.size));
2278     return result;
2279 }
2280
2281 // Table "Definition of TPM2B_MAX_BUFFER Structure" (Part 2: Structures)
2282 TPM_RC
2283 TPM2B_MAX_BUFFER_Unmarshal(TPM2B_MAX_BUFFER* target, BYTE** buffer, INT32* size)
2284 {
2285     TPM_RC result;
2286     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
2287     if((result == TPM_RC_SUCCESS) && (target->t.size > MAX_DIGEST_BUFFER))
2288         result = TPM_RC_SIZE;

```

```

2289     if(result == TPM_RC_SUCCESS)
2290         result = BYTE_Array_Unmarshal(
2291             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
2292     return result;
2293 }
2294 UINT16
2295 TPM2B_MAX_BUFFER_Marshal(TPM2B_MAX_BUFFER* source, BYTE** buffer, INT32* size)
2296 {
2297     UINT16 result = 0;
2298     result =
2299         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
2300     // if size equal to 0, the rest of the structure is a zero buffer
2301     if(source->t.size == 0)
2302         return result;
2303     result = (UINT16)(result
2304         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
2305             buffer,
2306             size,
2307             (INT32)source->t.size));
2308     return result;
2309 }
2310
2311 // Table "Definition of TPM2B_MAX_NV_BUFFER Structure" (Part 2: Structures)
2312 TPM_RC
2313 TPM2B_MAX_NV_BUFFER_Unmarshal(TPM2B_MAX_NV_BUFFER* target, BYTE** buffer, INT32* size)
2314 {
2315     TPM_RC result;
2316     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
2317     if((result == TPM_RC_SUCCESS) && (target->t.size > MAX_NV_BUFFER_SIZE))
2318         result = TPM_RC_SIZE;
2319     if(result == TPM_RC_SUCCESS)
2320         result = BYTE_Array_Unmarshal(
2321             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
2322     return result;
2323 }
2324 UINT16
2325 TPM2B_MAX_NV_BUFFER_Marshal(TPM2B_MAX_NV_BUFFER* source, BYTE** buffer, INT32* size)
2326 {
2327     UINT16 result = 0;
2328     result =
2329         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
2330     // if size equal to 0, the rest of the structure is a zero buffer
2331     if(source->t.size == 0)
2332         return result;
2333     result = (UINT16)(result
2334         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
2335             buffer,
2336             size,
2337             (INT32)source->t.size));
2338     return result;
2339 }
2340
2341 // Table "Definition of TPM2B_TIMEOUT Structure" (Part 2: Structures)
2342 TPM_RC
2343 TPM2B_TIMEOUT_Unmarshal(TPM2B_TIMEOUT* target, BYTE** buffer, INT32* size)
2344 {
2345     TPM_RC result;
2346     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
2347     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(UINT64)))
2348         result = TPM_RC_SIZE;
2349     if(result == TPM_RC_SUCCESS)
2350         result = BYTE_Array_Unmarshal(
2351             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
2352     return result;
2353 }
2354 UINT16

```

```

2355 TPM2B_TIMEOUT_Marshal(TPM2B_TIMEOUT* source, BYTE** buffer, INT32* size)
2356 {
2357     UINT16 result = 0;
2358     result =
2359         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
2360     // if size equal to 0, the rest of the structure is a zero buffer
2361     if(source->t.size == 0)
2362         return result;
2363     result = (UINT16)(result
2364         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
2365             buffer,
2366             size,
2367             (INT32)source->t.size));
2368     return result;
2369 }
2370
2371 // Table "Definition of TPM2B_IV Structure" (Part 2: Structures)
2372 TPM_RC
2373 TPM2B_IV_Unmarshal(TPM2B_IV* target, BYTE** buffer, INT32* size)
2374 {
2375     TPM_RC result;
2376     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
2377     if((result == TPM_RC_SUCCESS) && (target->t.size > MAX_SYM_BLOCK_SIZE))
2378         result = TPM_RC_SIZE;
2379     if(result == TPM_RC_SUCCESS)
2380         result = BYTE_Array_Unmarshal(
2381             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
2382     return result;
2383 }
2384
2385 UINT16
2386 TPM2B_IV_Marshal(TPM2B_IV* source, BYTE** buffer, INT32* size)
2387 {
2388     UINT16 result = 0;
2389     result =
2390         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
2391     // if size equal to 0, the rest of the structure is a zero buffer
2392     if(source->t.size == 0)
2393         return result;
2394     result = (UINT16)(result
2395         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
2396             buffer,
2397             size,
2398             (INT32)source->t.size));
2399     return result;
2400 }
2401
2402 // Table "Definition of TPM2B_VENDOR_PROPERTY Structure" (Part 2: Structures)
2403 TPM_RC
2404 TPM2B_VENDOR_PROPERTY_Unmarshal(
2405     TPM2B_VENDOR_PROPERTY* target, BYTE** buffer, INT32* size)
2406 {
2407     TPM_RC result;
2408     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
2409     if((result == TPM_RC_SUCCESS) && (target->t.size > 512))
2410         result = TPM_RC_SIZE;
2411     if(result == TPM_RC_SUCCESS)
2412         result = BYTE_Array_Unmarshal(
2413             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
2414     return result;
2415 }
2416
2417 TPM2B_VENDOR_PROPERTY_Marshal(
2418     TPM2B_VENDOR_PROPERTY* source, BYTE** buffer, INT32* size)
2419 {
2420     UINT16 result = 0;
2421     result =

```

```

2421     (UINT16) (result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
2422 // if size equal to 0, the rest of the structure is a zero buffer
2423 if(source->t.size == 0)
2424     return result;
2425 result = (UINT16) (result
2426                 + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
2427                                     buffer,
2428                                     size,
2429                                     (INT32) source->t.size));
2430 return result;
2431 }
2432
2433 // Table "Definition of TPM2B_NAME Structure" (Part 2: Structures)
2434 TPM_RC
2435 TPM2B_NAME_Unmarshal(TPM2B_NAME* target, BYTE** buffer, INT32* size)
2436 {
2437     TPM_RC result;
2438     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
2439     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(TPMU_NAME)))
2440         result = TPM_RC_SIZE;
2441     if(result == TPM_RC_SUCCESS)
2442         result = BYTE_Array_Unmarshal(
2443             (BYTE*)&(target->t.name), buffer, size, (INT32) target->t.size);
2444     return result;
2445 }
2446 UINT16
2447 TPM2B_NAME_Marshal(TPM2B_NAME* source, BYTE** buffer, INT32* size)
2448 {
2449     UINT16 result = 0;
2450     result =
2451         (UINT16) (result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
2452 // if size equal to 0, the rest of the structure is a zero buffer
2453 if(source->t.size == 0)
2454     return result;
2455 result =
2456     (UINT16) (result
2457             + BYTE_Array_Marshal(
2458                 (BYTE*)&(source->t.name), buffer, size, (INT32) source->t.size));
2459 return result;
2460 }
2461
2462 // Table "Definition of TPMS_PCR_SELECT Structure" (Part 2: Structures)
2463 TPM_RC
2464 TPMS_PCR_SELECT_Unmarshal(TPMS_PCR_SELECT* target, BYTE** buffer, INT32* size)
2465 {
2466     TPM_RC result;
2467     result = UINT8_Unmarshal((UINT8*)&(target->sizeofSelect), buffer, size);
2468     if((result == TPM_RC_SUCCESS)
2469        && ((target->sizeofSelect < PCR_SELECT_MIN)
2470            || (target->sizeofSelect > PCR_SELECT_MAX)))
2471         result = TPM_RC_VALUE;
2472     if(result == TPM_RC_SUCCESS)
2473         result = BYTE_Array_Unmarshal(
2474             (BYTE*)&(target->pcrSelect), buffer, size, (INT32) target->sizeofSelect);
2475     return result;
2476 }
2477 UINT16
2478 TPMS_PCR_SELECT_Marshal(TPMS_PCR_SELECT* source, BYTE** buffer, INT32* size)
2479 {
2480     UINT16 result = 0;
2481     result = (UINT16) (result
2482                     + UINT8_Marshal((UINT8*)&(source->sizeofSelect), buffer, size));
2483     result = (UINT16) (result
2484                     + BYTE_Array_Marshal((BYTE*)&(source->pcrSelect),
2485                                         buffer,
2486                                         size,

```



```

2487                                     (INT32)source->sizeofSelect));
2488     return result;
2489 }
2490
2491 // Table "Definition of TPMS_PCR_SELECTION Structure" (Part 2: Structures)
2492 TPM_RC
2493 TPMS_PCR_SELECTION_Unmarshal(TPMS_PCR_SELECTION* target, BYTE** buffer, INT32* size)
2494 {
2495     TPM_RC result;
2496     result =
2497         TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH*)&(target->hash), buffer, size, 0);
2498     if(result == TPM_RC_SUCCESS)
2499         result = UINT8_Unmarshal((UINT8*)&(target->sizeofSelect), buffer, size);
2500     if((result == TPM_RC_SUCCESS)
2501         && ((target->sizeofSelect < PCR_SELECT_MIN)
2502             || (target->sizeofSelect > PCR_SELECT_MAX)))
2503         result = TPM_RC_VALUE;
2504     if(result == TPM_RC_SUCCESS)
2505         result = BYTE_Array_Unmarshal(
2506             (BYTE*)&(target->pcrSelect), buffer, size, (INT32)target->sizeofSelect);
2507     return result;
2508 }
2509
2510 UINT16
2511 TPMS_PCR_SELECTION_Marshal(TPMS_PCR_SELECTION* source, BYTE** buffer, INT32* size)
2512 {
2513     UINT16 result = 0;
2514     result = (UINT16)(result
2515         + TPMI_ALG_HASH_Marshal(
2516             (TPMI_ALG_HASH*)&(source->hash), buffer, size));
2517     result = (UINT16)(result
2518         + UINT8_Marshal((UINT8*)&(source->sizeofSelect), buffer, size));
2519     result = (UINT16)(result
2520         + BYTE_Array_Marshal((BYTE*)&(source->pcrSelect),
2521             buffer,
2522             size,
2523             (INT32)source->sizeofSelect));
2524     return result;
2525 }
2526
2527 // Table "Definition of TPMT_TK_CREATION Structure" (Part 2: Structures)
2528 TPM_RC
2529 TPMT_TK_CREATION_Unmarshal(TPMT_TK_CREATION* target, BYTE** buffer, INT32* size)
2530 {
2531     TPM_RC result;
2532     result = TPM_ST_Unmarshal((TPM_ST*)&(target->tag), buffer, size);
2533     if((result == TPM_RC_SUCCESS) && (target->tag != TPM_ST_CREATION))
2534         result = TPM_RC_TAG;
2535     if(result == TPM_RC_SUCCESS)
2536         result = TPMI_RH_HIERARCHY_Unmarshal(
2537             (TPMI_RH_HIERARCHY*)&(target->hierarchy), buffer, size);
2538     if(result == TPM_RC_SUCCESS)
2539         result =
2540             TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)&(target->digest), buffer, size);
2541     return result;
2542 }
2543
2544 UINT16
2545 TPMT_TK_CREATION_Marshal(TPMT_TK_CREATION* source, BYTE** buffer, INT32* size)
2546 {
2547     UINT16 result = 0;
2548     result = (UINT16)(result + TPM_ST_Marshal((TPM_ST*)&(source->tag), buffer, size));
2549     result = (UINT16)(result
2550         + TPMI_RH_HIERARCHY_Marshal(
2551             (TPMI_RH_HIERARCHY*)&(source->hierarchy), buffer, size));
2552     result = (UINT16)(result
2553         + TPM2B_DIGEST_Marshal(
2554             (TPM2B_DIGEST*)&(source->digest), buffer, size));

```

```

2553     return result;
2554 }
2555
2556 // Table "Definition of TPMT_TK_VERIFIED Structure" (Part 2: Structures)
2557 TPM_RC
2558 TPMT_TK_VERIFIED_Unmarshal(TPMT_TK_VERIFIED* target, BYTE** buffer, INT32* size)
2559 {
2560     TPM_RC result;
2561     result = TPM_ST_Unmarshal((TPM_ST*)&(target->tag), buffer, size);
2562     if((result == TPM_RC_SUCCESS) && (target->tag != TPM_ST_VERIFIED))
2563         result = TPM_RC_TAG;
2564     if(result == TPM_RC_SUCCESS)
2565         result = TPMI_RH_HIERARCHY_Unmarshal(
2566             (TPMI_RH_HIERARCHY*)&(target->hierarchy), buffer, size);
2567     if(result == TPM_RC_SUCCESS)
2568         result =
2569             TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)&(target->digest), buffer, size);
2570     return result;
2571 }
2572 UINT16
2573 TPMT_TK_VERIFIED_Marshal(TPMT_TK_VERIFIED* source, BYTE** buffer, INT32* size)
2574 {
2575     UINT16 result = 0;
2576     result = (UINT16)(result + TPM_ST_Marshal((TPM_ST*)&(source->tag), buffer, size));
2577     result = (UINT16)(result
2578         + TPMI_RH_HIERARCHY_Marshal(
2579             (TPMI_RH_HIERARCHY*)&(source->hierarchy), buffer, size));
2580     result = (UINT16)(result
2581         + TPM2B_DIGEST_Marshal(
2582             (TPM2B_DIGEST*)&(source->digest), buffer, size));
2583     return result;
2584 }
2585
2586 // Table "Definition of TPMT_TK_AUTH Structure" (Part 2: Structures)
2587 TPM_RC
2588 TPMT_TK_AUTH_Unmarshal(TPMT_TK_AUTH* target, BYTE** buffer, INT32* size)
2589 {
2590     TPM_RC result;
2591     result = TPM_ST_Unmarshal((TPM_ST*)&(target->tag), buffer, size);
2592     if((result == TPM_RC_SUCCESS) && (target->tag != TPM_ST_AUTH_SIGNED)
2593         && (target->tag != TPM_ST_AUTH_SECRET))
2594         result = TPM_RC_TAG;
2595     if(result == TPM_RC_SUCCESS)
2596         result = TPMI_RH_HIERARCHY_Unmarshal(
2597             (TPMI_RH_HIERARCHY*)&(target->hierarchy), buffer, size);
2598     if(result == TPM_RC_SUCCESS)
2599         result =
2600             TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)&(target->digest), buffer, size);
2601     return result;
2602 }
2603 UINT16
2604 TPMT_TK_AUTH_Marshal(TPMT_TK_AUTH* source, BYTE** buffer, INT32* size)
2605 {
2606     UINT16 result = 0;
2607     result = (UINT16)(result + TPM_ST_Marshal((TPM_ST*)&(source->tag), buffer, size));
2608     result = (UINT16)(result
2609         + TPMI_RH_HIERARCHY_Marshal(
2610             (TPMI_RH_HIERARCHY*)&(source->hierarchy), buffer, size));
2611     result = (UINT16)(result
2612         + TPM2B_DIGEST_Marshal(
2613             (TPM2B_DIGEST*)&(source->digest), buffer, size));
2614     return result;
2615 }
2616
2617 // Table "Definition of TPMT_TK_HASHCHECK Structure" (Part 2: Structures)
2618 TPM_RC

```

```

2619 TPMT_TK_HASHCHECK_Unmarshal(TPMT_TK_HASHCHECK* target, BYTE** buffer, INT32* size)
2620 {
2621     TPM_RC result;
2622     result = TPM_ST_Unmarshal((TPM_ST*)&(target->tag), buffer, size);
2623     if((result == TPM_RC_SUCCESS) && (target->tag != TPM_ST_HASHCHECK))
2624         result = TPM_RC_TAG;
2625     if(result == TPM_RC_SUCCESS)
2626         result = TPMI_RH_HIERARCHY_Unmarshal(
2627             (TPMI_RH_HIERARCHY*)&(target->hierarchy), buffer, size);
2628     if(result == TPM_RC_SUCCESS)
2629         result =
2630             TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)&(target->digest), buffer, size);
2631     return result;
2632 }
2633
2634 TPMT_TK_HASHCHECK_Marshal(TPMT_TK_HASHCHECK* source, BYTE** buffer, INT32* size)
2635 {
2636     UINT16 result = 0;
2637     result = (UINT16)(result + TPM_ST_Marshal((TPM_ST*)&(source->tag), buffer, size));
2638     result = (UINT16)(result
2639         + TPMI_RH_HIERARCHY_Marshal(
2640             (TPMI_RH_HIERARCHY*)&(source->hierarchy), buffer, size));
2641     result = (UINT16)(result
2642         + TPM2B_DIGEST_Marshal(
2643             (TPM2B_DIGEST*)&(source->digest), buffer, size));
2644     return result;
2645 }
2646
2647 // Table "Definition of TPMS_ALG_PROPERTY Structure" (Part 2: Structures)
2648
2649 TPMS_ALG_PROPERTY_Marshal(TPMS_ALG_PROPERTY* source, BYTE** buffer, INT32* size)
2650 {
2651     UINT16 result = 0;
2652     result =
2653         (UINT16)(result
2654             + TPM_ALG_ID_Marshal((TPM_ALG_ID*)&(source->alg), buffer, size));
2655     result = (UINT16)(result
2656         + TPMA_ALGORITHM_Marshal(
2657             (TPMA_ALGORITHM*)&(source->algProperties), buffer, size));
2658     return result;
2659 }
2660
2661 // Table "Definition of TPMS_TAGGED_PROPERTY Structure" (Part 2: Structures)
2662
2663 TPMS_TAGGED_PROPERTY_Marshal(TPMS_TAGGED_PROPERTY* source, BYTE** buffer, INT32* size)
2664 {
2665     UINT16 result = 0;
2666     result =
2667         (UINT16)(result + TPM_PT_Marshal((TPM_PT*)&(source->property), buffer, size));
2668     result =
2669         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->value), buffer, size));
2670     return result;
2671 }
2672
2673 // Table "Definition of TPMS_TAGGED_PCR_SELECT Structure" (Part 2: Structures)
2674
2675 TPMS_TAGGED_PCR_SELECT_Marshal(
2676     TPMS_TAGGED_PCR_SELECT* source, BYTE** buffer, INT32* size)
2677 {
2678     UINT16 result = 0;
2679     result =
2680         (UINT16)(result
2681             + TPM_PT_PCR_Marshal((TPM_PT_PCR*)&(source->tag), buffer, size));
2682     result = (UINT16)(result
2683         + UINT8_Marshal((UINT8*)&(source->sizeofSelect), buffer, size));
2684     result = (UINT16)(result

```

```

2685         + BYTE_Array_Marshal((BYTE*)&(source->pcrSelect),
2686                               buffer,
2687                               size,
2688                               (INT32)source->sizeofSelect));
2689     return result;
2690 }
2691
2692 // Table "Definition of TPMS_TAGGED_POLICY Structure" (Part 2: Structures)
2693 UINT16
2694 TPMS_TAGGED_POLICY_Marshal(TPMS_TAGGED_POLICY* source, BYTE** buffer, INT32* size)
2695 {
2696     UINT16 result = 0;
2697     result =
2698         (UINT16)(result
2699                 + TPM_HANDLE_Marshal((TPM_HANDLE*)&(source->handle), buffer, size));
2700     result =
2701         (UINT16)(result
2702                 + TPMT_HA_Marshal((TPMT_HA*)&(source->policyHash), buffer, size));
2703     return result;
2704 }
2705
2706 // Table "Definition of TPMS_ACT_DATA Structure" (Part 2: Structures)
2707 UINT16
2708 TPMS_ACT_DATA_Marshal(TPMS_ACT_DATA* source, BYTE** buffer, INT32* size)
2709 {
2710     UINT16 result = 0;
2711     result =
2712         (UINT16)(result
2713                 + TPM_HANDLE_Marshal((TPM_HANDLE*)&(source->handle), buffer, size));
2714     result =
2715         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->timeout), buffer, size));
2716     result =
2717         (UINT16)(result
2718                 + TPMA_ACT_Marshal((TPMA_ACT*)&(source->attributes), buffer, size));
2719     return result;
2720 }
2721
2722 // Table "Definition of TPML_CC Structure" (Part 2: Structures)
2723 TPM_RC
2724 TPML_CC_Unmarshal(TPML_CC* target, BYTE** buffer, INT32* size)
2725 {
2726     TPM_RC result;
2727     result = UINT32_Unmarshal((UINT32*)&(target->count), buffer, size);
2728     if((result == TPM_RC_SUCCESS) && (target->count > MAX_CAP_CC))
2729         result = TPM_RC_SIZE;
2730     if(result == TPM_RC_SUCCESS)
2731         result = TPM_CC_Array_Unmarshal(
2732             (TPM_CC*)&(target->commandCodes), buffer, size, (INT32)target->count);
2733     return result;
2734 }
2735
2736 UINT16
2737 TPML_CC_Marshal(TPML_CC* source, BYTE** buffer, INT32* size)
2738 {
2739     UINT16 result = 0;
2740     result =
2741         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2742     result = (UINT16)(result
2743                     + TPM_CC_Array_Marshal((TPM_CC*)&(source->commandCodes),
2744                                             buffer,
2745                                             size,
2746                                             (INT32)source->count));
2747     return result;
2748 }
2749
2750 // Table "Definition of TPML_CCA Structure" (Part 2: Structures)
2751 UINT16

```

```

2751 TPML_CCA_Marshal(TPML_CCA* source, BYTE** buffer, INT32* size)
2752 {
2753     UINT16 result = 0;
2754     result =
2755         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2756     result = (UINT16)(result
2757         + TPMA_CC_Array_Marshal((TPMA_CC*)&(source->commandAttributes),
2758             buffer,
2759             size,
2760             (INT32)source->count));
2761     return result;
2762 }
2763
2764 // Table "Definition of TPML_ALG Structure" (Part 2: Structures)
2765 TPM_RC
2766 TPML_ALG_Unmarshal(TPML_ALG* target, BYTE** buffer, INT32* size)
2767 {
2768     TPM_RC result;
2769     result = UINT32_Unmarshal((UINT32*)&(target->count), buffer, size);
2770     if((result == TPM_RC_SUCCESS) && (target->count > MAX_ALG_LIST_SIZE))
2771         result = TPM_RC_SIZE;
2772     if(result == TPM_RC_SUCCESS)
2773         result = TPM_ALG_ID_Array_Unmarshal(
2774             (TPM_ALG_ID*)&(target->algorithms), buffer, size, (INT32)target->count);
2775     return result;
2776 }
2777
2778 TPM_RC
2779 TPML_ALG_Marshal(TPML_ALG* source, BYTE** buffer, INT32* size)
2780 {
2781     UINT16 result = 0;
2782     result =
2783         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2784     result = (UINT16)(result
2785         + TPM_ALG_ID_Array_Marshal((TPM_ALG_ID*)&(source->algorithms),
2786             buffer,
2787             size,
2788             (INT32)source->count));
2789     return result;
2790 }
2791
2792 // Table "Definition of TPML_HANDLE Structure" (Part 2: Structures)
2793 TPM_RC
2794 TPML_HANDLE_Marshal(TPML_HANDLE* source, BYTE** buffer, INT32* size)
2795 {
2796     UINT16 result = 0;
2797     result =
2798         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2799     result = (UINT16)(result
2800         + TPM_HANDLE_Array_Marshal((TPM_HANDLE*)&(source->handle),
2801             buffer,
2802             size,
2803             (INT32)source->count));
2804     return result;
2805 }
2806
2807 // Table "Definition of TPML_DIGEST Structure" (Part 2: Structures)
2808 TPM_RC
2809 TPML_DIGEST_Unmarshal(TPML_DIGEST* target, BYTE** buffer, INT32* size)
2810 {
2811     TPM_RC result;
2812     result = UINT32_Unmarshal((UINT32*)&(target->count), buffer, size);
2813     if((result == TPM_RC_SUCCESS) && ((target->count < 2) || (target->count > 8)))
2814         result = TPM_RC_SIZE;
2815     if(result == TPM_RC_SUCCESS)
2816         result = TPM2B_DIGEST_Array_Unmarshal(
2817             (TPM2B_DIGEST*)&(target->digests), buffer, size, (INT32)target->count);

```

```

2817     return result;
2818 }
2819 UINT16
2820 TPML_DIGEST_Marshal(TPML_DIGEST* source, BYTE** buffer, INT32* size)
2821 {
2822     UINT16 result = 0;
2823     result =
2824         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2825     result = (UINT16)(result
2826         + TPM2B_DIGEST_Array_Marshal((TPM2B_DIGEST*)&(source->digests),
2827         buffer,
2828         size,
2829         (INT32)source->count));
2830     return result;
2831 }
2832
2833 // Table "Definition of TPML_DIGEST_VALUES Structure" (Part 2: Structures)
2834 TPM_RC
2835 TPML_DIGEST_VALUES_Unmarshal(TPML_DIGEST_VALUES* target, BYTE** buffer, INT32* size)
2836 {
2837     TPM_RC result;
2838     result = UINT32_Unmarshal((UINT32*)&(target->count), buffer, size);
2839     if((result == TPM_RC_SUCCESS) && (target->count > HASH_COUNT))
2840         result = TPM_RC_SIZE;
2841     if(result == TPM_RC_SUCCESS)
2842         result = TPMT_HA_Array_Unmarshal(
2843             (TPMT_HA*)&(target->digests), buffer, size, 0, (INT32)target->count);
2844     return result;
2845 }
2846
2847 UINT16
2848 TPML_DIGEST_VALUES_Marshal(TPML_DIGEST_VALUES* source, BYTE** buffer, INT32* size)
2849 {
2850     UINT16 result = 0;
2851     result =
2852         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2853     result = (UINT16)(result
2854         + TPMT_HA_Array_Marshal((TPMT_HA*)&(source->digests),
2855         buffer,
2856         size,
2857         (INT32)source->count));
2858     return result;
2859 }
2860
2861 // Table "Definition of TPML_PCR_SELECTION Structure" (Part 2: Structures)
2862 TPM_RC
2863 TPML_PCR_SELECTION_Unmarshal(TPML_PCR_SELECTION* target, BYTE** buffer, INT32* size)
2864 {
2865     TPM_RC result;
2866     result = UINT32_Unmarshal((UINT32*)&(target->count), buffer, size);
2867     if((result == TPM_RC_SUCCESS) && (target->count > HASH_COUNT))
2868         result = TPM_RC_SIZE;
2869     if(result == TPM_RC_SUCCESS)
2870         result = TPMS_PCR_SELECTION_Array_Unmarshal(
2871             (TPMS_PCR_SELECTION*)&(target->pcrSelections),
2872             buffer,
2873             size,
2874             (INT32)target->count);
2875     return result;
2876 }
2877
2878 UINT16
2879 TPML_PCR_SELECTION_Marshal(TPML_PCR_SELECTION* source, BYTE** buffer, INT32* size)
2880 {
2881     UINT16 result = 0;
2882     result =
2883         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2884     result = (UINT16)(result

```



```

2883         + TPMS_PCR_SELECTION_Array_Marshal(
2884             (TPMS_PCR_SELECTION*)&(source->pcrSelections),
2885             buffer,
2886             size,
2887             (INT32)source->count));
2888     return result;
2889 }
2890
2891 // Table "Definition of TPML_ALG_PROPERTY Structure" (Part 2: Structures)
2892 UINT16
2893 TPML_ALG_PROPERTY_Marshal(TPML_ALG_PROPERTY* source, BYTE** buffer, INT32* size)
2894 {
2895     UINT16 result = 0;
2896     result =
2897         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2898     result = (UINT16)(result
2899         + TPMS_ALG_PROPERTY_Array_Marshal(
2900             (TPMS_ALG_PROPERTY*)&(source->algProperties),
2901             buffer,
2902             size,
2903             (INT32)source->count));
2904     return result;
2905 }
2906
2907 // Table "Definition of TPML_TAGGED_TPM_PROPERTY Structure" (Part 2: Structures)
2908 UINT16
2909 TPML_TAGGED_TPM_PROPERTY_Marshal(
2910     TPML_TAGGED_TPM_PROPERTY* source, BYTE** buffer, INT32* size)
2911 {
2912     UINT16 result = 0;
2913     result =
2914         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2915     result = (UINT16)(result
2916         + TPMS_TAGGED_PROPERTY_Array_Marshal(
2917             (TPMS_TAGGED_PROPERTY*)&(source->tpmProperty),
2918             buffer,
2919             size,
2920             (INT32)source->count));
2921     return result;
2922 }
2923
2924 // Table "Definition of TPML_TAGGED_PCR_PROPERTY Structure" (Part 2: Structures)
2925 UINT16
2926 TPML_TAGGED_PCR_PROPERTY_Marshal(
2927     TPML_TAGGED_PCR_PROPERTY* source, BYTE** buffer, INT32* size)
2928 {
2929     UINT16 result = 0;
2930     result =
2931         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2932     result = (UINT16)(result
2933         + TPMS_TAGGED_PCR_SELECT_Array_Marshal(
2934             (TPMS_TAGGED_PCR_SELECT*)&(source->pcrProperty),
2935             buffer,
2936             size,
2937             (INT32)source->count));
2938     return result;
2939 }
2940
2941 // Table "Definition of TPML_ECC_CURVE Structure" (Part 2: Structures)
2942 # if ALG_ECC
2943 UINT16
2944 TPML_ECC_CURVE_Marshal(TPML_ECC_CURVE* source, BYTE** buffer, INT32* size)
2945 {
2946     UINT16 result = 0;
2947     result =
2948         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));

```

```

2949     result =
2950         (UINT16) (result
2951             + TPM_ECC_CURVE_Array_Marshal((TPM_ECC_CURVE*)&(source->eccCurves),
2952                 buffer,
2953                 size,
2954                 (INT32)source->count));
2955     return result;
2956 }
2957 # endif // ALG_ECC
2958
2959 // Table "Definition of TPML_TAGGED_POLICY Structure" (Part 2: Structures)
2960 UINT16
2961 TPML_TAGGED_POLICY_Marshal(TPML_TAGGED_POLICY* source, BYTE** buffer, INT32* size)
2962 {
2963     UINT16 result = 0;
2964     result =
2965         (UINT16) (result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2966     result = (UINT16) (result
2967         + TPMS_TAGGED_POLICY_Array_Marshal(
2968             (TPMS_TAGGED_POLICY*)&(source->policies),
2969             buffer,
2970             size,
2971             (INT32)source->count));
2972     return result;
2973 }
2974
2975 // Table "Definition of TPML_ACT_DATA Structure" (Part 2: Structures)
2976 UINT16
2977 TPML_ACT_DATA_Marshal(TPML_ACT_DATA* source, BYTE** buffer, INT32* size)
2978 {
2979     UINT16 result = 0;
2980     result =
2981         (UINT16) (result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
2982     result =
2983         (UINT16) (result
2984             + TPMS_ACT_DATA_Array_Marshal((TPMS_ACT_DATA*)&(source->actData),
2985                 buffer,
2986                 size,
2987                 (INT32)source->count));
2988     return result;
2989 }
2990
2991 // Table "Definition of TPML_VENDOR_PROPERTY Structure" (Part 2: Structures)
2992 TPM_RC
2993 TPML_VENDOR_PROPERTY_Unmarshal(
2994     TPML_VENDOR_PROPERTY* target, BYTE** buffer, INT32* size)
2995 {
2996     TPM_RC result;
2997     result = UINT32_Unmarshal((UINT32*)&(target->count), buffer, size);
2998     if((result == TPM_RC_SUCCESS) && (target->count > MAX_VENDOR_PROPERTY))
2999         result = TPM_RC_VALUE;
3000     if(result == TPM_RC_SUCCESS)
3001         result = TPM2B_VENDOR_PROPERTY_Array_Unmarshal(
3002             (TPM2B_VENDOR_PROPERTY*)&(target->vendorData),
3003             buffer,
3004             size,
3005             (INT32)target->count);
3006     return result;
3007 }
3008
3009 UINT16
3010 TPML_VENDOR_PROPERTY_Marshal(TPML_VENDOR_PROPERTY* source, BYTE** buffer, INT32* size)
3011 {
3012     UINT16 result = 0;
3013     result =
3014         (UINT16) (result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
3015     result = (UINT16) (result

```

```

3015         + TPM2B_VENDOR_PROPERTY Array_Marshal(
3016             (TPM2B_VENDOR_PROPERTY*)&(source->vendorData),
3017             buffer,
3018             size,
3019             (INT32)source->count));
3020     return result;
3021 }
3022
3023 // Table "Definition of TPMU_CAPABILITIES Union" (Part 2: Structures)
3024 UINT16
3025 TPMU_CAPABILITIES_Marshal(
3026     TPMU_CAPABILITIES* source, BYTE** buffer, INT32* size, UINT32 selector)
3027 {
3028     switch(selector)
3029     {
3030         case TPM_CAP_ALGS:
3031             return TPML_ALG_PROPERTY_Marshal(
3032                 (TPML_ALG_PROPERTY*)&(source->algorithms), buffer, size);
3033         case TPM_CAP_HANDLES:
3034             return TPML_HANDLE_Marshal(
3035                 (TPML_HANDLE*)&(source->handles), buffer, size);
3036         case TPM_CAP_COMMANDS:
3037             return TPML_CCA_Marshal((TPML_CCA*)&(source->command), buffer, size);
3038         case TPM_CAP_PP_COMMANDS:
3039             return TPML_CC_Marshal((TPML_CC*)&(source->ppCommands), buffer, size);
3040         case TPM_CAP_AUDIT_COMMANDS:
3041             return TPML_CC_Marshal((TPML_CC*)&(source->auditCommands), buffer, size);
3042         case TPM_CAP_PCERS:
3043             return TPML_PCR_SELECTION_Marshal(
3044                 (TPML_PCR_SELECTION*)&(source->assignedPCR), buffer, size);
3045         case TPM_CAP_TPM_PROPERTIES:
3046             return TPML_TAGGED_TPM_PROPERTY_Marshal(
3047                 (TPML_TAGGED_TPM_PROPERTY*)&(source->tpmProperties), buffer, size);
3048         case TPM_CAP_PCR_PROPERTIES:
3049             return TPML_TAGGED_PCR_PROPERTY_Marshal(
3050                 (TPML_TAGGED_PCR_PROPERTY*)&(source->pcrProperties), buffer, size);
3051 # if ALG_ECC
3052         case TPM_CAP_ECC_CURVES:
3053             return TPML_ECC_CURVE_Marshal(
3054                 (TPML_ECC_CURVE*)&(source->eccCurves), buffer, size);
3055 # endif // ALG_ECC
3056         case TPM_CAP_AUTH_POLICIES:
3057             return TPML_TAGGED_POLICY_Marshal(
3058                 (TPML_TAGGED_POLICY*)&(source->authPolicies), buffer, size);
3059         case TPM_CAP_ACT:
3060             return TPML_ACT_DATA_Marshal(
3061                 (TPML_ACT_DATA*)&(source->actData), buffer, size);
3062     }
3063     return 0;
3064 }
3065
3066 // Table "Definition of TPMS_CAPABILITY_DATA Structure" (Part 2: Structures)
3067 UINT16
3068 TPMS_CAPABILITY_DATA_Marshal(TPMS_CAPABILITY_DATA* source, BYTE** buffer, INT32* size)
3069 {
3070     UINT16 result = 0;
3071     result =
3072         (UINT16)(result
3073             + TPM_CAP_Marshal((TPM_CAP*)&(source->capability), buffer, size));
3074     result = (UINT16)(result
3075         + TPMU_CAPABILITIES_Marshal((TPMU_CAPABILITIES*)&(source->data),
3076             buffer,
3077             size,
3078             (UINT32)source->capability));
3079     return result;
3080 }

```

```

3081
3082 // Defined in an additional Capabilities registry
3083 TPM_RC
3084 TPMU_SET_CAPABILITIES_Unmarshal(
3085     TPMU_SET_CAPABILITIES* target, BYTE** buffer, INT32* size, UINT32 selector)
3086 {
3087     // No settable capabilities are currently defined in the reference code.
3088     return TPM_RC_SELECTOR;
3089 }
3090
3091 // Table "Definition of TPMS_SET_CAPABILITY_DATA Structure" (Part 2: Structures)
3092 TPM_RC
3093 TPMS_SET_CAPABILITY_DATA_Unmarshal(
3094     TPMS_SET_CAPABILITY_DATA* target, BYTE** buffer, INT32* size)
3095 {
3096     TPM_RC result;
3097     result = TPM_CAP_Unmarshal(&target->setCapability, buffer, size);
3098     if(result == TPM_RC_SUCCESS)
3099     {
3100         result = TPMU_SET_CAPABILITIES_Unmarshal(
3101             &target->data, buffer, size, (UINT32)target->setCapability);
3102     }
3103     return result;
3104 }
3105
3106 // Table "Definition of TPM2B_SET_CAPABILITY_DATA Structure" (Part 2: Structures)
3107 TPM_RC
3108 TPM2B_SET_CAPABILITY_DATA_Unmarshal(
3109     TPM2B_SET_CAPABILITY_DATA* target, BYTE** buffer, INT32* size)
3110 {
3111     TPM_RC result;
3112     result = UINT16_Unmarshal((UINT16*)&(target->size), buffer, size);
3113     if(result == TPM_RC_SUCCESS)
3114     {
3115         // if size is zero, then the required structure is missing
3116         if(target->size == 0)
3117             result = TPM_RC_SIZE;
3118         else
3119         {
3120             INT32 startSize = *size;
3121             result = TPMS_SET_CAPABILITY_DATA_Unmarshal(
3122                 &target->setCapabilityData, buffer, size);
3123             if((result == TPM_RC_SUCCESS) && (target->size != (startSize - *size)))
3124                 result = TPM_RC_SIZE;
3125         }
3126     }
3127     return result;
3128 }
3129
3130 // Table "Definition of TPMS_CLOCK_INFO Structure" (Part 2: Structures)
3131 TPM_RC
3132 TPMS_CLOCK_INFO_Unmarshal(TPMS_CLOCK_INFO* target, BYTE** buffer, INT32* size)
3133 {
3134     TPM_RC result;
3135     result = UINT64_Unmarshal((UINT64*)&(target->clock), buffer, size);
3136     if(result == TPM_RC_SUCCESS)
3137         result = UINT32_Unmarshal((UINT32*)&(target->resetCount), buffer, size);
3138     if(result == TPM_RC_SUCCESS)
3139         result = UINT32_Unmarshal((UINT32*)&(target->restartCount), buffer, size);
3140     if(result == TPM_RC_SUCCESS)
3141         result = TPMI_YES_NO_Unmarshal((TPMI_YES_NO*)&(target->safe), buffer, size);
3142     return result;
3143 }
3144
3145 TPM_RC
3146 TPMS_CLOCK_INFO_Marshal(TPMS_CLOCK_INFO* source, BYTE** buffer, INT32* size)
3147 {

```

```

3147     UINT16 result = 0;
3148     result =
3149         (UINT16)(result + UINT64_Marshal((UINT64*)&(source->clock), buffer, size));
3150     result = (UINT16)(result
3151         + UINT32_Marshal((UINT32*)&(source->resetCount), buffer, size));
3152     result =
3153         (UINT16)(result
3154         + UINT32_Marshal((UINT32*)&(source->restartCount), buffer, size));
3155     result =
3156         (UINT16)(result
3157         + TPMI_YES_NO_Marshal((TPMI_YES_NO*)&(source->safe), buffer, size));
3158     return result;
3159 }
3160
3161 // Table "Definition of TPMS_TIME_INFO Structure" (Part 2: Structures)
3162 TPM_RC
3163 TPMS_TIME_INFO_Unmarshal(TPMS_TIME_INFO* target, BYTE** buffer, INT32* size)
3164 {
3165     TPM_RC result;
3166     result = UINT64_Unmarshal((UINT64*)&(target->time), buffer, size);
3167     if(result == TPM_RC_SUCCESS)
3168         result = TPMS_CLOCK_INFO_Unmarshal(
3169             (TPMS_CLOCK_INFO*)&(target->clockInfo), buffer, size);
3170     return result;
3171 }
3172
3173 UINT16
3174 TPMS_TIME_INFO_Marshal(TPMS_TIME_INFO* source, BYTE** buffer, INT32* size)
3175 {
3176     UINT16 result = 0;
3177     result =
3178         (UINT16)(result + UINT64_Marshal((UINT64*)&(source->time), buffer, size));
3179     result = (UINT16)(result
3180         + TPMS_CLOCK_INFO_Marshal(
3181             (TPMS_CLOCK_INFO*)&(source->clockInfo), buffer, size));
3182     return result;
3183 }
3184
3185 // Table "Definition of TPMS_TIME_ATTEST_INFO Structure" (Part 2: Structures)
3186
3187 TPM_RC
3188 TPMS_TIME_ATTEST_INFO_Marshal(
3189     TPMS_TIME_ATTEST_INFO* source, BYTE** buffer, INT32* size)
3190 {
3191     TPM_RC result = 0;
3192     result = (UINT16)(result
3193         + TPMS_TIME_INFO_Marshal(
3194             (TPMS_TIME_INFO*)&(source->time), buffer, size));
3195     result =
3196         (UINT16)(result
3197         + UINT64_Marshal((UINT64*)&(source->firmwareVersion), buffer, size));
3198     return result;
3199 }
3200
3201 // Table "Definition of TPMS_CERTIFY_INFO Structure" (Part 2: Structures)
3202
3203 TPM_RC
3204 TPMS_CERTIFY_INFO_Marshal(TPMS_CERTIFY_INFO* source, BYTE** buffer, INT32* size)
3205 {
3206     TPM_RC result = 0;
3207     result =
3208         (UINT16)(result
3209         + TPM2B_NAME_Marshal((TPM2B_NAME*)&(source->name), buffer, size));
3210     result = (UINT16)(result
3211         + TPM2B_NAME_Marshal(
3212             (TPM2B_NAME*)&(source->qualifiedName), buffer, size));
3213     return result;
3214 }

```

```

3213 // Table "Definition of TPMS_QUOTE_INFO Structure" (Part 2: Structures)
3214 UINT16
3215 TPMS_QUOTE_INFO_Marshal(TPMS_QUOTE_INFO* source, BYTE** buffer, INT32* size)
3216 {
3217     UINT16 result = 0;
3218     result = (UINT16)(result
3219         + TPML_PCR_SELECTION_Marshal(
3220             (TPML_PCR_SELECTION*)&(source->pcrSelect), buffer, size));
3221     result = (UINT16)(result
3222         + TPM2B_DIGEST_Marshal(
3223             (TPM2B_DIGEST*)&(source->pcrDigest), buffer, size));
3224     return result;
3225 }
3226
3227 // Table "Definition of TPMS_COMMAND_AUDIT_INFO Structure" (Part 2: Structures)
3228 UINT16
3229 TPMS_COMMAND_AUDIT_INFO_Marshal(
3230     TPMS_COMMAND_AUDIT_INFO* source, BYTE** buffer, INT32* size)
3231 {
3232     UINT16 result = 0;
3233     result =
3234         (UINT16)(result
3235             + UINT64_Marshal((UINT64*)&(source->auditCounter), buffer, size));
3236     result = (UINT16)(result
3237         + TPM_ALG_ID_Marshal(
3238             (TPM_ALG_ID*)&(source->digestAlg), buffer, size));
3239     result = (UINT16)(result
3240         + TPM2B_DIGEST_Marshal(
3241             (TPM2B_DIGEST*)&(source->auditDigest), buffer, size));
3242     result = (UINT16)(result
3243         + TPM2B_DIGEST_Marshal(
3244             (TPM2B_DIGEST*)&(source->commandDigest), buffer, size));
3245     return result;
3246 }
3247
3248 // Table "Definition of TPMS_SESSION_AUDIT_INFO Structure" (Part 2: Structures)
3249 UINT16
3250 TPMS_SESSION_AUDIT_INFO_Marshal(
3251     TPMS_SESSION_AUDIT_INFO* source, BYTE** buffer, INT32* size)
3252 {
3253     UINT16 result = 0;
3254     result = (UINT16)(result
3255         + TPMI_YES_NO_Marshal(
3256             (TPMI_YES_NO*)&(source->exclusiveSession), buffer, size));
3257     result = (UINT16)(result
3258         + TPM2B_DIGEST_Marshal(
3259             (TPM2B_DIGEST*)&(source->sessionDigest), buffer, size));
3260     return result;
3261 }
3262
3263 // Table "Definition of TPMS_CREATION_INFO Structure" (Part 2: Structures)
3264 UINT16
3265 TPMS_CREATION_INFO_Marshal(TPMS_CREATION_INFO* source, BYTE** buffer, INT32* size)
3266 {
3267     UINT16 result = 0;
3268     result = (UINT16)(result
3269         + TPM2B_NAME_Marshal(
3270             (TPM2B_NAME*)&(source->objectName), buffer, size));
3271     result = (UINT16)(result
3272         + TPM2B_DIGEST_Marshal(
3273             (TPM2B_DIGEST*)&(source->creationHash), buffer, size));
3274     return result;
3275 }
3276
3277 // Table "Definition of TPMS_NV_CERTIFY_INFO Structure" (Part 2: Structures)
3278 UINT16

```



```

3279 TPMS_NV_CERTIFY_INFO_Marshal(TPMS_NV_CERTIFY_INFO* source, BYTE** buffer, INT32* size)
3280 {
3281     UINT16 result = 0;
3282     result = (UINT16)(result
3283         + TPM2B_NAME_Marshal(
3284             (TPM2B_NAME*)&(source->indexName), buffer, size));
3285     result =
3286         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->offset), buffer, size));
3287     result = (UINT16)(result
3288         + TPM2B_MAX_NV_BUFFER_Marshal(
3289             (TPM2B_MAX_NV_BUFFER*)&(source->nvContents), buffer, size));
3290     return result;
3291 }
3292
3293 // Table "Definition of TPMS_NV_DIGEST_CERTIFY_INFO Structure" (Part 2: Structures)
3294 UINT16
3295 TPMS_NV_DIGEST_CERTIFY_INFO_Marshal(
3296     TPMS_NV_DIGEST_CERTIFY_INFO* source, BYTE** buffer, INT32* size)
3297 {
3298     UINT16 result = 0;
3299     result = (UINT16)(result
3300         + TPM2B_NAME_Marshal(
3301             (TPM2B_NAME*)&(source->indexName), buffer, size));
3302     result = (UINT16)(result
3303         + TPM2B_DIGEST_Marshal(
3304             (TPM2B_DIGEST*)&(source->nvDigest), buffer, size));
3305     return result;
3306 }
3307
3308 // Table "Definition of TPMI_ST_ATTEST Type" (Part 2: Structures)
3309 # if !USE_MARSHALING_DEFINES
3310 UINT16
3311 TPMI_ST_ATTEST_Marshal(TPMI_ST_ATTEST* source, BYTE** buffer, INT32* size)
3312 {
3313     return TPM_ST_Marshal((TPM_ST*)source, buffer, size);
3314 }
3315 # endif // !USE_MARSHALING_DEFINES
3316
3317 // Table "Definition of TPMU_ATTEST Union" (Part 2: Structures)
3318 UINT16
3319 TPMU_ATTEST_Marshal(TPMU_ATTEST* source, BYTE** buffer, INT32* size, UINT32 selector)
3320 {
3321     switch(selector)
3322     {
3323     case TPM_ST_ATTEST_CERTIFY:
3324         return TPMS_CERTIFY_INFO_Marshal(
3325             (TPMS_CERTIFY_INFO*)&(source->certify), buffer, size);
3326     case TPM_ST_ATTEST_CREATION:
3327         return TPMS_CREATION_INFO_Marshal(
3328             (TPMS_CREATION_INFO*)&(source->creation), buffer, size);
3329     case TPM_ST_ATTEST_QUOTE:
3330         return TPMS_QUOTE_INFO_Marshal(
3331             (TPMS_QUOTE_INFO*)&(source->quote), buffer, size);
3332     case TPM_ST_ATTEST_COMMAND_AUDIT:
3333         return TPMS_COMMAND_AUDIT_INFO_Marshal(
3334             (TPMS_COMMAND_AUDIT_INFO*)&(source->commandAudit), buffer, size);
3335     case TPM_ST_ATTEST_SESSION_AUDIT:
3336         return TPMS_SESSION_AUDIT_INFO_Marshal(
3337             (TPMS_SESSION_AUDIT_INFO*)&(source->sessionAudit), buffer, size);
3338     case TPM_ST_ATTEST_TIME:
3339         return TPMS_TIME_ATTEST_INFO_Marshal(
3340             (TPMS_TIME_ATTEST_INFO*)&(source->time), buffer, size);
3341     case TPM_ST_ATTEST_NV:
3342         return TPMS_NV_CERTIFY_INFO_Marshal(
3343             (TPMS_NV_CERTIFY_INFO*)&(source->nv), buffer, size);
3344     case TPM_ST_ATTEST_NV_DIGEST:

```

```

3345         return TPMS_NV_DIGEST_CERTIFY_INFO_Marshal(
3346             (TPMS_NV_DIGEST_CERTIFY_INFO*)&(source->nvDigest), buffer, size);
3347     }
3348     return 0;
3349 }
3350
3351 // Table "Definition of TPMS_ATTEST Structure" (Part 2: Structures)
3352 UINT16
3353 TPMS_ATTEST_Marshal(TPMS_ATTEST* source, BYTE** buffer, INT32* size)
3354 {
3355     UINT16 result = 0;
3356     result = (UINT16)(result
3357         + TPM_CONSTANTS32_Marshal(
3358             (TPM_CONSTANTS32*)&(source->magic), buffer, size));
3359     result = (UINT16)(result
3360         + TPMI_ST_ATTEST_Marshal(
3361             (TPMI_ST_ATTEST*)&(source->type), buffer, size));
3362     result = (UINT16)(result
3363         + TPM2B_NAME_Marshal(
3364             (TPM2B_NAME*)&(source->qualifiedSigner), buffer, size));
3365     result = (UINT16)(result
3366         + TPM2B_DATA_Marshal(
3367             (TPM2B_DATA*)&(source->extraData), buffer, size));
3368     result = (UINT16)(result
3369         + TPMS_CLOCK_INFO_Marshal(
3370             (TPMS_CLOCK_INFO*)&(source->clockInfo), buffer, size));
3371     result =
3372         (UINT16)(result
3373             + UINT64_Marshal((UINT64*)&(source->firmwareVersion), buffer, size));
3374     result = (UINT16)(result
3375         + TPMU_ATTEST_Marshal((TPMU_ATTEST*)&(source->attested),
3376             buffer,
3377             size,
3378             (UINT32)source->type));
3379     return result;
3380 }
3381
3382 // Table "Definition of TPM2B_ATTEST Structure" (Part 2: Structures)
3383 UINT16
3384 TPM2B_ATTEST_Marshal(TPM2B_ATTEST* source, BYTE** buffer, INT32* size)
3385 {
3386     UINT16 result = 0;
3387     result =
3388         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
3389     // if size equal to 0, the rest of the structure is a zero buffer
3390     if(source->t.size == 0)
3391         return result;
3392     result = (UINT16)(result
3393         + BYTE_Array_Marshal((BYTE*)&(source->t.attestationData),
3394             buffer,
3395             size,
3396             (INT32)source->t.size));
3397     return result;
3398 }
3399
3400 // Table "Definition of TPMS_AUTH_COMMAND Structure" (Part 2: Structures)
3401 TPM_RC
3402 TPMS_AUTH_COMMAND_Unmarshal(TPMS_AUTH_COMMAND* target, BYTE** buffer, INT32* size)
3403 {
3404     TPM_RC result;
3405     result = TPMI_SH_AUTH_SESSION_Unmarshal(
3406         (TPMI_SH_AUTH_SESSION*)&(target->sessionHandle), buffer, size, 1);
3407     if(result == TPM_RC_SUCCESS)
3408         result = TPM2B_NONCE_Unmarshal((TPM2B_NONCE*)&(target->nonce), buffer, size);
3409     if(result == TPM_RC_SUCCESS)
3410         result = TPMA_SESSION_Unmarshal(

```

```

3411         (TPMA_SESSION*)&(target->sessionAttributes), buffer, size);
3412     if(result == TPM_RC_SUCCESS)
3413         result = TPM2B_AUTH_Unmarshal((TPM2B_AUTH*)&(target->hmac), buffer, size);
3414     return result;
3415 }
3416
3417 // Table "Definition of TPMS_AUTH_RESPONSE Structure" (Part 2: Structures)
3418 UINT16
3419 TPMS_AUTH_RESPONSE_Marshal(TPMS_AUTH_RESPONSE* source, BYTE** buffer, INT32* size)
3420 {
3421     UINT16 result = 0;
3422     result =
3423         (UINT16)(result
3424             + TPM2B_NONCE_Marshal((TPM2B_NONCE*)&(source->nonce), buffer, size));
3425     result = (UINT16)(result
3426         + TPMA_SESSION_Marshal(
3427             (TPMA_SESSION*)&(source->sessionAttributes), buffer, size));
3428     result =
3429         (UINT16)(result
3430             + TPM2B_AUTH_Marshal((TPM2B_AUTH*)&(source->hmac), buffer, size));
3431     return result;
3432 }
3433
3434 // Table "Definition of TPMI_AES_KEY_BITS Type" (Part 2: Structures)
3435 # if ALG_AES
3436 TPM_RC
3437 TPMI_AES_KEY_BITS_Unmarshal(TPMI_AES_KEY_BITS* target, BYTE** buffer, INT32* size)
3438 {
3439     TPM_RC result;
3440     result = TPM_KEY_BITS_Unmarshal((TPM_KEY_BITS*)target, buffer, size);
3441     if(result == TPM_RC_SUCCESS)
3442     {
3443         switch(*target)
3444         {
3445             # if AES_128
3446                 case 128:
3447             # endif // AES_128
3448             # if AES_192
3449                 case 192:
3450             # endif // AES_192
3451             # if AES_256
3452                 case 256:
3453             # endif // AES_256
3454                 break;
3455             default:
3456                 result = TPM_RC_VALUE;
3457                 break;
3458         }
3459     }
3460     return result;
3461 }
3462 # if !USE_MARSHALING_DEFINES
3463 UINT16
3464 TPMI_AES_KEY_BITS_Marshal(TPMI_AES_KEY_BITS* source, BYTE** buffer, INT32* size)
3465 {
3466     return TPM_KEY_BITS_Marshal((TPM_KEY_BITS*)source, buffer, size);
3467 }
3468 # endif // !USE_MARSHALING_DEFINES
3469 # endif // ALG_AES
3470
3471 // Table "Definition of TPMI_SM4_KEY_BITS Type" (Part 2: Structures)
3472 # if ALG_SM4
3473 TPM_RC
3474 TPMI_SM4_KEY_BITS_Unmarshal(TPMI_SM4_KEY_BITS* target, BYTE** buffer, INT32* size)
3475 {
3476     TPM_RC result;

```

```

3477     result = TPM_KEY_BITS_Unmarshal((TPM_KEY_BITS*)target, buffer, size);
3478     if(result == TPM_RC_SUCCESS)
3479     {
3480         switch(*target)
3481         {
3482             # if SM4_128
3483                 case 128:
3484             # endif // SM4_128
3485                 break;
3486             default:
3487                 result = TPM_RC_VALUE;
3488                 break;
3489         }
3490     }
3491     return result;
3492 }
3493 # if !USE_MARSHALING_DEFINES
3494 UINT16
3495 TPMI_SM4_KEY_BITS_Marshal(TPMI_SM4_KEY_BITS* source, BYTE** buffer, INT32* size)
3496 {
3497     return TPM_KEY_BITS_Marshal((TPM_KEY_BITS*)source, buffer, size);
3498 }
3499 # endif // !USE_MARSHALING_DEFINES
3500 # endif // ALG_SM4
3501
3502 // Table "Definition of TPMI_CAMELLIA_KEY_BITS Type" (Part 2: Structures)
3503 # if ALG_CAMELLIA
3504 TPM_RC
3505 TPMI_CAMELLIA_KEY_BITS_Unmarshal(
3506     TPMI_CAMELLIA_KEY_BITS* target, BYTE** buffer, INT32* size)
3507 {
3508     TPM_RC result;
3509     result = TPM_KEY_BITS_Unmarshal((TPM_KEY_BITS*)target, buffer, size);
3510     if(result == TPM_RC_SUCCESS)
3511     {
3512         switch(*target)
3513         {
3514             # if CAMELLIA_128
3515                 case 128:
3516             # endif // CAMELLIA_128
3517             # if CAMELLIA_192
3518                 case 192:
3519             # endif // CAMELLIA_192
3520             # if CAMELLIA_256
3521                 case 256:
3522             # endif // CAMELLIA_256
3523                 break;
3524             default:
3525                 result = TPM_RC_VALUE;
3526                 break;
3527         }
3528     }
3529     return result;
3530 }
3531 # if !USE_MARSHALING_DEFINES
3532 UINT16
3533 TPMI_CAMELLIA_KEY_BITS_Marshal(
3534     TPMI_CAMELLIA_KEY_BITS* source, BYTE** buffer, INT32* size)
3535 {
3536     return TPM_KEY_BITS_Marshal((TPM_KEY_BITS*)source, buffer, size);
3537 }
3538 # endif // !USE_MARSHALING_DEFINES
3539 # endif // ALG_CAMELLIA
3540
3541 // Table "Definition of TPMU_SYM_KEY_BITS Union" (Part 2: Structures)
3542 TPM_RC

```

```

3543 TPMU_SYM_KEY_BITS_Unmarshal(
3544     TPMU_SYM_KEY_BITS* target, BYTE** buffer, INT32* size, UINT32 selector)
3545 {
3546     switch(selector)
3547     {
3548     # if ALG_AES
3549         case TPM_ALG_AES:
3550             return TPMI_AES_KEY_BITS_Unmarshal(
3551                 (TPMI_AES_KEY_BITS*)&(target->aes), buffer, size);
3552     # endif // ALG_AES
3553     # if ALG_SM4
3554         case TPM_ALG_SM4:
3555             return TPMI_SM4_KEY_BITS_Unmarshal(
3556                 (TPMI_SM4_KEY_BITS*)&(target->sm4), buffer, size);
3557     # endif // ALG_SM4
3558     # if ALG_CAMELLIA
3559         case TPM_ALG_CAMELLIA:
3560             return TPMI_CAMELLIA_KEY_BITS_Unmarshal(
3561                 (TPMI_CAMELLIA_KEY_BITS*)&(target->camellia), buffer, size);
3562     # endif // ALG_CAMELLIA
3563     # if ALG_XOR
3564         case TPM_ALG_XOR:
3565             return TPMI_ALG_HASH_Unmarshal(
3566                 (TPMI_ALG_HASH*)&(target->xor), buffer, size, 0);
3567     # endif // ALG_XOR
3568         case TPM_ALG_NULL:
3569             return TPM_RC_SUCCESS;
3570     }
3571     return TPM_RC_SELECTOR;
3572 }
3573
3574 TPMU_SYM_KEY_BITS_Marshal(
3575     TPMU_SYM_KEY_BITS* source, BYTE** buffer, INT32* size, UINT32 selector)
3576 {
3577     switch(selector)
3578     {
3579     # if ALG_AES
3580         case TPM_ALG_AES:
3581             return TPMI_AES_KEY_BITS_Marshal(
3582                 (TPMI_AES_KEY_BITS*)&(source->aes), buffer, size);
3583     # endif // ALG_AES
3584     # if ALG_SM4
3585         case TPM_ALG_SM4:
3586             return TPMI_SM4_KEY_BITS_Marshal(
3587                 (TPMI_SM4_KEY_BITS*)&(source->sm4), buffer, size);
3588     # endif // ALG_SM4
3589     # if ALG_CAMELLIA
3590         case TPM_ALG_CAMELLIA:
3591             return TPMI_CAMELLIA_KEY_BITS_Marshal(
3592                 (TPMI_CAMELLIA_KEY_BITS*)&(source->camellia), buffer, size);
3593     # endif // ALG_CAMELLIA
3594     # if ALG_XOR
3595         case TPM_ALG_XOR:
3596             return TPMI_ALG_HASH_Marshal(
3597                 (TPMI_ALG_HASH*)&(source->xor), buffer, size);
3598     # endif // ALG_XOR
3599     }
3600     return 0;
3601 }
3602
3603 // Table "Definition of TPMU_SYM_MODE Union" (Part 2: Structures)
3604 TPM_RC
3605 TPMU_SYM_MODE_Unmarshal(
3606     TPMU_SYM_MODE* target, BYTE** buffer, INT32* size, UINT32 selector)
3607 {
3608     switch(selector)

```

```

3609     {
3610     # if ALG_AES
3611         case TPM_ALG_AES:
3612             return TPMI_ALG_SYM_MODE_Unmarshal(
3613                 (TPMI_ALG_SYM_MODE*)&(target->aes), buffer, size, 1);
3614     # endif // ALG_AES
3615     # if ALG_SM4
3616         case TPM_ALG_SM4:
3617             return TPMI_ALG_SYM_MODE_Unmarshal(
3618                 (TPMI_ALG_SYM_MODE*)&(target->sm4), buffer, size, 1);
3619     # endif // ALG_SM4
3620     # if ALG_CAMELLIA
3621         case TPM_ALG_CAMELLIA:
3622             return TPMI_ALG_SYM_MODE_Unmarshal(
3623                 (TPMI_ALG_SYM_MODE*)&(target->camellia), buffer, size, 1);
3624     # endif // ALG_CAMELLIA
3625     # if ALG_XOR
3626         case TPM_ALG_XOR:
3627             return TPM_RC_SUCCESS;
3628     # endif // ALG_XOR
3629         case TPM_ALG_NULL:
3630             return TPM_RC_SUCCESS;
3631     }
3632     return TPM_RC_SELECTOR;
3633 }
3634 UINT16
3635 TPMU_SYM_MODE_Marshal(
3636     TPMU_SYM_MODE* source, BYTE** buffer, INT32* size, UINT32 selector)
3637 {
3638     switch(selector)
3639     {
3640     # if ALG_AES
3641         case TPM_ALG_AES:
3642             return TPMI_ALG_SYM_MODE_Marshal(
3643                 (TPMI_ALG_SYM_MODE*)&(source->aes), buffer, size);
3644     # endif // ALG_AES
3645     # if ALG_SM4
3646         case TPM_ALG_SM4:
3647             return TPMI_ALG_SYM_MODE_Marshal(
3648                 (TPMI_ALG_SYM_MODE*)&(source->sm4), buffer, size);
3649     # endif // ALG_SM4
3650     # if ALG_CAMELLIA
3651         case TPM_ALG_CAMELLIA:
3652             return TPMI_ALG_SYM_MODE_Marshal(
3653                 (TPMI_ALG_SYM_MODE*)&(source->camellia), buffer, size);
3654     # endif // ALG_CAMELLIA
3655     }
3656     return 0;
3657 }
3658
3659 // Table "Definition of TPMT_SYM_DEF Structure" (Part 2: Structures)
3660 TPM_RC
3661 TPMT_SYM_DEF_Unmarshal(TPMT_SYM_DEF* target, BYTE** buffer, INT32* size, BOOL flag)
3662 {
3663     TPM_RC result;
3664     result = TPMI_ALG_SYM_Unmarshal(
3665         (TPMI_ALG_SYM*)&(target->algorithm), buffer, size, flag);
3666     if(result == TPM_RC_SUCCESS)
3667         result = TPMU_SYM_KEY_BITS_Unmarshal((TPMU_SYM_KEY_BITS*)&(target->keyBits),
3668             buffer,
3669             size,
3670             (UINT32)target->algorithm);
3671     if(result == TPM_RC_SUCCESS)
3672         result = TPMU_SYM_MODE_Unmarshal(
3673             (TPMU_SYM_MODE*)&(target->mode), buffer, size, (UINT32)target->algorithm);
3674     return result;

```



```

3675 }
3676 UINT16
3677 TPMT_SYM_DEF_Marshal(TPMT_SYM_DEF* source, BYTE** buffer, INT32* size)
3678 {
3679     UINT16 result = 0;
3680     result = (UINT16)(result
3681         + TPMI_ALG_SYM_Marshal(
3682             (TPMI_ALG_SYM*)&(source->algorithm), buffer, size));
3683     result =
3684         (UINT16)(result
3685             + TPMU_SYM_KEY_BITS_Marshal((TPMU_SYM_KEY_BITS*)&(source->keyBits),
3686                 buffer,
3687                 size,
3688                 (UINT32)source->algorithm));
3689     result = (UINT16)(result
3690         + TPMU_SYM_MODE_Marshal((TPMU_SYM_MODE*)&(source->mode),
3691             buffer,
3692             size,
3693             (UINT32)source->algorithm));
3694     return result;
3695 }
3696
3697 // Table "Definition of TPMT_SYM_DEF_OBJECT Structure" (Part 2: Structures)
3698 TPM_RC
3699 TPMT_SYM_DEF_OBJECT_Unmarshal(
3700     TPMT_SYM_DEF_OBJECT* target, BYTE** buffer, INT32* size, BOOL flag)
3701 {
3702     TPM_RC result;
3703     result = TPMI_ALG_SYM_OBJECT_Unmarshal(
3704         (TPMI_ALG_SYM_OBJECT*)&(target->algorithm), buffer, size, flag);
3705     if(result == TPM_RC_SUCCESS)
3706         result = TPMU_SYM_KEY_BITS_Unmarshal((TPMU_SYM_KEY_BITS*)&(target->keyBits),
3707             buffer,
3708             size,
3709             (UINT32)target->algorithm);
3710     if(result == TPM_RC_SUCCESS)
3711         result = TPMU_SYM_MODE_Unmarshal(
3712             (TPMU_SYM_MODE*)&(target->mode), buffer, size, (UINT32)target->algorithm);
3713     return result;
3714 }
3715
3716 TPMU_SYM_DEF_OBJECT_Marshal(TPMT_SYM_DEF_OBJECT* source, BYTE** buffer, INT32* size)
3717 {
3718     UINT16 result = 0;
3719     result = (UINT16)(result
3720         + TPMI_ALG_SYM_OBJECT_Marshal(
3721             (TPMI_ALG_SYM_OBJECT*)&(source->algorithm), buffer, size));
3722     result =
3723         (UINT16)(result
3724             + TPMU_SYM_KEY_BITS_Marshal((TPMU_SYM_KEY_BITS*)&(source->keyBits),
3725                 buffer,
3726                 size,
3727                 (UINT32)source->algorithm));
3728     result = (UINT16)(result
3729         + TPMU_SYM_MODE_Marshal((TPMU_SYM_MODE*)&(source->mode),
3730             buffer,
3731             size,
3732             (UINT32)source->algorithm));
3733     return result;
3734 }
3735
3736 // Table "Definition of TPM2B_SYM_KEY Structure" (Part 2: Structures)
3737 TPM_RC
3738 TPM2B_SYM_KEY_Unmarshal(TPM2B_SYM_KEY* target, BYTE** buffer, INT32* size)
3739 {
3740     TPM_RC result;

```

```

3741     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
3742     if((result == TPM_RC_SUCCESS) && (target->t.size > MAX_SYM_KEY_BYTES))
3743         result = TPM_RC_SIZE;
3744     if(result == TPM_RC_SUCCESS)
3745         result = BYTE_Array_Unmarshal(
3746             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
3747     return result;
3748 }
3749 UINT16
3750 TPM2B_SYM_KEY_Marshal(TPM2B_SYM_KEY* source, BYTE** buffer, INT32* size)
3751 {
3752     UINT16 result = 0;
3753     result =
3754         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
3755     // if size equal to 0, the rest of the structure is a zero buffer
3756     if(source->t.size == 0)
3757         return result;
3758     result = (UINT16)(result
3759         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
3760             buffer,
3761             size,
3762             (INT32)source->t.size));
3763     return result;
3764 }
3765
3766 // Table "Definition of TPMS_SYMCIPHER_PARMS Structure" (Part 2: Structures)
3767 TPM_RC
3768 TPMS_SYMCIPHER_PARMS_Unmarshal(
3769     TPMS_SYMCIPHER_PARMS* target, BYTE** buffer, INT32* size)
3770 {
3771     TPM_RC result;
3772     result = TPMT_SYM_DEF_OBJECT_Unmarshal(
3773         (TPMT_SYM_DEF_OBJECT*)&(target->sym), buffer, size, 0);
3774     return result;
3775 }
3776 UINT16
3777 TPMS_SYMCIPHER_PARMS_Marshal(TPMS_SYMCIPHER_PARMS* source, BYTE** buffer, INT32* size)
3778 {
3779     UINT16 result = 0;
3780     result = (UINT16)(result
3781         + TPMT_SYM_DEF_OBJECT_Marshal(
3782             (TPMT_SYM_DEF_OBJECT*)&(source->sym), buffer, size));
3783     return result;
3784 }
3785
3786 // Table "Definition of TPM2B_LABEL Structure" (Part 2: Structures)
3787 TPM_RC
3788 TPM2B_LABEL_Unmarshal(TPM2B_LABEL* target, BYTE** buffer, INT32* size)
3789 {
3790     TPM_RC result;
3791     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
3792     if((result == TPM_RC_SUCCESS) && (target->t.size > LABEL_MAX_BUFFER))
3793         result = TPM_RC_SIZE;
3794     if(result == TPM_RC_SUCCESS)
3795         result = BYTE_Array_Unmarshal(
3796             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
3797     return result;
3798 }
3799 UINT16
3800 TPM2B_LABEL_Marshal(TPM2B_LABEL* source, BYTE** buffer, INT32* size)
3801 {
3802     UINT16 result = 0;
3803     result =
3804         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
3805     // if size equal to 0, the rest of the structure is a zero buffer
3806     if(source->t.size == 0)

```

```

3807         return result;
3808     result = (UINT16)(result
3809         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
3810             buffer,
3811             size,
3812             (INT32)source->t.size));
3813     return result;
3814 }
3815
3816 // Table "Definition of TPMS_DERIVE Structure" (Part 2: Structures)
3817 TPM_RC
3818 TPMS_DERIVE_Unmarshal(TPMS_DERIVE* target, BYTE** buffer, INT32* size)
3819 {
3820     TPM_RC result;
3821     result = TPM2B_LABEL_Unmarshal((TPM2B_LABEL*)&(target->label), buffer, size);
3822     if(result == TPM_RC_SUCCESS)
3823         result =
3824             TPM2B_LABEL_Unmarshal((TPM2B_LABEL*)&(target->context), buffer, size);
3825     return result;
3826 }
3827
3828 TPM_RC
3829 TPMS_DERIVE_Marshal(TPMS_DERIVE* source, BYTE** buffer, INT32* size)
3830 {
3831     UINT16 result = 0;
3832     result =
3833         (UINT16)(result
3834             + TPM2B_LABEL_Marshal((TPM2B_LABEL*)&(source->label), buffer, size));
3835     result = (UINT16)(result
3836         + TPM2B_LABEL_Marshal(
3837             (TPM2B_LABEL*)&(source->context), buffer, size));
3838     return result;
3839 }
3840
3841 // Table "Definition of TPM2B_DERIVE Structure" (Part 2: Structures)
3842 TPM_RC
3843 TPM2B_DERIVE_Unmarshal(TPM2B_DERIVE* target, BYTE** buffer, INT32* size)
3844 {
3845     TPM_RC result;
3846     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
3847     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(TPMS_DERIVE)))
3848         result = TPM_RC_SIZE;
3849     if(result == TPM_RC_SUCCESS)
3850         result = BYTE_Array_Unmarshal(
3851             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
3852     return result;
3853 }
3854
3855 TPM_RC
3856 TPM2B_DERIVE_Marshal(TPM2B_DERIVE* source, BYTE** buffer, INT32* size)
3857 {
3858     UINT16 result = 0;
3859     result =
3860         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
3861     // if size equal to 0, the rest of the structure is a zero buffer
3862     if(source->t.size == 0)
3863         return result;
3864     result = (UINT16)(result
3865         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
3866             buffer,
3867             size,
3868             (INT32)source->t.size));
3869     return result;
3870 }
3871
3872 // Table "Definition of TPM2B_SENSITIVE_DATA Structure" (Part 2: Structures)
3873 TPM_RC
3874 TPM2B_SENSITIVE_DATA_Unmarshal(

```

```

3873     TPM2B_SENSITIVE_DATA* target, BYTE** buffer, INT32* size)
3874 {
3875     TPM_RC result;
3876     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
3877     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(TPMU_SENSITIVE_CREATE)))
3878         result = TPM_RC_SIZE;
3879     if(result == TPM_RC_SUCCESS)
3880         result = BYTE_Array_Unmarshal(
3881             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
3882     return result;
3883 }
3884
3885 UINT16
3886 TPM2B_SENSITIVE_DATA_Marshal(TPM2B_SENSITIVE_DATA* source, BYTE** buffer, INT32* size)
3887 {
3888     UINT16 result = 0;
3889     result =
3890         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
3891     // if size equal to 0, the rest of the structure is a zero buffer
3892     if(source->t.size == 0)
3893         return result;
3894     result = (UINT16)(result
3895         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
3896             buffer,
3897             size,
3898             (INT32)source->t.size));
3899     return result;
3900 }
3901
3902 // Table "Definition of TPMS_SENSITIVE_CREATE Structure" (Part 2: Structures)
3903 TPM_RC
3904 TPMS_SENSITIVE_CREATE_Unmarshal(
3905     TPMS_SENSITIVE_CREATE* target, BYTE** buffer, INT32* size)
3906 {
3907     TPM_RC result;
3908     result = TPM2B_AUTH_Unmarshal((TPM2B_AUTH*)&(target->userAuth), buffer, size);
3909     if(result == TPM_RC_SUCCESS)
3910         result = TPM2B_SENSITIVE_DATA_Unmarshal(
3911             (TPM2B_SENSITIVE_DATA*)&(target->data), buffer, size);
3912     return result;
3913 }
3914
3915 // Table "Definition of TPM2B_SENSITIVE_CREATE Structure" (Part 2: Structures)
3916 TPM_RC
3917 TPM2B_SENSITIVE_CREATE_Unmarshal(
3918     TPM2B_SENSITIVE_CREATE* target, BYTE** buffer, INT32* size)
3919 {
3920     TPM_RC result;
3921     result = UINT16_Unmarshal((UINT16*)&(target->size), buffer, size);
3922     if(result == TPM_RC_SUCCESS)
3923     {
3924         // if size is zero, then the required structure is missing
3925         if(target->size == 0)
3926             result = TPM_RC_SIZE;
3927         else
3928         {
3929             INT32 startSize = *size;
3930             result = TPMS_SENSITIVE_CREATE_Unmarshal(
3931                 (TPMS_SENSITIVE_CREATE*)&(target->sensitive), buffer, size);
3932             if((result == TPM_RC_SUCCESS) && (target->size != (startSize - *size)))
3933                 result = TPM_RC_SIZE;
3934         }
3935     }
3936     return result;
3937 }
3938
3939 // Table "Definition of TPMS_SCHEME_HASH Structure" (Part 2: Structures)

```

```

3939 TPM_RC
3940 TPMS_SCHEME_HASH_Unmarshal(TPMS_SCHEME_HASH* target, BYTE** buffer, INT32* size)
3941 {
3942     TPM_RC result;
3943     result =
3944         TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH*)&(target->hashAlg), buffer, size, 0);
3945     return result;
3946 }
3947 UINT16
3948 TPMS_SCHEME_HASH_Marshal(TPMS_SCHEME_HASH* source, BYTE** buffer, INT32* size)
3949 {
3950     UINT16 result = 0;
3951     result = (UINT16)(result
3952         + TPMI_ALG_HASH_Marshal(
3953             (TPMI_ALG_HASH*)&(source->hashAlg), buffer, size));
3954     return result;
3955 }
3956
3957 // Table "Definition of TPMS_SCHEME_ECDSA Structure" (Part 2: Structures)
3958 # if ALG_ECC
3959 TPM_RC
3960 TPMS_SCHEME_ECDSA_Unmarshal(TPMS_SCHEME_ECDSA* target, BYTE** buffer, INT32* size)
3961 {
3962     TPM_RC result;
3963     result =
3964         TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH*)&(target->hashAlg), buffer, size, 0);
3965     if(result == TPM_RC_SUCCESS)
3966         result = UINT16_Unmarshal((UINT16*)&(target->count), buffer, size);
3967     return result;
3968 }
3969 UINT16
3970 TPMS_SCHEME_ECDSA_Marshal(TPMS_SCHEME_ECDSA* source, BYTE** buffer, INT32* size)
3971 {
3972     UINT16 result = 0;
3973     result = (UINT16)(result
3974         + TPMI_ALG_HASH_Marshal(
3975             (TPMI_ALG_HASH*)&(source->hashAlg), buffer, size));
3976     result =
3977         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->count), buffer, size));
3978     return result;
3979 }
3980 # endif // ALG_ECC
3981
3982 // Table "Definition of TPMI_ALG_KEYEDHASH_SCHEME Type" (Part 2: Structures)
3983 TPM_RC
3984 TPMI_ALG_KEYEDHASH_SCHEME_Unmarshal(
3985     TPMI_ALG_KEYEDHASH_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
3986 {
3987     TPM_RC result;
3988     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
3989     if(result == TPM_RC_SUCCESS)
3990     {
3991         switch(*target)
3992         {
3993             # if ALG_HMAC
3994                 case TPM_ALG_HMAC:
3995             # endif // ALG_HMAC
3996             # if ALG_XOR
3997                 case TPM_ALG_XOR:
3998             # endif // ALG_XOR
3999                 break;
4000             default:
4001                 if((*target != TPM_ALG_NULL) || !flag)
4002                     result = TPM_RC_VALUE;
4003                 break;
4004         }

```

```

4005     }
4006     return result;
4007 }
4008 # if !USE_MARSHALING_DEFINES
4009 UINT16
4010 TPMI_ALG_KEYEDHASH_SCHEME_Marshal(
4011     TPMI_ALG_KEYEDHASH_SCHEME* source, BYTE** buffer, INT32* size)
4012 {
4013     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
4014 }
4015 # endif // !USE_MARSHALING_DEFINES
4016
4017 // Table "Definition of Types for HMAC_SIG_SCHEME" (Part 2: Structures)
4018 # if !USE_MARSHALING_DEFINES
4019 TPM_RC
4020 TPMS_SCHEME_HMAC_Unmarshal(TPMS_SCHEME_HMAC* target, BYTE** buffer, INT32* size)
4021 {
4022     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4023 }
4024
4025 TPMS_SCHEME_HMAC_Marshal(TPMS_SCHEME_HMAC* source, BYTE** buffer, INT32* size)
4026 {
4027     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4028 }
4029 # endif // !USE_MARSHALING_DEFINES
4030
4031 // Table "Definition of TPMS_SCHEME_XOR Structure" (Part 2: Structures)
4032 TPM_RC
4033 TPMS_SCHEME_XOR_Unmarshal(TPMS_SCHEME_XOR* target, BYTE** buffer, INT32* size)
4034 {
4035     TPM_RC result;
4036     result =
4037         TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH*)&(target->hashAlg), buffer, size, 0);
4038     if(result == TPM_RC_SUCCESS)
4039         result =
4040             TPMI_ALG_KDF_Unmarshal((TPMI_ALG_KDF*)&(target->kdf), buffer, size, 1);
4041     return result;
4042 }
4043
4044 TPMS_SCHEME_XOR_Marshal(TPMS_SCHEME_XOR* source, BYTE** buffer, INT32* size)
4045 {
4046     UINT16 result = 0;
4047     result = (UINT16)(result
4048         + TPMI_ALG_HASH_Marshal(
4049             (TPMI_ALG_HASH*)&(source->hashAlg), buffer, size));
4050     result =
4051         (UINT16)(result
4052             + TPMI_ALG_KDF_Marshal((TPMI_ALG_KDF*)&(source->kdf), buffer, size));
4053     return result;
4054 }
4055
4056 // Table "Definition of TPMU_SCHEME_KEYEDHASH Union" (Part 2: Structures)
4057 TPM_RC
4058 TPMU_SCHEME_KEYEDHASH_Unmarshal(
4059     TPMU_SCHEME_KEYEDHASH* target, BYTE** buffer, INT32* size, UINT32 selector)
4060 {
4061     switch(selector)
4062     {
4063     # if ALG_HMAC
4064         case TPM_ALG_HMAC:
4065             return TPMS_SCHEME_HMAC_Unmarshal(
4066                 (TPMS_SCHEME_HMAC*)&(target->hmac), buffer, size);
4067     # endif // ALG_HMAC
4068     # if ALG_XOR
4069         case TPM_ALG_XOR:
4070             return TPMS_SCHEME_XOR_Unmarshal(

```



```

4071         (TPMS_SCHEME_XOR*)&(target->xor), buffer, size);
4072 # endif // ALG_XOR
4073     case TPM_ALG_NULL:
4074         return TPM_RC_SUCCESS;
4075     }
4076     return TPM_RC_SELECTOR;
4077 }
4078 UINT16
4079 TPMU_SCHEME_KEYEDHASH_Marshal(
4080     TPMU_SCHEME_KEYEDHASH* source, BYTE** buffer, INT32* size, UINT32 selector)
4081 {
4082     switch(selector)
4083     {
4084 # if ALG_HMAC
4085         case TPM_ALG_HMAC:
4086             return TPMS_SCHEME_HMAC_Marshal(
4087                 (TPMS_SCHEME_HMAC*)&(source->hmac), buffer, size);
4088 # endif // ALG_HMAC
4089 # if ALG_XOR
4090         case TPM_ALG_XOR:
4091             return TPMS_SCHEME_XOR_Marshal(
4092                 (TPMS_SCHEME_XOR*)&(source->xor), buffer, size);
4093 # endif // ALG_XOR
4094     }
4095     return 0;
4096 }
4097
4098 // Table "Definition of TPMT_KEYEDHASH_SCHEME Structure" (Part 2: Structures)
4099 TPM_RC
4100 TPMT_KEYEDHASH_SCHEME_Unmarshal(
4101     TPMT_KEYEDHASH_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
4102 {
4103     TPM_RC result;
4104     result = TPMI_ALG_KEYEDHASH_SCHEME_Unmarshal(
4105         (TPMI_ALG_KEYEDHASH_SCHEME*)&(target->scheme), buffer, size, flag);
4106     if(result == TPM_RC_SUCCESS)
4107         result = TPMU_SCHEME_KEYEDHASH_Unmarshal(
4108             (TPMU_SCHEME_KEYEDHASH*)&(target->details),
4109             buffer,
4110             size,
4111             (UINT32)target->scheme);
4112     return result;
4113 }
4114
4115 TPMI_ALG_KEYEDHASH_SCHEME_Marshal(
4116     TPMT_KEYEDHASH_SCHEME* source, BYTE** buffer, INT32* size)
4117 {
4118     UINT16 result = 0;
4119     result =
4120         (UINT16)(result
4121             + TPMI_ALG_KEYEDHASH_SCHEME_Marshal(
4122                 (TPMI_ALG_KEYEDHASH_SCHEME*)&(source->scheme), buffer, size));
4123     result = (UINT16)(result
4124         + TPMU_SCHEME_KEYEDHASH_Marshal(
4125             (TPMU_SCHEME_KEYEDHASH*)&(source->details),
4126             buffer,
4127             size,
4128             (UINT32)source->scheme));
4129     return result;
4130 }
4131
4132 // Table "Definition of Types for RSA Signature Schemes" (Part 2: Structures)
4133 # if !USE_MARSHALING_DEFINES
4134 TPM_RC
4135 TPMS_SIG_SCHEME_RSASSA_Unmarshal(
4136     TPMS_SIG_SCHEME_RSASSA* target, BYTE** buffer, INT32* size)

```

```

4137 {
4138     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4139 }
4140 UINT16
4141 TPMS_SIG_SCHEME_RSASSA_Marshal(
4142     TPMS_SIG_SCHEME_RSASSA* source, BYTE** buffer, INT32* size)
4143 {
4144     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4145 }
4146 TPM_RC
4147 TPMS_SIG_SCHEME_RSAPSS_Unmarshal(
4148     TPMS_SIG_SCHEME_RSAPSS* target, BYTE** buffer, INT32* size)
4149 {
4150     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4151 }
4152 UINT16
4153 TPMS_SIG_SCHEME_RSAPSS_Marshal(
4154     TPMS_SIG_SCHEME_RSAPSS* source, BYTE** buffer, INT32* size)
4155 {
4156     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4157 }
4158 # endif // !USE_MARSHALING_DEFINES
4159
4160 // Table "Definition of Types for ECC Signature Schemes" (Part 2: Structures)
4161 # if !USE_MARSHALING_DEFINES
4162 TPM_RC
4163 TPMS_SIG_SCHEME_ECDSA_Unmarshal(
4164     TPMS_SIG_SCHEME_ECDSA* target, BYTE** buffer, INT32* size)
4165 {
4166     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4167 }
4168 UINT16
4169 TPMS_SIG_SCHEME_ECDSA_Marshal(
4170     TPMS_SIG_SCHEME_ECDSA* source, BYTE** buffer, INT32* size)
4171 {
4172     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4173 }
4174 TPM_RC
4175 TPMS_SIG_SCHEME_ECDSA_Unmarshal(
4176     TPMS_SIG_SCHEME_ECDSA* target, BYTE** buffer, INT32* size)
4177 {
4178     return TPMS_SCHEME_ECDSA_Unmarshal((TPMS_SCHEME_ECDSA*)target, buffer, size);
4179 }
4180 UINT16
4181 TPMS_SIG_SCHEME_ECDSA_Marshal(
4182     TPMS_SIG_SCHEME_ECDSA* source, BYTE** buffer, INT32* size)
4183 {
4184     return TPMS_SCHEME_ECDSA_Marshal((TPMS_SCHEME_ECDSA*)source, buffer, size);
4185 }
4186 TPM_RC
4187 TPMS_SIG_SCHEME_SM2_Unmarshal(TPMS_SIG_SCHEME_SM2* target, BYTE** buffer, INT32* size)
4188 {
4189     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4190 }
4191 UINT16
4192 TPMS_SIG_SCHEME_SM2_Marshal(TPMS_SIG_SCHEME_SM2* source, BYTE** buffer, INT32* size)
4193 {
4194     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4195 }
4196 TPM_RC
4197 TPMS_SIG_SCHEME_ECSCHNORR_Unmarshal(
4198     TPMS_SIG_SCHEME_ECSCHNORR* target, BYTE** buffer, INT32* size)
4199 {
4200     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4201 }
4202 UINT16

```

```

4203 TPMS_SIG_SCHEME_EC Schnorr_Marshal(
4204     TPMS_SIG_SCHEME_EC Schnorr* source, BYTE** buffer, INT32* size)
4205 {
4206     return TPMS_SIG_SCHEME_HASH_Marshal((TPMS_SIG_SCHEME_HASH*)source, buffer, size);
4207 }
4208 TPM_RC
4209 TPMS_SIG_SCHEME_EDDSA_Unmarshal(
4210     TPMS_SIG_SCHEME_EDDSA* target, BYTE** buffer, INT32* size)
4211 {
4212     return TPMS_SIG_SCHEME_HASH_Unmarshal((TPMS_SIG_SCHEME_HASH*)target, buffer, size);
4213 }
4214 UINT16
4215 TPMS_SIG_SCHEME_EDDSA_Marshal(
4216     TPMS_SIG_SCHEME_EDDSA* source, BYTE** buffer, INT32* size)
4217 {
4218     return TPMS_SIG_SCHEME_HASH_Marshal((TPMS_SIG_SCHEME_HASH*)source, buffer, size);
4219 }
4220 TPM_RC
4221 TPMS_SIG_SCHEME_EDDSA_PH_Unmarshal(
4222     TPMS_SIG_SCHEME_EDDSA_PH* target, BYTE** buffer, INT32* size)
4223 {
4224     return TPMS_SIG_SCHEME_HASH_Unmarshal((TPMS_SIG_SCHEME_HASH*)target, buffer, size);
4225 }
4226 UINT16
4227 TPMS_SIG_SCHEME_EDDSA_PH_Marshal(
4228     TPMS_SIG_SCHEME_EDDSA_PH* source, BYTE** buffer, INT32* size)
4229 {
4230     return TPMS_SIG_SCHEME_HASH_Marshal((TPMS_SIG_SCHEME_HASH*)source, buffer, size);
4231 }
4232 # endif // !USE_MARSHALING_DEFINES
4233
4234 // Table "Definition of TPMU_SIG_SCHEME Union" (Part 2: Structures)
4235 TPM_RC
4236 TPMU_SIG_SCHEME_Unmarshal(
4237     TPMU_SIG_SCHEME* target, BYTE** buffer, INT32* size, UINT32 selector)
4238 {
4239     switch(selector)
4240     {
4241 # if ALG_HMAC
4242         case TPM_ALG_HMAC:
4243             return TPMS_SIG_SCHEME_HMAC_Unmarshal(
4244                 (TPMS_SIG_SCHEME_HMAC*)&(target->hmac), buffer, size);
4245 # endif // ALG_HMAC
4246 # if ALG_RSASSA
4247         case TPM_ALG_RSASSA:
4248             return TPMS_SIG_SCHEME_RSASSA_Unmarshal(
4249                 (TPMS_SIG_SCHEME_RSASSA*)&(target->rsassa), buffer, size);
4250 # endif // ALG_RSASSA
4251 # if ALG_RSAPSS
4252         case TPM_ALG_RSAPSS:
4253             return TPMS_SIG_SCHEME_RSAPSS_Unmarshal(
4254                 (TPMS_SIG_SCHEME_RSAPSS*)&(target->rsapss), buffer, size);
4255 # endif // ALG_RSAPSS
4256 # if ALG_ECDSA
4257         case TPM_ALG_ECDSA:
4258             return TPMS_SIG_SCHEME_ECDSA_Unmarshal(
4259                 (TPMS_SIG_SCHEME_ECDSA*)&(target->ecdsa), buffer, size);
4260 # endif // ALG_ECDSA
4261 # if ALG_ECDA
4262         case TPM_ALG_ECDA:
4263             return TPMS_SIG_SCHEME_ECDA_Unmarshal(
4264                 (TPMS_SIG_SCHEME_ECDA*)&(target->ecda), buffer, size);
4265 # endif // ALG_ECDA
4266 # if ALG_SM2
4267         case TPM_ALG_SM2:
4268             return TPMS_SIG_SCHEME_SM2_Unmarshal(

```

```

4269         (TPMS_SIG_SCHEME_SM2*)&(target->sm2), buffer, size);
4270 # endif // ALG_SM2
4271 # if ALG_EC Schnorr
4272     case TPM_ALG_EC Schnorr:
4273         return TPMS_SIG_SCHEME_EC Schnorr_Unmarshal(
4274             (TPMS_SIG_SCHEME_EC Schnorr*)&(target->ecschnorr), buffer, size);
4275 # endif // ALG_EC Schnorr
4276 # if ALG_EDDSA
4277     case TPM_ALG_EDDSA:
4278         return TPMS_SIG_SCHEME_EDDSA_Unmarshal(
4279             (TPMS_SIG_SCHEME_EDDSA*)&(target->eddsa), buffer, size);
4280 # endif // ALG_EDDSA
4281 # if ALG_EDDSA_PH
4282     case TPM_ALG_EDDSA_PH:
4283         return TPMS_SIG_SCHEME_EDDSA_PH_Unmarshal(
4284             (TPMS_SIG_SCHEME_EDDSA_PH*)&(target->eddsa_ph), buffer, size);
4285 # endif // ALG_EDDSA_PH
4286 # if ALG_LMS
4287     case TPM_ALG_LMS:
4288         return TPMS_SIG_SCHEME_LMS_Unmarshal(
4289             (TPMS_SIG_SCHEME_LMS*)&(target->lms), buffer, size);
4290 # endif // ALG_LMS
4291 # if ALG_XMSS
4292     case TPM_ALG_XMSS:
4293         return TPMS_SIG_SCHEME_XMSS_Unmarshal(
4294             (TPMS_SIG_SCHEME_XMSS*)&(target->xmss), buffer, size);
4295 # endif // ALG_XMSS
4296     case TPM_ALG_NULL:
4297         return TPM_RC_SUCCESS;
4298 }
4299 return TPM_RC_SELECTOR;
4300 }
4301 UINT16
4302 TPMU_SIG_SCHEME_Marshal(
4303     TPMU_SIG_SCHEME* source, BYTE** buffer, INT32* size, UINT32 selector)
4304 {
4305     switch(selector)
4306     {
4307 # if ALG_HMAC
4308     case TPM_ALG_HMAC:
4309         return TPMS_SCHEME_HMAC_Marshal(
4310             (TPMS_SCHEME_HMAC*)&(source->hmac), buffer, size);
4311 # endif // ALG_HMAC
4312 # if ALG_RSASSA
4313     case TPM_ALG_RSASSA:
4314         return TPMS_SIG_SCHEME_RSASSA_Marshal(
4315             (TPMS_SIG_SCHEME_RSASSA*)&(source->rsassa), buffer, size);
4316 # endif // ALG_RSASSA
4317 # if ALG_RSAPSS
4318     case TPM_ALG_RSAPSS:
4319         return TPMS_SIG_SCHEME_RSAPSS_Marshal(
4320             (TPMS_SIG_SCHEME_RSAPSS*)&(source->rsapss), buffer, size);
4321 # endif // ALG_RSAPSS
4322 # if ALG_ECDSA
4323     case TPM_ALG_ECDSA:
4324         return TPMS_SIG_SCHEME_ECDSA_Marshal(
4325             (TPMS_SIG_SCHEME_ECDSA*)&(source->ecdsa), buffer, size);
4326 # endif // ALG_ECDSA
4327 # if ALG_ECDA
4328     case TPM_ALG_ECDA:
4329         return TPMS_SIG_SCHEME_ECDA_Marshal(
4330             (TPMS_SIG_SCHEME_ECDA*)&(source->ecda), buffer, size);
4331 # endif // ALG_ECDA
4332 # if ALG_SM2
4333     case TPM_ALG_SM2:
4334         return TPMS_SIG_SCHEME_SM2_Marshal(

```

```

4335         (TPMS_SIG_SCHEME_SM2*)&(source->sm2), buffer, size);
4336 # endif // ALG_SM2
4337 # if ALG_ECSCNORR
4338     case TPM_ALG_ECSCNORR:
4339         return TPMS_SIG_SCHEME_ECSCNORR_Marshal(
4340             (TPMS_SIG_SCHEME_ECSCNORR*)&(source->ecschnorr), buffer, size);
4341 # endif // ALG_ECSCNORR
4342 # if ALG_EDDSA
4343     case TPM_ALG_EDDSA:
4344         return TPMS_SIG_SCHEME_EDDSA_Marshal(
4345             (TPMS_SIG_SCHEME_EDDSA*)&(source->eddsa), buffer, size);
4346 # endif // ALG_EDDSA
4347 # if ALG_EDDSA_PH
4348     case TPM_ALG_EDDSA_PH:
4349         return TPMS_SIG_SCHEME_EDDSA_PH_Marshal(
4350             (TPMS_SIG_SCHEME_EDDSA_PH*)&(source->eddsa_ph), buffer, size);
4351 # endif // ALG_EDDSA_PH
4352 # if ALG_LMS
4353     case TPM_ALG_LMS:
4354         return TPMS_SIG_SCHEME_LMS_Marshal(
4355             (TPMS_SIG_SCHEME_LMS*)&(source->lms), buffer, size);
4356 # endif // ALG_LMS
4357 # if ALG_XMSS
4358     case TPM_ALG_XMSS:
4359         return TPMS_SIG_SCHEME_XMSS_Marshal(
4360             (TPMS_SIG_SCHEME_XMSS*)&(source->xmss), buffer, size);
4361 # endif // ALG_XMSS
4362     }
4363     return 0;
4364 }
4365
4366 // Table "Definition of TPMT_SIG_SCHEME Structure" (Part 2: Structures)
4367 TPM_RC
4368 TPMT_SIG_SCHEME_Unmarshal(
4369     TPMT_SIG_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
4370 {
4371     TPM_RC result;
4372     result = TPMI_ALG_SIG_SCHEME_Unmarshal(
4373         (TPMI_ALG_SIG_SCHEME*)&(target->scheme), buffer, size, flag);
4374     if(result == TPM_RC_SUCCESS)
4375         result = TPMU_SIG_SCHEME_Unmarshal((TPMU_SIG_SCHEME*)&(target->details),
4376             buffer,
4377             size,
4378             (UINT32)target->scheme);
4379     return result;
4380 }
4381 UINT16
4382 TPMT_SIG_SCHEME_Marshal(TPMT_SIG_SCHEME* source, BYTE** buffer, INT32* size)
4383 {
4384     UINT16 result = 0;
4385     result = (UINT16)(result
4386         + TPMI_ALG_SIG_SCHEME_Marshal(
4387             (TPMI_ALG_SIG_SCHEME*)&(source->scheme), buffer, size));
4388     result = (UINT16)(result
4389         + TPMU_SIG_SCHEME_Marshal((TPMU_SIG_SCHEME*)&(source->details),
4390             buffer,
4391             size,
4392             (UINT32)source->scheme));
4393     return result;
4394 }
4395
4396 // Table "Definition of Types for Encryption Schemes" (Part 2: Structures)
4397 # if !USE_MARSHALING_DEFINES
4398 TPM_RC
4399 TPMS_ENC_SCHEME_RSAES_Unmarshal(
4400     TPMS_ENC_SCHEME_RSAES* target, BYTE** buffer, INT32* size)

```



```

4401 {
4402     return TPMS_EMPTY_Unmarshal((TPMS_EMPTY*)target, buffer, size);
4403 }
4404 UINT16
4405 TPMS_ENC_SCHEME_RSAES_Marshal(
4406     TPMS_ENC_SCHEME_RSAES* source, BYTE** buffer, INT32* size)
4407 {
4408     return TPMS_EMPTY_Marshal((TPMS_EMPTY*)source, buffer, size);
4409 }
4410 TPM_RC
4411 TPMS_ENC_SCHEME_OAEP_Unmarshal(
4412     TPMS_ENC_SCHEME_OAEP* target, BYTE** buffer, INT32* size)
4413 {
4414     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4415 }
4416 UINT16
4417 TPMS_ENC_SCHEME_OAEP_Marshal(TPMS_ENC_SCHEME_OAEP* source, BYTE** buffer, INT32* size)
4418 {
4419     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4420 }
4421 # endif // !USE_MARSHALING_DEFINES
4422
4423 // Table "Definition of Types for ECC Key Exchange" (Part 2: Structures)
4424 # if !USE_MARSHALING_DEFINES
4425 TPM_RC
4426 TPMS_KEY_SCHEME_ECDH_Unmarshal(
4427     TPMS_KEY_SCHEME_ECDH* target, BYTE** buffer, INT32* size)
4428 {
4429     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4430 }
4431 UINT16
4432 TPMS_KEY_SCHEME_ECDH_Marshal(TPMS_KEY_SCHEME_ECDH* source, BYTE** buffer, INT32* size)
4433 {
4434     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4435 }
4436 TPM_RC
4437 TPMS_KEY_SCHEME_SM2_Unmarshal(TPMS_KEY_SCHEME_SM2* target, BYTE** buffer, INT32* size)
4438 {
4439     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4440 }
4441 UINT16
4442 TPMS_KEY_SCHEME_SM2_Marshal(TPMS_KEY_SCHEME_SM2* source, BYTE** buffer, INT32* size)
4443 {
4444     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4445 }
4446 TPM_RC
4447 TPMS_KEY_SCHEME_ECMQV_Unmarshal(
4448     TPMS_KEY_SCHEME_ECMQV* target, BYTE** buffer, INT32* size)
4449 {
4450     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4451 }
4452 UINT16
4453 TPMS_KEY_SCHEME_ECMQV_Marshal(
4454     TPMS_KEY_SCHEME_ECMQV* source, BYTE** buffer, INT32* size)
4455 {
4456     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4457 }
4458 # endif // !USE_MARSHALING_DEFINES
4459
4460 // Table "Definition of Types for KDF Schemes" (Part 2: Structures)
4461 # if !USE_MARSHALING_DEFINES
4462 TPM_RC
4463 TPMS_KDF_SCHEME_MGF1_Unmarshal(
4464     TPMS_KDF_SCHEME_MGF1* target, BYTE** buffer, INT32* size)
4465 {
4466     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);

```



```

4467 }
4468 UINT16
4469 TPMS_KDF_SCHEME_MGF1_Marshal(TPMS_KDF_SCHEME_MGF1* source, BYTE** buffer, INT32* size)
4470 {
4471     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4472 }
4473 TPM_RC
4474 TPMS_KDF_SCHEME_KDF1_SP800_56A_Unmarshal(
4475     TPMS_KDF_SCHEME_KDF1_SP800_56A* target, BYTE** buffer, INT32* size)
4476 {
4477     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4478 }
4479 UINT16
4480 TPMS_KDF_SCHEME_KDF1_SP800_56A_Marshal(
4481     TPMS_KDF_SCHEME_KDF1_SP800_56A* source, BYTE** buffer, INT32* size)
4482 {
4483     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4484 }
4485 TPM_RC
4486 TPMS_KDF_SCHEME_KDF2_Unmarshal(
4487     TPMS_KDF_SCHEME_KDF2* target, BYTE** buffer, INT32* size)
4488 {
4489     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4490 }
4491 UINT16
4492 TPMS_KDF_SCHEME_KDF2_Marshal(TPMS_KDF_SCHEME_KDF2* source, BYTE** buffer, INT32* size)
4493 {
4494     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4495 }
4496 TPM_RC
4497 TPMS_KDF_SCHEME_KDF1_SP800_108_Unmarshal(
4498     TPMS_KDF_SCHEME_KDF1_SP800_108* target, BYTE** buffer, INT32* size)
4499 {
4500     return TPMS_SCHEME_HASH_Unmarshal((TPMS_SCHEME_HASH*)target, buffer, size);
4501 }
4502 UINT16
4503 TPMS_KDF_SCHEME_KDF1_SP800_108_Marshal(
4504     TPMS_KDF_SCHEME_KDF1_SP800_108* source, BYTE** buffer, INT32* size)
4505 {
4506     return TPMS_SCHEME_HASH_Marshal((TPMS_SCHEME_HASH*)source, buffer, size);
4507 }
4508 # endif // !USE_MARSHALING_DEFINES
4509
4510 // Table "Definition of TPMU_KDF_SCHEME Union" (Part 2: Structures)
4511 TPM_RC
4512 TPMU_KDF_SCHEME_Unmarshal(
4513     TPMU_KDF_SCHEME* target, BYTE** buffer, INT32* size, UINT32 selector)
4514 {
4515     switch(selector)
4516     {
4517 # if ALG_MGF1
4518         case TPM_ALG_MGF1:
4519             return TPMS_KDF_SCHEME_MGF1_Unmarshal(
4520                 (TPMS_KDF_SCHEME_MGF1*)&(target->mgf1), buffer, size);
4521 # endif // ALG_MGF1
4522 # if ALG_KDF1_SP800_56A
4523         case TPM_ALG_KDF1_SP800_56A:
4524             return TPMS_KDF_SCHEME_KDF1_SP800_56A_Unmarshal(
4525                 (TPMS_KDF_SCHEME_KDF1_SP800_56A*)&(target->kdf1_sp800_56a),
4526                 buffer,
4527                 size);
4528 # endif // ALG_KDF1_SP800_56A
4529 # if ALG_KDF2
4530         case TPM_ALG_KDF2:
4531             return TPMS_KDF_SCHEME_KDF2_Unmarshal(
4532                 (TPMS_KDF_SCHEME_KDF2*)&(target->kdf2), buffer, size);

```

```

4533 # endif // ALG_KDF2
4534 # if ALG_KDF1_SP800_108
4535     case TPM_ALG_KDF1_SP800_108:
4536         return TPMS_KDF_SCHEME_KDF1_SP800_108_Unmarshal(
4537             (TPMS_KDF_SCHEME_KDF1_SP800_108*)&(target->kdf1_sp800_108),
4538             buffer,
4539             size);
4540 # endif // ALG_KDF1_SP800_108
4541     case TPM_ALG_NULL:
4542         return TPM_RC_SUCCESS;
4543 }
4544 return TPM_RC_SELECTOR;
4545 }
4546 UINT16
4547 TPMU_KDF_SCHEME_Marshal(
4548     TPMU_KDF_SCHEME* source, BYTE** buffer, INT32* size, UINT32 selector)
4549 {
4550     switch(selector)
4551     {
4552 # if ALG_MGF1
4553         case TPM_ALG_MGF1:
4554             return TPMS_KDF_SCHEME_MGF1_Marshal(
4555                 (TPMS_KDF_SCHEME_MGF1*)&(source->mgf1), buffer, size);
4556 # endif // ALG_MGF1
4557 # if ALG_KDF1_SP800_56A
4558         case TPM_ALG_KDF1_SP800_56A:
4559             return TPMS_KDF_SCHEME_KDF1_SP800_56A_Marshal(
4560                 (TPMS_KDF_SCHEME_KDF1_SP800_56A*)&(source->kdf1_sp800_56a),
4561                 buffer,
4562                 size);
4563 # endif // ALG_KDF1_SP800_56A
4564 # if ALG_KDF2
4565         case TPM_ALG_KDF2:
4566             return TPMS_KDF_SCHEME_KDF2_Marshal(
4567                 (TPMS_KDF_SCHEME_KDF2*)&(source->kdf2), buffer, size);
4568 # endif // ALG_KDF2
4569 # if ALG_KDF1_SP800_108
4570         case TPM_ALG_KDF1_SP800_108:
4571             return TPMS_KDF_SCHEME_KDF1_SP800_108_Marshal(
4572                 (TPMS_KDF_SCHEME_KDF1_SP800_108*)&(source->kdf1_sp800_108),
4573                 buffer,
4574                 size);
4575 # endif // ALG_KDF1_SP800_108
4576     }
4577     return 0;
4578 }
4579
4580 // Table "Definition of TPMT_KDF_SCHEME Structure" (Part 2: Structures)
4581 TPM_RC
4582 TPMT_KDF_SCHEME_Unmarshal(
4583     TPMT_KDF_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
4584 {
4585     TPM_RC result;
4586     result =
4587         TPMI_ALG_KDF_Unmarshal((TPMI_ALG_KDF*)&(target->scheme), buffer, size, flag);
4588     if(result == TPM_RC_SUCCESS)
4589         result = TPMU_KDF_SCHEME_Unmarshal((TPMU_KDF_SCHEME*)&(target->details),
4590             buffer,
4591             size,
4592             (UINT32)target->scheme);
4593     return result;
4594 }
4595 UINT16
4596 TPMT_KDF_SCHEME_Marshal(TPMT_KDF_SCHEME* source, BYTE** buffer, INT32* size)
4597 {
4598     UINT16 result = 0;

```

```

4599     result      = (UINT16)(result
4600                   + TPMI_ALG_KDF_Marshal(
4601                       (TPMI_ALG_KDF*)&(source->scheme), buffer, size));
4602     result      = (UINT16)(result
4603                   + TPMU_KDF_SCHEME_Marshal((TPMU_KDF_SCHEME*)&(source->details),
4604                                               buffer,
4605                                               size,
4606                                               (UINT32)source->scheme));
4607     return result;
4608 }
4609
4610 // Table "Definition of TPMI_ALG_ASYM_SCHEME Type" (Part 2: Structures)
4611 TPM_RC
4612 TPMI_ALG_ASYM_SCHEME_Unmarshal(
4613     TPMI_ALG_ASYM_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
4614 {
4615     TPM_RC result;
4616     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
4617     if(result == TPM_RC_SUCCESS)
4618     {
4619         switch(*target)
4620         {
4621             # if ALG_RSASSA
4622                 case TPM_ALG_RSASSA:
4623             # endif // ALG_RSASSA
4624             # if ALG_RSAES
4625                 case TPM_ALG_RSAES:
4626             # endif // ALG_RSAES
4627             # if ALG_RSAPSS
4628                 case TPM_ALG_RSAPSS:
4629             # endif // ALG_RSAPSS
4630             # if ALG_OAEP
4631                 case TPM_ALG_OAEP:
4632             # endif // ALG_OAEP
4633             # if ALG_ECDSA
4634                 case TPM_ALG_ECDSA:
4635             # endif // ALG_ECDSA
4636             # if ALG_ECDH
4637                 case TPM_ALG_ECDH:
4638             # endif // ALG_ECDH
4639             # if ALG_ECDAA
4640                 case TPM_ALG_ECDAA:
4641             # endif // ALG_ECDAA
4642             # if ALG_SM2
4643                 case TPM_ALG_SM2:
4644             # endif // ALG_SM2
4645             # if ALG_ECSCHNORR
4646                 case TPM_ALG_ECSCHNORR:
4647             # endif // ALG_ECSCHNORR
4648             # if ALG_ECMQV
4649                 case TPM_ALG_ECMQV:
4650             # endif // ALG_ECMQV
4651             # if ALG_EDDSA
4652                 case TPM_ALG_EDDSA:
4653             # endif // ALG_EDDSA
4654             # if ALG_EDDSA_PH
4655                 case TPM_ALG_EDDSA_PH:
4656             # endif // ALG_EDDSA_PH
4657             # if ALG_LMS
4658                 case TPM_ALG_LMS:
4659             # endif // ALG_LMS
4660             # if ALG_XMSS
4661                 case TPM_ALG_XMSS:
4662             # endif // ALG_XMSS
4663                 break;
4664             default:

```

```

4665         if ((*target != TPM_ALG_NULL) || !flag)
4666             result = TPM_RC_VALUE;
4667         break;
4668     }
4669 }
4670 return result;
4671 }
4672 # if !USE_MARSHALING_DEFINES
4673 UINT16
4674 TPMI_ALG_ASYM_SCHEME_Marshal(TPMI_ALG_ASYM_SCHEME* source, BYTE** buffer, INT32* size)
4675 {
4676     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
4677 }
4678 # endif // !USE_MARSHALING_DEFINES
4679
4680 // Table "Definition of TPMU_ASYM_SCHEME Union" (Part 2: Structures)
4681 TPM_RC
4682 TPMU_ASYM_SCHEME_Unmarshal(
4683     TPMU_ASYM_SCHEME* target, BYTE** buffer, INT32* size, UINT32 selector)
4684 {
4685     switch(selector)
4686     {
4687     # if ALG_RSASSA
4688         case TPM_ALG_RSASSA:
4689             return TPMS_SIG_SCHEME_RSASSA_Unmarshal(
4690                 (TPMS_SIG_SCHEME_RSASSA*)&(target->rsassa), buffer, size);
4691     # endif // ALG_RSASSA
4692     # if ALG_RSAES
4693         case TPM_ALG_RSAES:
4694             return TPMS_ENC_SCHEME_RSAES_Unmarshal(
4695                 (TPMS_ENC_SCHEME_RSAES*)&(target->rsaes), buffer, size);
4696     # endif // ALG_RSAES
4697     # if ALG_RSAPSS
4698         case TPM_ALG_RSAPSS:
4699             return TPMS_SIG_SCHEME_RSAPSS_Unmarshal(
4700                 (TPMS_SIG_SCHEME_RSAPSS*)&(target->rsapss), buffer, size);
4701     # endif // ALG_RSAPSS
4702     # if ALG_OAEP
4703         case TPM_ALG_OAEP:
4704             return TPMS_ENC_SCHEME_OAEP_Unmarshal(
4705                 (TPMS_ENC_SCHEME_OAEP*)&(target->oaep), buffer, size);
4706     # endif // ALG_OAEP
4707     # if ALG_ECDSA
4708         case TPM_ALG_ECDSA:
4709             return TPMS_SIG_SCHEME_ECDSA_Unmarshal(
4710                 (TPMS_SIG_SCHEME_ECDSA*)&(target->ecdsa), buffer, size);
4711     # endif // ALG_ECDSA
4712     # if ALG_ECDH
4713         case TPM_ALG_ECDH:
4714             return TPMS_KEY_SCHEME_ECDH_Unmarshal(
4715                 (TPMS_KEY_SCHEME_ECDH*)&(target->ecdh), buffer, size);
4716     # endif // ALG_ECDH
4717     # if ALG_ECDAA
4718         case TPM_ALG_ECDAA:
4719             return TPMS_SIG_SCHEME_ECDAA_Unmarshal(
4720                 (TPMS_SIG_SCHEME_ECDAA*)&(target->ecdac), buffer, size);
4721     # endif // ALG_ECDAA
4722     # if ALG_SM2
4723         case TPM_ALG_SM2:
4724             return TPMS_KEY_SCHEME_SM2_Unmarshal(
4725                 (TPMS_KEY_SCHEME_SM2*)&(target->sm2), buffer, size);
4726     # endif // ALG_SM2
4727     # if ALG_ECSCHNORR
4728         case TPM_ALG_ECSCHNORR:
4729             return TPMS_SIG_SCHEME_ECSCHNORR_Unmarshal(
4730                 (TPMS_SIG_SCHEME_ECSCHNORR*)&(target->ecschnorr), buffer, size);
4731     # endif // ALG_ECSCHNORR
4732     }
4733     return TPM_RC_SELECTOR;
4734 }

```

```

4731 # endif // ALG_ECSCNORR
4732 # if ALG_ECMQV
4733     case TPM_ALG_ECMQV:
4734         return TPMS_KEY_SCHEME_ECMQV_Unmarshal(
4735             (TPMS_KEY_SCHEME_ECMQV*)&(target->ecmqv), buffer, size);
4736 # endif // ALG_ECMQV
4737 # if ALG_EDDSA
4738     case TPM_ALG_EDDSA:
4739         return TPMS_SIG_SCHEME_EDDSA_Unmarshal(
4740             (TPMS_SIG_SCHEME_EDDSA*)&(target->eddsa), buffer, size);
4741 # endif // ALG_EDDSA
4742 # if ALG_EDDSA_PH
4743     case TPM_ALG_EDDSA_PH:
4744         return TPMS_SIG_SCHEME_EDDSA_PH_Unmarshal(
4745             (TPMS_SIG_SCHEME_EDDSA_PH*)&(target->eddsa_ph), buffer, size);
4746 # endif // ALG_EDDSA_PH
4747 # if ALG_LMS
4748     case TPM_ALG_LMS:
4749         return TPMS_SIG_SCHEME_LMS_Unmarshal(
4750             (TPMS_SIG_SCHEME_LMS*)&(target->lms), buffer, size);
4751 # endif // ALG_LMS
4752 # if ALG_XMSS
4753     case TPM_ALG_XMSS:
4754         return TPMS_SIG_SCHEME_XMSS_Unmarshal(
4755             (TPMS_SIG_SCHEME_XMSS*)&(target->xmss), buffer, size);
4756 # endif // ALG_XMSS
4757     case TPM_ALG_NULL:
4758         return TPM_RC_SUCCESS;
4759 }
4760 return TPM_RC_SELECTOR;
4761 }
4762 UINT16
4763 TPMU_ASYM_SCHEME_Marshal(
4764     TPMU_ASYM_SCHEME* source, BYTE** buffer, INT32* size, UINT32 selector)
4765 {
4766     switch(selector)
4767     {
4768 # if ALG_RSASSA
4769         case TPM_ALG_RSASSA:
4770             return TPMS_SIG_SCHEME_RSASSA_Marshal(
4771                 (TPMS_SIG_SCHEME_RSASSA*)&(source->rsassa), buffer, size);
4772 # endif // ALG_RSASSA
4773 # if ALG_RSAES
4774         case TPM_ALG_RSAES:
4775             return TPMS_ENC_SCHEME_RSAES_Marshal(
4776                 (TPMS_ENC_SCHEME_RSAES*)&(source->rsaes), buffer, size);
4777 # endif // ALG_RSAES
4778 # if ALG_RSAPSS
4779         case TPM_ALG_RSAPSS:
4780             return TPMS_SIG_SCHEME_RSAPSS_Marshal(
4781                 (TPMS_SIG_SCHEME_RSAPSS*)&(source->rsapss), buffer, size);
4782 # endif // ALG_RSAPSS
4783 # if ALG_OAEP
4784         case TPM_ALG_OAEP:
4785             return TPMS_ENC_SCHEME_OAEP_Marshal(
4786                 (TPMS_ENC_SCHEME_OAEP*)&(source->oaep), buffer, size);
4787 # endif // ALG_OAEP
4788 # if ALG_ECDSA
4789         case TPM_ALG_ECDSA:
4790             return TPMS_SIG_SCHEME_ECDSA_Marshal(
4791                 (TPMS_SIG_SCHEME_ECDSA*)&(source->ecdsa), buffer, size);
4792 # endif // ALG_ECDSA
4793 # if ALG_ECDH
4794         case TPM_ALG_ECDH:
4795             return TPMS_KEY_SCHEME_ECDH_Marshal(
4796                 (TPMS_KEY_SCHEME_ECDH*)&(source->ecdh), buffer, size);

```

```

4797 # endif // ALG_ECDH
4798 # if ALG_ECDSA
4799     case TPM_ALG_ECDSA:
4800         return TPMS_SIG_SCHEME_ECDSA_Marshal(
4801             (TPMS_SIG_SCHEME_ECDSA*)&(source->ecdsa), buffer, size);
4802 # endif // ALG_ECDSA
4803 # if ALG_SM2
4804     case TPM_ALG_SM2:
4805         return TPMS_KEY_SCHEME_SM2_Marshal(
4806             (TPMS_KEY_SCHEME_SM2*)&(source->sm2), buffer, size);
4807 # endif // ALG_SM2
4808 # if ALG_ECSCHNORR
4809     case TPM_ALG_ECSCHNORR:
4810         return TPMS_SIG_SCHEME_ECSCHNORR_Marshal(
4811             (TPMS_SIG_SCHEME_ECSCHNORR*)&(source->ecschnor), buffer, size);
4812 # endif // ALG_ECSCHNORR
4813 # if ALG_ECMQV
4814     case TPM_ALG_ECMQV:
4815         return TPMS_KEY_SCHEME_ECMQV_Marshal(
4816             (TPMS_KEY_SCHEME_ECMQV*)&(source->ecmqv), buffer, size);
4817 # endif // ALG_ECMQV
4818 # if ALG_EDDSA
4819     case TPM_ALG_EDDSA:
4820         return TPMS_SIG_SCHEME_EDDSA_Marshal(
4821             (TPMS_SIG_SCHEME_EDDSA*)&(source->eddsa), buffer, size);
4822 # endif // ALG_EDDSA
4823 # if ALG_EDDSA_PH
4824     case TPM_ALG_EDDSA_PH:
4825         return TPMS_SIG_SCHEME_EDDSA_PH_Marshal(
4826             (TPMS_SIG_SCHEME_EDDSA_PH*)&(source->eddsa_ph), buffer, size);
4827 # endif // ALG_EDDSA_PH
4828 # if ALG_LMS
4829     case TPM_ALG_LMS:
4830         return TPMS_SIG_SCHEME_LMS_Marshal(
4831             (TPMS_SIG_SCHEME_LMS*)&(source->lms), buffer, size);
4832 # endif // ALG_LMS
4833 # if ALG_XMSS
4834     case TPM_ALG_XMSS:
4835         return TPMS_SIG_SCHEME_XMSS_Marshal(
4836             (TPMS_SIG_SCHEME_XMSS*)&(source->xmss), buffer, size);
4837 # endif // ALG_XMSS
4838     }
4839     return 0;
4840 }
4841
4842 // Table "Definition of TPMI_ALG_RSA_SCHEME Type" (Part 2: Structures)
4843 TPM_RC
4844 TPMI_ALG_RSA_SCHEME_Unmarshal(
4845     TPMI_ALG_RSA_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
4846 {
4847     TPM_RC result;
4848     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
4849     if(result == TPM_RC_SUCCESS)
4850     {
4851         switch(*target)
4852         {
4853 # if ALG_RSASSA
4854             case TPM_ALG_RSASSA:
4855 # endif // ALG_RSASSA
4856 # if ALG_RSAES
4857             case TPM_ALG_RSAES:
4858 # endif // ALG_RSAES
4859 # if ALG_RSAPSS
4860             case TPM_ALG_RSAPSS:
4861 # endif // ALG_RSAPSS
4862 # if ALG_OAEP

```



```

4863         case TPM_ALG_OAEP:
4864 # endif // ALG_OAEP
4865         break;
4866         default:
4867             if ((*target != TPM_ALG_NULL) || !flag)
4868                 result = TPM_RC_VALUE;
4869             break;
4870     }
4871 }
4872 return result;
4873 }
4874 # if !USE_MARSHALING_DEFINES
4875 UINT16
4876 TPMI_ALG_RSA_SCHEME_Marshal(TPMI_ALG_RSA_SCHEME* source, BYTE** buffer, INT32* size)
4877 {
4878     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
4879 }
4880 # endif // !USE_MARSHALING_DEFINES
4881
4882 // Table "Definition of TPMT_RSA_SCHEME Structure" (Part 2: Structures)
4883 TPM_RC
4884 TPMT_RSA_SCHEME_Unmarshal(
4885     TPMT_RSA_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
4886 {
4887     TPM_RC result;
4888     result = TPMI_ALG_RSA_SCHEME_Unmarshal(
4889         (TPMI_ALG_RSA_SCHEME*)&(target->scheme), buffer, size, flag);
4890     if (result == TPM_RC_SUCCESS)
4891         result = TPMU_ASYM_SCHEME_Unmarshal((TPMU_ASYM_SCHEME*)&(target->details),
4892             buffer,
4893             size,
4894             (UINT32)target->scheme);
4895     return result;
4896 }
4897
4898 UINT16
4899 TPMT_RSA_SCHEME_Marshal(TPMT_RSA_SCHEME* source, BYTE** buffer, INT32* size)
4900 {
4901     UINT16 result = 0;
4902     result = (UINT16)(result
4903         + TPMI_ALG_RSA_SCHEME_Marshal(
4904             (TPMI_ALG_RSA_SCHEME*)&(source->scheme), buffer, size));
4905     result =
4906         (UINT16)(result
4907             + TPMU_ASYM_SCHEME_Marshal((TPMU_ASYM_SCHEME*)&(source->details),
4908                 buffer,
4909                 size,
4910                 (UINT32)source->scheme));
4911     return result;
4912 }
4913
4914 // Table "Definition of TPMI_ALG_RSA_DECRYPT Type" (Part 2: Structures)
4915 TPM_RC
4916 TPMI_ALG_RSA_DECRYPT_Unmarshal(
4917     TPMI_ALG_RSA_DECRYPT* target, BYTE** buffer, INT32* size, BOOL flag)
4918 {
4919     TPM_RC result;
4920     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
4921     if (result == TPM_RC_SUCCESS)
4922     {
4923         switch(*target)
4924         {
4925 # if ALG_RSAES
4926             case TPM_ALG_RSAES:
4927 # endif // ALG_RSAES
4928             case TPM_ALG_OAEP:

```

```

4929 # endif // ALG_OAEP
4930     break;
4931     default:
4932         if ((*target != TPM_ALG_NULL) || !flag)
4933             result = TPM_RC_VALUE;
4934         break;
4935     }
4936 }
4937 return result;
4938 }
4939 # if !USE_MARSHALING_DEFINES
4940 UINT16
4941 TPMI_ALG_RSA_DECRYPT_Marshal(TPMI_ALG_RSA_DECRYPT* source, BYTE** buffer, INT32* size)
4942 {
4943     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
4944 }
4945 # endif // !USE_MARSHALING_DEFINES
4946
4947 // Table "Definition of TPMT_RSA_DECRYPT Structure" (Part 2: Structures)
4948 TPM_RC
4949 TPMT_RSA_DECRYPT_Unmarshal(
4950     TPMT_RSA_DECRYPT* target, BYTE** buffer, INT32* size, BOOL flag)
4951 {
4952     TPM_RC result;
4953     result = TPMI_ALG_RSA_DECRYPT_Unmarshal(
4954         (TPMI_ALG_RSA_DECRYPT*)&(target->scheme), buffer, size, flag);
4955     if (result == TPM_RC_SUCCESS)
4956         result = TPMU_ASYM_SCHEME_Unmarshal((TPMU_ASYM_SCHEME*)&(target->details),
4957             buffer,
4958             size,
4959             (UINT32)target->scheme);
4960     return result;
4961 }
4962 UINT16
4963 TPMT_RSA_DECRYPT_Marshal(TPMT_RSA_DECRYPT* source, BYTE** buffer, INT32* size)
4964 {
4965     UINT16 result = 0;
4966     result = (UINT16)(result
4967         + TPMI_ALG_RSA_DECRYPT_Marshal(
4968             (TPMI_ALG_RSA_DECRYPT*)&(source->scheme), buffer, size));
4969     result =
4970         (UINT16)(result
4971             + TPMU_ASYM_SCHEME_Marshal((TPMU_ASYM_SCHEME*)&(source->details),
4972                 buffer,
4973                 size,
4974                 (UINT32)source->scheme));
4975     return result;
4976 }
4977
4978 // Table "Definition of TPM2B_PUBLIC_KEY_RSA Structure" (Part 2: Structures)
4979 TPM_RC
4980 TPM2B_PUBLIC_KEY_RSA_Unmarshal(
4981     TPM2B_PUBLIC_KEY_RSA* target, BYTE** buffer, INT32* size)
4982 {
4983     TPM_RC result;
4984     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
4985     if ((result == TPM_RC_SUCCESS) && (target->t.size > MAX_RSA_KEY_BYTES))
4986         result = TPM_RC_SIZE;
4987     if (result == TPM_RC_SUCCESS)
4988         result = BYTE_Array_Unmarshal(
4989             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
4990     return result;
4991 }
4992 UINT16
4993 TPM2B_PUBLIC_KEY_RSA_Marshal(TPM2B_PUBLIC_KEY_RSA* source, BYTE** buffer, INT32* size)
4994 {

```

```

4995     UINT16 result = 0;
4996     result =
4997         (UINT16) (result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
4998     // if size equal to 0, the rest of the structure is a zero buffer
4999     if(source->t.size == 0)
5000         return result;
5001     result = (UINT16) (result
5002         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
5003             buffer,
5004             size,
5005             (INT32) source->t.size));
5006     return result;
5007 }
5008
5009 // Table "Definition of TPMI_RSA_KEY_BITS Type" (Part 2: Structures)
5010 TPM_RC
5011 TPMI_RSA_KEY_BITS_Unmarshal(TPMI_RSA_KEY_BITS* target, BYTE** buffer, INT32* size)
5012 {
5013     TPM_RC result;
5014     result = TPM_KEY_BITS_Unmarshal((TPM_KEY_BITS*)target, buffer, size);
5015     if(result == TPM_RC_SUCCESS)
5016     {
5017         switch(*target)
5018         {
5019 # if RSA_1024
5020             case 1024:
5021 # endif // RSA_1024
5022 # if RSA_16384
5023             case 16384:
5024 # endif // RSA_16384
5025 # if RSA_2048
5026             case 2048:
5027 # endif // RSA_2048
5028 # if RSA_3072
5029             case 3072:
5030 # endif // RSA_3072
5031 # if RSA_4096
5032             case 4096:
5033 # endif // RSA_4096
5034             break;
5035             default:
5036                 result = TPM_RC_VALUE;
5037                 break;
5038         }
5039     }
5040     return result;
5041 }
5042 # if !USE_MARSHALING_DEFINES
5043 UINT16
5044 TPMI_RSA_KEY_BITS_Marshal(TPMI_RSA_KEY_BITS* source, BYTE** buffer, INT32* size)
5045 {
5046     return TPM_KEY_BITS_Marshal((TPM_KEY_BITS*)source, buffer, size);
5047 }
5048 # endif // !USE_MARSHALING_DEFINES
5049
5050 // Table "Definition of TPM2B_PRIVATE_KEY_RSA Structure" (Part 2: Structures)
5051 TPM_RC
5052 TPM2B_PRIVATE_KEY_RSA_Unmarshal(
5053     TPM2B_PRIVATE_KEY_RSA* target, BYTE** buffer, INT32* size)
5054 {
5055     TPM_RC result;
5056     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
5057     if((result == TPM_RC_SUCCESS) && (target->t.size > RSA_PRIVATE_SIZE))
5058         result = TPM_RC_SIZE;
5059     if(result == TPM_RC_SUCCESS)
5060         result = BYTE_Array_Unmarshal(

```

```

5061         (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
5062     return result;
5063 }
5064 UINT16
5065 TPM2B_PRIVATE_KEY_RSA_Marshal(
5066     TPM2B_PRIVATE_KEY_RSA* source, BYTE** buffer, INT32* size)
5067 {
5068     UINT16 result = 0;
5069     result =
5070         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
5071     // if size equal to 0, the rest of the structure is a zero buffer
5072     if(source->t.size == 0)
5073         return result;
5074     result = (UINT16)(result
5075         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
5076             buffer,
5077             size,
5078             (INT32)source->t.size));
5079     return result;
5080 }
5081
5082 // Table "Definition of TPM2B_ECC_PARAMETER Structure" (Part 2: Structures)
5083 # if ALG_ECC
5084 TPM_RC
5085 TPM2B_ECC_PARAMETER_Unmarshal(TPM2B_ECC_PARAMETER* target, BYTE** buffer, INT32* size)
5086 {
5087     TPM_RC result;
5088     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
5089     if((result == TPM_RC_SUCCESS) && (target->t.size > MAX_ECC_KEY_BYTES))
5090         result = TPM_RC_SIZE;
5091     if(result == TPM_RC_SUCCESS)
5092         result = BYTE_Array_Unmarshal(
5093             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
5094     return result;
5095 }
5096 UINT16
5097 TPM2B_ECC_PARAMETER_Marshal(TPM2B_ECC_PARAMETER* source, BYTE** buffer, INT32* size)
5098 {
5099     UINT16 result = 0;
5100     result =
5101         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
5102     // if size equal to 0, the rest of the structure is a zero buffer
5103     if(source->t.size == 0)
5104         return result;
5105     result = (UINT16)(result
5106         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
5107             buffer,
5108             size,
5109             (INT32)source->t.size));
5110     return result;
5111 }
5112 # endif // ALG_ECC
5113
5114 // Table "Definition of TPMS_ECC_POINT Structure" (Part 2: Structures)
5115 # if ALG_ECC
5116 TPM_RC
5117 TPMS_ECC_POINT_Unmarshal(TPMS_ECC_POINT* target, BYTE** buffer, INT32* size)
5118 {
5119     TPM_RC result;
5120     result = TPM2B_ECC_PARAMETER_Unmarshal(
5121         (TPM2B_ECC_PARAMETER*)&(target->x), buffer, size);
5122     if(result == TPM_RC_SUCCESS)
5123         result = TPM2B_ECC_PARAMETER_Unmarshal(
5124             (TPM2B_ECC_PARAMETER*)&(target->y), buffer, size);
5125     return result;
5126 }

```

```

5127 UINT16
5128 TPMS_ECC_POINT_Marshal(TPMS_ECC_POINT* source, BYTE** buffer, INT32* size)
5129 {
5130     UINT16 result = 0;
5131     result = (UINT16)(result
5132         + TPM2B_ECC_PARAMETER_Marshal(
5133             (TPM2B_ECC_PARAMETER*)&(source->x), buffer, size));
5134     result = (UINT16)(result
5135         + TPM2B_ECC_PARAMETER_Marshal(
5136             (TPM2B_ECC_PARAMETER*)&(source->y), buffer, size));
5137     return result;
5138 }
5139 # endif // ALG_ECC
5140
5141 // Table "Definition of TPM2B_ECC_POINT Structure" (Part 2: Structures)
5142 # if ALG_ECC
5143 TPM_RC
5144 TPM2B_ECC_POINT_Unmarshal(TPM2B_ECC_POINT* target, BYTE** buffer, INT32* size)
5145 {
5146     TPM_RC result;
5147     result = UINT16_Unmarshal((UINT16*)&(target->size), buffer, size);
5148     if(result == TPM_RC_SUCCESS)
5149     {
5150         // if size is zero, then the required structure is missing
5151         if(target->size == 0)
5152             result = TPM_RC_SIZE;
5153         else
5154         {
5155             INT32 startSize = *size;
5156             result = TPMS_ECC_POINT_Unmarshal(
5157                 (TPMS_ECC_POINT*)&(target->point), buffer, size);
5158             if((result == TPM_RC_SUCCESS) && (target->size != (startSize - *size)))
5159                 result = TPM_RC_SIZE;
5160         }
5161     }
5162     return result;
5163 }
5164
5165 UINT16
5166 TPM2B_ECC_POINT_Marshal(TPM2B_ECC_POINT* source, BYTE** buffer, INT32* size)
5167 {
5168     UINT16 result = 0;
5169     // Marshal a dummy value of the 2B size. This makes sure that 'buffer'
5170     // and 'size' are advanced as necessary (i.e., if they are present)
5171     result = UINT16_Marshal(&result, buffer, size);
5172     // Marshal the structure
5173     result = (UINT16)(result
5174         + TPMS_ECC_POINT_Marshal(
5175             (TPMS_ECC_POINT*)&(source->point), buffer, size));
5176     // if a buffer was provided, go back and fill in the actual size
5177     if(buffer != NULL)
5178         UINT16_TO_BYTE_ARRAY((result - 2), (*buffer - result));
5179     return result;
5180 }
5181 # endif // ALG_ECC
5182
5183 // Table "Definition of TPMI_ALG_ECC_SCHEME Type" (Part 2: Structures)
5184 TPM_RC
5185 TPMI_ALG_ECC_SCHEME_Unmarshal(
5186     TPMI_ALG_ECC_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
5187 {
5188     TPM_RC result;
5189     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
5190     if(result == TPM_RC_SUCCESS)
5191     {
5192         switch(*target)
5193         {

```

```

5193 # if ALG_ECDSA
5194     case TPM_ALG_ECDSA:
5195 # endif // ALG_ECDSA
5196 # if ALG_ECDH
5197     case TPM_ALG_ECDH:
5198 # endif // ALG_ECDH
5199 # if ALG_ECDAA
5200     case TPM_ALG_ECDAA:
5201 # endif // ALG_ECDAA
5202 # if ALG_SM2
5203     case TPM_ALG_SM2:
5204 # endif // ALG_SM2
5205 # if ALG_ECSCHNORR
5206     case TPM_ALG_ECSCHNORR:
5207 # endif // ALG_ECSCHNORR
5208 # if ALG_ECMQV
5209     case TPM_ALG_ECMQV:
5210 # endif // ALG_ECMQV
5211 # if ALG_EDDSA
5212     case TPM_ALG_EDDSA:
5213 # endif // ALG_EDDSA
5214 # if ALG_EDDSA_PH
5215     case TPM_ALG_EDDSA_PH:
5216 # endif // ALG_EDDSA_PH
5217     break;
5218     default:
5219         if ((*target != TPM_ALG_NULL) || !flag)
5220             result = TPM_RC_SCHEME;
5221         break;
5222     }
5223 }
5224 return result;
5225 }
5226 # if !USE_MARSHALING_DEFINES
5227 UINT16
5228 TPMI_ALG_ECC_SCHEME_Marshal(TPMI_ALG_ECC_SCHEME* source, BYTE** buffer, INT32* size)
5229 {
5230     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
5231 }
5232 # endif // !USE_MARSHALING_DEFINES
5233
5234 // Table "Definition of TPMI_ECC_CURVE Type" (Part 2: Structures)
5235 TPM_RC
5236 TPMI_ECC_CURVE_Unmarshal(
5237     TPMI_ECC_CURVE* target, BYTE** buffer, INT32* size, BOOL flag)
5238 {
5239     TPM_RC result;
5240     result = TPM_ECC_CURVE_Unmarshal((TPM_ECC_CURVE*)target, buffer, size);
5241     if (result == TPM_RC_SUCCESS)
5242     {
5243         switch(*target)
5244         {
5245 # if ECC_NIST_P192
5246             case TPM_ECC_NIST_P192:
5247 # endif // ECC_NIST_P192
5248 # if ECC_NIST_P224
5249             case TPM_ECC_NIST_P224:
5250 # endif // ECC_NIST_P224
5251 # if ECC_NIST_P256
5252             case TPM_ECC_NIST_P256:
5253 # endif // ECC_NIST_P256
5254 # if ECC_NIST_P384
5255             case TPM_ECC_NIST_P384:
5256 # endif // ECC_NIST_P384
5257 # if ECC_NIST_P521
5258             case TPM_ECC_NIST_P521:

```



```

5259 # endif // ECC_NIST_P521
5260 # if ECC_BN_P256
5261     case TPM_ECC_BN_P256:
5262 # endif // ECC_BN_P256
5263 # if ECC_BN_P638
5264     case TPM_ECC_BN_P638:
5265 # endif // ECC_BN_P638
5266 # if ECC_SM2_P256
5267     case TPM_ECC_SM2_P256:
5268 # endif // ECC_SM2_P256
5269 # if ECC_BP_P256_R1
5270     case TPM_ECC_BP_P256_R1:
5271 # endif // ECC_BP_P256_R1
5272 # if ECC_BP_P384_R1
5273     case TPM_ECC_BP_P384_R1:
5274 # endif // ECC_BP_P384_R1
5275 # if ECC_BP_P512_R1
5276     case TPM_ECC_BP_P512_R1:
5277 # endif // ECC_BP_P512_R1
5278 # if ECC_CURVE_25519
5279     case TPM_ECC_CURVE_25519:
5280 # endif // ECC_CURVE_25519
5281 # if ECC_CURVE_448
5282     case TPM_ECC_CURVE_448:
5283 # endif // ECC_CURVE_448
5284     break;
5285     default:
5286         if ((*target != TPM_ECC_NONE) || !flag)
5287             result = TPM_RC_CURVE;
5288         break;
5289     }
5290 }
5291 return result;
5292 }
5293 # if !USE_MARSHALING_DEFINES
5294 UINT16
5295 TPMI_ECC_CURVE_Marshal(TPMI_ECC_CURVE* source, BYTE** buffer, INT32* size)
5296 {
5297     return TPM_ECC_CURVE_Marshal((TPM_ECC_CURVE*)source, buffer, size);
5298 }
5299 # endif // !USE_MARSHALING_DEFINES
5300
5301 // Table "Definition of TPMT_ECC_SCHEME Structure" (Part 2: Structures)
5302 # if ALG_ECC
5303 TPM_RC
5304 TPMT_ECC_SCHEME_Unmarshal(
5305     TPMT_ECC_SCHEME* target, BYTE** buffer, INT32* size, BOOL flag)
5306 {
5307     TPM_RC result;
5308     result = TPMI_ALG_ECC_SCHEME_Unmarshal(
5309         (TPMI_ALG_ECC_SCHEME*)&(target->scheme), buffer, size, flag);
5310     if (result == TPM_RC_SUCCESS)
5311         result = TPMU_ASYM_SCHEME_Unmarshal((TPMU_ASYM_SCHEME*)&(target->details),
5312             buffer,
5313             size,
5314             (UINT32)target->scheme);
5315     return result;
5316 }
5317 UINT16
5318 TPMT_ECC_SCHEME_Marshal(TPMT_ECC_SCHEME* source, BYTE** buffer, INT32* size)
5319 {
5320     UINT16 result = 0;
5321     result = (UINT16)(result
5322         + TPMI_ALG_ECC_SCHEME_Marshal(
5323             (TPMI_ALG_ECC_SCHEME*)&(source->scheme), buffer, size));
5324     result =

```

```

5325         (UINT16) (result
5326             + TPMU_ASYM_SCHEME_Marshal((TPMU_ASYM_SCHEME*)&(source->details),
5327                                         buffer,
5328                                         size,
5329                                         (UINT32)source->scheme));
5330     return result;
5331 }
5332 # endif // ALG_ECC
5333
5334 // Table "Definition of TPMS_ALGORITHM_DETAIL_ECC Structure" (Part 2: Structures)
5335 # if ALG_ECC
5336 UINT16
5337 TPMS_ALGORITHM_DETAIL_ECC_Marshal(
5338     TPMS_ALGORITHM_DETAIL_ECC* source, BYTE** buffer, INT32* size)
5339 {
5340     UINT16 result = 0;
5341     result = (UINT16) (result
5342         + TPM_ECC_CURVE_Marshal(
5343             (TPM_ECC_CURVE*)&(source->curveID), buffer, size));
5344     result =
5345         (UINT16) (result + UINT16_Marshal((UINT16*)&(source->keySize), buffer, size));
5346     result = (UINT16) (result
5347         + TPMT_KDF_SCHEME_Marshal(
5348             (TPMT_KDF_SCHEME*)&(source->kdf), buffer, size));
5349     result = (UINT16) (result
5350         + TPMT_ECC_SCHEME_Marshal(
5351             (TPMT_ECC_SCHEME*)&(source->sign), buffer, size));
5352     result = (UINT16) (result
5353         + TPM2B_ECC_PARAMETER_Marshal(
5354             (TPM2B_ECC_PARAMETER*)&(source->p), buffer, size));
5355     result = (UINT16) (result
5356         + TPM2B_ECC_PARAMETER_Marshal(
5357             (TPM2B_ECC_PARAMETER*)&(source->a), buffer, size));
5358     result = (UINT16) (result
5359         + TPM2B_ECC_PARAMETER_Marshal(
5360             (TPM2B_ECC_PARAMETER*)&(source->b), buffer, size));
5361     result = (UINT16) (result
5362         + TPM2B_ECC_PARAMETER_Marshal(
5363             (TPM2B_ECC_PARAMETER*)&(source->gX), buffer, size));
5364     result = (UINT16) (result
5365         + TPM2B_ECC_PARAMETER_Marshal(
5366             (TPM2B_ECC_PARAMETER*)&(source->gY), buffer, size));
5367     result = (UINT16) (result
5368         + TPM2B_ECC_PARAMETER_Marshal(
5369             (TPM2B_ECC_PARAMETER*)&(source->n), buffer, size));
5370     result = (UINT16) (result
5371         + TPM2B_ECC_PARAMETER_Marshal(
5372             (TPM2B_ECC_PARAMETER*)&(source->h), buffer, size));
5373     return result;
5374 }
5375 # endif // ALG_ECC
5376
5377 // Table "Definition of TPMS_SIGNATURE_RSA Structure" (Part 2: Structures)
5378 TPM_RC
5379 TPMS_SIGNATURE_RSA_Unmarshal(TPMS_SIGNATURE_RSA* target, BYTE** buffer, INT32* size)
5380 {
5381     TPM_RC result;
5382     result =
5383         TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH*)&(target->hash), buffer, size, 0);
5384     if(result == TPM_RC_SUCCESS)
5385         result = TPM2B_PUBLIC_KEY_RSA_Unmarshal(
5386             (TPM2B_PUBLIC_KEY_RSA*)&(target->sig), buffer, size);
5387     return result;
5388 }
5389
5390 UINT16
5391 TPMS_SIGNATURE_RSA_Marshal(TPMS_SIGNATURE_RSA* source, BYTE** buffer, INT32* size)

```

```

5391 {
5392     UINT16 result = 0;
5393     result      = (UINT16) (result
5394         + TPMI_ALG_HASH_Marshal(
5395             (TPMI_ALG_HASH*)&(source->hash), buffer, size));
5396     result      = (UINT16) (result
5397         + TPM2B_PUBLIC_KEY_RSA_Marshal(
5398             (TPM2B_PUBLIC_KEY_RSA*)&(source->sig), buffer, size));
5399     return result;
5400 }
5401
5402 // Table "Definition of Types for Signature" (Part 2: Structures)
5403 # if !USE_MARSHALING_DEFINES
5404 TPM_RC
5405 TPMS_SIGNATURE_RSASSA_Unmarshal(
5406     TPMS_SIGNATURE_RSASSA* target, BYTE** buffer, INT32* size)
5407 {
5408     return TPMS_SIGNATURE_RSA_Unmarshal((TPMS_SIGNATURE_RSA*)target, buffer, size);
5409 }
5410
5411 TPM_RC
5412 TPMS_SIGNATURE_RSASSA_Marshal(
5413     TPMS_SIGNATURE_RSASSA* source, BYTE** buffer, INT32* size)
5414 {
5415     return TPMS_SIGNATURE_RSA_Marshal((TPMS_SIGNATURE_RSA*)source, buffer, size);
5416 }
5417
5418 TPM_RC
5419 TPMS_SIGNATURE_RSAPSS_Unmarshal(
5420     TPMS_SIGNATURE_RSAPSS* target, BYTE** buffer, INT32* size)
5421 {
5422     return TPMS_SIGNATURE_RSA_Unmarshal((TPMS_SIGNATURE_RSA*)target, buffer, size);
5423 }
5424
5425 TPM_RC
5426 TPMS_SIGNATURE_RSAPSS_Marshal(
5427     TPMS_SIGNATURE_RSAPSS* source, BYTE** buffer, INT32* size)
5428 {
5429     return TPMS_SIGNATURE_RSA_Marshal((TPMS_SIGNATURE_RSA*)source, buffer, size);
5430 }
5431 # endif // !USE_MARSHALING_DEFINES
5432
5433 // Table "Definition of TPMS_SIGNATURE_ECC Structure" (Part 2: Structures)
5434 # if ALG_ECC
5435 TPM_RC
5436 TPMS_SIGNATURE_ECC_Unmarshal(TPMS_SIGNATURE_ECC* target, BYTE** buffer, INT32* size)
5437 {
5438     TPM_RC result;
5439     result =
5440         TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH*)&(target->hash), buffer, size, 0);
5441     if(result == TPM_RC_SUCCESS)
5442         result = TPM2B_ECC_PARAMETER_Unmarshal(
5443             (TPM2B_ECC_PARAMETER*)&(target->signatureR), buffer, size);
5444     if(result == TPM_RC_SUCCESS)
5445         result = TPM2B_ECC_PARAMETER_Unmarshal(
5446             (TPM2B_ECC_PARAMETER*)&(target->signatureS), buffer, size);
5447     return result;
5448 }
5449
5450 TPM_RC
5451 TPMS_SIGNATURE_ECC_Marshal(TPMS_SIGNATURE_ECC* source, BYTE** buffer, INT32* size)
5452 {
5453     TPMI_ALG_HASH result = 0;
5454     result      = (UINT16) (result
5455         + TPMI_ALG_HASH_Marshal(
5456             (TPMI_ALG_HASH*)&(source->hash), buffer, size));
5457     result      = (UINT16) (result
5458         + TPM2B_ECC_PARAMETER_Marshal(
5459             (TPM2B_ECC_PARAMETER*)&(source->signatureR), buffer, size));
5460     result      = (UINT16) (result

```

```

5457         + TPM2B_ECC_PARAMETER Marshal(
5458             (TPM2B_ECC_PARAMETER*)&(source->signatureS), buffer, size));
5459     return result;
5460 }
5461 # endif // ALG_ECC
5462
5463 // Table "Definition of Types for TPMS_SIGNATURE_ECC" (Part 2: Structures)
5464 # if !USE_MARSHALING_DEFINES
5465 TPM_RC
5466 TPMS_SIGNATURE_ECDSA_Unmarshal(
5467     TPMS_SIGNATURE_ECDSA* target, BYTE** buffer, INT32* size)
5468 {
5469     return TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)target, buffer, size);
5470 }
5471 UINT16
5472 TPMS_SIGNATURE_ECDSA_Marshal(TPMS_SIGNATURE_ECDSA* source, BYTE** buffer, INT32* size)
5473 {
5474     return TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)source, buffer, size);
5475 }
5476 TPM_RC
5477 TPMS_SIGNATURE_ECDSA_Unmarshal(
5478     TPMS_SIGNATURE_ECDSA* target, BYTE** buffer, INT32* size)
5479 {
5480     return TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)target, buffer, size);
5481 }
5482 UINT16
5483 TPMS_SIGNATURE_ECDSA_Marshal(TPMS_SIGNATURE_ECDSA* source, BYTE** buffer, INT32* size)
5484 {
5485     return TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)source, buffer, size);
5486 }
5487 TPM_RC
5488 TPMS_SIGNATURE_SM2_Unmarshal(TPMS_SIGNATURE_SM2* target, BYTE** buffer, INT32* size)
5489 {
5490     return TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)target, buffer, size);
5491 }
5492 UINT16
5493 TPMS_SIGNATURE_SM2_Marshal(TPMS_SIGNATURE_SM2* source, BYTE** buffer, INT32* size)
5494 {
5495     return TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)source, buffer, size);
5496 }
5497 TPM_RC
5498 TPMS_SIGNATURE_ECSCNORR_Unmarshal(
5499     TPMS_SIGNATURE_ECSCNORR* target, BYTE** buffer, INT32* size)
5500 {
5501     return TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)target, buffer, size);
5502 }
5503 UINT16
5504 TPMS_SIGNATURE_ECSCNORR_Marshal(
5505     TPMS_SIGNATURE_ECSCNORR* source, BYTE** buffer, INT32* size)
5506 {
5507     return TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)source, buffer, size);
5508 }
5509 TPM_RC
5510 TPMS_SIGNATURE_EDDSA_Unmarshal(
5511     TPMS_SIGNATURE_EDDSA* target, BYTE** buffer, INT32* size)
5512 {
5513     return TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)target, buffer, size);
5514 }
5515 UINT16
5516 TPMS_SIGNATURE_EDDSA_Marshal(TPMS_SIGNATURE_EDDSA* source, BYTE** buffer, INT32* size)
5517 {
5518     return TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)source, buffer, size);
5519 }
5520 TPM_RC
5521 TPMS_SIGNATURE_EDDSA_PH_Unmarshal(
5522     TPMS_SIGNATURE_EDDSA_PH* target, BYTE** buffer, INT32* size)

```

```

5523 {
5524     return TPMS_SIGNATURE_ECC_Unmarshal((TPMS_SIGNATURE_ECC*)target, buffer, size);
5525 }
5526 UINT16
5527 TPMS_SIGNATURE_EDDSA_PH_Marshal(
5528     TPMS_SIGNATURE_EDDSA_PH* source, BYTE** buffer, INT32* size)
5529 {
5530     return TPMS_SIGNATURE_ECC_Marshal((TPMS_SIGNATURE_ECC*)source, buffer, size);
5531 }
5532 # endif // !USE_MARSHALING_DEFINES
5533
5534 // Table "Definition of TPMU_SIGNATURE Union" (Part 2: Structures)
5535 TPM_RC
5536 TPMU_SIGNATURE_Unmarshal(
5537     TPMU_SIGNATURE* target, BYTE** buffer, INT32* size, UINT32 selector)
5538 {
5539     switch(selector)
5540     {
5541 # if ALG_HMAC
5542         case TPM_ALG_HMAC:
5543             return TPMT_HA_Unmarshal((TPMT_HA*)&(target->hmac), buffer, size, 0);
5544 # endif // ALG_HMAC
5545 # if ALG_RSASSA
5546         case TPM_ALG_RSASSA:
5547             return TPMS_SIGNATURE_RSASSA_Unmarshal(
5548                 (TPMS_SIGNATURE_RSASSA*)&(target->rsassa), buffer, size);
5549 # endif // ALG_RSASSA
5550 # if ALG_RSAPSS
5551         case TPM_ALG_RSAPSS:
5552             return TPMS_SIGNATURE_RSAPSS_Unmarshal(
5553                 (TPMS_SIGNATURE_RSAPSS*)&(target->rsapss), buffer, size);
5554 # endif // ALG_RSAPSS
5555 # if ALG_ECDSA
5556         case TPM_ALG_ECDSA:
5557             return TPMS_SIGNATURE_ECDSA_Unmarshal(
5558                 (TPMS_SIGNATURE_ECDSA*)&(target->ecdsa), buffer, size);
5559 # endif // ALG_ECDSA
5560 # if ALG_ECDAA
5561         case TPM_ALG_ECDAA:
5562             return TPMS_SIGNATURE_ECDAA_Unmarshal(
5563                 (TPMS_SIGNATURE_ECDAA*)&(target->ecdac), buffer, size);
5564 # endif // ALG_ECDAA
5565 # if ALG_SM2
5566         case TPM_ALG_SM2:
5567             return TPMS_SIGNATURE_SM2_Unmarshal(
5568                 (TPMS_SIGNATURE_SM2*)&(target->sm2), buffer, size);
5569 # endif // ALG_SM2
5570 # if ALG_ECSCHNORR
5571         case TPM_ALG_ECSCHNORR:
5572             return TPMS_SIGNATURE_ECSCHNORR_Unmarshal(
5573                 (TPMS_SIGNATURE_ECSCHNORR*)&(target->ecschnorr), buffer, size);
5574 # endif // ALG_ECSCHNORR
5575 # if ALG_EDDSA
5576         case TPM_ALG_EDDSA:
5577             return TPMS_SIGNATURE_EDDSA_Unmarshal(
5578                 (TPMS_SIGNATURE_EDDSA*)&(target->eddsa), buffer, size);
5579 # endif // ALG_EDDSA
5580 # if ALG_EDDSA_PH
5581         case TPM_ALG_EDDSA_PH:
5582             return TPMS_SIGNATURE_EDDSA_PH_Unmarshal(
5583                 (TPMS_SIGNATURE_EDDSA_PH*)&(target->eddsa_ph), buffer, size);
5584 # endif // ALG_EDDSA_PH
5585 # if ALG_LMS
5586         case TPM_ALG_LMS:
5587             return TPMS_SIGNATURE_LMS_Unmarshal(
5588                 (TPMS_SIGNATURE_LMS*)&(target->lms), buffer, size);

```



```

5589 # endif // ALG_LMS
5590 # if ALG_XMSS
5591     case TPM_ALG_XMSS:
5592         return TPMS_SIGNATURE_XMSS_Unmarshal(
5593             (TPMS_SIGNATURE_XMSS*)&(target->xmss), buffer, size);
5594 # endif // ALG_XMSS
5595     case TPM_ALG_NULL:
5596         return TPM_RC_SUCCESS;
5597     }
5598     return TPM_RC_SELECTOR;
5599 }
5600 UINT16
5601 TPMU_SIGNATURE_Marshal(
5602     TPMU_SIGNATURE* source, BYTE** buffer, INT32* size, UINT32 selector)
5603 {
5604     switch(selector)
5605     {
5606 # if ALG_HMAC
5607         case TPM_ALG_HMAC:
5608             return TPMT_HA_Marshal((TPMT_HA*)&(source->hmac), buffer, size);
5609 # endif // ALG_HMAC
5610 # if ALG_RSASSA
5611         case TPM_ALG_RSASSA:
5612             return TPMS_SIGNATURE_RSASSA_Marshal(
5613                 (TPMS_SIGNATURE_RSASSA*)&(source->rsassa), buffer, size);
5614 # endif // ALG_RSASSA
5615 # if ALG_RSAPSS
5616         case TPM_ALG_RSAPSS:
5617             return TPMS_SIGNATURE_RSAPSS_Marshal(
5618                 (TPMS_SIGNATURE_RSAPSS*)&(source->rsapss), buffer, size);
5619 # endif // ALG_RSAPSS
5620 # if ALG_ECDSA
5621         case TPM_ALG_ECDSA:
5622             return TPMS_SIGNATURE_ECDSA_Marshal(
5623                 (TPMS_SIGNATURE_ECDSA*)&(source->ecdsa), buffer, size);
5624 # endif // ALG_ECDSA
5625 # if ALG_ECDAA
5626         case TPM_ALG_ECDAA:
5627             return TPMS_SIGNATURE_ECDAA_Marshal(
5628                 (TPMS_SIGNATURE_ECDAA*)&(source->ecdac), buffer, size);
5629 # endif // ALG_ECDAA
5630 # if ALG_SM2
5631         case TPM_ALG_SM2:
5632             return TPMS_SIGNATURE_SM2_Marshal(
5633                 (TPMS_SIGNATURE_SM2*)&(source->sm2), buffer, size);
5634 # endif // ALG_SM2
5635 # if ALG_ECSCHNORR
5636         case TPM_ALG_ECSCHNORR:
5637             return TPMS_SIGNATURE_ECSCHNORR_Marshal(
5638                 (TPMS_SIGNATURE_ECSCHNORR*)&(source->ecschnorr), buffer, size);
5639 # endif // ALG_ECSCHNORR
5640 # if ALG_EDDSA
5641         case TPM_ALG_EDDSA:
5642             return TPMS_SIGNATURE_EDDSA_Marshal(
5643                 (TPMS_SIGNATURE_EDDSA*)&(source->eddsa), buffer, size);
5644 # endif // ALG_EDDSA
5645 # if ALG_EDDSA_PH
5646         case TPM_ALG_EDDSA_PH:
5647             return TPMS_SIGNATURE_EDDSA_PH_Marshal(
5648                 (TPMS_SIGNATURE_EDDSA_PH*)&(source->eddsa_ph), buffer, size);
5649 # endif // ALG_EDDSA_PH
5650 # if ALG_LMS
5651         case TPM_ALG_LMS:
5652             return TPMS_SIGNATURE_LMS_Marshal(
5653                 (TPMS_SIGNATURE_LMS*)&(source->lms), buffer, size);
5654 # endif // ALG_LMS

```



```

5655 # if ALG_XMSS
5656     case TPM_ALG_XMSS:
5657         return TPMS_SIGNATURE_XMSS_Marshal(
5658             (TPMS_SIGNATURE_XMSS*)&(source->xmss), buffer, size);
5659 # endif // ALG_XMSS
5660     }
5661     return 0;
5662 }
5663
5664 // Table "Definition of TPMT_SIGNATURE Structure" (Part 2: Structures)
5665 TPM_RC
5666 TPMT_SIGNATURE_Unmarshal(
5667     TPMT_SIGNATURE* target, BYTE** buffer, INT32* size, BOOL flag)
5668 {
5669     TPM_RC result;
5670     result = TPMI_ALG_SIG_SCHEME_Unmarshal(
5671         (TPMI_ALG_SIG_SCHEME*)&(target->sigAlg), buffer, size, flag);
5672     if(result == TPM_RC_SUCCESS)
5673         result = TPMU_SIGNATURE_Unmarshal((TPMU_SIGNATURE*)&(target->signature),
5674             buffer,
5675             size,
5676             (UINT32)target->sigAlg);
5677     return result;
5678 }
5679
5680 UINT16
5681 TPMT_SIGNATURE_Marshal(TPMT_SIGNATURE* source, BYTE** buffer, INT32* size)
5682 {
5683     UINT16 result = 0;
5684     result = (UINT16)(result
5685         + TPMI_ALG_SIG_SCHEME_Marshal(
5686             (TPMI_ALG_SIG_SCHEME*)&(source->sigAlg), buffer, size));
5687     result = (UINT16)(result
5688         + TPMU_SIGNATURE_Marshal((TPMU_SIGNATURE*)&(source->signature),
5689             buffer,
5690             size,
5691             (UINT32)source->sigAlg));
5692     return result;
5693 }
5694
5695 // Table "Definition of TPMU_ENCRYPTED_SECRET Union" (Part 2: Structures)
5696 TPM_RC
5697 TPMU_ENCRYPTED_SECRET_Unmarshal(
5698     TPMU_ENCRYPTED_SECRET* target, BYTE** buffer, INT32* size, UINT32 selector)
5699 {
5700     switch(selector)
5701     {
5702     # if ALG_ECC
5703         case TPM_ALG_ECC:
5704             return BYTE_Array_Unmarshal(
5705                 (BYTE*)&(target->ecc), buffer, size, (INT32)sizeof(TPMS_ECC_POINT));
5706     # endif // ALG_ECC
5707     # if ALG_RSA
5708         case TPM_ALG_RSA:
5709             return BYTE_Array_Unmarshal(
5710                 (BYTE*)&(target->rsa), buffer, size, (INT32)MAX_RSA_KEY_BYTES);
5711     # endif // ALG_RSA
5712     # if ALG_SYMCIPHER
5713         case TPM_ALG_SYMCIPHER:
5714             return BYTE_Array_Unmarshal((BYTE*)&(target->symmetric),
5715                 buffer,
5716                 size,
5717                 (INT32)sizeof(TPM2B_DIGEST));
5718     # endif // ALG_SYMCIPHER
5719     # if ALG_KEYEDHASH
5720         case TPM_ALG_KEYEDHASH:
5721             return BYTE_Array_Unmarshal((BYTE*)&(target->keyedHash),

```

```

5721         buffer,
5722         size,
5723         (INT32) sizeof(TPM2B_DIGEST));
5724 # endif // ALG_KEYEDHASH
5725     }
5726     return TPM_RC_SELECTOR;
5727 }
5728 UINT16
5729 TPMU_ENCRYPTED_SECRET_Marshal(
5730     TPMU_ENCRYPTED_SECRET* source, BYTE** buffer, INT32* size, UINT32 selector)
5731 {
5732     switch(selector)
5733     {
5734 # if ALG_ECC
5735         case TPM_ALG_ECC:
5736             return BYTE_Array_Marshal(
5737                 (BYTE*)&(source->ecc), buffer, size, (INT32) sizeof(TPMS_ECC_POINT));
5738 # endif // ALG_ECC
5739 # if ALG_RSA
5740         case TPM_ALG_RSA:
5741             return BYTE_Array_Marshal(
5742                 (BYTE*)&(source->rsa), buffer, size, (INT32) MAX_RSA_KEY_BYTES);
5743 # endif // ALG_RSA
5744 # if ALG_SYMCIPHER
5745         case TPM_ALG_SYMCIPHER:
5746             return BYTE_Array_Marshal((BYTE*)&(source->symmetric),
5747                 buffer,
5748                 size,
5749                 (INT32) sizeof(TPM2B_DIGEST));
5750 # endif // ALG_SYMCIPHER
5751 # if ALG_KEYEDHASH
5752         case TPM_ALG_KEYEDHASH:
5753             return BYTE_Array_Marshal((BYTE*)&(source->keyedHash),
5754                 buffer,
5755                 size,
5756                 (INT32) sizeof(TPM2B_DIGEST));
5757 # endif // ALG_KEYEDHASH
5758     }
5759     return 0;
5760 }
5761
5762 // Table "Definition of TPM2B_ENCRYPTED_SECRET Structure" (Part 2: Structures)
5763 TPM_RC
5764 TPM2B_ENCRYPTED_SECRET_Unmarshal(
5765     TPM2B_ENCRYPTED_SECRET* target, BYTE** buffer, INT32* size)
5766 {
5767     TPM_RC result;
5768     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
5769     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(TPMU_ENCRYPTED_SECRET)))
5770         result = TPM_RC_SIZE;
5771     if(result == TPM_RC_SUCCESS)
5772         result = BYTE_Array_Unmarshal(
5773             (BYTE*)&(target->t.secret), buffer, size, (INT32) target->t.size);
5774     return result;
5775 }
5776 UINT16
5777 TPM2B_ENCRYPTED_SECRET_Marshal(
5778     TPM2B_ENCRYPTED_SECRET* source, BYTE** buffer, INT32* size)
5779 {
5780     UINT16 result = 0;
5781     result =
5782         (UINT16) (result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
5783     // if size equal to 0, the rest of the structure is a zero buffer
5784     if(source->t.size == 0)
5785         return result;
5786     result = (UINT16) (result

```

```

5787         + BYTE_Array_Marshal((BYTE*)&(source->t.secret),
5788                               buffer,
5789                               size,
5790                               (INT32)source->t.size));
5791     return result;
5792 }
5793
5794 // Table "Definition of TPMI_ALG_PUBLIC Type" (Part 2: Structures)
5795 TPM_RC
5796 TPMI_ALG_PUBLIC_Unmarshal(TPMI_ALG_PUBLIC* target, BYTE** buffer, INT32* size)
5797 {
5798     TPM_RC result;
5799     result = TPM_ALG_ID_Unmarshal((TPM_ALG_ID*)target, buffer, size);
5800     if(result == TPM_RC_SUCCESS)
5801     {
5802         switch(*target)
5803         {
5804 # if ALG_RSA
5805             case TPM_ALG_RSA:
5806 # endif // ALG_RSA
5807 # if ALG_KEYEDHASH
5808             case TPM_ALG_KEYEDHASH:
5809 # endif // ALG_KEYEDHASH
5810 # if ALG_ECC
5811             case TPM_ALG_ECC:
5812 # endif // ALG_ECC
5813 # if ALG_SYMCIPHER
5814             case TPM_ALG_SYMCIPHER:
5815 # endif // ALG_SYMCIPHER
5816             break;
5817             default:
5818                 result = TPM_RC_TYPE;
5819                 break;
5820         }
5821     }
5822     return result;
5823 }
5824 # if !USE_MARSHALING_DEFINES
5825 UINT16
5826 TPMI_ALG_PUBLIC_Marshal(TPMI_ALG_PUBLIC* source, BYTE** buffer, INT32* size)
5827 {
5828     return TPM_ALG_ID_Marshal((TPM_ALG_ID*)source, buffer, size);
5829 }
5830 # endif // !USE_MARSHALING_DEFINES
5831
5832 // Table "Definition of TPMU_PUBLIC_ID Union" (Part 2: Structures)
5833 TPM_RC
5834 TPMU_PUBLIC_ID_Unmarshal(
5835     TPMU_PUBLIC_ID* target, BYTE** buffer, INT32* size, UINT32 selector)
5836 {
5837     switch(selector)
5838     {
5839 # if ALG_KEYEDHASH
5840         case TPM_ALG_KEYEDHASH:
5841             return TPM2B_DIGEST_Unmarshal(
5842                 (TPM2B_DIGEST*)&(target->keyedHash), buffer, size);
5843 # endif // ALG_KEYEDHASH
5844 # if ALG_SYMCIPHER
5845         case TPM_ALG_SYMCIPHER:
5846             return TPM2B_DIGEST_Unmarshal(
5847                 (TPM2B_DIGEST*)&(target->sym), buffer, size);
5848 # endif // ALG_SYMCIPHER
5849 # if ALG_RSA
5850         case TPM_ALG_RSA:
5851             return TPM2B_PUBLIC_KEY_RSA_Unmarshal(
5852                 (TPM2B_PUBLIC_KEY_RSA*)&(target->rsa), buffer, size);

```

```

5853 # endif // ALG_RSA
5854 # if ALG_ECC
5855     case TPM_ALG_ECC:
5856         return TPMS_ECC_POINT_Unmarshal(
5857             (TPMS_ECC_POINT*)&(target->ecc), buffer, size);
5858 # endif // ALG_ECC
5859     }
5860     return TPM_RC_SELECTOR;
5861 }
5862 UINT16
5863 TPMU_PUBLIC_ID_Marshal(
5864     TPMU_PUBLIC_ID* source, BYTE** buffer, INT32* size, UINT32 selector)
5865 {
5866     switch(selector)
5867     {
5868 # if ALG_KEYEDHASH
5869         case TPM_ALG_KEYEDHASH:
5870             return TPM2B_DIGEST_Marshal(
5871                 (TPM2B_DIGEST*)&(source->keyedHash), buffer, size);
5872 # endif // ALG_KEYEDHASH
5873 # if ALG_SYMCIPHER
5874         case TPM_ALG_SYMCIPHER:
5875             return TPM2B_DIGEST_Marshal((TPM2B_DIGEST*)&(source->sym), buffer, size);
5876 # endif // ALG_SYMCIPHER
5877 # if ALG_RSA
5878         case TPM_ALG_RSA:
5879             return TPM2B_PUBLIC_KEY_RSA_Marshal(
5880                 (TPM2B_PUBLIC_KEY_RSA*)&(source->rsa), buffer, size);
5881 # endif // ALG_RSA
5882 # if ALG_ECC
5883         case TPM_ALG_ECC:
5884             return TPMS_ECC_POINT_Marshal(
5885                 (TPMS_ECC_POINT*)&(source->ecc), buffer, size);
5886 # endif // ALG_ECC
5887     }
5888     return 0;
5889 }
5890
5891 // Table "Definition of TPMS_KEYEDHASH_PARMS Structure" (Part 2: Structures)
5892 TPM_RC
5893 TPMS_KEYEDHASH_PARMS_Unmarshal(
5894     TPMS_KEYEDHASH_PARMS* target, BYTE** buffer, INT32* size)
5895 {
5896     TPM_RC result;
5897     result = TPMT_KEYEDHASH_SCHEME_Unmarshal(
5898         (TPMT_KEYEDHASH_SCHEME*)&(target->scheme), buffer, size, 1);
5899     return result;
5900 }
5901 UINT16
5902 TPMS_KEYEDHASH_PARMS_Marshal(TPMS_KEYEDHASH_PARMS* source, BYTE** buffer, INT32* size)
5903 {
5904     UINT16 result = 0;
5905     result = (UINT16)(result
5906         + TPMT_KEYEDHASH_SCHEME_Marshal(
5907             (TPMT_KEYEDHASH_SCHEME*)&(source->scheme), buffer, size));
5908     return result;
5909 }
5910
5911 // Table "Definition of TPMS_RSA_PARMS Structure" (Part 2: Structures)
5912 TPM_RC
5913 TPMS_RSA_PARMS_Unmarshal(TPMS_RSA_PARMS* target, BYTE** buffer, INT32* size)
5914 {
5915     TPM_RC result;
5916     result = TPMT_SYM_DEF_OBJECT_Unmarshal(
5917         (TPMT_SYM_DEF_OBJECT*)&(target->symmetric), buffer, size, 1);
5918     if(result == TPM_RC_SUCCESS)

```

```

5919         result = TPMT_RSA_SCHEME_Unmarshal(
5920             (TPMT_RSA_SCHEME*)&(target->scheme), buffer, size, 1);
5921     if(result == TPM_RC_SUCCESS)
5922         result = TPMI_RSA_KEY_BITS_Unmarshal(
5923             (TPMI_RSA_KEY_BITS*)&(target->keyBits), buffer, size);
5924     if(result == TPM_RC_SUCCESS)
5925         result = UINT32_Unmarshal((UINT32*)&(target->exponent), buffer, size);
5926     return result;
5927 }
5928 UINT16
5929 TPMS_RSA_PARMS_Marshal(TPMS_RSA_PARMS* source, BYTE** buffer, INT32* size)
5930 {
5931     UINT16 result = 0;
5932     result = (UINT16)(result
5933         + TPMT_SYM_DEF_OBJECT_Marshal(
5934             (TPMT_SYM_DEF_OBJECT*)&(source->symmetric), buffer, size));
5935     result = (UINT16)(result
5936         + TPMT_RSA_SCHEME_Marshal(
5937             (TPMT_RSA_SCHEME*)&(source->scheme), buffer, size));
5938     result = (UINT16)(result
5939         + TPMI_RSA_KEY_BITS_Marshal(
5940             (TPMI_RSA_KEY_BITS*)&(source->keyBits), buffer, size));
5941     result =
5942         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->exponent), buffer, size));
5943     return result;
5944 }
5945
5946 // Table "Definition of TPMS_ECC_PARMS Structure" (Part 2: Structures)
5947 TPM_RC
5948 TPMS_ECC_PARMS_Unmarshal(TPMS_ECC_PARMS* target, BYTE** buffer, INT32* size)
5949 {
5950     TPM_RC result;
5951     result = TPMT_SYM_DEF_OBJECT_Unmarshal(
5952         (TPMT_SYM_DEF_OBJECT*)&(target->symmetric), buffer, size, 1);
5953     if(result == TPM_RC_SUCCESS)
5954         result = TPMT_ECC_SCHEME_Unmarshal(
5955             (TPMT_ECC_SCHEME*)&(target->scheme), buffer, size, 1);
5956     if(result == TPM_RC_SUCCESS)
5957         result = TPMI_ECC_CURVE_Unmarshal(
5958             (TPMI_ECC_CURVE*)&(target->curveID), buffer, size, 0);
5959     if(result == TPM_RC_SUCCESS)
5960         result = TPMT_KDF_SCHEME_Unmarshal(
5961             (TPMT_KDF_SCHEME*)&(target->kdf), buffer, size, 1);
5962     return result;
5963 }
5964 UINT16
5965 TPMS_ECC_PARMS_Marshal(TPMS_ECC_PARMS* source, BYTE** buffer, INT32* size)
5966 {
5967     UINT16 result = 0;
5968     result = (UINT16)(result
5969         + TPMT_SYM_DEF_OBJECT_Marshal(
5970             (TPMT_SYM_DEF_OBJECT*)&(source->symmetric), buffer, size));
5971     result = (UINT16)(result
5972         + TPMT_ECC_SCHEME_Marshal(
5973             (TPMT_ECC_SCHEME*)&(source->scheme), buffer, size));
5974     result = (UINT16)(result
5975         + TPMI_ECC_CURVE_Marshal(
5976             (TPMI_ECC_CURVE*)&(source->curveID), buffer, size));
5977     result = (UINT16)(result
5978         + TPMT_KDF_SCHEME_Marshal(
5979             (TPMT_KDF_SCHEME*)&(source->kdf), buffer, size));
5980     return result;
5981 }
5982
5983 // Table "Definition of TPMU_PUBLIC_PARMS Union" (Part 2: Structures)
5984 TPM_RC

```

```

5985 TPMU_PUBLIC_PARMS_Unmarshal(
5986     TPMU_PUBLIC_PARMS* target, BYTE** buffer, INT32* size, UINT32 selector)
5987 {
5988     switch(selector)
5989     {
5990 # if ALG_KEYEDHASH
5991         case TPM_ALG_KEYEDHASH:
5992             return TPMS_KEYEDHASH_PARMS_Unmarshal(
5993                 (TPMS_KEYEDHASH_PARMS*)&(target->keyedHashDetail), buffer, size);
5994 # endif // ALG_KEYEDHASH
5995 # if ALG_SYMCIPHER
5996         case TPM_ALG_SYMCIPHER:
5997             return TPMS_SYMCIPHER_PARMS_Unmarshal(
5998                 (TPMS_SYMCIPHER_PARMS*)&(target->symDetail), buffer, size);
5999 # endif // ALG_SYMCIPHER
6000 # if ALG_RSA
6001         case TPM_ALG_RSA:
6002             return TPMS_RSA_PARMS_Unmarshal(
6003                 (TPMS_RSA_PARMS*)&(target->rsaDetail), buffer, size);
6004 # endif // ALG_RSA
6005 # if ALG_ECC
6006         case TPM_ALG_ECC:
6007             return TPMS_ECC_PARMS_Unmarshal(
6008                 (TPMS_ECC_PARMS*)&(target->eccDetail), buffer, size);
6009 # endif // ALG_ECC
6010     }
6011     return TPM_RC_SELECTOR;
6012 }
6013
6014 TPMU_PUBLIC_PARMS_Marshal(
6015     TPMU_PUBLIC_PARMS* source, BYTE** buffer, INT32* size, UINT32 selector)
6016 {
6017     switch(selector)
6018     {
6019 # if ALG_KEYEDHASH
6020         case TPM_ALG_KEYEDHASH:
6021             return TPMS_KEYEDHASH_PARMS_Marshal(
6022                 (TPMS_KEYEDHASH_PARMS*)&(source->keyedHashDetail), buffer, size);
6023 # endif // ALG_KEYEDHASH
6024 # if ALG_SYMCIPHER
6025         case TPM_ALG_SYMCIPHER:
6026             return TPMS_SYMCIPHER_PARMS_Marshal(
6027                 (TPMS_SYMCIPHER_PARMS*)&(source->symDetail), buffer, size);
6028 # endif // ALG_SYMCIPHER
6029 # if ALG_RSA
6030         case TPM_ALG_RSA:
6031             return TPMS_RSA_PARMS_Marshal(
6032                 (TPMS_RSA_PARMS*)&(source->rsaDetail), buffer, size);
6033 # endif // ALG_RSA
6034 # if ALG_ECC
6035         case TPM_ALG_ECC:
6036             return TPMS_ECC_PARMS_Marshal(
6037                 (TPMS_ECC_PARMS*)&(source->eccDetail), buffer, size);
6038 # endif // ALG_ECC
6039     }
6040     return 0;
6041 }
6042
6043 // Table "Definition of TPMT_PUBLIC_PARMS Structure" (Part 2: Structures)
6044 TPM_RC
6045 TPMT_PUBLIC_PARMS_Unmarshal(TPMT_PUBLIC_PARMS* target, BYTE** buffer, INT32* size)
6046 {
6047     TPM_RC result;
6048     result =
6049         TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC*)&(target->type), buffer, size);
6050     if(result == TPM_RC_SUCCESS)

```



```

6051         result =
6052             TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS*)&(target->parameters),
6053                                         buffer,
6054                                         size,
6055                                         (UINT32)target->type);
6056     return result;
6057 }
6058 UINT16
6059 TPMT_PUBLIC_PARMS_Marshal(TPMT_PUBLIC_PARMS* source, BYTE** buffer, INT32* size)
6060 {
6061     UINT16 result = 0;
6062     result = (UINT16)(result
6063                     + TPMI_ALG_PUBLIC_Marshal(
6064                         (TPMI_ALG_PUBLIC*)&(source->type), buffer, size));
6065     result = (UINT16)(result
6066                     + TPMU_PUBLIC_PARMS_Marshal(
6067                         (TPMU_PUBLIC_PARMS*)&(source->parameters),
6068                         buffer,
6069                         size,
6070                         (UINT32)source->type));
6071     return result;
6072 }
6073
6074 // Table "Definition of TPMT_PUBLIC Structure" (Part 2: Structures)
6075 TPM_RC
6076 TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC* target, BYTE** buffer, INT32* size, BOOL flag)
6077 {
6078     TPM_RC result;
6079     result =
6080         TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC*)&(target->type), buffer, size);
6081     if(result == TPM_RC_SUCCESS)
6082         result = TPMI_ALG_HASH_Unmarshal(
6083             (TPMI_ALG_HASH*)&(target->nameAlg), buffer, size, flag);
6084     if(result == TPM_RC_SUCCESS)
6085         result = TPMA_OBJECT_Unmarshal(
6086             (TPMA_OBJECT*)&(target->objectAttributes), buffer, size);
6087     if(result == TPM_RC_SUCCESS)
6088         result = TPM2B_DIGEST_Unmarshal(
6089             (TPM2B_DIGEST*)&(target->authPolicy), buffer, size);
6090     if(result == TPM_RC_SUCCESS)
6091         result =
6092             TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS*)&(target->parameters),
6093                                         buffer,
6094                                         size,
6095                                         (UINT32)target->type);
6096     if(result == TPM_RC_SUCCESS)
6097         result = TPMU_PUBLIC_ID_Unmarshal(
6098             (TPMU_PUBLIC_ID*)&(target->unique), buffer, size, (UINT32)target->type);
6099     return result;
6100 }
6101 UINT16
6102 TPMT_PUBLIC_Marshal(TPMT_PUBLIC* source, BYTE** buffer, INT32* size)
6103 {
6104     UINT16 result = 0;
6105     result = (UINT16)(result
6106                     + TPMI_ALG_PUBLIC_Marshal(
6107                         (TPMI_ALG_PUBLIC*)&(source->type), buffer, size));
6108     result = (UINT16)(result
6109                     + TPMI_ALG_HASH_Marshal(
6110                         (TPMI_ALG_HASH*)&(source->nameAlg), buffer, size));
6111     result = (UINT16)(result
6112                     + TPMA_OBJECT_Marshal(
6113                         (TPMA_OBJECT*)&(source->objectAttributes), buffer, size));
6114     result = (UINT16)(result
6115                     + TPM2B_DIGEST_Marshal(
6116                         (TPM2B_DIGEST*)&(source->authPolicy), buffer, size));

```

```

6117     result      = (UINT16) (result
6118                          + TPMU_PUBLIC_PARMS_Marshal(
6119                            (TPMU_PUBLIC_PARMS*)&(source->parameters),
6120                            buffer,
6121                            size,
6122                            (UINT32)source->type));
6123     result      = (UINT16) (result
6124                          + TPMU_PUBLIC_ID_Marshal((TPMU_PUBLIC_ID*)&(source->unique),
6125                                                    buffer,
6126                                                    size,
6127                                                    (UINT32)source->type));
6128     return result;
6129 }
6130
6131 // Table "Definition of TPM2B_PUBLIC Structure" (Part 2: Structures)
6132 TPM_RC
6133 TPM2B_PUBLIC_Unmarshal(TPM2B_PUBLIC* target, BYTE** buffer, INT32* size, BOOL flag)
6134 {
6135     TPM_RC result;
6136     result = UINT16_Unmarshal((UINT16*)&(target->size), buffer, size);
6137     if(result == TPM_RC_SUCCESS)
6138     {
6139         // if size is zero, then the required structure is missing
6140         if(target->size == 0)
6141             result = TPM_RC_SIZE;
6142         else
6143         {
6144             INT32 startSize = *size;
6145             result = TPMT_PUBLIC_Unmarshal(
6146                 (TPMT_PUBLIC*)&(target->publicArea), buffer, size, flag);
6147             if((result == TPM_RC_SUCCESS) && (target->size != (startSize - *size)))
6148                 result = TPM_RC_SIZE;
6149         }
6150     }
6151     return result;
6152 }
6153
6154 UINT16
6155 TPM2B_PUBLIC_Marshal(TPM2B_PUBLIC* source, BYTE** buffer, INT32* size)
6156 {
6157     UINT16 result = 0;
6158     // Marshal a dummy value of the 2B size. This makes sure that 'buffer'
6159     // and 'size' are advanced as necessary (i.e., if they are present)
6160     result = UINT16_Marshal(&result, buffer, size);
6161     // Marshal the structure
6162     result = (UINT16) (result
6163                      + TPMT_PUBLIC_Marshal(
6164                        (TPMT_PUBLIC*)&(source->publicArea), buffer, size));
6165     // if a buffer was provided, go back and fill in the actual size
6166     if(buffer != NULL)
6167         UINT16_TO_BYTE_ARRAY((result - 2), (*buffer - result));
6168     return result;
6169 }
6170
6171 // Table "Definition of TPM2B_TEMPLATE Structure" (Part 2: Structures)
6172 TPM_RC
6173 TPM2B_TEMPLATE_Unmarshal(TPM2B_TEMPLATE* target, BYTE** buffer, INT32* size)
6174 {
6175     TPM_RC result;
6176     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
6177     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(TPMT_PUBLIC)))
6178         result = TPM_RC_SIZE;
6179     if(result == TPM_RC_SUCCESS)
6180         result = BYTE_Array_Unmarshal(
6181             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
6182     return result;
6183 }

```

```

6183  UINT16
6184  TPM2B_TEMPLATE_Marshal(TPM2B_TEMPLATE* source, BYTE** buffer, INT32* size)
6185  {
6186      UINT16 result = 0;
6187      result =
6188          (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
6189      // if size equal to 0, the rest of the structure is a zero buffer
6190      if(source->t.size == 0)
6191          return result;
6192      result = (UINT16)(result
6193                  + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
6194                                          buffer,
6195                                          size,
6196                                          (INT32)source->t.size));
6197      return result;
6198  }
6199
6200  // Table "Definition of TPM2B_PRIVATE_VENDOR_SPECIFIC Structure" (Part 2: Structures)
6201  TPM_RC
6202  TPM2B_PRIVATE_VENDOR_SPECIFIC_Unmarshal(
6203      TPM2B_PRIVATE_VENDOR_SPECIFIC* target, BYTE** buffer, INT32* size)
6204  {
6205      TPM_RC result;
6206      result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
6207      if((result == TPM_RC_SUCCESS) && (target->t.size > PRIVATE_VENDOR_SPECIFIC_BYTES))
6208          result = TPM_RC_SIZE;
6209      if(result == TPM_RC_SUCCESS)
6210          result = BYTE_Array_Unmarshal(
6211              (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
6212      return result;
6213  }
6214  UINT16
6215  TPM2B_PRIVATE_VENDOR_SPECIFIC_Marshal(
6216      TPM2B_PRIVATE_VENDOR_SPECIFIC* source, BYTE** buffer, INT32* size)
6217  {
6218      UINT16 result = 0;
6219      result =
6220          (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
6221      // if size equal to 0, the rest of the structure is a zero buffer
6222      if(source->t.size == 0)
6223          return result;
6224      result = (UINT16)(result
6225                  + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
6226                                          buffer,
6227                                          size,
6228                                          (INT32)source->t.size));
6229      return result;
6230  }
6231
6232  // Table "Definition of TPMU_SENSITIVE_COMPOSITE Union" (Part 2: Structures)
6233  TPM_RC
6234  TPMU_SENSITIVE_COMPOSITE_Unmarshal(
6235      TPMU_SENSITIVE_COMPOSITE* target, BYTE** buffer, INT32* size, UINT32 selector)
6236  {
6237      switch(selector)
6238      {
6239          # if ALG_RSA
6240              case TPM_ALG_RSA:
6241                  return TPM2B_PRIVATE_KEY_RSA_Unmarshal(
6242                      (TPM2B_PRIVATE_KEY_RSA*)&(target->rsa), buffer, size);
6243          # endif // ALG_RSA
6244          # if ALG_ECC
6245              case TPM_ALG_ECC:
6246                  return TPM2B_ECC_PARAMETER_Unmarshal(
6247                      (TPM2B_ECC_PARAMETER*)&(target->ecc), buffer, size);
6248          # endif // ALG_ECC

```

```

6249 # if ALG_KEYEDHASH
6250     case TPM_ALG_KEYEDHASH:
6251         return TPM2B_SENSITIVE_DATA_Unmarshal(
6252             (TPM2B_SENSITIVE_DATA*)&(target->bits), buffer, size);
6253 # endif // ALG_KEYEDHASH
6254 # if ALG_SYMCIPHER
6255     case TPM_ALG_SYMCIPHER:
6256         return TPM2B_SYM_KEY_Unmarshal(
6257             (TPM2B_SYM_KEY*)&(target->sym), buffer, size);
6258 # endif // ALG_SYMCIPHER
6259 }
6260 return TPM_RC_SELECTOR;
6261 }
6262 UINT16
6263 TPMU_SENSITIVE_COMPOSITE_Marshal(
6264     TPMU_SENSITIVE_COMPOSITE* source, BYTE** buffer, INT32* size, UINT32 selector)
6265 {
6266     switch(selector)
6267     {
6268 # if ALG_RSA
6269         case TPM_ALG_RSA:
6270             return TPM2B_PRIVATE_KEY_RSA_Marshal(
6271                 (TPM2B_PRIVATE_KEY_RSA*)&(source->rsa), buffer, size);
6272 # endif // ALG_RSA
6273 # if ALG_ECC
6274         case TPM_ALG_ECC:
6275             return TPM2B_ECC_PARAMETER_Marshal(
6276                 (TPM2B_ECC_PARAMETER*)&(source->ecc), buffer, size);
6277 # endif // ALG_ECC
6278 # if ALG_KEYEDHASH
6279         case TPM_ALG_KEYEDHASH:
6280             return TPM2B_SENSITIVE_DATA_Marshal(
6281                 (TPM2B_SENSITIVE_DATA*)&(source->bits), buffer, size);
6282 # endif // ALG_KEYEDHASH
6283 # if ALG_SYMCIPHER
6284         case TPM_ALG_SYMCIPHER:
6285             return TPM2B_SYM_KEY_Marshal(
6286                 (TPM2B_SYM_KEY*)&(source->sym), buffer, size);
6287 # endif // ALG_SYMCIPHER
6288     }
6289     return 0;
6290 }
6291
6292 // Table "Definition of TPMT_SENSITIVE Structure" (Part 2: Structures)
6293 TPM_RC
6294 TPMT_SENSITIVE_Unmarshal(TPMT_SENSITIVE* target, BYTE** buffer, INT32* size)
6295 {
6296     TPM_RC result;
6297     result = TPMT_ALG_PUBLIC_Unmarshal(
6298         (TPMT_ALG_PUBLIC*)&(target->sensitiveType), buffer, size);
6299     if(result == TPM_RC_SUCCESS)
6300         result =
6301             TPM2B_AUTH_Unmarshal((TPM2B_AUTH*)&(target->authValue), buffer, size);
6302     if(result == TPM_RC_SUCCESS)
6303         result =
6304             TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)&(target->seedValue), buffer, size);
6305     if(result == TPM_RC_SUCCESS)
6306         result = TPMU_SENSITIVE_COMPOSITE_Unmarshal(
6307             (TPMU_SENSITIVE_COMPOSITE*)&(target->sensitive),
6308             buffer,
6309             size,
6310             (UINT32)target->sensitiveType);
6311     return result;
6312 }
6313
6314 TPMU_SENSITIVE_Marshal(TPMT_SENSITIVE* source, BYTE** buffer, INT32* size)

```

```

6315 {
6316     UINT16 result = 0;
6317     result = (UINT16) (result
6318         + TPMI_ALG_PUBLIC_Marshal(
6319             (TPMI_ALG_PUBLIC*)&(source->sensitiveType), buffer, size));
6320     result = (UINT16) (result
6321         + TPM2B_AUTH_Marshal(
6322             (TPM2B_AUTH*)&(source->authValue), buffer, size));
6323     result = (UINT16) (result
6324         + TPM2B_DIGEST_Marshal(
6325             (TPM2B_DIGEST*)&(source->seedValue), buffer, size));
6326     result = (UINT16) (result
6327         + TPMU_SENSITIVE_COMPOSITE_Marshal(
6328             (TPMU_SENSITIVE_COMPOSITE*)&(source->sensitive),
6329             buffer,
6330             size,
6331             (UINT32)source->sensitiveType));
6332     return result;
6333 }
6334
6335 // Table "Definition of TPM2B_SENSITIVE Structure" (Part 2: Structures)
6336 TPM_RC
6337 TPM2B_SENSITIVE_Unmarshal(TPM2B_SENSITIVE* target, BYTE** buffer, INT32* size)
6338 {
6339     TPM_RC result;
6340     result = UINT16_Unmarshal((UINT16*)&(target->size), buffer, size);
6341     // if there was an error or if target->size equal to 0,
6342     // skip unmarshaling of the structure
6343     if((result == TPM_RC_SUCCESS) && (target->size != 0))
6344     {
6345         INT32 startSize = *size;
6346         result = TPMT_SENSITIVE_Unmarshal(
6347             (TPMT_SENSITIVE*)&(target->sensitiveArea), buffer, size);
6348         if((result == TPM_RC_SUCCESS) && (target->size != (startSize - *size)))
6349             result = TPM_RC_SIZE;
6350     }
6351     return result;
6352 }
6353
6354 UINT16
6355 TPM2B_SENSITIVE_Marshal(TPM2B_SENSITIVE* source, BYTE** buffer, INT32* size)
6356 {
6357     UINT16 result = 0;
6358     // Marshal a dummy value of the 2B size. This makes sure that 'buffer'
6359     // and 'size' are advanced as necessary (i.e., if they are present)
6360     result = UINT16_Marshal(&result, buffer, size);
6361     // Marshal the structure
6362     result = (UINT16) (result
6363         + TPMT_SENSITIVE_Marshal(
6364             (TPMT_SENSITIVE*)&(source->sensitiveArea), buffer, size));
6365     // if a buffer was provided, go back and fill in the actual size
6366     if(buffer != NULL)
6367         UINT16_TO_BYTE_ARRAY((result - 2), (*buffer - result));
6368     return result;
6369 }
6370
6371 // Table "Definition of TPM2B_PRIVATE Structure" (Part 2: Structures)
6372 TPM_RC
6373 TPM2B_PRIVATE_Unmarshal(TPM2B_PRIVATE* target, BYTE** buffer, INT32* size)
6374 {
6375     TPM_RC result;
6376     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
6377     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(_PRIVATE)))
6378         result = TPM_RC_SIZE;
6379     if(result == TPM_RC_SUCCESS)
6380         result = BYTE_Array_Unmarshal(
6381             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);

```

```

6381     return result;
6382 }
6383 UINT16
6384 TPM2B_PRIVATE_Marshal(TPM2B_PRIVATE* source, BYTE** buffer, INT32* size)
6385 {
6386     UINT16 result = 0;
6387     result =
6388         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
6389     // if size equal to 0, the rest of the structure is a zero buffer
6390     if(source->t.size == 0)
6391         return result;
6392     result = (UINT16)(result
6393         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
6394             buffer,
6395             size,
6396             (INT32)source->t.size));
6397     return result;
6398 }
6399
6400 // Table "Definition of TPM2B_ID_OBJECT Structure" (Part 2: Structures)
6401 TPM_RC
6402 TPM2B_ID_OBJECT_Unmarshal(TPM2B_ID_OBJECT* target, BYTE** buffer, INT32* size)
6403 {
6404     TPM_RC result;
6405     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
6406     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(TPMS_ID_OBJECT)))
6407         result = TPM_RC_SIZE;
6408     if(result == TPM_RC_SUCCESS)
6409         result = BYTE_Array_Unmarshal(
6410             (BYTE*)&(target->t.credential), buffer, size, (INT32)target->t.size);
6411     return result;
6412 }
6413 UINT16
6414 TPM2B_ID_OBJECT_Marshal(TPM2B_ID_OBJECT* source, BYTE** buffer, INT32* size)
6415 {
6416     UINT16 result = 0;
6417     result =
6418         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
6419     // if size equal to 0, the rest of the structure is a zero buffer
6420     if(source->t.size == 0)
6421         return result;
6422     result = (UINT16)(result
6423         + BYTE_Array_Marshal((BYTE*)&(source->t.credential),
6424             buffer,
6425             size,
6426             (INT32)source->t.size));
6427     return result;
6428 }
6429
6430 // Table "Definition of TPMS_NV_PIN_COUNTER_PARAMETERS Structure" (Part 2: Structures)
6431 TPM_RC
6432 TPMS_NV_PIN_COUNTER_PARAMETERS_Unmarshal(
6433     TPMS_NV_PIN_COUNTER_PARAMETERS* target, BYTE** buffer, INT32* size)
6434 {
6435     TPM_RC result;
6436     result = UINT32_Unmarshal((UINT32*)&(target->pinCount), buffer, size);
6437     if(result == TPM_RC_SUCCESS)
6438         result = UINT32_Unmarshal((UINT32*)&(target->pinLimit), buffer, size);
6439     return result;
6440 }
6441 UINT16
6442 TPMS_NV_PIN_COUNTER_PARAMETERS_Marshal(
6443     TPMS_NV_PIN_COUNTER_PARAMETERS* source, BYTE** buffer, INT32* size)
6444 {
6445     UINT16 result = 0;
6446     result =

```



```

6447         (UINT16) (result + UINT32_Marshal((UINT32*)&(source->pinCount), buffer, size));
6448     result =
6449         (UINT16) (result + UINT32_Marshal((UINT32*)&(source->pinLimit), buffer, size));
6450     return result;
6451 }
6452
6453 // Table "Definition of TPMA_NV Bits" (Part 2: Structures)
6454 TPM_RC
6455 TPMA_NV_Unmarshal(TPMA_NV* target, BYTE** buffer, INT32* size)
6456 {
6457     TPM_RC result;
6458     result = UINT32_Unmarshal((UINT32*)target, buffer, size);
6459     if(result == TPM_RC_SUCCESS)
6460     {
6461         // check that no reserved bits are set
6462         if(*(UINT32*)target) & (UINT32)0x01f00300)
6463             result = TPM_RC_RESERVED_BITS;
6464     }
6465     return result;
6466 }
6467 # if !USE_MARSHALING_DEFINES
6468 UINT16
6469 TPMA_NV_Marshal(TPMA_NV* source, BYTE** buffer, INT32* size)
6470 {
6471     return UINT32_Marshal((UINT32*)source, buffer, size);
6472 }
6473 # endif // !USE_MARSHALING_DEFINES
6474
6475 // Table "Definition of TPMA_NV_EXP Bits" (Part 2: Structures)
6476 TPM_RC
6477 TPMA_NV_EXP_Unmarshal(TPMA_NV_EXP* target, BYTE** buffer, INT32* size)
6478 {
6479     TPM_RC result;
6480     result = UINT64_Unmarshal((UINT64*)target, buffer, size);
6481     if(result == TPM_RC_SUCCESS)
6482     {
6483         // check that no reserved bits are set
6484         if(*(UINT64*)target) & (UINT64)0xffffffff801f00300)
6485             result = TPM_RC_RESERVED_BITS;
6486     }
6487     return result;
6488 }
6489 # if !USE_MARSHALING_DEFINES
6490 UINT16
6491 TPMA_NV_EXP_Marshal(TPMA_NV_EXP* source, BYTE** buffer, INT32* size)
6492 {
6493     return UINT64_Marshal((UINT64*)source, buffer, size);
6494 }
6495 # endif // !USE_MARSHALING_DEFINES
6496
6497 // Table "Definition of TPMS_NV_PUBLIC Structure" (Part 2: Structures)
6498 TPM_RC
6499 TPMS_NV_PUBLIC_Unmarshal(TPMS_NV_PUBLIC* target, BYTE** buffer, INT32* size)
6500 {
6501     TPM_RC result;
6502     result = TPMS_NV_PUBLIC_Unmarshal(
6503         (TPMS_NV_PUBLIC*)&(target->nvIndex), buffer, size);
6504     if(result == TPM_RC_SUCCESS)
6505         result = TPMS_NV_PUBLIC_Unmarshal(
6506             (TPMS_NV_PUBLIC*)&(target->nameAlg), buffer, size, 0);
6507     if(result == TPM_RC_SUCCESS)
6508         result = TPMS_NV_PUBLIC_Unmarshal((TPMS_NV_PUBLIC*)&(target->attributes), buffer, size);
6509     if(result == TPM_RC_SUCCESS)
6510         result = TPMS_NV_PUBLIC_Unmarshal(
6511             (TPMS_NV_PUBLIC*)&(target->authPolicy), buffer, size);
6512     if(result == TPM_RC_SUCCESS)

```

```

6513         result = UINT16_Unmarshal((UINT16*)&(target->dataSize), buffer, size);
6514         if((result == TPM_RC_SUCCESS) && (target->dataSize > MAX_NV_INDEX_SIZE))
6515             result = TPM_RC_SIZE;
6516         return result;
6517     }
6518     UINT16
6519     TPMS_NV_PUBLIC_Marshal(TPMS_NV_PUBLIC* source, BYTE** buffer, INT32* size)
6520     {
6521         UINT16 result = 0;
6522         result =
6523             (UINT16)(result
6524                 + TPMI_RH_NV_LEGACY_INDEX_Marshal(
6525                     (TPMI_RH_NV_LEGACY_INDEX*)&(source->nvIndex), buffer, size));
6526         result = (UINT16)(result
6527             + TPMI_ALG_HASH_Marshal(
6528                 (TPMI_ALG_HASH*)&(source->nameAlg), buffer, size));
6529         result =
6530             (UINT16)(result
6531                 + TPMA_NV_Marshal((TPMA_NV*)&(source->attributes), buffer, size));
6532         result = (UINT16)(result
6533             + TPM2B_DIGEST_Marshal(
6534                 (TPM2B_DIGEST*)&(source->authPolicy), buffer, size));
6535         result =
6536             (UINT16)(result + UINT16_Marshal((UINT16*)&(source->dataSize), buffer, size));
6537         return result;
6538     }
6539
6540     // Table "Definition of TPM2B_NV_PUBLIC Structure" (Part 2: Structures)
6541     TPM_RC
6542     TPM2B_NV_PUBLIC_Unmarshal(TPM2B_NV_PUBLIC* target, BYTE** buffer, INT32* size)
6543     {
6544         TPM_RC result;
6545         result = UINT16_Unmarshal((UINT16*)&(target->size), buffer, size);
6546         if(result == TPM_RC_SUCCESS)
6547         {
6548             // if size is zero, then the required structure is missing
6549             if(target->size == 0)
6550                 result = TPM_RC_SIZE;
6551             else
6552             {
6553                 INT32 startSize = *size;
6554                 result = TPMS_NV_PUBLIC_Unmarshal(
6555                     (TPMS_NV_PUBLIC*)&(target->nvPublic), buffer, size);
6556                 if((result == TPM_RC_SUCCESS) && (target->size != (startSize - *size)))
6557                     result = TPM_RC_SIZE;
6558             }
6559         }
6560         return result;
6561     }
6562     UINT16
6563     TPM2B_NV_PUBLIC_Marshal(TPM2B_NV_PUBLIC* source, BYTE** buffer, INT32* size)
6564     {
6565         UINT16 result = 0;
6566         // Marshal a dummy value of the 2B size. This makes sure that 'buffer'
6567         // and 'size' are advanced as necessary (i.e., if they are present)
6568         result = UINT16_Marshal(&result, buffer, size);
6569         // Marshal the structure
6570         result = (UINT16)(result
6571             + TPMS_NV_PUBLIC_Marshal(
6572                 (TPMS_NV_PUBLIC*)&(source->nvPublic), buffer, size));
6573         // if a buffer was provided, go back and fill in the actual size
6574         if(buffer != NULL)
6575             UINT16_TO_BYTE_ARRAY((result - 2), (*buffer - result));
6576         return result;
6577     }
6578

```

```

6579 // Table "Definition of TPMS_NV_PUBLIC_EXP_ATTR Structure" (Part 2: Structures)
6580 TPM_RC
6581 TPMS_NV_PUBLIC_EXP_ATTR_Unmarshal(
6582     TPMS_NV_PUBLIC_EXP_ATTR* target, BYTE** buffer, INT32* size)
6583 {
6584     TPM_RC result;
6585     result = TPMI_RH_NV_EXP_INDEX_Unmarshal(
6586         (TPMI_RH_NV_EXP_INDEX*)&(target->nvIndex), buffer, size);
6587     if(result == TPM_RC_SUCCESS)
6588         result = TPMI_ALG_HASH_Unmarshal(
6589             (TPMI_ALG_HASH*)&(target->nameAlg), buffer, size, 0);
6590     if(result == TPM_RC_SUCCESS)
6591         result =
6592             TPMA_NV_EXP_Unmarshal((TPMA_NV_EXP*)&(target->attributes), buffer, size);
6593     if(result == TPM_RC_SUCCESS)
6594         result = TPM2B_DIGEST_Unmarshal(
6595             (TPM2B_DIGEST*)&(target->authPolicy), buffer, size);
6596     if(result == TPM_RC_SUCCESS)
6597         result = UINT16_Unmarshal((UINT16*)&(target->dataSize), buffer, size);
6598     if((result == TPM_RC_SUCCESS) && (target->dataSize > MAX_NV_INDEX_SIZE))
6599         result = TPM_RC_SIZE;
6600     return result;
6601 }
6602 UINT16
6603 TPMS_NV_PUBLIC_EXP_ATTR_Marshal(
6604     TPMS_NV_PUBLIC_EXP_ATTR* source, BYTE** buffer, INT32* size)
6605 {
6606     UINT16 result = 0;
6607     result = (UINT16)(result
6608         + TPMI_RH_NV_EXP_INDEX_Marshal(
6609             (TPMI_RH_NV_EXP_INDEX*)&(source->nvIndex), buffer, size));
6610     result = (UINT16)(result
6611         + TPMI_ALG_HASH_Marshal(
6612             (TPMI_ALG_HASH*)&(source->nameAlg), buffer, size));
6613     result = (UINT16)(result
6614         + TPMA_NV_EXP_Marshal(
6615             (TPMA_NV_EXP*)&(source->attributes), buffer, size));
6616     result = (UINT16)(result
6617         + TPM2B_DIGEST_Marshal(
6618             (TPM2B_DIGEST*)&(source->authPolicy), buffer, size));
6619     result =
6620         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->dataSize), buffer, size));
6621     return result;
6622 }
6623
6624 // Table "Definition of TPMU_NV_PUBLIC_2 Union" (Part 2: Structures)
6625 TPM_RC
6626 TPMU_NV_PUBLIC_2_Unmarshal(
6627     TPMU_NV_PUBLIC_2* target, BYTE** buffer, INT32* size, UINT32 selector)
6628 {
6629     switch(selector)
6630     {
6631         case TPM_HT_NV_INDEX:
6632             return TPMS_NV_PUBLIC_Unmarshal(
6633                 (TPMS_NV_PUBLIC*)&(target->nvIndex), buffer, size);
6634         case TPM_HT_EXTERNAL_NV:
6635             return TPMS_NV_PUBLIC_EXP_ATTR_Unmarshal(
6636                 (TPMS_NV_PUBLIC_EXP_ATTR*)&(target->externalNV), buffer, size);
6637         case TPM_HT_PERMANENT_NV:
6638             return TPMS_NV_PUBLIC_Unmarshal(
6639                 (TPMS_NV_PUBLIC*)&(target->permanentNV), buffer, size);
6640     }
6641     return TPM_RC_SELECTOR;
6642 }
6643 UINT16
6644 TPMU_NV_PUBLIC_2_Marshal(

```

```

6645     TPMU_NV_PUBLIC_2* source, BYTE** buffer, INT32* size, UINT32 selector)
6646 {
6647     switch(selector)
6648     {
6649         case TPM_HT_NV_INDEX:
6650             return TPMS_NV_PUBLIC_Marshal(
6651                 (TPMS_NV_PUBLIC*)&(source->nvIndex), buffer, size);
6652         case TPM_HT_EXTERNAL_NV:
6653             return TPMS_NV_PUBLIC_EXP_ATTR_Marshal(
6654                 (TPMS_NV_PUBLIC_EXP_ATTR*)&(source->externalNV), buffer, size);
6655         case TPM_HT_PERMANENT_NV:
6656             return TPMS_NV_PUBLIC_Marshal(
6657                 (TPMS_NV_PUBLIC*)&(source->permanentNV), buffer, size);
6658     }
6659     return 0;
6660 }
6661
6662 // Table "Definition of TPMT_NV_PUBLIC_2 Structure" (Part 2: Structures)
6663 TPM_RC
6664 TPMT_NV_PUBLIC_2_Unmarshal(TPMT_NV_PUBLIC_2* target, BYTE** buffer, INT32* size)
6665 {
6666     TPM_RC result;
6667     result = TPM_HT_Unmarshal((TPM_HT*)&(target->handleType), buffer, size);
6668     if(result == TPM_RC_SUCCESS)
6669         result = TPMU_NV_PUBLIC_2_Unmarshal((TPMU_NV_PUBLIC_2*)&(target->nvPublic2),
6670             buffer,
6671             size,
6672             (UINT32)target->handleType);
6673     return result;
6674 }
6675
6676 UINT16
6677 TPMT_NV_PUBLIC_2_Marshal(TPMT_NV_PUBLIC_2* source, BYTE** buffer, INT32* size)
6678 {
6679     UINT16 result = 0;
6680     result = (UINT16)(result
6681         + TPM_HT_Marshal((TPM_HT*)&(source->handleType), buffer, size));
6682     result =
6683         (UINT16)(result
6684             + TPMU_NV_PUBLIC_2_Marshal((TPMU_NV_PUBLIC_2*)&(source->nvPublic2),
6685                 buffer,
6686                 size,
6687                 (UINT32)source->handleType));
6688     return result;
6689 }
6690
6691 // Table "Definition of TPM2B_NV_PUBLIC_2 Structure" (Part 2: Structures)
6692 TPM_RC
6693 TPM2B_NV_PUBLIC_2_Unmarshal(TPM2B_NV_PUBLIC_2* target, BYTE** buffer, INT32* size)
6694 {
6695     TPM_RC result;
6696     result = UINT16_Unmarshal((UINT16*)&(target->size), buffer, size);
6697     if(result == TPM_RC_SUCCESS)
6698     {
6699         // if size is zero, then the required structure is missing
6700         if(target->size == 0)
6701             result = TPM_RC_SIZE;
6702         else
6703         {
6704             INT32 startSize = *size;
6705             result = TPMT_NV_PUBLIC_2_Unmarshal(
6706                 (TPMT_NV_PUBLIC_2*)&(target->nvPublic2), buffer, size);
6707             if((result == TPM_RC_SUCCESS) && (target->size != (startSize - *size)))
6708                 result = TPM_RC_SIZE;
6709         }
6710     }
6711     return result;

```

```

6711 }
6712 UINT16
6713 TPM2B_NV_PUBLIC_2_Marshal(TPM2B_NV_PUBLIC_2* source, BYTE** buffer, INT32* size)
6714 {
6715     UINT16 result = 0;
6716     // Marshal a dummy value of the 2B size. This makes sure that 'buffer'
6717     // and 'size' are advanced as necessary (i.e., if they are present)
6718     result = UINT16_Marshal(&result, buffer, size);
6719     // Marshal the structure
6720     result = (UINT16)(result
6721         + TPMT_NV_PUBLIC_2_Marshal(
6722             (TPMT_NV_PUBLIC_2*)&(source->nvPublic2), buffer, size));
6723     // if a buffer was provided, go back and fill in the actual size
6724     if(buffer != NULL)
6725         UINT16_TO_BYTE_ARRAY((result - 2), (*buffer - result));
6726     return result;
6727 }
6728
6729 // Table "Definition of TPM2B_CONTEXT_SENSITIVE Structure" (Part 2: Structures)
6730 TPM_RC
6731 TPM2B_CONTEXT_SENSITIVE_Unmarshal(
6732     TPM2B_CONTEXT_SENSITIVE* target, BYTE** buffer, INT32* size)
6733 {
6734     TPM_RC result;
6735     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
6736     if((result == TPM_RC_SUCCESS) && (target->t.size > MAX_CONTEXT_SIZE))
6737         result = TPM_RC_SIZE;
6738     if(result == TPM_RC_SUCCESS)
6739         result = BYTE_Array_Unmarshal(
6740             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
6741     return result;
6742 }
6743
6744 TPM2B_CONTEXT_SENSITIVE_Marshal(
6745     TPM2B_CONTEXT_SENSITIVE* source, BYTE** buffer, INT32* size)
6746 {
6747     UINT16 result = 0;
6748     result =
6749         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
6750     // if size equal to 0, the rest of the structure is a zero buffer
6751     if(source->t.size == 0)
6752         return result;
6753     result = (UINT16)(result
6754         + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
6755             buffer,
6756             size,
6757             (INT32)source->t.size));
6758     return result;
6759 }
6760
6761 // Table "Definition of TPMS_CONTEXT_DATA Structure" (Part 2: Structures)
6762 TPM_RC
6763 TPMS_CONTEXT_DATA_Unmarshal(TPMS_CONTEXT_DATA* target, BYTE** buffer, INT32* size)
6764 {
6765     TPM_RC result;
6766     result =
6767         TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST*)&(target->integrity), buffer, size);
6768     if(result == TPM_RC_SUCCESS)
6769         result = TPM2B_CONTEXT_SENSITIVE_Unmarshal(
6770             (TPM2B_CONTEXT_SENSITIVE*)&(target->encrypted), buffer, size);
6771     return result;
6772 }
6773
6774 TPMS_CONTEXT_DATA_Marshal(TPMS_CONTEXT_DATA* source, BYTE** buffer, INT32* size)
6775 {
6776     UINT16 result = 0;

```



```

6777     result          = (UINT16)(result
6778                         + TPM2B_DIGEST_Marshal(
6779                             (TPM2B_DIGEST*)&(source->integrity), buffer, size));
6780     result =
6781         (UINT16)(result
6782                 + TPM2B_CONTEXT_SENSITIVE_Marshal(
6783                     (TPM2B_CONTEXT_SENSITIVE*)&(source->encrypted), buffer, size));
6784     return result;
6785 }
6786
6787 // Table "Definition of TPM2B_CONTEXT_DATA Structure" (Part 2: Structures)
6788 TPM_RC
6789 TPM2B_CONTEXT_DATA_Unmarshal(TPM2B_CONTEXT_DATA* target, BYTE** buffer, INT32* size)
6790 {
6791     TPM_RC result;
6792     result = UINT16_Unmarshal((UINT16*)&(target->t.size), buffer, size);
6793     if((result == TPM_RC_SUCCESS) && (target->t.size > sizeof(TPMS_CONTEXT_DATA)))
6794         result = TPM_RC_SIZE;
6795     if(result == TPM_RC_SUCCESS)
6796         result = BYTE_Array_Unmarshal(
6797             (BYTE*)&(target->t.buffer), buffer, size, (INT32)target->t.size);
6798     return result;
6799 }
6800
6801 UINT16
6802 TPM2B_CONTEXT_DATA_Marshal(TPM2B_CONTEXT_DATA* source, BYTE** buffer, INT32* size)
6803 {
6804     UINT16 result = 0;
6805     result =
6806         (UINT16)(result + UINT16_Marshal((UINT16*)&(source->t.size), buffer, size));
6807     // if size equal to 0, the rest of the structure is a zero buffer
6808     if(source->t.size == 0)
6809         return result;
6810     result = (UINT16)(result
6811                     + BYTE_Array_Marshal((BYTE*)&(source->t.buffer),
6812                                           buffer,
6813                                           size,
6814                                           (INT32)source->t.size));
6815     return result;
6816 }
6817
6818 // Table "Definition of TPMS_CONTEXT Structure" (Part 2: Structures)
6819 TPM_RC
6820 TPMS_CONTEXT_Unmarshal(TPMS_CONTEXT* target, BYTE** buffer, INT32* size)
6821 {
6822     TPM_RC result;
6823     result = UINT64_Unmarshal((UINT64*)&(target->sequence), buffer, size);
6824     if(result == TPM_RC_SUCCESS)
6825         result = TPMS_CONTEXT_Unmarshal(
6826             (TPMS_CONTEXT*)&(target->savedHandle), buffer, size);
6827     if(result == TPM_RC_SUCCESS)
6828         result = TPMS_CONTEXT_Unmarshal(
6829             (TPMS_CONTEXT*)&(target->hierarchy), buffer, size);
6830     if(result == TPM_RC_SUCCESS)
6831         result = TPM2B_CONTEXT_DATA_Unmarshal(
6832             (TPM2B_CONTEXT_DATA*)&(target->contextBlob), buffer, size);
6833     return result;
6834 }
6835
6836 TPM_RC
6837 TPMS_CONTEXT_Marshal(TPMS_CONTEXT* source, BYTE** buffer, INT32* size)
6838 {
6839     UINT16 result = 0;
6840     result =
6841         (UINT16)(result + UINT64_Marshal((UINT64*)&(source->sequence), buffer, size));
6842     result = (UINT16)(result
6843                     + TPM2B_CONTEXT_DATA_Marshal(
6844                         (TPM2B_CONTEXT_DATA*)&(source->contextBlob), buffer, size));

```



```

6843     result = (UINT16) (result
6844         + TPMI_RH_HIERARCHY_Marshal(
6845             (TPMI_RH_HIERARCHY*)&(source->hierarchy), buffer, size));
6846     result = (UINT16) (result
6847         + TPM2B_CONTEXT_DATA_Marshal(
6848             (TPM2B_CONTEXT_DATA*)&(source->contextBlob), buffer, size));
6849     return result;
6850 }
6851
6852 // Table "Definition of TPMS_CREATION_DATA Structure" (Part 2: Structures)
6853 UINT16
6854 TPMS_CREATION_DATA_Marshal(TPMS_CREATION_DATA* source, BYTE** buffer, INT32* size)
6855 {
6856     UINT16 result = 0;
6857     result = (UINT16) (result
6858         + TPML_PCR_SELECTION_Marshal(
6859             (TPML_PCR_SELECTION*)&(source->pcrSelect), buffer, size));
6860     result = (UINT16) (result
6861         + TPM2B_DIGEST_Marshal(
6862             (TPM2B_DIGEST*)&(source->pcrDigest), buffer, size));
6863     result = (UINT16) (result
6864         + TPMA_LOCALITY_Marshal(
6865             (TPMA_LOCALITY*)&(source->locality), buffer, size));
6866     result = (UINT16) (result
6867         + TPM_ALG_ID_Marshal(
6868             (TPM_ALG_ID*)&(source->parentNameAlg), buffer, size));
6869     result = (UINT16) (result
6870         + TPM2B_NAME_Marshal(
6871             (TPM2B_NAME*)&(source->parentName), buffer, size));
6872     result = (UINT16) (result
6873         + TPM2B_NAME_Marshal(
6874             (TPM2B_NAME*)&(source->parentQualifiedName), buffer, size));
6875     result = (UINT16) (result
6876         + TPM2B_DATA_Marshal(
6877             (TPM2B_DATA*)&(source->outsideInfo), buffer, size));
6878     return result;
6879 }
6880
6881 // Table "Definition of TPM2B_CREATION_DATA Structure" (Part 2: Structures)
6882 UINT16
6883 TPM2B_CREATION_DATA_Marshal(TPM2B_CREATION_DATA* source, BYTE** buffer, INT32* size)
6884 {
6885     UINT16 result = 0;
6886     // Marshal a dummy value of the 2B size. This makes sure that 'buffer'
6887     // and 'size' are advanced as necessary (i.e., if they are present)
6888     result = UINT16_Marshal(&result, buffer, size);
6889     // Marshal the structure
6890     result =
6891         (UINT16) (result
6892             + TPMS_CREATION_DATA_Marshal(
6893                 (TPMS_CREATION_DATA*)&(source->creationData), buffer, size));
6894     // if a buffer was provided, go back and fill in the actual size
6895     if(buffer != NULL)
6896         UINT16_TO_BYTE_ARRAY((result - 2), (*buffer - result));
6897     return result;
6898 }
6899
6900 // Table "Definition of TPM_AT Constants" (Part 2: Structures)
6901 TPM_RC
6902 TPM_AT_Unmarshal(TPM_AT* target, BYTE** buffer, INT32* size)
6903 {
6904     TPM_RC result;
6905     result = UINT32_Unmarshal((UINT32*)target, buffer, size);
6906     if(result == TPM_RC_SUCCESS)
6907     {
6908         switch(*target)

```

```

6909     {
6910         case TPM_AT_ANY:
6911         case TPM_AT_ERROR:
6912         case TPM_AT_PV1:
6913         case TPM_AT_VEND:
6914             break;
6915         default:
6916             result = TPM_RC_VALUE;
6917             break;
6918     }
6919 }
6920 return result;
6921 }
6922 # if !USE_MARSHALING_DEFINES
6923 UINT16
6924 TPM_AT_Marshal(TPM_AT* source, BYTE** buffer, INT32* size)
6925 {
6926     return UINT32_Marshal((UINT32*)source, buffer, size);
6927 }
6928 # endif // !USE_MARSHALING_DEFINES
6929
6930 // Table "Definition of TPM_AE Constants" (Part 2: Structures)
6931 # if !USE_MARSHALING_DEFINES
6932 UINT16
6933 TPM_AE_Marshal(TPM_AE* source, BYTE** buffer, INT32* size)
6934 {
6935     return UINT32_Marshal((UINT32*)source, buffer, size);
6936 }
6937 # endif // !USE_MARSHALING_DEFINES
6938
6939 // Table "Definition of TPMS_AC_OUTPUT Structure" (Part 2: Structures)
6940 UINT16
6941 TPMS_AC_OUTPUT_Marshal(TPMS_AC_OUTPUT* source, BYTE** buffer, INT32* size)
6942 {
6943     UINT16 result = 0;
6944     result = (UINT16)(result + TPM_AT_Marshal((TPM_AT*)&(source->tag), buffer, size));
6945     result =
6946         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->data), buffer, size));
6947     return result;
6948 }
6949
6950 // Table "Definition of TPML_AC_CAPABILITIES Structure" (Part 2: Structures)
6951 UINT16
6952 TPML_AC_CAPABILITIES_Marshal(TPML_AC_CAPABILITIES* source, BYTE** buffer, INT32* size)
6953 {
6954     UINT16 result = 0;
6955     result =
6956         (UINT16)(result + UINT32_Marshal((UINT32*)&(source->count), buffer, size));
6957     result = (UINT16)(result
6958         + TPMS_AC_OUTPUT_Array_Marshal(
6959             (TPMS_AC_OUTPUT*)&(source->acCapabilities),
6960             buffer,
6961             size,
6962             (INT32)source->count));
6963     return result;
6964 }
6965
6966 // For structures that unmarshals/marshals an array, the code calls an
6967 // un/marshaling function to process the array of the defined type.
6968 // This section contains the functions that perform that operation
6969 // Array Unmarshal/Marshal for BYTE
6970 TPM_RC
6971 BYTE_Array_Unmarshal(BYTE* target, BYTE** buffer, INT32* size, INT32 count)
6972 {
6973     if(*size < count)
6974         return TPM_RC_INSUFFICIENT;

```

```

6975     memcpy(target, *buffer, count);
6976     *size -= count;
6977     *buffer += count;
6978     return TPM_RC_SUCCESS;
6979 }
6980 UINT16
6981 BYTE_Array_Marshal(BYTE* source, BYTE** buffer, INT32* size, INT32 count)
6982 {
6983     if(buffer != 0)
6984     {
6985         if((size == 0) || ((*size -= count) >= 0))
6986         {
6987             memcpy(*buffer, source, count);
6988             *buffer += count;
6989         }
6990         pAssert((size == 0) || (*size >= 0));
6991     }
6992     pAssert(count < INT16_MAX);
6993     return ((UINT16)count);
6994 }
6995
6996 // Array Unmarshal and Marshal for TPM_ALG_ID
6997 TPM_RC
6998 TPM_ALG_ID_Array_Unmarshal(
6999     TPM_ALG_ID* target, BYTE** buffer, INT32* size, INT32 count)
7000 {
7001     TPM_RC result = TPM_RC_SUCCESS;
7002     INT32 i;
7003     for(i = 0; ((result == TPM_RC_SUCCESS) && (i < count)); i++)
7004     {
7005         result = TPM_ALG_ID_Unmarshal(&target[i], buffer, size);
7006     }
7007     return result;
7008 }
7009
7010 UINT16
7011 TPM_ALG_ID_Array_Marshal(TPM_ALG_ID* source, BYTE** buffer, INT32* size, INT32 count)
7012 {
7013     UINT16 result = 0;
7014     INT32 i;
7015     for(i = 0; i < count; i++)
7016     {
7017         result += TPM_ALG_ID_Marshal(&source[i], buffer, size);
7018     }
7019     return result;
7020 }
7021
7022 // Array Unmarshal and Marshal for TPM_CC
7023 TPM_RC
7024 TPM_CC_Array_Unmarshal(TPM_CC* target, BYTE** buffer, INT32* size, INT32 count)
7025 {
7026     TPM_RC result = TPM_RC_SUCCESS;
7027     INT32 i;
7028     for(i = 0; ((result == TPM_RC_SUCCESS) && (i < count)); i++)
7029     {
7030         result = TPM_CC_Unmarshal(&target[i], buffer, size);
7031     }
7032     return result;
7033 }
7034
7035 UINT16
7036 TPM_CC_Array_Marshal(TPM_CC* source, BYTE** buffer, INT32* size, INT32 count)
7037 {
7038     UINT16 result = 0;
7039     INT32 i;
7040     for(i = 0; i < count; i++)
7041     {
7042         result += TPM_CC_Marshal(&source[i], buffer, size);
7043     }

```

```

7041     }
7042     return result;
7043 }
7044
7045 // Array Marshal for TPM_ECC_CURVE
7046 # if ALG_ECC
7047 UINT16
7048 TPM_ECC_CURVE_Array_Marshal(
7049     TPM_ECC_CURVE* source, BYTE** buffer, INT32* size, INT32 count)
7050 {
7051     UINT16 result = 0;
7052     INT32 i;
7053     for(i = 0; i < count; i++)
7054     {
7055         result += TPM_ECC_CURVE_Marshal(&source[i], buffer, size);
7056     }
7057     return result;
7058 }
7059 # endif // ALG_ECC
7060
7061 // Array Marshal for TPM_HANDLE
7062 UINT16
7063 TPM_HANDLE_Array_Marshal(TPM_HANDLE* source, BYTE** buffer, INT32* size, INT32 count)
7064 {
7065     UINT16 result = 0;
7066     INT32 i;
7067     for(i = 0; i < count; i++)
7068     {
7069         result += TPM_HANDLE_Marshal(&source[i], buffer, size);
7070     }
7071     return result;
7072 }
7073
7074 // Array Unmarshal and Marshal for TPM2B_DIGEST
7075 TPM_RC
7076 TPM2B_DIGEST_Array_Unmarshal(
7077     TPM2B_DIGEST* target, BYTE** buffer, INT32* size, INT32 count)
7078 {
7079     TPM_RC result = TPM_RC_SUCCESS;
7080     INT32 i;
7081     for(i = 0; ((result == TPM_RC_SUCCESS) && (i < count)); i++)
7082     {
7083         result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
7084     }
7085     return result;
7086 }
7087
7088 TPM2B_DIGEST_Array_Marshal(
7089     TPM2B_DIGEST* source, BYTE** buffer, INT32* size, INT32 count)
7090 {
7091     UINT16 result = 0;
7092     INT32 i;
7093     for(i = 0; i < count; i++)
7094     {
7095         result += TPM2B_DIGEST_Marshal(&source[i], buffer, size);
7096     }
7097     return result;
7098 }
7099
7100 // Array Unmarshal and Marshal for TPM2B_VENDOR_PROPERTY
7101 TPM_RC
7102 TPM2B_VENDOR_PROPERTY_Array_Unmarshal(
7103     TPM2B_VENDOR_PROPERTY* target, BYTE** buffer, INT32* size, INT32 count)
7104 {
7105     TPM_RC result = TPM_RC_SUCCESS;
7106     INT32 i;

```

```

7107     for(i = 0; ((result == TPM_RC_SUCCESS) && (i < count)); i++)
7108     {
7109         result = TPM2B_VENDOR_PROPERTY_Unmarshal(&target[i], buffer, size);
7110     }
7111     return result;
7112 }
7113 UINT16
7114 TPM2B_VENDOR_PROPERTY_Array_Marshal(
7115     TPM2B_VENDOR_PROPERTY* source, BYTE** buffer, INT32* size, INT32 count)
7116 {
7117     UINT16 result = 0;
7118     INT32 i;
7119     for(i = 0; i < count; i++)
7120     {
7121         result += TPM2B_VENDOR_PROPERTY_Marshal(&source[i], buffer, size);
7122     }
7123     return result;
7124 }
7125
7126 // Array Marshal for TPMA_CC
7127 UINT16
7128 TPMA_CC_Array_Marshal(TPMA_CC* source, BYTE** buffer, INT32* size, INT32 count)
7129 {
7130     UINT16 result = 0;
7131     INT32 i;
7132     for(i = 0; i < count; i++)
7133     {
7134         result += TPMA_CC_Marshal(&source[i], buffer, size);
7135     }
7136     return result;
7137 }
7138
7139 // Array Marshal for TPMS_AC_OUTPUT
7140 UINT16
7141 TPMS_AC_OUTPUT_Array_Marshal(
7142     TPMS_AC_OUTPUT* source, BYTE** buffer, INT32* size, INT32 count)
7143 {
7144     UINT16 result = 0;
7145     INT32 i;
7146     for(i = 0; i < count; i++)
7147     {
7148         result += TPMS_AC_OUTPUT_Marshal(&source[i], buffer, size);
7149     }
7150     return result;
7151 }
7152
7153 // Array Marshal for TPMS_ACT_DATA
7154 UINT16
7155 TPMS_ACT_DATA_Array_Marshal(
7156     TPMS_ACT_DATA* source, BYTE** buffer, INT32* size, INT32 count)
7157 {
7158     UINT16 result = 0;
7159     INT32 i;
7160     for(i = 0; i < count; i++)
7161     {
7162         result += TPMS_ACT_DATA_Marshal(&source[i], buffer, size);
7163     }
7164     return result;
7165 }
7166
7167 // Array Marshal for TPMS_ALG_PROPERTY
7168 UINT16
7169 TPMS_ALG_PROPERTY_Array_Marshal(
7170     TPMS_ALG_PROPERTY* source, BYTE** buffer, INT32* size, INT32 count)
7171 {
7172     UINT16 result = 0;

```

```

7173     INT32 i;
7174     for(i = 0; i < count; i++)
7175     {
7176         result += TPMS_ALG_PROPERTY_Marshal(&source[i], buffer, size);
7177     }
7178     return result;
7179 }
7180
7181 // Array Unmarshal and Marshal for TPMS_PCR_SELECTION
7182 TPM_RC
7183 TPMS_PCR_SELECTION_Array_Unmarshal(
7184     TPMS_PCR_SELECTION* target, BYTE** buffer, INT32* size, INT32 count)
7185 {
7186     TPM_RC result = TPM_RC_SUCCESS;
7187     INT32 i;
7188     for(i = 0; ((result == TPM_RC_SUCCESS) && (i < count)); i++)
7189     {
7190         result = TPMS_PCR_SELECTION_Unmarshal(&target[i], buffer, size);
7191     }
7192     return result;
7193 }
7194
7195 UINT16
7196 TPMS_PCR_SELECTION_Array_Marshal(
7197     TPMS_PCR_SELECTION* source, BYTE** buffer, INT32* size, INT32 count)
7198 {
7199     UINT16 result = 0;
7200     INT32 i;
7201     for(i = 0; i < count; i++)
7202     {
7203         result += TPMS_PCR_SELECTION_Marshal(&source[i], buffer, size);
7204     }
7205     return result;
7206 }
7207
7208 // Array Marshal for TPMS_TAGGED_PCR_SELECT
7209 UINT16
7210 TPMS_TAGGED_PCR_SELECT_Array_Marshal(
7211     TPMS_TAGGED_PCR_SELECT* source, BYTE** buffer, INT32* size, INT32 count)
7212 {
7213     UINT16 result = 0;
7214     INT32 i;
7215     for(i = 0; i < count; i++)
7216     {
7217         result += TPMS_TAGGED_PCR_SELECT_Marshal(&source[i], buffer, size);
7218     }
7219     return result;
7220 }
7221
7222 // Array Marshal for TPMS_TAGGED_POLICY
7223 UINT16
7224 TPMS_TAGGED_POLICY_Array_Marshal(
7225     TPMS_TAGGED_POLICY* source, BYTE** buffer, INT32* size, INT32 count)
7226 {
7227     UINT16 result = 0;
7228     INT32 i;
7229     for(i = 0; i < count; i++)
7230     {
7231         result += TPMS_TAGGED_POLICY_Marshal(&source[i], buffer, size);
7232     }
7233     return result;
7234 }
7235
7236 // Array Marshal for TPMS_TAGGED_PROPERTY
7237 UINT16
7238 TPMS_TAGGED_PROPERTY_Array_Marshal(
7239     TPMS_TAGGED_PROPERTY* source, BYTE** buffer, INT32* size, INT32 count)

```



```

7239 {
7240     UINT16 result = 0;
7241     INT32 i;
7242     for(i = 0; i < count; i++)
7243     {
7244         result += TPMS_TAGGED_PROPERTY_Marshal(&source[i], buffer, size);
7245     }
7246     return result;
7247 }
7248
7249 // Array Unmarshal and Marshal for TPMT_HA
7250 TPM_RC
7251 TPMT_HA_Array_Unmarshal(
7252     TPMT_HA* target, BYTE** buffer, INT32* size, BOOL flag, INT32 count)
7253 {
7254     TPM_RC result = TPM_RC_SUCCESS;
7255     INT32 i;
7256     for(i = 0; ((result == TPM_RC_SUCCESS) && (i < count)); i++)
7257     {
7258         result = TPMT_HA_Unmarshal(&target[i], buffer, size, flag);
7259     }
7260     return result;
7261 }
7262
7263 UINT16
7264 TPMT_HA_Array_Marshal(TPMT_HA* source, BYTE** buffer, INT32* size, INT32 count)
7265 {
7266     UINT16 result = 0;
7267     INT32 i;
7268     for(i = 0; i < count; i++)
7269     {
7270         result += TPMT_HA_Marshal(&source[i], buffer, size);
7271     }
7272     return result;
7273 }
7274 #endif // !TABLE_DRIVEN_MARSHAL

```

7.190 /tpm/src/support/MathOnByteBuffers.c

```

1  /*** Introduction
2  /**
3  // This file contains implementation of the math functions that are performed
4  // with canonical integers in byte buffers. The canonical integer is
5  // big-endian bytes.
6  /**
7  #include "Tpm.h"
8  #include "TpmMath_Util_fp.h"
9
10 /*** Functions
11
12 /**** UnsignedCmpB
13 // This function compare two unsigned values. The values are byte-aligned,
14 // big-endian numbers (e.g, a hash).
15 // Return Type: int
16 //      1      if (a > b)
17 //      0      if (a = b)
18 //     -1      if (a < b)
19 LIB_EXPORT int UnsignedCompareB(UINT32      aSize, // IN: size of a
20                                const BYTE* a,    // IN: a
21                                UINT32      bSize, // IN: size of b
22                                const BYTE* b,    // IN: b
23 )
24 {
25     UINT32 i;
26     if(aSize > bSize)
27         return 1;

```

```

28     else if(aSize < bSize)
29         return -1;
30     else
31     {
32         for(i = 0; i < aSize; i++)
33         {
34             if(a[i] != b[i])
35                 return (a[i] > b[i]) ? 1 : -1;
36         }
37     }
38     // Will return == if sizes are both zero
39     return 0;
40 }
41
42 /***SignedCompareB()
43 // Compare two signed integers:
44 // Return Type: int
45 //     1       if a > b
46 //     0       if a = b
47 //    -1       if a < b
48 int SignedCompareB(const UINT32 aSize, // IN: size of a
49                   const BYTE* a,      // IN: a buffer
50                   const UINT32 bSize, // IN: size of b
51                   const BYTE* b,      // IN: b buffer
52 )
53 {
54     // are the signs different ?
55     if(((a[0] ^ b[0]) & 0x80) > 0)
56         // if the signs are different, then a is less than b if a is negative.
57         return a[0] & 0x80 ? -1 : 1;
58     else
59         // do unsigned compare function
60         return UnsignedCompareB(aSize, a, bSize, b);
61 }
62
63 #if ALG_RSA
64 /*** ModExpB
65 // This function is used to do modular exponentiation in support of RSA.
66 // The most typical uses are: 'c' = 'm'^'e' mod 'n' (RSA encrypt) and
67 // 'm' = 'c'^'d' mod 'n' (RSA decrypt). When doing decryption, the 'e' parameter
68 // of the function will contain the private exponent 'd' instead of the public
69 // exponent 'e'.
70 //
71 // If the results will not fit in the provided buffer,
72 // an error is returned (CRYPT_ERROR_UNDERFLOW). If the results is smaller
73 // than the buffer, the results is de-normalized.
74 //
75 // This version is intended for use with RSA and requires that 'm' be
76 // less than 'n'.
77 //
78 // Return Type: TPM_RC
79 //     TPM_RC_SIZE      number to exponentiate is larger than the modulus
80 //     TPM_RC_NO_RESULT result will not fit into the provided buffer
81 //
82 TPM_RC
83 ModExpB(UINT32 cSize, // IN: the size of the output buffer. It will
84         //          need to be the same size as the modulus
85         BYTE* c,      // OUT: the buffer to receive the results
86         //          (c->size must be set to the maximum size
87         //          for the returned value)
88         const UINT32 mSize,
89         const BYTE* m, // IN: number to exponentiate
90         const UINT32 eSize,
91         const BYTE* e, // IN: power
92         const UINT32 nSize,
93         const BYTE* n // IN: modulus

```

```

94  )
95  {
96      CRYPT_INT_MAX(bnC);
97      CRYPT_INT_MAX(bnM);
98      CRYPT_INT_MAX(bnE);
99      CRYPT_INT_MAX(bnN);
100     NUMBYTES tSize = (NUMBYTES)nSize;
101     TPM_RC   retVal = TPM_RC_SUCCESS;
102
103     // Convert input parameters
104     ExtMath_IntFromBytes(bnM, m, (NUMBYTES)mSize);
105     ExtMath_IntFromBytes(bnE, e, (NUMBYTES)eSize);
106     ExtMath_IntFromBytes(bnN, n, (NUMBYTES)nSize);
107
108     // Make sure that the output is big enough to hold the result
109     // and that 'm' is less than 'n' (the modulus)
110     if(cSize < nSize)
111         ERROR_EXIT(TPM_RC_NO_RESULT);
112     if(ExtMath_UnsignedCmp(bnM, bnN) >= 0)
113         ERROR_EXIT(TPM_RC_SIZE);
114     ExtMath_ModExp(bnC, bnM, bnE, bnN);
115     ExtMath_IntToBytes(bnC, c, &tSize);
116 Exit:
117     return retVal;
118 }
119 #endif // ALG_RSA
120
121 /*** DivideB()
122 // Divide an integer ('n') by an integer ('d') producing a quotient ('q') and
123 // a remainder ('r'). If 'q' or 'r' is not needed, then the pointer to them
124 // may be set to NULL.
125 //
126 // Return Type: TPM_RC
127 //      TPM_RC_NO_RESULT      'q' or 'r' is too small to receive the result
128 //
129 LIB_EXPORT TPM_RC DivideB(const TPM2B* n, // IN: numerator
130                          const TPM2B* d, // IN: denominator
131                          TPM2B* q, // OUT: quotient
132                          TPM2B* r // OUT: remainder
133 )
134 {
135     CRYPT_INT_MAX_INITIALIZED(bnN, n);
136     CRYPT_INT_MAX_INITIALIZED(bnD, d);
137     CRYPT_INT_MAX(bnQ);
138     CRYPT_INT_MAX(bnR);
139     //
140     // Do divide with converted values
141     ExtMath_Divide(bnQ, bnR, bnN, bnD);
142
143     // Convert the Crypt_Int* result back to 2B format using the size of the original
144     // number
145     if(q != NULL)
146         if(!TpmMath_IntTo2B(bnQ, q, q->size))
147             return TPM_RC_NO_RESULT;
148     if(r != NULL)
149         if(!TpmMath_IntTo2B(bnR, r, r->size))
150             return TPM_RC_NO_RESULT;
151     return TPM_RC_SUCCESS;
152 }
153
154 /*** AdjustNumberB()
155 // Remove/add leading zeros from a number in a TPM2B. Will try to make the number
156 // by adding or removing leading zeros. If the number is larger than the requested
157 // size, it will make the number as small as possible. Setting 'requestedSize' to
158 // zero is equivalent to requesting that the number be normalized.
159 UINT16

```

```

160 AdjustNumberB(TPM2B* num, UINT16 requestedSize)
161 {
162     BYTE* from;
163     UINT16 i;
164     // See if number is already the requested size
165     if(num->size == requestedSize)
166         return requestedSize;
167     from = num->buffer;
168     if(num->size > requestedSize)
169     {
170         // This is a request to shift the number to the left (remove leading zeros)
171         // Find the first non-zero byte. Don't look past the point where removing
172         // more zeros would make the number smaller than requested, and don't throw
173         // away any significant digits.
174         for(i = num->size; *from == 0 && i > requestedSize; from++, i--)
175             ;
176         if(i < num->size)
177         {
178             num->size = i;
179             MemoryCopy(num->buffer, from, i);
180         }
181     }
182     // This is a request to shift the number to the right (add leading zeros)
183     else
184     {
185         MemoryCopy(&num->buffer[requestedSize - num->size], num->buffer, num->size);
186         MemorySet(num->buffer, 0, requestedSize - num->size);
187         num->size = requestedSize;
188     }
189     return num->size;
190 }
191
192 /*** ShiftLeft()
193 // This function shifts a byte buffer (a TPM2B) one byte to the left. That is,
194 // the most significant bit of the most significant byte is lost.
195 TPM2B* ShiftLeft(TPM2B* value // IN/OUT: value to shift and shifted value out
196 )
197 {
198     UINT16 count = value->size;
199     BYTE* buffer = value->buffer;
200     if(count > 0)
201     {
202         for(count -= 1; count > 0; buffer++, count--)
203         {
204             buffer[0] = (buffer[0] << 1) + ((buffer[1] & 0x80) ? 1 : 0);
205         }
206         *buffer <<= 1;
207     }
208     return value;
209 }

```

7.191 /tpm/src/support/Memory.c

```

1  /*** Description
2  // This file contains a set of miscellaneous memory manipulation routines. Many
3  // of the functions have the same semantics as functions defined in string.h.
4  // Those functions are not used directly in the TPM because they are not 'safe'
5  //
6  // This version uses string.h after adding guards. This is because the math
7  // libraries invariably use those functions so it is not practical to prevent
8  // those library functions from being pulled into the build.
9
10 /*** Includes and Data Definitions
11 #include "Tpm.h"
12 #include "Memory_fp.h"

```

```

13
14 /** Functions
15
16 /** MemoryCopy()
17 // This is an alias for memmove. This is used in place of memcpy because
18 // some of the moves may overlap and rather than try to make sure that
19 // memmove is used when necessary, it is always used.
20 void MemoryCopy(void* dest, const void* src, int sSize)
21 {
22     if(dest != src)
23         memmove(dest, src, sSize);
24 }
25
26 /** MemoryEqual()
27 // This function indicates if two buffers have the same values in the indicated
28 // number of bytes.
29 // Return Type: BOOL
30 //     TRUE(1)      all octets are the same
31 //     FALSE(0)     all octets are not the same
32 BOOL MemoryEqual(const void* buffer1, // IN: compare buffer1
33                 const void* buffer2, // IN: compare buffer2
34                 unsigned int size    // IN: size of bytes being compared
35 )
36 {
37     BYTE equal = 0;
38     const BYTE* b1 = (BYTE*)buffer1;
39     const BYTE* b2 = (BYTE*)buffer2;
40     //
41     // Compare all bytes so that there is no leakage of information
42     // due to timing differences.
43     for(; size > 0; size--)
44         equal |= (*b1++ ^ *b2++);
45     return (equal == 0);
46 }
47
48 /** MemoryCopy2B()
49 // This function copies a TPM2B. This can be used when the TPM2B types are
50 // the same or different.
51 //
52 // This function returns the number of octets in the data buffer of the TPM2B.
53 LIB_EXPORT INT16 MemoryCopy2B(TPM2B* dest, // OUT: receiving TPM2B
54                               const TPM2B* source, // IN: source TPM2B
55                               unsigned int dSize // IN: size of the receiving buffer
56 )
57 {
58     pAssert(dest != NULL);
59     if(source == NULL)
60         dest->size = 0;
61     else
62     {
63         pAssert(source->size <= dSize);
64         MemoryCopy(dest->buffer, source->buffer, source->size);
65         dest->size = source->size;
66     }
67     return dest->size;
68 }
69
70 /** MemoryConcat2B()
71 // This function will concatenate the buffer contents of a TPM2B to
72 // the buffer contents of another TPM2B and adjust the size accordingly
73 // ('a' := ('a' | 'b')).
74 void MemoryConcat2B(
75     TPM2B* aInOut, // IN/OUT: destination 2B
76     TPM2B* bIn,    // IN: second 2B
77     unsigned int aMaxSize // IN: The size of aInOut.buffer (max values for
78                          // aInOut.size)

```

```

79 )
80 {
81     pAssert(bIn->size <= aMaxSize - aInOut->size);
82     MemoryCopy(&aInOut->buffer[aInOut->size], &bIn->buffer, bIn->size);
83     aInOut->size = aInOut->size + bIn->size;
84     return;
85 }
86
87 /*** MemoryEqual2B()
88 // This function will compare two TPM2B structures. To be equal, they
89 // need to be the same size and the buffer contexts need to be the same
90 // in all octets.
91 // Return Type: BOOL
92 //     TRUE(1)         size and buffer contents are the same
93 //     FALSE(0)        size or buffer contents are not the same
94 BOOL MemoryEqual2B(const TPM2B* aIn, // IN: compare value
95                   const TPM2B* bIn  // IN: compare value
96 )
97 {
98     if(aIn->size != bIn->size)
99         return FALSE;
100    return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
101 }
102
103 /*** MemorySet()
104 // This function will set all the octets in the specified memory range to
105 // the specified octet value.
106 // Note: A previous version had an additional parameter (dSize) that was
107 // intended to make sure that the destination would not be overrun. The
108 // problem is that, in use, all that was happening was that the value of
109 // size was used for dSize so there was no benefit in the extra parameter.
110 void MemorySet(void* dest, int value, size_t size)
111 {
112     memset(dest, value, size);
113 }
114
115 /*** MemoryPad2B()
116 // Function to pad a TPM2B with zeros and adjust the size.
117 void MemoryPad2B(TPM2B* b, UINT16 newSize)
118 {
119     MemorySet(&b->buffer[b->size], 0, newSize - b->size);
120     b->size = newSize;
121 }
122
123 /*** Uint16ToByteArray()
124 // Function to write an integer to a byte array
125 void Uint16ToByteArray(UINT16 i, BYTE* a)
126 {
127     a[1] = (BYTE) (i);
128     i >>= 8;
129     a[0] = (BYTE) (i);
130 }
131
132 /*** Uint32ToByteArray()
133 // Function to write an integer to a byte array
134 void Uint32ToByteArray(UINT32 i, BYTE* a)
135 {
136     a[3] = (BYTE) (i);
137     i >>= 8;
138     a[2] = (BYTE) (i);
139     i >>= 8;
140     a[1] = (BYTE) (i);
141     i >>= 8;
142     a[0] = (BYTE) (i);
143 }
144

```



```

145  /*** Uint64ToByteArray()
146  // Function to write an integer to a byte array
147  void Uint64ToByteArray(UINT64 i, BYTE* a)
148  {
149      a[7] = (BYTE) (i);
150      i >>= 8;
151      a[6] = (BYTE) (i);
152      i >>= 8;
153      a[5] = (BYTE) (i);
154      i >>= 8;
155      a[4] = (BYTE) (i);
156      i >>= 8;
157      a[3] = (BYTE) (i);
158      i >>= 8;
159      a[2] = (BYTE) (i);
160      i >>= 8;
161      a[1] = (BYTE) (i);
162      i >>= 8;
163      a[0] = (BYTE) (i);
164  }
165
166  /*** ByteArrayToUint8()
167  // Function to write a UINT8 to a byte array. This is included for completeness
168  // and to allow certain macro expansions
169  UINT8
170  ByteArrayToUint8(BYTE* a)
171  {
172      return *a;
173  }
174
175  /*** ByteArrayToUint16()
176  // Function to write an integer to a byte array
177  UINT16
178  ByteArrayToUint16(BYTE* a)
179  {
180      return ((UINT16)a[0] << 8) + a[1];
181  }
182
183  /*** ByteArrayToUint32()
184  // Function to write an integer to a byte array
185  UINT32
186  ByteArrayToUint32(BYTE* a)
187  {
188      return (UINT32)(((((UINT32)a[0] << 8) + a[1]) << 8) + (UINT32)a[2]) << 8) + a[3];
189  }
190
191  /*** ByteArrayToUint64()
192  // Function to write an integer to a byte array
193  UINT64
194  ByteArrayToUint64(BYTE* a)
195  {
196      return (((UINT64)BYTE_ARRAY_TO_UINT32(a)) << 32) + BYTE_ARRAY_TO_UINT32(&a[4]);
197  }

```

7.192 /tpm/src/support/Power.c

```

1  /*** Description
2
3  // This file contains functions that receive the simulated power state
4  // transitions of the TPM.
5
6  /*** Includes and Data Definitions
7  #define POWER_C
8  #include "Tpm.h"
9

```

```

10  /*** Functions
11
12  /*** TPMInit()
13  // This function is used to process a power on event.
14  void TPMInit(void)
15  {
16      // Set state as not initialized. This means that Startup is required
17      g_initialized = FALSE;
18      return;
19  }
20
21  /*** TPMRegisterStartup()
22  // This function registers the fact that the TPM has been initialized
23  // (a TPM2_Startup() has completed successfully).
24  BOOL TPMRegisterStartup(void)
25  {
26      g_initialized = TRUE;
27      return TRUE;
28  }
29
30  /*** TPMIsStarted()
31  // Indicates if the TPM has been initialized (a TPM2_Startup() has completed
32  // successfully after a _TPM_Init).
33  // Return Type: BOOL
34  //     TRUE(1)           TPM has been initialized
35  //     FALSE(0)          TPM has not been initialized
36  BOOL TPMIsStarted(void)
37  {
38      return g_initialized;
39  }

```

7.193 /tpm/src/support/PropertyCap.c

```

1  /*** Description
2  // This file contains the functions that are used for accessing the
3  // TPM_CAP_TPM_PROPERTY values.
4
5  /*** Includes
6
7  #include "Tpm.h"
8
9  /*** Functions
10
11  /*** TPMPropertyIsDefined()
12  // This function accepts a property selection and, if so, sets 'value'
13  // to the value of the property.
14  //
15  // All the fixed values are vendor dependent or determined by a
16  // platform-specific specification. The values in the table below
17  // are examples and should be changed by the vendor.
18  // Return Type: BOOL
19  //     TRUE(1)           referenced property exists and 'value' set
20  //     FALSE(0)          referenced property does not exist
21  static BOOL TPMPropertyIsDefined(TPM_PT property, // IN: property
22                                  UINT32* value    // OUT: property value
23  )
24  {
25      switch(property)
26      {
27          case TPM_PT_FAMILY_INDICATOR:
28              // from the title page of the specification
29              // For this specification, the value is "2.0".
30              *value = TPM_SPEC_FAMILY;
31              break;
32          case TPM_PT_LEVEL:

```

```

33         // from the title page of the specification
34         *value = TPM_SPEC_LEVEL;
35         break;
36     case TPM_PT_REVISION:
37         // from the title page of the specification
38         *value = TPM_SPEC_VERSION;
39         break;
40     case TPM_PT_DAY_OF_YEAR:
41         // computed from the date value on the title page of the specification
42         *value = TPM_SPEC_DAY_OF_YEAR;
43         break;
44     case TPM_PT_YEAR:
45         // from the title page of the specification
46         *value = TPM_SPEC_YEAR;
47         break;
48
49     case TPM_PT_MANUFACTURER:
50         // the vendor ID unique to each TPM manufacturer
51         *value = _plat__GetManufacturerCapabilityCode();
52         break;
53
54     case TPM_PT_VENDOR_STRING_1:
55         // the first four characters of the vendor ID string
56         *value = _plat__GetVendorCapabilityCode(1);
57         break;
58
59     case TPM_PT_VENDOR_STRING_2:
60         // the second four characters of the vendor ID string
61         *value = _plat__GetVendorCapabilityCode(2);
62         break;
63
64     case TPM_PT_VENDOR_STRING_3:
65         // the third four characters of the vendor ID string
66         *value = _plat__GetVendorCapabilityCode(3);
67         break;
68
69     case TPM_PT_VENDOR_STRING_4:
70         // the fourth four characters of the vendor ID string
71         *value = _plat__GetVendorCapabilityCode(4);
72         break;
73
74     case TPM_PT_VENDOR_TPM_TYPE:
75         // vendor-defined value indicating the TPM model
76         // We just make up a number here
77         *value = _plat__GetTpmType();
78         break;
79
80     case TPM_PT_FIRMWARE_VERSION_1:
81         // more significant 32-bits of a vendor-specific value
82         *value = gp.firmwareV1;
83         break;
84     case TPM_PT_FIRMWARE_VERSION_2:
85         // less significant 32-bits of a vendor-specific value
86         *value = gp.firmwareV2;
87         break;
88     case TPM_PT_INPUT_BUFFER:
89         // maximum size of TPM2B_MAX_BUFFER
90         *value = MAX_DIGEST_BUFFER;
91         break;
92     case TPM_PT_HR_TRANSIENT_MIN:
93         // minimum number of transient objects that can be held in TPM
94         // RAM
95         *value = MAX_LOADED_OBJECTS;
96         break;
97     case TPM_PT_HR_PERSISTENT_MIN:
98         // minimum number of persistent objects that can be held in

```

```

99         // TPM NV memory
100        // In this implementation, there is no minimum number of
101        // persistent objects.
102        *value = MIN_EVICT_OBJECTS;
103        break;
104    case TPM_PT_HR_LOADED_MIN:
105        // minimum number of authorization sessions that can be held in
106        // TPM RAM
107        *value = MAX_LOADED_SESSIONS;
108        break;
109    case TPM_PT_ACTIVE_SESSIONS_MAX:
110        // number of authorization sessions that may be active at a time
111        *value = MAX_ACTIVE_SESSIONS;
112        break;
113    case TPM_PT_PCR_COUNT:
114        // number of PCR implemented
115        *value = IMPLEMENTATION_PCR;
116        break;
117    case TPM_PT_PCR_SELECT_MIN:
118        // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
119        *value = PCR_SELECT_MIN;
120        break;
121    case TPM_PT_CONTEXT_GAP_MAX:
122        // maximum allowed difference (unsigned) between the contextID
123        // values of two saved session contexts
124        *value = ((UINT32)1 << (sizeof(CONTEXT_SLOT) * 8)) - 1;
125        break;
126    case TPM_PT_NV_COUNTERS_MAX:
127        // maximum number of NV indexes that are allowed to have the
128        // TPMA_NV_COUNTER attribute SET
129        // In this implementation, there is no limitation on the number
130        // of counters, except for the size of the NV Index memory.
131        *value = 0;
132        break;
133    case TPM_PT_NV_INDEX_MAX:
134        // maximum size of an NV index data area
135        *value = MAX_NV_INDEX_SIZE;
136        break;
137    case TPM_PT_MEMORY:
138        // a TPMA_MEMORY indicating the memory management method for the TPM
139        {
140            union
141            {
142                TPMA_MEMORY att;
143                UINT32      u32;
144            } attributes = {TPMA_ZERO_INITIALIZER()};
145            SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, sharedNV);
146            SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, objectCopiedToRam);
147
148            // Note: For a LSB0 machine, the bits in a bit field are in the
correct
149            // order even if the machine is MSB0. For a MSB0 machine, a TPMA will
150            // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
151            // be NO) so the bits are manipulate correctly.
152            *value = attributes.u32;
153            break;
154        }
155    case TPM_PT_CLOCK_UPDATE:
156        // interval, in seconds, between updates to the copy of
157        // TPMS_TIME_INFO .clock in NV
158        *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
159        break;
160    case TPM_PT_CONTEXT_HASH:
161        // algorithm used for the integrity hash on saved contexts and
162        // for digesting the fuData of TPM2_FirmwareRead()
163        *value = CONTEXT_INTEGRITY_HASH_ALG;

```

```

164         break;
165     case TPM_PT_CONTEXT_SYM:
166         // algorithm used for encryption of saved contexts
167         *value = CONTEXT_ENCRYPT_ALG;
168         break;
169     case TPM_PT_CONTEXT_SYM_SIZE:
170         // size of the key used for encryption of saved contexts
171         *value = CONTEXT_ENCRYPT_KEY_BITS;
172         break;
173     case TPM_PT_ORDERLY_COUNT:
174         // maximum difference between the volatile and non-volatile
175         // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
176         *value = MAX_ORDERLY_COUNT;
177         break;
178     case TPM_PT_MAX_COMMAND_SIZE:
179         // maximum value for 'commandSize'
180         *value = MAX_COMMAND_SIZE;
181         break;
182     case TPM_PT_MAX_RESPONSE_SIZE:
183         // maximum value for 'responseSize'
184         *value = MAX_RESPONSE_SIZE;
185         break;
186     case TPM_PT_MAX_DIGEST:
187         // maximum size of a digest that can be produced by the TPM
188         *value = sizeof(TPMU_HA);
189         break;
190     case TPM_PT_MAX_OBJECT_CONTEXT:
191         // Header has 'sequence', 'handle' and 'hierarchy'
192         #define SIZE_OF_CONTEXT_HEADER \
193             sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) + sizeof(TPMI_RH_HIERARCHY)
194         #define SIZE_OF_CONTEXT_INTEGRITY (sizeof(UINT16) + CONTEXT_INTEGRITY_HASH_SIZE)
195         #define SIZE_OF_FINGERPRINT      sizeof(UINT64)
196         #define SIZE_OF_CONTEXT_BLOB_OVERHEAD \
197             (sizeof(UINT16) + SIZE_OF_CONTEXT_INTEGRITY + SIZE_OF_FINGERPRINT)
198         #define SIZE_OF_CONTEXT_OVERHEAD \
199             (SIZE_OF_CONTEXT_HEADER + SIZE_OF_CONTEXT_BLOB_OVERHEAD)
200         #if 0
201             // maximum size of a TPMS_CONTEXT that will be returned by
202             // TPM2_ContextSave for object context
203             *value = 0;
204             // adding sequence, saved handle and hierarchy
205             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
206                 sizeof(TPMI_RH_HIERARCHY);
207             // add size field in TPM2B_CONTEXT
208             *value += sizeof(UINT16);
209             // add integrity hash size
210             *value += sizeof(UINT16) +
211                 CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
212             // Add fingerprint size, which is the same as sequence size
213             *value += sizeof(UINT64);
214             // Add OBJECT structure size
215             *value += sizeof(OBJECT);
216         #else
217             // the maximum size of a TPMS_CONTEXT that will be returned by
218             // TPM2_ContextSave for object context
219             *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(OBJECT);
220         #endif
221         break;
222     case TPM_PT_MAX_SESSION_CONTEXT:
223         #if 0
224             // the maximum size of a TPMS_CONTEXT that will be returned by
225             // TPM2_ContextSave for object context
226             *value = 0;
227             // adding sequence, saved handle and hierarchy
228             *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +

```

```

230         sizeof(TPMI_RH_HIERARCHY);
231         // Add size field in TPM2B_CONTEXT
232         *value += sizeof(UINT16);
233     // Add integrity hash size
234     *value += sizeof(UINT16) +
235         CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
236     // Add fingerprint size, which is the same as sequence size
237     *value += sizeof(UINT64);
238     // Add SESSION structure size
239     *value += sizeof(SESSION);
240 #else
241     // the maximum size of a TPMS_CONTEXT that will be returned by
242     // TPM2_ContextSave for object context
243     *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(SESSION);
244 #endif
245     break;
246 case TPM_PT_PS_FAMILY_INDICATOR:
247     // platform specific values for the TPM_PT_PS parameters from
248     // the relevant platform-specific specification
249     // In this reference implementation, all of these values are 0.
250     *value = PLATFORM_FAMILY;
251     break;
252 case TPM_PT_PS_LEVEL:
253     // level of the platform-specific specification
254     *value = PLATFORM_LEVEL;
255     break;
256 case TPM_PT_PS_REVISION:
257     // specification Revision times 100 for the platform-specific
258     // specification
259     *value = PLATFORM_VERSION;
260     break;
261 case TPM_PT_PS_DAY_OF_YEAR:
262     // platform-specific specification day of year using TCG calendar
263     *value = PLATFORM_DAY_OF_YEAR;
264     break;
265 case TPM_PT_PS_YEAR:
266     // platform-specific specification year using the CE
267     *value = PLATFORM_YEAR;
268     break;
269 case TPM_PT_SPLIT_MAX:
270     // number of split signing operations supported by the TPM
271     *value = 0;
272 #if ALG_ECC
273     *value = sizeof(gr.commitArray) * 8;
274 #endif
275     break;
276 case TPM_PT_TOTAL_COMMANDS:
277     // total number of commands implemented in the TPM
278     // Since the reference implementation does not have any
279     // vendor-defined commands, this will be the same as the
280     // number of library commands.
281     {
282 #if COMPRESSED_LISTS
283         (*value) = COMMAND_COUNT;
284 #else
285         COMMAND_INDEX commandIndex;
286         *value = 0;
287
288         // scan all implemented commands
289         for(commandIndex = GetClosestCommandIndex(0);
290             commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
291             commandIndex = GetNextCommandIndex(commandIndex))
292         {
293             (*value)++; // count of all implemented
294         }
295 #endif

```



```

296         break;
297     }
298     case TPM_PT_LIBRARY_COMMANDS:
299         // number of commands from the TPM library that are implemented
300         {
301             #if COMPRESSED_LISTS
302                 *value = LIBRARY_COMMAND_ARRAY_SIZE;
303             #else
304                 COMMAND_INDEX commandIndex;
305                 *value = 0;
306
307                 // scan all implemented commands
308                 for (commandIndex = GetClosestCommandIndex(0);
309                     commandIndex < LIBRARY_COMMAND_ARRAY_SIZE;
310                     commandIndex = GetNextCommandIndex(commandIndex))
311                 {
312                     (*value)++;
313                 }
314             #endif
315             break;
316         }
317     case TPM_PT_VENDOR_COMMANDS:
318         // number of vendor commands that are implemented
319         *value = VENDOR_COMMAND_ARRAY_SIZE;
320         break;
321     case TPM_PT_NV_BUFFER_MAX:
322         // Maximum data size in an NV write command
323         *value = MAX_NV_BUFFER_SIZE;
324         break;
325     case TPM_PT_MODES:
326     {
327         union
328         {
329             TPMA_MODES attr;
330             UINT32 u32;
331             } flags = {TPMA_ZERO_INITIALIZER()};
332     #if FIPS_COMPLIANT
333         SET_ATTRIBUTE(flags.attr, TPMA_MODES, FIPS_140_2);
334     #endif
335         *value = flags.u32;
336         break;
337     }
338     case TPM_PT_MAX_CAP_BUFFER:
339         *value = MAX_CAP_BUFFER;
340         break;
341     case TPM_PT_FIRMWARE_SVN:
342         *value = _plat_GetTpmFirmwareSvn();
343         break;
344     case TPM_PT_FIRMWARE_MAX_SVN:
345         *value = _plat_GetTpmFirmwareMaxSvn();
346         break;
347
348     // Start of variable commands
349     case TPM_PT_PERMANENT:
350         // TPMA_PERMANENT
351         {
352             union
353             {
354                 TPMA_PERMANENT attr;
355                 UINT32 u32;
356             } flags = {TPMA_ZERO_INITIALIZER()};
357             if (gp.ownerAuth.t.size != 0)
358                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, ownerAuthSet);
359             if (gp.endorsementAuth.t.size != 0)
360                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, endorsementAuthSet);
361             if (gp.lockoutAuth.t.size != 0)

```

```

362         SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, lockoutAuthSet);
363     if(gp.disableClear)
364         SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, disableClear);
365     if(gp.failedTries >= gp.maxTries)
366         SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, inLockout);
367     // In this implementation, EPS is always generated by TPM
368     SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, tpmGeneratedEPS);
369
370     // Note: For a LSb0 machine, the bits in a bit field are in the
correct
371     // order even if the machine is MSB0. For a MSb0 machine, a TPMA will
372     // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
373     // be NO) so the bits are manipulate correctly.
374     *value = flags.u32;
375     break;
376 }
377 case TPM_PT_STARTUP_CLEAR:
378     // TPMA_STARTUP_CLEAR
379     {
380         union
381         {
382             TPMA_STARTUP_CLEAR attr;
383             UINT32 u32;
384         } flags = {TPMA_ZERO_INITIALIZER()};
385         //
386         if(g_phEnable)
387             SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnable);
388         if(gc.shEnable)
389             SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, shEnable);
390         if(gc.ehEnable)
391             SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, ehEnable);
392         if(gc.phEnableNV)
393             SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnableNV);
394         if(g_prevOrderlyState != SU_NONE_VALUE)
395             SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, orderly);
396
397         // Note: For a LSb0 machine, the bits in a bit field are in the
correct
398         // order even if the machine is MSB0. For a MSb0 machine, a TPMA will
399         // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
400         // be NO) so the bits are manipulate correctly.
401         *value = flags.u32;
402         break;
403     }
404 case TPM_PT_HR_NV_INDEX:
405     // number of NV indexes currently defined
406     *value = NvCapGetIndexNumber();
407     break;
408 case TPM_PT_HR_LOADED:
409     // number of authorization sessions currently loaded into TPM
410     // RAM
411     *value = SessionCapGetLoadedNumber();
412     break;
413 case TPM_PT_HR_LOADED_AVAIL:
414     // number of additional authorization sessions, of any type,
415     // that could be loaded into TPM RAM
416     *value = SessionCapGetLoadedAvail();
417     break;
418 case TPM_PT_HR_ACTIVE:
419     // number of active authorization sessions currently being
420     // tracked by the TPM
421     *value = SessionCapGetActiveNumber();
422     break;
423 case TPM_PT_HR_ACTIVE_AVAIL:
424     // number of additional authorization sessions, of any type,
425     // that could be created

```

```

426         *value = SessionCapGetActiveAvail();
427         break;
428     case TPM_PT_HR_TRANSIENT_AVAIL:
429         // estimate of the number of additional transient objects that
430         // could be loaded into TPM RAM
431         *value = ObjectCapGetTransientAvail();
432         break;
433     case TPM_PT_HR_PERSISTENT:
434         // number of persistent objects currently loaded into TPM
435         // NV memory
436         *value = NvCapGetPersistentNumber();
437         break;
438     case TPM_PT_HR_PERSISTENT_AVAIL:
439         // number of additional persistent objects that could be loaded
440         // into NV memory
441         *value = NvCapGetPersistentAvail();
442         break;
443     case TPM_PT_NV_COUNTERS:
444         // number of defined NV indexes that have NV TPMA_NV_COUNTER
445         // attribute SET
446         *value = NvCapGetCounterNumber();
447         break;
448     case TPM_PT_NV_COUNTERS_AVAIL:
449         // number of additional NV indexes that can be defined with their
450         // TPMA_NV_COUNTER attribute SET
451         *value = NvCapGetCounterAvail();
452         break;
453     case TPM_PT_ALGORITHM_SET:
454         // region code for the TPM
455         *value = gp.algorithmSet;
456         break;
457     case TPM_PT_LOADED_CURVES:
458 #if ALG_ECC
459         // number of loaded ECC curves
460         *value = ECC_CURVE_COUNT;
461 #else // ALG_ECC
462         *value = 0;
463 #endif // ALG_ECC
464         break;
465     case TPM_PT_LOCKOUT_COUNTER:
466         // current value of the lockout counter
467         *value = gp.failedTries;
468         break;
469     case TPM_PT_MAX_AUTH_FAIL:
470         // number of authorization failures before DA lockout is invoked
471         *value = gp.maxTries;
472         break;
473     case TPM_PT_LOCKOUT_INTERVAL:
474         // number of seconds before the value reported by
475         // TPM_PT_LOCKOUT_COUNTER is decremented
476         *value = gp.recoveryTime;
477         break;
478     case TPM_PT_LOCKOUT_RECOVERY:
479         // number of seconds after a lockoutAuth failure before use of
480         // lockoutAuth may be attempted again
481         *value = gp.lockoutRecovery;
482         break;
483     case TPM_PT_NV_WRITE_RECOVERY:
484         // number of milliseconds before the TPM will accept another command
485         // that will modify NV.
486         // This should make a call to the platform code that is doing rate
487         // limiting of NV. Rate limiting is not implemented in the reference
488         // code so no call is made.
489         *value = 0;
490         break;
491     case TPM_PT_AUDIT_COUNTER_0:

```

```

492         // high-order 32 bits of the command audit counter
493         *value = (UINT32) (gp.auditCounter >> 32);
494         break;
495     case TPM_PT_AUDIT_COUNTER_1:
496         // low-order 32 bits of the command audit counter
497         *value = (UINT32) (gp.auditCounter);
498         break;
499     default:
500         // property is not defined
501         return FALSE;
502         break;
503 }
504 return TRUE;
505 }
506
507 /*** TPMCapGetProperties()
508 // This function is used to get the TPM_PT values. The search of properties will
509 // start at 'property' and continue until 'propertyList' has as many values as
510 // will fit, or the last property has been reported, or the list has as many
511 // values as requested in 'count'.
512 // Return Type: TPMT_YES_NO
513 // YES         more properties are available
514 // NO          no more properties to be reported
515 TPMT_YES_NO
516 TPMCapGetProperties(TPM_PT property, // IN: the starting TPM property
517                    UINT32 count,    // IN: maximum number of returned
518                               // properties
519                    TPML_TAGGED_TPM_PROPERTY* propertyList // OUT: property list
520 )
521 {
522     TPMT_YES_NO more = NO;
523     UINT32 i;
524     UINT32 nextGroup;
525
526     // initialize output property list
527     propertyList->count = 0;
528
529     // maximum count of properties we may return is MAX_PCR_PROPERTIES
530     if(count > MAX_TPM_PROPERTIES)
531         count = MAX_TPM_PROPERTIES;
532
533     // if property is less than PT_FIXED, start from PT_FIXED
534     if(property < PT_FIXED)
535         property = PT_FIXED;
536     // There is only the fixed and variable groups with the variable group coming
537     // last
538     if(property >= (PT_VAR + PT_GROUP))
539         return more;
540
541     // Don't read past the end of the selected group
542     nextGroup = ((property / PT_GROUP) * PT_GROUP) + PT_GROUP;
543
544     // Scan through the TPM properties of the requested group.
545     for(i = property; i < nextGroup; i++)
546     {
547         UINT32 value;
548         // if we have hit the end of the group, quit
549         if(i != property && ((i % PT_GROUP) == 0))
550             break;
551         if(TPMPropertyIsDefined((TPM_PT)i, &value))
552         {
553             if(propertyList->count < count)
554             {
555                 // If the list is not full, add this property
556                 propertyList->tpmProperty[propertyList->count].property = (TPM_PT)i;
557                 propertyList->tpmProperty[propertyList->count].value = value;

```

```

558         propertyList->count++;
559     }
560     else
561     {
562         // If the return list is full but there are more properties
563         // available, set the indication and exit the loop.
564         more = YES;
565         break;
566     }
567 }
568 }
569 return more;
570 }
571
572 /*** TPMCapGetOneProperty()
573 // This function returns a single TPM property, if present.
574 BOOL TPMCapGetOneProperty(TPM_PT pt, // IN: the TPM property
575                           TPMS_TAGGED_PROPERTY* property // OUT: tagged property
576 )
577 {
578     UINT32 value;
579
580     if(TPMPropertyIsDefined((TPM_PT)pt, &value))
581     {
582         property->property = (TPM_PT)pt;
583         property->value     = value;
584         return TRUE;
585     }
586
587     return FALSE;
588 }

```

7.194 /tpm/src/support/Response.c

```

1  /*** Description
2  // This file contains the common code for building a response header, including
3  // setting the size of the structure. 'command' may be NULL if result is
4  // not TPM_RC_SUCCESS.
5
6  /*** Includes and Defines
7  #include "Tpm.h"
8  #include "Marshal.h"
9
10 /*** BuildResponseHeader()
11 // Adds the response header to the response. It will update command->parameterSize
12 // to indicate the total size of the response.
13 void BuildResponseHeader(COMMAND* command, // IN: main control structure
14                          BYTE* buffer,    // OUT: the output buffer
15                          TPM_RC result    // IN: the response code
16 )
17 {
18     TPM_ST tag;
19     UINT32 size;
20
21     if(result != TPM_RC_SUCCESS)
22     {
23         tag = TPM_ST_NO_SESSIONS;
24         size = 10;
25     }
26     else
27     {
28         tag = command->tag;
29         // Compute the overall size of the response
30         size = STD_RESPONSE_HEADER + command->handleNum * sizeof(TPM_HANDLE);
31         size += command->parameterSize;

```

```

32         size += (command->tag == TPM_ST_SESSIONS) ? command->authSize + sizeof(UINT32)
33               : 0;
34     }
35     TPM_ST_Marshal(&tag, &buffer, NULL);
36     UINT32_Marshal(&size, &buffer, NULL);
37     TPM_RC_Marshal(&result, &buffer, NULL);
38     if(result == TPM_RC_SUCCESS)
39     {
40         if(command->handleNum > 0)
41             TPM_HANDLE_Marshal(&command->handles[0], &buffer, NULL);
42         if(tag == TPM_ST_SESSIONS)
43             UINT32_Marshal((UINT32*)&command->parameterSize, &buffer, NULL);
44     }
45     command->parameterSize = size;
46 }

```

7.195 /tpm/src/support/ResponseCodeProcessing.c

```

1  /** Description
2  // This file contains the miscellaneous functions for processing response codes.
3  // NOTE: Currently, there is only one.
4
5  /** Includes and Defines
6  #include "Tpm.h"
7
8  /** RcSafeAddToResult()
9  // Adds a modifier to a response code as long as the response code allows a modifier
10 // and no modifier has already been added.
11 TPM_RC
12 RcSafeAddToResult(TPM_RC responseCode, TPM_RC modifier)
13 {
14     if((responseCode & RC_FMT1) && !(responseCode & 0xf40))
15         return responseCode + modifier;
16     else
17         return responseCode;
18 }

```

7.196 /tpm/src/support/TableDrivenMarshal.c

```

1  #include <assert.h>
2
3  #include "Tpm.h"
4  #include "Marshal.h"
5  #include "TableMarshal.h"
6
7  #if TABLE_DRIVEN_MARSHAL
8
9  extern ArrayMarshal_mst ArrayLookupTable[];
10
11 extern UINT16          MarshalLookupTable[];
12
13 typedef struct
14 {
15     int a;
16 } External_Structure_t;
17
18 extern struct External_Structure_t MarshalData;
19
20 # define IS_SUCCESS(UNMARSHAL_FUNCTION) \
21     (TPM_RC_SUCCESS == (result = (UNMARSHAL_FUNCTION)))
22
23 marshalIndex_t IntegerDispatch[] = {UINT8_MARSHAL_REF,
24                                     UINT16_MARSHAL_REF,
25                                     UINT32_MARSHAL_REF,

```



```

26             UINT64_MARSHAL_REF,
27             INT8_MARSHAL_REF,
28             INT16_MARSHAL_REF,
29             INT32_MARSHAL_REF,
30             INT64_MARSHAL_REF};
31
32 # if 1
33 #   define GetDescriptor(reference) \
34     ((MarshalHeader_mst*)((BYTE*)&MarshalData) + (reference & NULL_MASK))
35 # else
36 static const MarshalHeader_mst* GetDescriptor(marshalIndex_t index)
37 {
38     const MarshalHeader_mst* mst = MarshalLookupTable[index & NULL_MASK];
39     return mst;
40 }
41 # endif
42
43 # define GetUnionDescriptor(_index_) ((UnionMarshal_mst*)GetDescriptor(_index_))
44 # define GetArrayDescriptor(_index_) \
45     ((ArrayMarshal_mst*)(ArrayLookupTable[_index_] & NULL_MASK))
46
47 /*** GetUnmarshaledInteger()
48 // Gets the unmarshaled value and normalizes it to a UIN32 for other
49 // processing (comparisons and such).
50 static UINT32 GetUnmarshaledInteger(marshalIndex_t type, const void* target)
51 {
52     int size = (type & SIZE_MASK);
53     //
54     if(size == FOUR_BYTES)
55         return *((UINT32*)target);
56     if(type & IS_SIGNED)
57     {
58         if(size == TWO_BYTES)
59             return (UINT32) * ((int16_t*)target);
60         return (UINT32) * ((int8_t*)target);
61     }
62     if(size == TWO_BYTES)
63         return (UINT32) * ((UINT16*)target);
64     return (UINT32) * ((UINT8*)target);
65 }
66
67 static UINT32 GetSelector(void* structure, const UINT16* values, UINT16 descriptor)
68 {
69     unsigned sel = GET_ELEMENT_NUMBER(descriptor);
70     // Get the offset of the value in the unmarshaled structure
71     const UINT16* entry = &values[(sel * 3)];
72     //
73     return GetUnmarshaledInteger(GET_ELEMENT_SIZE(entry[0]),
74                                 ((UINT8*)structure) + entry[2]);
75 }
76
77 static TPM_RC UnmarshalBytes(
78     UINT8* target, // IN/OUT: place to put the bytes
79     UINT8** buffer, // IN/OUT: source of the input data
80     INT32* size, // IN/OUT: remaining bytes in the input buffer
81     int count // IN: number of bytes to get
82 )
83 {
84     if((*size -- count) >= 0)
85     {
86         memcpy(target, *buffer, count);
87         *buffer += count;
88         return TPM_RC_SUCCESS;
89     }
90     return TPM_RC_INSUFFICIENT;
91 }

```

```

92
93 /*** MarshalBytes()
94 // Marshal an array of bytes.
95 static UINT16 MarshalBytes(UINT8* source, UINT8** buffer, INT32* size, int32_t count)
96 {
97     if(buffer != NULL)
98     {
99         if(size != NULL && (size -= count) < 0)
100             return 0;
101         memcpy(*buffer, source, count);
102         *buffer += count;
103     }
104     return (UINT16)count;
105 }
106
107 /*** ArrayUnmarshal()
108 // Unmarshal an array. The 'index' is of the form: 'type'_ARRAY_MARSHAL_INDEX.
109 static TPM_RC ArrayUnmarshal(UINT16 index, // IN: the type of the array
110                             UINT8* target, // IN: target for the data
111                             UINT8** buffer, // IN/OUT: place to get the data
112                             INT32* size, // IN/OUT: remaining unmarshal data
113                             UINT32 count // IN: number of values of 'index' to
114                                         // unmarshal
115 )
116 {
117     marshalIndex_t which = ArrayLookupTable[index & NULL_MASK].type;
118     UINT16 stride = ArrayLookupTable[index & NULL_MASK].stride;
119     TPM_RC result;
120     //
121     if(stride == 1) // A byte array
122         result = UnmarshalBytes(target, buffer, size, count);
123     else
124     {
125         which |= index & NULL_FLAG;
126         for(result = TPM_RC_SUCCESS; count > 0; target += stride, count--)
127             if(!IS_SUCCESS(Unmarshal(which, target, buffer, size)))
128                 break;
129     }
130     return result;
131 }
132
133 /*** ArrayMarshal()
134 static UINT16 ArrayMarshal(UINT16 index, // IN: the type of the array
135                           UINT8* source, // IN: source of the data
136                           UINT8** buffer, // IN/OUT: place to put the data
137                           INT32* size, // IN/OUT: amount of space for the data
138                           UINT32 count // IN: number of values of 'index' to marshal
139 )
140 {
141     marshalIndex_t which = ArrayLookupTable[index & NULL_MASK].type;
142     UINT16 stride = ArrayLookupTable[index & NULL_MASK].stride;
143     UINT16 retVal;
144     //
145     if(stride == 1) // A byte array
146         return MarshalBytes(source, buffer, size, count);
147     which |= index & NULL_FLAG;
148     for(retVal = 0; count > 0; source += stride, count--)
149         retVal += Marshal(which, source, buffer, size);
150
151     return retVal;
152 }
153
154 /***UnmarshalUnion()
155 TPM_RC
156 UnmarshalUnion(UINT16 typeIndex, // IN: the thing to unmarshal
157                void* target, // IN: where the data goes to

```

```

158         UINT8** buffer,      // IN/OUT: the data source buffer
159         INT32*  size,        // IN/OUT: the remaining size
160         UINT32  selector)
161 {
162     int i;
163     UnionMarshal_mst* ut = GetUnionDescriptor(typeIndex);
164     marshalIndex_t selected;
165     //
166     for(i = 0; i < ut->countOfselectors; i++)
167     {
168         if(selector == ut->selectors[i])
169         {
170             UINT8* offset = ((UINT8*)ut) + ut->offsetOfUnmarshalTypes;
171             // Get the selected thing to unmarshal
172             selected = ((marshalIndex_t*)offset)[i];
173             if(ut->modifiers & IS_ARRAY_UNION)
174                 return UnmarshalBytes(target, buffer, size, selected);
175             else
176             {
177                 // Propagate NULL_FLAG if the null flag was
178                 // propagated to the structure containing the union
179                 selected |= (typeIndex & NULL_FLAG);
180                 return Unmarshal(selected, target, buffer, size);
181             }
182         }
183     }
184     // Didn't find the value.
185     return TPM_RC_SELECTOR;
186 }
187
188 /*** MarshalUnion()
189  *
190  * MarshalUnion(UINT16 typeIndex, // IN: the thing to marshal
191  *               void* source,    // IN: where the data comes from
192  *               UINT8** buffer,  // IN/OUT: the data source buffer
193  *               INT32* size,     // IN/OUT: the remaining size
194  *               UINT32 selector // IN: the union selector
195  * )
196  *
197  *
198  *
199  *
200  *
201  *
202  *
203  *
204  *
205  *
206  *
207  *
208  *
209  *
210  *
211  *
212  *
213  *
214  *
215  *
216  *
217  *
218  *
219  *
220  *
221  *
222  *
223  *

```

```

224         UINT32* value    // OUT: optional copy of 'target'
225     )
226 {
227     // This is just to save typing
228     # define _MB_ (*buffer)
229     // The size is a power of two so convert to regular integer
230     int bytes = (1 << (iSize & SIZE_MASK));
231     //
232     // Check to see if there is enough data to fulfill the request
233     if((*size -= bytes) >= 0)
234     {
235         // The most common size
236         if(bytes == 4)
237         {
238             *((UINT32*)target) =
239                 (UINT32) ((((_MB_[0] << 8) | _MB_[1]) << 8) | _MB_[2]) << 8)
240                     | _MB_[3]);
241             // If a copy is needed, copy it.
242             if(value != NULL)
243                 *value = *((UINT32*)target);
244         }
245         else if(bytes == 2)
246         {
247             *((UINT16*)target) = (UINT16) ((_MB_[0] << 8) | _MB_[1]);
248             // If a copy is needed, copy with the appropriate sign extension
249             if(value != NULL)
250             {
251                 if(iSize & IS_SIGNED)
252                     *value = (UINT32) *((INT16*)target);
253                 else
254                     *value = (UINT32) *((UINT16*)target);
255             }
256         }
257         else if(bytes == 1)
258         {
259             *((UINT8*)target) = (UINT8) _MB_[0];
260             // If a copy is needed, copy with the appropriate sign extension
261             if(value != NULL)
262             {
263                 if(iSize & IS_SIGNED)
264                     *value = (UINT32) *((INT8*)target);
265                 else
266                     *value = (UINT32) *((UINT8*)target);
267             }
268         }
269         else
270         {
271             // There is no input type that is a 64-bit value other than a UINT64. So
272             // there is no reason to do anything other than unmarshal it.
273             *((UINT64*)target) = BYTE_ARRAY_TO_UINT64(*buffer);
274         }
275         *buffer += bytes;
276         return TPM_RC_SUCCESS;
277     # undef _MB_
278     }
279     return TPM_RC_INSUFFICIENT;
280 }
281
282 /*** Unmarshal()
283 // This is the function that performs unmarshaling of different numbered types. Each
284 // TPM type has a number. The number is used to lookup the address of the data
285 // structure that describes how to unmarshal that data type.
286 //
287 TPM_RC
288 Unmarshal(UINT16 typeIndex, // IN: the thing to marshal
289           void* target,    // IN: where the data goes from

```

```

290     UINT8** buffer,      // IN/OUT: the data source buffer
291     INT32*  size         // IN/OUT: the remaining size
292 )
293 {
294     const MarshalHeader_mst* sel;
295     TPM_RC                    result;
296     //
297     sel = GetDescriptor(typeIndex);
298     switch(sel->marshalType)
299     {
300     case UINT_MTYPE:
301     {
302         // A simple signed or unsigned integer value.
303         return UnmarshalInteger(sel->modifiers, target, buffer, size, NULL);
304     }
305     case VALUES_MTYPE:
306     {
307         // This is the general-purpose structure that can handle things like
308         // TPMI_DH_PARENT that has multiple ranges, multiple singles and a
309         // 'null' value. When things cover a large range with holes in the range
310         // they can be turned into multiple ranges. There is no option for a bit
311         // field.
312         // The structure is:
313         // typedef const struct ValuesMarshal_mst
314         // {
315         //     UINT8      marshalType;          // VALUES_MTYPE
316         //     UINT8      modifiers;
317         //     UINT8      errorCode;
318         //     UINT8      ranges;
319         //     UINT8      singles;
320         //     UINT32     values[1];
321         // } ValuesMarshal_mst;
322         // Unmarshal the base type
323         UINT32 val;
324         if(IS_SUCCESS(
325             UnmarshalInteger(sel->modifiers, target, buffer, size, &val)))
326         {
327             ValuesMarshal_mst* vmt = ((ValuesMarshal_mst*)sel);
328             const UINT32* check = vmt->values;
329             //
330             // if the TAKES_NULL flag is set, then the first entry in the values
331             // list is the NULL value. It is not included in the 'ranges' or
332             // 'singles' count.
333             if((vmt->modifiers & TAKES_NULL) && (val == *check++))
334             {
335                 if((typeIndex & NULL_FLAG) == 0)
336                     result = (TPM_RC)(sel->errorCode);
337             }
338             // No NULL value or input is not the NULL value
339             else
340             {
341                 int i;
342                 //
343                 // Check all the min-max ranges.
344                 for(i = vmt->ranges - 1; i >= 0; check = &check[2], i--)
345                     if((UINT32)(val - check[0]) <= check[1])
346                         break;
347                 // if the input is in a selected range, then i >= 0
348                 if(i < 0)
349                 {
350                     // Not in any range, so check sigles
351                     for(i = vmt->singles - 1; i >= 0; i--)
352                         if(val == check[i])
353                             break;
354                 }
355                 // If input not in range and not in any single so return error

```

```

356         if(i < 0)
357             result = (TPM_RC) (sel->errorCode);
358     }
359 }
360 break;
361 }
362 case TABLE_MTYPE:
363 {
364     // This is a table with or without bit checking. The input is checked
365     // against each value in the table. If the value is in the table, and
366     // a bits table is present, then the bit field is checked to see if the
367     // indicated value is implemented. For example, if there is a table of
368     // allowed RSA key sizes and the 2nd entry matches, then the 2nd bit in
369     // the bit field is checked to see if that allowed size is implemented
370     // in this TPM.
371     // typedef const struct TableMarshal_mst
372     // {
373     //     UINT8      marshalType;        // TABLE_MTYPE
374     //     UINT8      modifiers;
375     //     UINT8      errorCode;
376     //     UINT8      singles;
377     //     UINT32     values[singles + 1 if TAKES_NULL];
378     // } TableMarshal_mst;
379
380     UINT32 val;
381     //
382     // Unmarshal the base type
383     if(IS_SUCCESS(
384         UnmarshalInteger(sel->modifiers, target, buffer, size, &val)))
385     {
386         TableMarshal_mst* tmt = ((TableMarshal_mst*)sel);
387         const UINT32* check = tmt->values;
388         //
389         // If this type has a null value, then it is the first value in the
390         // list of values. It does not count in the count of values
391         if((tmt->modifiers & TAKES_NULL) && (val == *check++))
392         {
393             if((typeIndex & NULL_FLAG) == 0)
394                 result = (TPM_RC) (sel->errorCode);
395         }
396         else
397         {
398             int i;
399             //
400             // Process the singles
401             for(i = tmt->singles - 1; i >= 0; i--)
402             {
403                 // does the input value match the value in the table
404                 if(val == check[i])
405                 {
406                     // If there is an associated bit table, make sure that
407                     // the corresponding bit is SET
408                     if((HAS_BITS & tmt->modifiers)
409                         && (!IS_BIT_SET32(i, &(check[tmt->singles]))))
410                         // if not SET, then this is a failure.
411                         i = -1;
412                     break;
413                 }
414             }
415             // error if not found or bit not SET
416             if(i < 0)
417                 result = (TPM_RC) (sel->errorCode);
418         }
419     }
420     break;
421 }

```



```

422     case MIN_MAX_MTYPE:
423     {
424         // A MIN_MAX is a range. It can have a bit field and a NULL value that is
425         // outside of the range. If the input value is in the min-max range then
426         // it is valid unless there is an associated bit field. Otherwise, it
427         // it is only valid if the corresponding value in the bit field is SET.
428         // The min value is 'values[0]' or 'values[1]' if there is a NULL value.
429         // The max value is the value after min. The max value is in the table as
430         // max minus min. This allows 'val' to be subtracted from min and then
431         // checked against max with one unsigned comparison. If present, the bit
432         // field will be the first 'values' after max.
433         // typedef const struct MinMaxMarshal_mst
434         // {
435         //     UINT8      marshalType;        // MIN_MAX_MTYPE
436         //     UINT8      modifiers;
437         //     UINT8      errorCode;
438         //     UINT32     values[2 + 1 if TAKES_NULL];
439         // } MinMaxMarshal_mst;
440         UINT32 val;
441         //
442         // A min-max has a range. It can have a bit-field that is indexed to the
443         // min value (something that matches min has a bit at 0. This is useful
444         // for algorithms. The min-max define a range of algorithms to be checked
445         // and the bit field can check to see if the algorithm in that range is
446         // allowed.
447         if(IS_SUCCESS(
448             UnmarshalInteger(sel->modifiers, target, buffer, size, &val)))
449         {
450             MinMaxMarshal_mst* mmt = (MinMaxMarshal_mst*)sel;
451             const UINT32* check = mmt->values;
452             //
453             // If this type takes a NULL, see if it matches. This
454             if((mmt->modifiers & TAKES_NULL) && (val == *check++))
455             {
456                 if((typeIndex & NULL_FLAG) == 0)
457                     result = (TPM_RC) (mmt->errorCode);
458             }
459             else
460             {
461                 val -= *check;
462                 if((val > check[1])
463                     || ((mmt->modifiers & HAS_BITS)
464                         && !IS_BIT_SET32(val, &check[2])))
465                     result = (TPM_RC) (mmt->errorCode);
466             }
467         }
468         break;
469     }
470     case ATTRIBUTES_MTYPE:
471     {
472         // This is used for TPMA values.
473         UINT32 mask;
474         AttributesMarshal_mst* amt = (AttributesMarshal_mst*)sel;
475         //
476         if(IS_SUCCESS(
477             UnmarshalInteger(sel->modifiers, target, buffer, size, &mask)))
478         {
479             if((mask & amt->attributeMask) != 0)
480                 result = TPM_RC_RESERVED_BITS;
481         }
482         break;
483     }
484     case STRUCTURE_MTYPE:
485     {
486         // A structure (not a union). A structure has elements (one defined per
487         // row). Three UINT16 values are used for each row. The first indicates

```

```

488 // the type of the entry. The choices are: simple, union, or array. A
489 // simple type can be a simple integer or another structure. It can also
490 // be a specific "interface type." For example, when a structure entry is
491 // a value that is used define the dimension of an array, the entry of
492 // the structure will reference a "synthetic" interface type, most often
493 // a min-max value. If the type of the entry is union or array, then the
494 // first value indicates which of the previous elements provides the union
495 // selector or the array dimension. That previous entry is referenced in
496 // the unmarshaled structure in memory (Not the marshaled buffer). The
497 // previous entry indicates the location in the structure of the value.
498 // The second entry of each structure entry indicated the index of the
499 // type associated with the entry. This is an index into the array of
500 // arrays or the union table (merged with the normal table in this
501 // implementation). The final entry is the offset in the unmarshaled
502 // structure where the value is located. This is the offsetof(STRUCTURE,
503 // element). This value is added to the input 'target' or 'source' value
504 // to determine where the value goes.
505 StructMarshal_mst* mst = (StructMarshal_mst*)sel;
506 int i;
507 const UINT16* value = mst->values;
508 //
509 for(result = TPM_RC_SUCCESS, i = mst->elements;
510 (TPM_RC_SUCCESS == result) && (i > 0);
511 value = &value[3], i--)
512 {
513     UINT16 descriptor = value[0];
514     marshalIndex_t index = value[1];
515     // The offset of the object in the structure is in the last value in
516     // the triplet. Add that value to the start of the structure
517     UINT8* offset = ((UINT8*)target) + value[2];
518     //
519     if((ELEMENT_PROPAGATE & descriptor) && (typeIndex & NULL_FLAG))
520         index |= NULL_FLAG;
521     switch(GET_ELEMENT_TYPE(descriptor))
522     {
523     case SIMPLE_TYPE:
524     {
525         result = Unmarshal(index, offset, buffer, size);
526         break;
527     }
528     case UNION_TYPE:
529     {
530         UINT32 choice;
531         //
532         // Get the selector or array dimension value
533         choice = GetSelector(target, mst->values, descriptor);
534         result = UnmarshalUnion(index, offset, buffer, size, choice);
535         break;
536     }
537     case ARRAY_TYPE:
538     {
539         UINT32 dimension;
540         //
541         dimension = GetSelector(target, mst->values, descriptor);
542         result =
543             ArrayUnmarshal(index, offset, buffer, size, dimension);
544         break;
545     }
546     default:
547         result = TPM_RC_FAILURE;
548         break;
549     }
550 }
551 break;
552 }
553 case TPM2B_MTYPE:

```

```

554     {
555         // A primitive TPM2B. A size and byte buffer. The single value (other than
556         // the tag) references the synthetic 'interface' value for the size
557         // parameter.
558         Tpm2bMarshal_mst* m2bt = (Tpm2bMarshal_mst*)sel;
559         //
560         if(IS_SUCCESS(Unmarshal(m2bt->sizeIndex, target, buffer, size)))
561             result = UnmarshalBytes(
562                 ((TPM2B*)target)->buffer, buffer, size, *((UINT16*)target));
563         break;
564     }
565     case TPM2BS_MTYPE:
566     {
567         // This is used when a TPM2B contains a structure.
568         Tpm2bsMarshal_mst* m2bst = (Tpm2bsMarshal_mst*)sel;
569         INT32 count;
570         //
571         if(IS_SUCCESS(Unmarshal(m2bst->sizeIndex, target, buffer, size)))
572         {
573             // fetch the size value and convert it to a 32-bit count value
574             count = (int32_t) * ((UINT16*)target);
575             if(count == 0)
576             {
577                 if(m2bst->modifiers & SIZE_EQUAL)
578                     result = TPM_RC_SIZE;
579             }
580             else if((*size == count) >= 0)
581             {
582                 marshalIndex_t index = m2bst->dataIndex;
583                 //
584                 // If this type propagates a null (PROPIGATE_NULL), propagate it
585                 if((m2bst->modifiers & PROPAGATE_NULL) && (typeIndex & typeIndex))
586                     index |= NULL_FLAG;
587                 // The structure might not start two bytes after the start of the
588                 // size field. The offset to the start of the structure is between
589                 // 2 and 8 bytes. This is encoded into the low 4 bits of the
590                 // modifiers byte
591                 if(IS_SUCCESS(Unmarshal(
592                     index,
593                     ((UINT8*)target) + (m2bst->modifiers & OFFSET_MASK),
594                     buffer,
595                     &count)))
596                 {
597                     if(count != 0)
598                         result = TPM_RC_SIZE;
599                 }
600             }
601             else
602                 result = TPM_RC_INSUFFICIENT;
603         }
604         break;
605     }
606     case LIST_MTYPE:
607     {
608         // Used for a list. A list is a qualified 32-bit 'count' value followed
609         // by a type indicator.
610         ListMarshal_mst* mlt = (ListMarshal_mst*)sel;
611         marshalIndex_t index = mlt->arrayRef;
612         //
613         if(IS_SUCCESS(Unmarshal(mlt->sizeIndex, target, buffer, size)))
614         {
615             // If this type propagates a null (PROPIGATE_NULL), propagate it
616             if((mlt->modifiers & PROPAGATE_NULL) && (typeIndex & NULL_FLAG))
617                 index |= NULL_FLAG;
618             result =
619                 ArrayUnmarshal(index,

```

```

620         ((UINT8*)target) + (mct->modifiers & OFFSET_MASK),
621         buffer,
622         size,
623         *((UINT32*)target));
624     }
625     break;
626 }
627 case NULL_MTYPE:
628 {
629     result = TPM_RC_SUCCESS;
630     break;
631 }
632 # if 0
633 case COMPOSITE_MTYPE:
634 {
635     CompositeMarshal_mst    *mct = (CompositeMarshal_mst *)sel;
636     int                      i;
637     UINT8                   *buf = *buffer;
638     INT32                   sz = *size;
639     //
640     result = TPM_RC_VALUE;
641     for(i = GET_ELEMENT_COUNT(mct->modifiers) - 1; i >= 0; i--)
642     {
643         marshalIndex_t      index = mct->types[i];
644         //
645         // This type might take a null so set it in each called value, just
646         // in case it is needed in that value. Only one value in each
647         // composite should have the takes null SET.
648         index |= typeIndex & NULL_MASK;
649         result = Unmarshal(index, target, buffer, size);
650         if(result == TPM_RC_SUCCESS)
651             break;
652         // Each of the composite values does its own unmarshaling. This
653         // has some execution overhead if it is unmarshaled multiple times
654         // but it saves code size in not having to reproduce the various
655         // unmarshaling types that can be in a composite. So, what this means
656         // is that the buffer pointer and size have to be reset for each
657         // unmarshaled value.
658         *buffer = buf;
659         *size = sz;
660     }
661     break;
662 }
663 # endif // 0
664 default:
665 {
666     result = TPM_RC_FAILURE;
667     break;
668 }
669 }
670 return result;
671 }
672
673 /*** Marshal()
674 // This is the function that drives marshaling of output. Because there is no
675 // validation of the output, there is a lot less code.
676 UINT16 Marshal(UINT16 typeIndex, // IN: the thing to marshal
677               void* source,      // IN: where the data comes from
678               UINT8** buffer,    // IN/OUT: the data source buffer
679               INT32* size        // IN/OUT: the remaining size
680 )
681 {
682     # define _source ((UINT8*)source)
683
684     const MarshalHeader_mst* sel;
685     UINT16                    retVal;

```

```

686 //
687 sel = GetDescriptor(typeIndex);
688 switch(sel->marshalType)
689 {
690     case VALUES_MTYPE:
691     case UINT_MTYPE:
692     case TABLE_MTYPE:
693     case MIN_MAX_MTYPE:
694     case ATTRIBUTES_MTYPE:
695     case COMPOSITE_MTYPE:
696     {
697 # if BIG_ENDIAN_TPM
698 #     define MM16 0
699 #     define MM32 0
700 #     define MM64 0
701 # else
702 // These flip the constant index values so that they count in reverse order when doing
703 // little-endian stuff
704 #     define MM16 1
705 #     define MM32 3
706 #     define MM64 7
707 # endif
708 // Just change the name and cast the type of the input parameters for typing purposes
709 # define mb (*buffer)
710 # define _source ((UINT8*)source)
711     retVal = (1 << (sel->modifiers & SIZE_MASK));
712     if(buffer != NULL)
713     {
714         if((size == NULL) || ((*size -- retVal) >= 0))
715         {
716             if(retVal == 4)
717             {
718                 mb[0 ^ MM32] = _source[0];
719                 mb[1 ^ MM32] = _source[1];
720                 mb[2 ^ MM32] = _source[2];
721                 mb[3 ^ MM32] = _source[3];
722             }
723             else if(retVal == 2)
724             {
725                 mb[0 ^ MM16] = _source[0];
726                 mb[1 ^ MM16] = _source[1];
727             }
728             else if(retVal == 1)
729                 mb[0] = _source[0];
730             else
731             {
732                 mb[0 ^ MM64] = _source[0];
733                 mb[1 ^ MM64] = _source[1];
734                 mb[2 ^ MM64] = _source[2];
735                 mb[3 ^ MM64] = _source[3];
736                 mb[4 ^ MM64] = _source[4];
737                 mb[5 ^ MM64] = _source[5];
738                 mb[6 ^ MM64] = _source[6];
739                 mb[7 ^ MM64] = _source[7];
740             }
741             *buffer += retVal;
742         }
743     }
744     break;
745 }
746 case STRUCTURE_MTYPE:
747 {
748     // #define _mst ((StructMarshal_mst *)sel)
749     StructMarshal_mst* mst = ((StructMarshal_mst*)sel);
750     int i;
751     const UINT16* value = mst->values;

```

```

752
753 //
754 for(retVal = 0, i = mst->elements; i > 0; value = &value[3], i--)
755 {
756     UINT16      des      = value[0];
757     marshalIndex_t index  = value[1];
758     UINT8*      offset   = _source + value[2];
759     //
760     switch(GET_ELEMENT_TYPE(des))
761     {
762         case UNION_STYPE:
763         {
764             UINT32 choice;
765             //
766             choice = GetSelector(source, mst->values, des);
767             retVal += MarshalUnion(index, offset, buffer, size, choice);
768             break;
769         }
770         case ARRAY_STYPE:
771         {
772             UINT32 count;
773             //
774             count = GetSelector(source, mst->values, des);
775             retVal += ArrayMarshal(index, offset, buffer, size, count);
776             break;
777         }
778         case SIMPLE_STYPE:
779         default:
780         {
781             // This is either another structure or a simple type
782             retVal += Marshal(index, offset, buffer, size);
783             break;
784         }
785     }
786 }
787 break;
788 }
789 case TPM2B_MTYPE:
790 {
791     // Get the number of bytes being marshaled
792     INT32 val = (int32_t) * ((UINT16*)source);
793     //
794     retVal = Marshal(UINT16_MARSHAL_REF, source, buffer, size);
795
796     // This is a standard 2B with a byte buffer
797     retVal += MarshalBytes(((TPM2B*)_source)->buffer, buffer, size, val);
798     break;
799 }
800 case TPM2BS_MTYPE: // A structure in a TPM2B
801 {
802     Tpm2bsMarshal_mst* m2bst = (Tpm2bsMarshal_mst*)sel;
803     UINT8*      offset;
804     UINT16      amount;
805     UINT8*      marshaledSize;
806     //
807     // Save the address of where the size should go
808     marshaledSize = *buffer;
809
810     // marshal the size (checks the space and advanced the pointer)
811     retVal = Marshal(UINT16_MARSHAL_REF, source, buffer, size);
812
813     // This gets the 'offsetof' the structure to marshal. It was placed in the
814     // modifiers byte because the offset from the start of the TPM2B to the
815     // start of the structure is going to be less than 8 and the modifiers
816     // byte isn't needed for anything else.
817     offset = _source + (m2bst->modifiers & SIGNED_MASK);

```



```

818
819 // Marshal the structure and get its size
820 amount = Marshal(m2bst->dataIndex, offset, buffer, size);
821
822 // put the size in the space used when the size was marshaled.
823 if(buffer != NULL)
824     UINT16_TO_BYTE_ARRAY(amount, marshaledSize);
825 retVal += amount;
826 break;
827
828 case LIST_MTYPE:
829 {
830     ListMarshal_mst* mlt = ((ListMarshal_mst*)sel);
831     UINT8* offset = _source + (mlt->modifiers & SIGNED_MASK);
832     retVal = Marshal(UINT32_MARSHAL_REF, source, buffer, size);
833     retVal += ArrayMarshal((marshalIndex_t) (mlt->arrayRef),
834                           offset,
835                           buffer,
836                           size,
837                           *((UINT32*)source));
838     break;
839 }
840 case NULL_MTYPE:
841     retVal = 0;
842     break;
843 case ERROR_MTYPE:
844 default:
845 {
846     if(size != NULL)
847         *size = -1;
848     retVal = 0;
849     break;
850 }
851 }
852 return retVal;
853 }
854
855 #endif // TABLE_DRIVEN_MARSHAL

```

7.197 /tpm/src/support/TableMarshalData.c

```

1 // clang-format off
2 /* (Auto-generated)
3  * Created by NewMarshal; Version 1.4 Apr 7, 2019
4  * Date: Mar 6, 2020 Time: 01:50:10PM
5  */
6
7 // This file contains the data initializer used for the table-driven marshaling code.
8
9 #include "Tpm.h"
10
11
12 #if TABLE_DRIVEN_MARSHAL
13 #include "TableMarshal.h"
14 #include "Marshal.h"
15
16 // The array marshaling table
17 ArrayMarshal_mst ArrayLookupTable[] = {
18     ARRAY_MARSHAL_ENTRY(UINT8),
19     ARRAY_MARSHAL_ENTRY(TPM_CC),
20     ARRAY_MARSHAL_ENTRY(TPMA_CC),
21     ARRAY_MARSHAL_ENTRY(TPM_ALG_ID),
22     ARRAY_MARSHAL_ENTRY(TPM_HANDLE),
23     ARRAY_MARSHAL_ENTRY(TPM2B_DIGEST),
24     ARRAY_MARSHAL_ENTRY(TPMT_HA),

```

```

25     ARRAY_MARSHAL_ENTRY(TPMS_PCR_SELECTION),
26     ARRAY_MARSHAL_ENTRY(TPMS_ALG_PROPERTY),
27     ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_PROPERTY),
28     ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_PCR_SELECT),
29     ARRAY_MARSHAL_ENTRY(TPM_ECC_CURVE),
30     ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_POLICY),
31     ARRAY_MARSHAL_ENTRY(TPMS_ACT_DATA),
32     ARRAY_MARSHAL_ENTRY(TPMS_AC_OUTPUT) };
33
34 // The main marshaling structure
35 MarshalData_st MarshalData = {
36 // UINT8_DATA
37 {UINT_MTYPE, 0},
38 // UINT16_DATA
39 {UINT_MTYPE, 1},
40 // UINT32_DATA
41 {UINT_MTYPE, 2},
42 // UINT64_DATA
43 {UINT_MTYPE, 3},
44 // INT8_DATA
45 {UINT_MTYPE, 0 + IS_SIGNED},
46 // INT16_DATA
47 {UINT_MTYPE, 1 + IS_SIGNED},
48 // INT32_DATA
49 {UINT_MTYPE, 2 + IS_SIGNED},
50 // INT64_DATA
51 {UINT_MTYPE, 3 + IS_SIGNED},
52 // UINT0_DATA
53 {NULL_MTYPE, 0},
54 // TPM_ECC_CURVE_DATA
55 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_CURVE,
56 {TPM_ECC_NONE,
57 RANGE(1, 32, UINT16),
58 (UINT32)((ECC_NIST_P192 << 0) | (ECC_NIST_P224 << 1) | (ECC_NIST_P256 << 2) |
59 (ECC_NIST_P384 << 3) | (ECC_NIST_P521 << 4) | (ECC_BN_P256 << 15) |
60 (ECC_BN_P638 << 16) | (ECC_SM2_P256 << 31))}},
61 // TPM_CLOCK_ADJUST_DATA
62 {MIN_MAX_MTYPE, ONE_BYTES|IS_SIGNED, (UINT8)TPM_RC_VALUE,
63 {RANGE(TPM_CLOCK_COARSE_SLOWER, TPM_CLOCK_COARSE_FASTER, INT8)}},
64 // TPM_EO_DATA
65 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE,
66 {RANGE(TPM_EO_EQ, TPM_EO_BITCLEAR, UINT16)}},
67 // TPM_SU_DATA
68 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 2,
69 {TPM_SU_CLEAR, TPM_SU_STATE}},
70 // TPM_SE_DATA
71 {TABLE_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE, 3,
72 {TPM_SE_HMAC, TPM_SE_POLICY, TPM_SE_TRIAL}},
73 // TPM_CAP_DATA
74 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 1,
75 {RANGE(TPM_CAP_ALGS, TPM_CAP_ACT, UINT32),
76 TPM_CAP_VENDOR_PROPERTY}},
77 // TPMA_ALGORITHM_DATA
78 {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFFFF8F0},
79 // TPMA_OBJECT_DATA
80 {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFFF0F309},
81 // TPMA_SESSION_DATA
82 {ATTRIBUTES_MTYPE, ONE_BYTES, 0x00000018},
83 // TPMA_ACT_DATA
84 {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFFFFFFFFC},
85 // TPMI_YES_NO_DATA
86 {TABLE_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE, 2,
87 {NO, YES}},
88 // TPMI_DH_OBJECT_DATA
89 {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 0,
90 {TPM_RH_NULL,

```

```

91     RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
92     RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32)}},
93 // TPMI_DH_PARENT_DATA
94 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 6, 8,
95     {RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
96     RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32),
97     RANGE(SVN_OWNER_FIRST, SVN_OWNER_LAST, UINT32),
98     RANGE(SVN_ENDORSEMENT_FIRST, SVN_ENDORSEMENT_LAST, UINT32),
99     RANGE(SVN_PLATFORM_FIRST, SVN_PLATFORM_LAST, UINT32),
100     RANGE(SVN_NULL_FIRST, SVN_NULL_LAST, UINT32),
101     TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM, TPM_RH_NULL,
102     TPM_RH_FW_OWNER, TPM_RH_FW_ENDORSEMENT, TPM_RH_FW_PLATFORM,
103     TPM_RH_FW_NULL}}},
104 // TPMI_DH_PERSISTENT_DATA
105 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
106     {RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32)}},
107 // TPMI_DH_ENTITY_DATA
108 {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 5, 4,
109     {TPM_RH_NULL,
110     RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
111     RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32),
112     RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32),
113     RANGE(PCR_FIRST, PCR_LAST, UINT32),
114     RANGE(TPM_RH_AUTH_00, TPM_RH_AUTH_FF, UINT32),
115     TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}}},
116 // TPMI_DH_PCR_DATA
117 {MIN_MAX_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE,
118     {TPM_RH_NULL,
119     RANGE(PCR_FIRST, PCR_LAST, UINT32)}},
120 // TPMI_SH_AUTH_SESSION_DATA
121 {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 0,
122     {TPM_RS_PW,
123     RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
124     RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32)}},
125 // TPMI_SH_HMAC_DATA
126 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
127     {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32)}},
128 // TPMI_SH_POLICY_DATA
129 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
130     {RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32)}},
131 // TPMI_DH_CONTEXT_DATA
132 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 3, 0,
133     {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
134     RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32),
135     RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32)}},
136 // TPMI_DH_SAVED_DATA
137 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2, 3,
138     {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
139     RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32),
140     0x80000000, 0x80000001, 0x80000002}},
141 // TPMI_RH_HIERARCHY_DATA
142 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 4, 8,
143     {RANGE(SVN_OWNER_FIRST, SVN_OWNER_LAST, UINT32),
144     RANGE(SVN_ENDORSEMENT_FIRST, SVN_ENDORSEMENT_LAST, UINT32),
145     RANGE(SVN_PLATFORM_FIRST, SVN_PLATFORM_LAST, UINT32),
146     RANGE(SVN_NULL_FIRST, SVN_NULL_LAST, UINT32),
147     TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM, TPM_RH_NULL,
148     TPM_RH_FW_OWNER, TPM_RH_FW_ENDORSEMENT, TPM_RH_FW_PLATFORM,
149     TPM_RH_FW_NULL}}},
150 // TPMI_RH_ENABLES_DATA
151 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 4,
152     {TPM_RH_NULL,
153     TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM, TPM_RH_PLATFORM_NV}},
154 // TPMI_RH_HIERARCHY_AUTH_DATA
155 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 4,
156     {TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},

```

```

157 // TPMI_RH_HIERARCHY_POLICY_DATA
158 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 4,
159   {RANGE(TPM_RH_ACT_0, TPM_RH_ACT_F, UINT32),
160    TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}}},
161 // TPMI_RH_BASE_HIERARCHY_DATA
162 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 3,
163   {TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}}},
164 // TPMI_RH_PLATFORM_DATA
165 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1,
166   {TPM_RH_PLATFORM}}},
167 // TPMI_RH_OWNER_DATA
168 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 1,
169   {TPM_RH_NULL,
170    TPM_RH_OWNER}}},
171 // TPMI_RH_ENDORSEMENT_DATA
172 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 1,
173   {TPM_RH_NULL,
174    TPM_RH_ENDORSEMENT}}},
175 // TPMI_RH_PROVISION_DATA
176 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2,
177   {TPM_RH_OWNER, TPM_RH_PLATFORM}}},
178 // TPMI_RH_CLEAR_DATA
179 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2,
180   {TPM_RH_LOCKOUT, TPM_RH_PLATFORM}}},
181 // TPMI_RH_NV_AUTH_DATA
182 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 2,
183   {RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32),
184    TPM_RH_OWNER, TPM_RH_PLATFORM}}},
185 // TPMI_RH_LOCKOUT_DATA
186 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1,
187   {TPM_RH_LOCKOUT}}},
188 // TPMI_RH_NV_INDEX_DATA
189 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
190   {RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32)}}},
191 // TPMI_RH_AC_DATA
192 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
193   {RANGE(AC_FIRST, AC_LAST, UINT32)}}},
194 // TPMI_RH_ACT_DATA
195 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
196   {RANGE(TPM_RH_ACT_0, TPM_RH_ACT_F, UINT32)}}},
197 // TPMI_ALG_HASH_DATA
198 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_HASH,
199   {TPM_ALG_NULL,
200    RANGE(4, 41, UINT16),
201    (UINT32)((ALG_SHA1 << 0) | (ALG_SHA256 << 7) | (ALG_SHA384 << 8) |
202             (ALG_SHA512 << 9) | (ALG_SM3_256 << 14)),
203    (UINT32)((ALG_SHA3_256 << 3) | (ALG_SHA3_384 << 4) | (ALG_SHA3_512 << 5))}},
204 // TPMI_ALG_ASYM_DATA
205 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_ASYMMETRIC,
206   {TPM_ALG_NULL,
207    RANGE(1, 35, UINT16),
208    (UINT32)((ALG_RSA << 0)),
209    (UINT32)((ALG_ECC << 2))}},
210 // TPMI_ALG_SYM_DATA
211 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
212   {TPM_ALG_NULL,
213    RANGE(3, 38, UINT16),
214    (UINT32)((ALG_AES << 3) | (ALG_XOR << 7) | (ALG_SM4 << 16)),
215    (UINT32)((ALG_CAMELLIA << 3))}},
216 // TPMI_ALG_SYM_OBJECT_DATA
217 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
218   {TPM_ALG_NULL,
219    RANGE(3, 38, UINT16),
220    (UINT32)((ALG_AES << 3) | (ALG_SM4 << 16)),
221    (UINT32)((ALG_CAMELLIA << 3))}},
222 // TPMI_ALG_SYM_MODE_DATA

```

```

223 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_MODE,
224     {TPM_ALG_NULL,
225         RANGE(63, 68, UINT16),
226         (UINT32)((ALG_CMAC << 0) | (ALG_CTR << 1) | (ALG_OFB << 2) |
227             (ALG_CBC << 3) | (ALG_CFB << 4) | (ALG_ECB << 5))}},
228 // TPMI_ALG_KDF_DATA
229 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_KDF,
230     {TPM_ALG_NULL,
231         RANGE(7, 34, UINT16),
232         (UINT32)((ALG_MGF1 << 0) | (ALG_KDF1_SP800_56A << 25) |
233             (ALG_KDF2 << 26) | (ALG_KDF1_SP800_108 << 27))}},
234 // TPMI_ALG_SIG_SCHEME_DATA
235 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
236     {TPM_ALG_NULL,
237         RANGE(5, 28, UINT16),
238         (UINT32)((ALG_HMAC << 0) | (ALG_RSASSA << 15) | (ALG_RSAPSS << 17) |
239             (ALG_ECDSA << 19) | (ALG_ECDSA << 21) | (ALG_SM2 << 22) |
240             (ALG_ECSCHNORR << 23))}},
241 // TPMI_ECC_KEY_EXCHANGE_DATA
242 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
243     {TPM_ALG_NULL,
244         RANGE(25, 29, UINT16),
245         (UINT32)((ALG_ECDH << 0) | (ALG_SM2 << 2) | (ALG_ECMQV << 4))}},
246 // TPMI_ST_COMMAND_TAG_DATA
247 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_BAD_TAG, 2,
248     {TPM_ST_NO_SESSIONS, TPM_ST_SESSIONS}},
249 // TPMI_ALG_MAC_SCHEME_DATA
250 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
251     {TPM_ALG_NULL,
252         RANGE(4, 63, UINT16),
253         (UINT32)((ALG_SHA1 << 0) | (ALG_SHA256 << 7) | (ALG_SHA384 << 8) |
254             (ALG_SHA512 << 9) | (ALG_SM3_256 << 14)),
255         (UINT32)((ALG_SHA3_256 << 3) | (ALG_SHA3_384 << 4) | (ALG_SHA3_512 << 5) |
256             (ALG_CMAC << 27))}},
257 // TPMI_ALG_CIPHER_MODE_DATA
258 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_MODE,
259     {TPM_ALG_NULL,
260         RANGE(64, 68, UINT16),
261         (UINT32)((ALG_CTR << 0) | (ALG_OFB << 1) | (ALG_CBC << 2) | (ALG_CFB << 3) |
262             (ALG_ECB << 4))}},
263 // TPMS_EMPTY_DATA
264 {STRUCTURE_MTYPE, 1,
265     {SET_ELEMENT_TYPE(SIMPLE_TYPE), UINT0_MARSHAL_REF, 0}},
266 // TPMS_ALGORITHM_DESCRIPTION_DATA
267 {STRUCTURE_MTYPE, 2, {
268     SET_ELEMENT_TYPE(SIMPLE_TYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
269     TPM_ALG_ID_MARSHAL_REF,
270     (UINT16)(offsetof(TPMS_ALGORITHM_DESCRIPTION, alg)),
271     SET_ELEMENT_TYPE(SIMPLE_TYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
272     TPMA_ALGORITHM_MARSHAL_REF,
273     (UINT16)(offsetof(TPMS_ALGORITHM_DESCRIPTION, attributes))}},
274 // TPMU_HA_DATA
275 {9, IS_ARRAY_UNION, (UINT16)(offsetof(TPMU_HA_mst, marshalingTypes)),
276     {(UINT32)TPM_ALG_SHA1, (UINT32)TPM_ALG_SHA256, (UINT32)TPM_ALG_SHA384,
277         (UINT32)TPM_ALG_SHA512, (UINT32)TPM_ALG_SM3_256, (UINT32)TPM_ALG_SHA3_256,
278         (UINT32)TPM_ALG_SHA3_384, (UINT32)TPM_ALG_SHA3_512, (UINT32)TPM_ALG_NULL},
279     {(UINT16)(SHA1_DIGEST_SIZE), (UINT16)(SHA256_DIGEST_SIZE),
280         (UINT16)(SHA384_DIGEST_SIZE), (UINT16)(SHA512_DIGEST_SIZE),
281         (UINT16)(SM3_256_DIGEST_SIZE), (UINT16)(SHA3_256_DIGEST_SIZE),
282         (UINT16)(SHA3_384_DIGEST_SIZE), (UINT16)(SHA3_512_DIGEST_SIZE),
283         (UINT16)(0)}},
284 },
285 // TPMT_HA_DATA
286 {STRUCTURE_MTYPE, 2, {
287     SET_ELEMENT_TYPE(SIMPLE_TYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
288     TPMI_ALG_HASH_MARSHAL_REF,

```



```

289         (UINT16) (offsetof(TPMT_HA, hashAlg)),
290         SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
291         TPMU_HA_MARSHAL_REF,
292         (UINT16) (offsetof(TPMT_HA, digest))},
293 // TPM2B_DIGEST_DATA
294 {TPM2B_MTYPE, Type00_MARSHAL_REF},
295 // TPM2B_DATA_DATA
296 {TPM2B_MTYPE, Type01_MARSHAL_REF},
297 // TPM2B_EVENT_DATA
298 {TPM2B_MTYPE, Type02_MARSHAL_REF},
299 // TPM2B_MAX_BUFFER_DATA
300 {TPM2B_MTYPE, Type03_MARSHAL_REF},
301 // TPM2B_MAX_NV_BUFFER_DATA
302 {TPM2B_MTYPE, Type04_MARSHAL_REF},
303 // TPM2B_TIMEOUT_DATA
304 {TPM2B_MTYPE, Type05_MARSHAL_REF},
305 // TPM2B_IV_DATA
306 {TPM2B_MTYPE, Type06_MARSHAL_REF},
307 // NULL_UNION_DATA
308 {0},
309 // TPM2B_NAME_DATA
310 {TPM2B_MTYPE, Type07_MARSHAL_REF},
311 // TPMS_PCR_SELECT_DATA
312 {STRUCTURE_MTYPE, 2, {
313     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
314     Type08_MARSHAL_REF,
315     (UINT16) (offsetof(TPMS_PCR_SELECT, sizeofSelect)),
316     SET_ELEMENT_TYPE(ARRAY_STYPE) | SET_ELEMENT_NUMBER(0),
317     UINT8_ARRAY_MARSHAL_INDEX,
318     (UINT16) (offsetof(TPMS_PCR_SELECT, pcrSelect))}},
319 // TPMS_PCR_SELECTION_DATA
320 {STRUCTURE_MTYPE, 3, {
321     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
322     TPMI_ALG_HASH_MARSHAL_REF,
323     (UINT16) (offsetof(TPMS_PCR_SELECTION, hash)),
324     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
325     Type08_MARSHAL_REF,
326     (UINT16) (offsetof(TPMS_PCR_SELECTION, sizeofSelect)),
327     SET_ELEMENT_TYPE(ARRAY_STYPE) | SET_ELEMENT_NUMBER(1),
328     UINT8_ARRAY_MARSHAL_INDEX,
329     (UINT16) (offsetof(TPMS_PCR_SELECTION, pcrSelect))}},
330 // TPMT_TK_CREATION_DATA
331 {STRUCTURE_MTYPE, 3, {
332     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
333     Type10_MARSHAL_REF,
334     (UINT16) (offsetof(TPMT_TK_CREATION, tag)),
335     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
336     TPMI_RH_HIERARCHY_MARSHAL_REF | NULL_FLAG,
337     (UINT16) (offsetof(TPMT_TK_CREATION, hierarchy)),
338     SET_ELEMENT_TYPE(SIMPLE_STYPE),
339     TPM2B_DIGEST_MARSHAL_REF,
340     (UINT16) (offsetof(TPMT_TK_CREATION, digest))}},
341 // TPMT_TK_VERIFIED_DATA
342 {STRUCTURE_MTYPE, 3, {
343     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
344     Type11_MARSHAL_REF,
345     (UINT16) (offsetof(TPMT_TK_VERIFIED, tag)),
346     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
347     TPMI_RH_HIERARCHY_MARSHAL_REF | NULL_FLAG,
348     (UINT16) (offsetof(TPMT_TK_VERIFIED, hierarchy)),
349     SET_ELEMENT_TYPE(SIMPLE_STYPE),
350     TPM2B_DIGEST_MARSHAL_REF,
351     (UINT16) (offsetof(TPMT_TK_VERIFIED, digest))}},
352 // TPMT_TK_AUTH_DATA
353 {STRUCTURE_MTYPE, 3, {
354     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),

```



```

355         Type12_MARSHAL_REF,
356         (UINT16) (offsetof(TPMT_TK_AUTH, tag)),
357     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
358     TPMI_RH_HIERARCHY_MARSHAL_REF | NULL_FLAG,
359     (UINT16) (offsetof(TPMT_TK_AUTH, hierarchy)),
360     SET_ELEMENT_TYPE(SIMPLE_TYPE),
361     TPM2B_DIGEST_MARSHAL_REF,
362     (UINT16) (offsetof(TPMT_TK_AUTH, digest))}},
363 // TPMT_TK_HASHCHECK_DATA
364 {STRUCTURE_MTYPE, 3, {
365     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
366     Type13_MARSHAL_REF,
367     (UINT16) (offsetof(TPMT_TK_HASHCHECK, tag)),
368     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
369     TPMI_RH_HIERARCHY_MARSHAL_REF | NULL_FLAG,
370     (UINT16) (offsetof(TPMT_TK_HASHCHECK, hierarchy)),
371     SET_ELEMENT_TYPE(SIMPLE_TYPE),
372     TPM2B_DIGEST_MARSHAL_REF,
373     (UINT16) (offsetof(TPMT_TK_HASHCHECK, digest))}},
374 // TPMS_ALG_PROPERTY_DATA
375 {STRUCTURE_MTYPE, 2, {
376     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
377     TPM_ALG_ID_MARSHAL_REF,
378     (UINT16) (offsetof(TPMS_ALG_PROPERTY, alg)),
379     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
380     TPMA_ALGORITHM_MARSHAL_REF,
381     (UINT16) (offsetof(TPMS_ALG_PROPERTY, algProperties))}},
382 // TPMS_TAGGED_PROPERTY_DATA
383 {STRUCTURE_MTYPE, 2, {
384     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
385     TPM_PT_MARSHAL_REF,
386     (UINT16) (offsetof(TPMS_TAGGED_PROPERTY, property)),
387     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
388     UINT32_MARSHAL_REF,
389     (UINT16) (offsetof(TPMS_TAGGED_PROPERTY, value))}},
390 // TPMS_TAGGED_PCR_SELECT_DATA
391 {STRUCTURE_MTYPE, 3, {
392     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
393     TPM_PT_PCR_MARSHAL_REF,
394     (UINT16) (offsetof(TPMS_TAGGED_PCR_SELECT, tag)),
395     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
396     Type08_MARSHAL_REF,
397     (UINT16) (offsetof(TPMS_TAGGED_PCR_SELECT, sizeofSelect)),
398     SET_ELEMENT_TYPE(ARRAY_TYPE) | SET_ELEMENT_NUMBER(1),
399     UINT8_ARRAY_MARSHAL_INDEX,
400     (UINT16) (offsetof(TPMS_TAGGED_PCR_SELECT, pcrSelect))}},
401 // TPMS_TAGGED_POLICY_DATA
402 {STRUCTURE_MTYPE, 2, {
403     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
404     TPM_HANDLE_MARSHAL_REF,
405     (UINT16) (offsetof(TPMS_TAGGED_POLICY, handle)),
406     SET_ELEMENT_TYPE(SIMPLE_TYPE),
407     TPMT_HA_MARSHAL_REF,
408     (UINT16) (offsetof(TPMS_TAGGED_POLICY, policyHash))}},
409 // TPMS_ACT_DATA_DATA
410 {STRUCTURE_MTYPE, 3, {
411     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
412     TPM_HANDLE_MARSHAL_REF,
413     (UINT16) (offsetof(TPMS_ACT_DATA, handle)),
414     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
415     UINT32_MARSHAL_REF,
416     (UINT16) (offsetof(TPMS_ACT_DATA, timeout)),
417     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
418     TPMA_ACT_MARSHAL_REF,
419     (UINT16) (offsetof(TPMS_ACT_DATA, attributes))}},
420 // TPML_CC_DATA

```

```

421 {LIST_MTYPE,
422     (UINT8) (offsetof(TPML_CC, commandCodes)),
423     Type15_MARSHAL_REF,
424     TPM_CC_ARRAY_MARSHAL_INDEX},
425 // TPML_CCA_DATA
426 {LIST_MTYPE,
427     (UINT8) (offsetof(TPML_CCA, commandAttributes)),
428     Type15_MARSHAL_REF,
429     TPMA_CC_ARRAY_MARSHAL_INDEX},
430 // TPML_ALG_DATA
431 {LIST_MTYPE,
432     (UINT8) (offsetof(TPML_ALG, algorithms)),
433     Type17_MARSHAL_REF,
434     TPM_ALG_ID_ARRAY_MARSHAL_INDEX},
435 // TPML_HANDLE_DATA
436 {LIST_MTYPE,
437     (UINT8) (offsetof(TPML_HANDLE, handle)),
438     Type18_MARSHAL_REF,
439     TPM_HANDLE_ARRAY_MARSHAL_INDEX},
440 // TPML_DIGEST_DATA
441 {LIST_MTYPE,
442     (UINT8) (offsetof(TPML_DIGEST, digests)),
443     Type19_MARSHAL_REF,
444     TPM2B_DIGEST_ARRAY_MARSHAL_INDEX},
445 // TPML_DIGEST_VALUES_DATA
446 {LIST_MTYPE,
447     (UINT8) (offsetof(TPML_DIGEST_VALUES, digests)),
448     Type20_MARSHAL_REF,
449     TPMT_HA_ARRAY_MARSHAL_INDEX},
450 // TPML_PCR_SELECTION_DATA
451 {LIST_MTYPE,
452     (UINT8) (offsetof(TPML_PCR_SELECTION, pcrSelections)),
453     Type20_MARSHAL_REF,
454     TPMS_PCR_SELECTION_ARRAY_MARSHAL_INDEX},
455 // TPML_ALG_PROPERTY_DATA
456 {LIST_MTYPE,
457     (UINT8) (offsetof(TPML_ALG_PROPERTY, algProperties)),
458     Type22_MARSHAL_REF,
459     TPMS_ALG_PROPERTY_ARRAY_MARSHAL_INDEX},
460 // TPML_TAGGED_TPM_PROPERTY_DATA
461 {LIST_MTYPE,
462     (UINT8) (offsetof(TPML_TAGGED_TPM_PROPERTY, tpmProperty)),
463     Type23_MARSHAL_REF,
464     TPMS_TAGGED_PROPERTY_ARRAY_MARSHAL_INDEX},
465 // TPML_TAGGED_PCR_PROPERTY_DATA
466 {LIST_MTYPE,
467     (UINT8) (offsetof(TPML_TAGGED_PCR_PROPERTY, pcrProperty)),
468     Type24_MARSHAL_REF,
469     TPMS_TAGGED_PCR_SELECT_ARRAY_MARSHAL_INDEX},
470 // TPML_ECC_CURVE_DATA
471 {LIST_MTYPE,
472     (UINT8) (offsetof(TPML_ECC_CURVE, eccCurves)),
473     Type25_MARSHAL_REF,
474     TPM_ECC_CURVE_ARRAY_MARSHAL_INDEX},
475 // TPML_TAGGED_POLICY_DATA
476 {LIST_MTYPE,
477     (UINT8) (offsetof(TPML_TAGGED_POLICY, policies)),
478     Type26_MARSHAL_REF,
479     TPMS_TAGGED_POLICY_ARRAY_MARSHAL_INDEX},
480 // TPML_ACT_DATA_DATA
481 {LIST_MTYPE,
482     (UINT8) (offsetof(TPML_ACT_DATA, actData)),
483     Type27_MARSHAL_REF,
484     TPMS_ACT_DATA_ARRAY_MARSHAL_INDEX},
485 // TPMU_CAPABILITIES_DATA
486 {11, 0, (UINT16) (offsetof(TPMU_CAPABILITIES_mst, marshalingTypes)),

```

```

487     { (UINT32)TPM_CAP_ALGS,                (UINT32)TPM_CAP_HANDLES,
488       (UINT32)TPM_CAP_COMMANDS,            (UINT32)TPM_CAP_PP_COMMANDS,
489       (UINT32)TPM_CAP_AUDIT_COMMANDS,      (UINT32)TPM_CAP_PCERS,
490       (UINT32)TPM_CAP_TPM_PROPERTIES,      (UINT32)TPM_CAP_PCR_PROPERTIES,
491       (UINT32)TPM_CAP_ECC_CURVES,          (UINT32)TPM_CAP_AUTH_POLICIES,
492       (UINT32)TPM_CAP_ACT},
493     { (UINT16) (TPML_ALG_PROPERTY_MARSHAL_REF),
494       (UINT16) (TPML_HANDLE_MARSHAL_REF),
495       (UINT16) (TPML_CCA_MARSHAL_REF),
496       (UINT16) (TPML_CC_MARSHAL_REF),
497       (UINT16) (TPML_CC_MARSHAL_REF),
498       (UINT16) (TPML_PCR_SELECTION_MARSHAL_REF),
499       (UINT16) (TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF),
500       (UINT16) (TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF),
501       (UINT16) (TPML_ECC_CURVE_MARSHAL_REF),
502       (UINT16) (TPML_TAGGED_POLICY_MARSHAL_REF),
503       (UINT16) (TPML_ACT_DATA_MARSHAL_REF) }
504 },
505 // TPMS_CAPABILITY_DATA_DATA
506 {STRUCTURE_MTYPE, 2, {
507     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
508     TPM_CAP_MARSHAL_REF,
509     (UINT16) (offsetof(TPMS_CAPABILITY_DATA, capability)),
510     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
511     TPMU_CAPABILITIES_MARSHAL_REF,
512     (UINT16) (offsetof(TPMS_CAPABILITY_DATA, data))}},
513 // TPMS_CLOCK_INFO_DATA
514 {STRUCTURE_MTYPE, 4, {
515     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
516     UINT64_MARSHAL_REF,
517     (UINT16) (offsetof(TPMS_CLOCK_INFO, clock)),
518     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
519     UINT32_MARSHAL_REF,
520     (UINT16) (offsetof(TPMS_CLOCK_INFO, resetCount)),
521     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
522     UINT32_MARSHAL_REF,
523     (UINT16) (offsetof(TPMS_CLOCK_INFO, restartCount)),
524     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
525     TPMI_YES_NO_MARSHAL_REF,
526     (UINT16) (offsetof(TPMS_CLOCK_INFO, safe))}},
527 // TPMS_TIME_INFO_DATA
528 {STRUCTURE_MTYPE, 2, {
529     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
530     UINT64_MARSHAL_REF,
531     (UINT16) (offsetof(TPMS_TIME_INFO, time)),
532     SET_ELEMENT_TYPE(SIMPLE_STYPE),
533     TPMS_CLOCK_INFO_MARSHAL_REF,
534     (UINT16) (offsetof(TPMS_TIME_INFO, clockInfo))}},
535 // TPMS_TIME_ATTEST_DATA
536 {STRUCTURE_MTYPE, 2, {
537     SET_ELEMENT_TYPE(SIMPLE_STYPE),
538     TPMS_TIME_INFO_MARSHAL_REF,
539     (UINT16) (offsetof(TPMS_TIME_ATTEST_INFO, time)),
540     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
541     UINT64_MARSHAL_REF,
542     (UINT16) (offsetof(TPMS_TIME_ATTEST_INFO, firmwareVersion))}},
543 // TPMS_CERTIFY_INFO_DATA
544 {STRUCTURE_MTYPE, 2, {
545     SET_ELEMENT_TYPE(SIMPLE_STYPE),
546     TPM2B_NAME_MARSHAL_REF,
547     (UINT16) (offsetof(TPMS_CERTIFY_INFO, name)),
548     SET_ELEMENT_TYPE(SIMPLE_STYPE),
549     TPM2B_NAME_MARSHAL_REF,
550     (UINT16) (offsetof(TPMS_CERTIFY_INFO, qualifiedName))}},
551 // TPMS_QUOTE_INFO_DATA
552 {STRUCTURE_MTYPE, 2, {

```

```

553     SET_ELEMENT_TYPE(SIMPLE_TYPE),
554     TPML_PCR_SELECTION MARSHAL_REF,
555     (UINT16) (offsetof(TPMS_QUOTE_INFO, pcrSelect)),
556     SET_ELEMENT_TYPE(SIMPLE_TYPE),
557     TPM2B_DIGEST MARSHAL_REF,
558     (UINT16) (offsetof(TPMS_QUOTE_INFO, pcrDigest))}},
559 // TPMS_COMMAND_AUDIT_INFO_DATA
560 {STRUCTURE_MTYPE, 4, {
561     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
562     UINT64 MARSHAL_REF,
563     (UINT16) (offsetof(TPMS_COMMAND_AUDIT_INFO, auditCounter)),
564     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
565     TPM_ALG_ID MARSHAL_REF,
566     (UINT16) (offsetof(TPMS_COMMAND_AUDIT_INFO, digestAlg)),
567     SET_ELEMENT_TYPE(SIMPLE_TYPE),
568     TPM2B_DIGEST MARSHAL_REF,
569     (UINT16) (offsetof(TPMS_COMMAND_AUDIT_INFO, auditDigest)),
570     SET_ELEMENT_TYPE(SIMPLE_TYPE),
571     TPM2B_DIGEST MARSHAL_REF,
572     (UINT16) (offsetof(TPMS_COMMAND_AUDIT_INFO, commandDigest))}},
573 // TPMS_SESSION_AUDIT_INFO_DATA
574 {STRUCTURE_MTYPE, 2, {
575     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
576     TPMI_YES_NO MARSHAL_REF,
577     (UINT16) (offsetof(TPMS_SESSION_AUDIT_INFO, exclusiveSession)),
578     SET_ELEMENT_TYPE(SIMPLE_TYPE),
579     TPM2B_DIGEST MARSHAL_REF,
580     (UINT16) (offsetof(TPMS_SESSION_AUDIT_INFO, sessionDigest))}},
581 // TPMS_CREATION_INFO_DATA
582 {STRUCTURE_MTYPE, 2, {
583     SET_ELEMENT_TYPE(SIMPLE_TYPE),
584     TPM2B_NAME MARSHAL_REF,
585     (UINT16) (offsetof(TPMS_CREATION_INFO, objectName)),
586     SET_ELEMENT_TYPE(SIMPLE_TYPE),
587     TPM2B_DIGEST MARSHAL_REF,
588     (UINT16) (offsetof(TPMS_CREATION_INFO, creationHash))}},
589 // TPMS_NV_CERTIFY_INFO_DATA
590 {STRUCTURE_MTYPE, 3, {
591     SET_ELEMENT_TYPE(SIMPLE_TYPE),
592     TPM2B_NAME MARSHAL_REF,
593     (UINT16) (offsetof(TPMS_NV_CERTIFY_INFO, indexName)),
594     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
595     UINT16 MARSHAL_REF,
596     (UINT16) (offsetof(TPMS_NV_CERTIFY_INFO, offset)),
597     SET_ELEMENT_TYPE(SIMPLE_TYPE),
598     TPM2B_MAX_NV_BUFFER MARSHAL_REF,
599     (UINT16) (offsetof(TPMS_NV_CERTIFY_INFO, nvContents))}},
600 // TPMS_NV_DIGEST_CERTIFY_INFO_DATA
601 {STRUCTURE_MTYPE, 2, {
602     SET_ELEMENT_TYPE(SIMPLE_TYPE),
603     TPM2B_NAME MARSHAL_REF,
604     (UINT16) (offsetof(TPMS_NV_DIGEST_CERTIFY_INFO, indexName)),
605     SET_ELEMENT_TYPE(SIMPLE_TYPE),
606     TPM2B_DIGEST MARSHAL_REF,
607     (UINT16) (offsetof(TPMS_NV_DIGEST_CERTIFY_INFO, nvDigest))}},
608 // TPMI_ST_ATTEST_DATA
609 {VALUES_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1, 1,
610     {RANGE(TPM_ST_ATTEST_NV, TPM_ST_ATTEST_CREATION, UINT16),
611     TPM_ST_ATTEST_NV_DIGEST}},
612 // TPMU_ATTEST_DATA
613 {8, 0, (UINT16) (offsetof(TPMU_ATTEST_mst, marshalingTypes)),
614     {(UINT32)TPM_ST_ATTEST_CERTIFY, (UINT32)TPM_ST_ATTEST_CREATION,
615     (UINT32)TPM_ST_ATTEST_QUOTE, (UINT32)TPM_ST_ATTEST_COMMAND_AUDIT,
616     (UINT32)TPM_ST_ATTEST_SESSION_AUDIT, (UINT32)TPM_ST_ATTEST_TIME,
617     (UINT32)TPM_ST_ATTEST_NV, (UINT32)TPM_ST_ATTEST_NV_DIGEST},
618     {(UINT16) (TPMS_CERTIFY_INFO_MARSHAL_REF),

```

```

619     (UINT16) (TPMS_CREATION_INFO_MARSHAL_REF),
620     (UINT16) (TPMS_QUOTE_INFO_MARSHAL_REF),
621     (UINT16) (TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF),
622     (UINT16) (TPMS_SESSION_AUDIT_INFO_MARSHAL_REF),
623     (UINT16) (TPMS_TIME_ATTEST_INFO_MARSHAL_REF),
624     (UINT16) (TPMS_NV_CERTIFY_INFO_MARSHAL_REF),
625     (UINT16) (TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF) }
626 },
627 // TPMS_ATTEST_DATA
628 {STRUCTURE_MTYPE, 7, {
629     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
630     TPM_CONSTANTS32_MARSHAL_REF,
631     (UINT16) (offsetof(TPMS_ATTEST, magic)),
632     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
633     TPMI_ST_ATTEST_MARSHAL_REF,
634     (UINT16) (offsetof(TPMS_ATTEST, type)),
635     SET_ELEMENT_TYPE(SIMPLE_STYPE),
636     TPM2B_NAME_MARSHAL_REF,
637     (UINT16) (offsetof(TPMS_ATTEST, qualifiedSigner)),
638     SET_ELEMENT_TYPE(SIMPLE_STYPE),
639     TPM2B_DATA_MARSHAL_REF,
640     (UINT16) (offsetof(TPMS_ATTEST, extraData)),
641     SET_ELEMENT_TYPE(SIMPLE_STYPE),
642     TPMS_CLOCK_INFO_MARSHAL_REF,
643     (UINT16) (offsetof(TPMS_ATTEST, clockInfo)),
644     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
645     UINT64_MARSHAL_REF,
646     (UINT16) (offsetof(TPMS_ATTEST, firmwareVersion)),
647     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(1),
648     TPMU_ATTEST_MARSHAL_REF,
649     (UINT16) (offsetof(TPMS_ATTEST, attested))}},
650 // TPM2B_ATTEST_DATA
651 {TPM2B_MTYPE, Type28_MARSHAL_REF},
652 // TPMS_AUTH_COMMAND_DATA
653 {STRUCTURE_MTYPE, 4, {
654     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
655     TPMI_SH_AUTH_SESSION_MARSHAL_REF | NULL_FLAG,
656     (UINT16) (offsetof(TPMS_AUTH_COMMAND, sessionHandle)),
657     SET_ELEMENT_TYPE(SIMPLE_STYPE),
658     TPM2B_NONCE_MARSHAL_REF,
659     (UINT16) (offsetof(TPMS_AUTH_COMMAND, nonce)),
660     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
661     TPMA_SESSION_MARSHAL_REF,
662     (UINT16) (offsetof(TPMS_AUTH_COMMAND, sessionAttributes)),
663     SET_ELEMENT_TYPE(SIMPLE_STYPE),
664     TPM2B_AUTH_MARSHAL_REF,
665     (UINT16) (offsetof(TPMS_AUTH_COMMAND, hmac))}},
666 // TPMS_AUTH_RESPONSE_DATA
667 {STRUCTURE_MTYPE, 3, {
668     SET_ELEMENT_TYPE(SIMPLE_STYPE),
669     TPM2B_NONCE_MARSHAL_REF,
670     (UINT16) (offsetof(TPMS_AUTH_RESPONSE, nonce)),
671     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
672     TPMA_SESSION_MARSHAL_REF,
673     (UINT16) (offsetof(TPMS_AUTH_RESPONSE, sessionAttributes)),
674     SET_ELEMENT_TYPE(SIMPLE_STYPE),
675     TPM2B_AUTH_MARSHAL_REF,
676     (UINT16) (offsetof(TPMS_AUTH_RESPONSE, hmac))}},
677 // TPMI_AES_KEY_BITS_DATA
678 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
679     {192*AES_192, 128*AES_128, 256*AES_256}},
680 // TPMI_SM4_KEY_BITS_DATA
681 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1,
682     {128*SM4_128}},
683 // TPMI_CAMELLIA_KEY_BITS_DATA
684 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,

```



```

685     {192*CAMELLIA_192, 128*CAMELLIA_128, 256*CAMELLIA_256}},
686 // TPMU_SYM_KEY_BITS_DATA
687 {6, 0, (UINT16)(offsetof(TPMU_SYM_KEY_BITS_mst, marshalingTypes)),
688     {(UINT32)TPM_ALG_AES, (UINT32)TPM_ALG_SM4,
689      (UINT32)TPM_ALG_CAMELLIA, (UINT32)TPM_ALG_XOR, (UINT32)TPM_ALG_NULL},
690     {(UINT16)(TPMI_AES_KEY_BITS_MARSHAL_REF),
691      (UINT16)(TPMI_SM4_KEY_BITS_MARSHAL_REF),
692      (UINT16)(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF),
693      (UINT16)(TPMI_ALG_HASH_MARSHAL_REF),
694      (UINT16)(UINT0_MARSHAL_REF)}},
695 },
696 // TPMU_SYM_MODE_DATA
697 {6, 0, (UINT16)(offsetof(TPMU_SYM_MODE_mst, marshalingTypes)),
698     {(UINT32)TPM_ALG_AES, (UINT32)TPM_ALG_SM4,
699      (UINT32)TPM_ALG_CAMELLIA, (UINT32)TPM_ALG_XOR, (UINT32)TPM_ALG_NULL},
700     {(UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
701      (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
702      (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
703      (UINT16)(UINT0_MARSHAL_REF),
704      (UINT16)(UINT0_MARSHAL_REF)}},
705 },
706 // TPMT_SYM_DEF_DATA
707 {STRUCTURE_MTYPE, 3, {
708     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
709     TPMI_ALG_SYM_MARSHAL_REF,
710     (UINT16)(offsetof(TPMT_SYM_DEF, algorithm)),
711     SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
712     TPMU_SYM_KEY_BITS_MARSHAL_REF,
713     (UINT16)(offsetof(TPMT_SYM_DEF, keyBits)),
714     SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
715     TPMU_SYM_MODE_MARSHAL_REF,
716     (UINT16)(offsetof(TPMT_SYM_DEF, mode))}},
717 // TPMT_SYM_DEF_OBJECT_DATA
718 {STRUCTURE_MTYPE, 3, {
719     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
720     TPMI_ALG_SYM_OBJECT_MARSHAL_REF,
721     (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, algorithm)),
722     SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
723     TPMU_SYM_KEY_BITS_MARSHAL_REF,
724     (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, keyBits)),
725     SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
726     TPMU_SYM_MODE_MARSHAL_REF,
727     (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, mode))}},
728 // TPM2B_SYM_KEY_DATA
729 {TPM2B_MTYPE, Type29_MARSHAL_REF},
730 // TPMS_SYMCIPHER_PARMS_DATA
731 {STRUCTURE_MTYPE, 1, {
732     SET_ELEMENT_TYPE(SIMPLE_STYPE),
733     TPMT_SYM_DEF_OBJECT_MARSHAL_REF,
734     (UINT16)(offsetof(TPMS_SYMCIPHER_PARMS, sym))}},
735 // TPM2B_LABEL_DATA
736 {TPM2B_MTYPE, Type30_MARSHAL_REF},
737 // TPMS_DERIVE_DATA
738 {STRUCTURE_MTYPE, 2, {
739     SET_ELEMENT_TYPE(SIMPLE_STYPE),
740     TPM2B_LABEL_MARSHAL_REF,
741     (UINT16)(offsetof(TPMS_DERIVE, label)),
742     SET_ELEMENT_TYPE(SIMPLE_STYPE),
743     TPM2B_LABEL_MARSHAL_REF,
744     (UINT16)(offsetof(TPMS_DERIVE, context))}},
745 // TPM2B_DERIVE_DATA
746 {TPM2B_MTYPE, Type31_MARSHAL_REF},
747 // TPM2B_SENSITIVE_DATA_DATA
748 {TPM2B_MTYPE, Type32_MARSHAL_REF},
749 // TPMS_SENSITIVE_CREATE_DATA
750 {STRUCTURE_MTYPE, 2, {

```



```

751     SET_ELEMENT_TYPE(SIMPLE_STYPE),
752     TPM2B_AUTH MARSHAL_REF,
753     (UINT16) (offsetof(TPMS_SENSITIVE_CREATE, userAuth)),
754     SET_ELEMENT_TYPE(SIMPLE_STYPE),
755     TPM2B_SENSITIVE_DATA MARSHAL_REF,
756     (UINT16) (offsetof(TPMS_SENSITIVE_CREATE, data))}},
757 // TPM2B_SENSITIVE_CREATE_DATA
758 {TPM2BS_MTYPE,
759     (UINT8) (offsetof(TPM2B_SENSITIVE_CREATE, sensitive))|SIZE_EQUAL,
760     UINT16_MARSHAL_REF,
761     TPMS_SENSITIVE_CREATE_MARSHAL_REF},
762 // TPMS_SCHEME_HASH_DATA
763 {STRUCTURE_MTYPE, 1, {
764     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
765     TPMI_ALG_HASH MARSHAL_REF,
766     (UINT16) (offsetof(TPMS_SCHEME_HASH, hashAlg))}},
767 // TPMS_SCHEME_ECDSA_DATA
768 {STRUCTURE_MTYPE, 2, {
769     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
770     TPMI_ALG_HASH MARSHAL_REF,
771     (UINT16) (offsetof(TPMS_SCHEME_ECDSA, hashAlg)),
772     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
773     UINT16_MARSHAL_REF,
774     (UINT16) (offsetof(TPMS_SCHEME_ECDSA, count))}},
775 // TPMI_ALG_KEYEDHASH_SCHEME_DATA
776 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
777     {TPM_ALG_NULL,
778     RANGE(5, 10, UINT16),
779     (UINT32) ((ALG_HMAC << 0) | (ALG_XOR << 5))}},
780 // TPMS_SCHEME_XOR_DATA
781 {STRUCTURE_MTYPE, 2, {
782     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
783     TPMI_ALG_HASH MARSHAL_REF,
784     (UINT16) (offsetof(TPMS_SCHEME_XOR, hashAlg)),
785     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
786     TPMI_ALG_KDF MARSHAL_REF|NULL_FLAG,
787     (UINT16) (offsetof(TPMS_SCHEME_XOR, kdf))}},
788 // TPMU_SCHEME_KEYEDHASH_DATA
789 {3, 0, (UINT16) (offsetof(TPMU_SCHEME_KEYEDHASH_mst, marshalingTypes)),
790     {(UINT32)TPM_ALG_HMAC, (UINT32)TPM_ALG_XOR, (UINT32)TPM_ALG_NULL},
791     {(UINT16) (TPMS_SCHEME_HMAC_MARSHAL_REF),
792     (UINT16) (TPMS_SCHEME_XOR_MARSHAL_REF),
793     (UINT16) (UINT0_MARSHAL_REF)}},
794 },
795 // TPMT_KEYEDHASH_SCHEME_DATA
796 {STRUCTURE_MTYPE, 2, {
797     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
798     TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF,
799     (UINT16) (offsetof(TPMT_KEYEDHASH_SCHEME, scheme)),
800     SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
801     TPMU_SCHEME_KEYEDHASH_MARSHAL_REF,
802     (UINT16) (offsetof(TPMT_KEYEDHASH_SCHEME, details))}},
803 // TPMU_SIG_SCHEME_DATA
804 {8, 0, (UINT16) (offsetof(TPMU_SIG_SCHEME_mst, marshalingTypes)),
805     {(UINT32)TPM_ALG_ECDSA, (UINT32)TPM_ALG_RSASSA,
806     (UINT32)TPM_ALG_RSAPSS, (UINT32)TPM_ALG_ECDSA,
807     (UINT32)TPM_ALG_SM2, (UINT32)TPM_ALG_ECSCHNORR,
808     (UINT32)TPM_ALG_HMAC, (UINT32)TPM_ALG_NULL},
809     {(UINT16) (TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF),
810     (UINT16) (TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF),
811     (UINT16) (TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF),
812     (UINT16) (TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF),
813     (UINT16) (TPMS_SIG_SCHEME_SM2_MARSHAL_REF),
814     (UINT16) (TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF),
815     (UINT16) (TPMS_SCHEME_HMAC_MARSHAL_REF),
816     (UINT16) (UINT0_MARSHAL_REF)}},

```

```

817 },
818 // TPMT_SIG_SCHEME_DATA
819 {STRUCTURE_MTYPE, 2, {
820     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
821     TPMI_ALG_SIG_SCHEME MARSHAL_REF,
822     (UINT16) (offsetof(TPMT_SIG_SCHEME, scheme)),
823     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
824     TPMU_SIG_SCHEME MARSHAL_REF,
825     (UINT16) (offsetof(TPMT_SIG_SCHEME, details))}},
826 // TPMU_KDF_SCHEME_DATA
827 {5, 0, (UINT16) (offsetof(TPMU_KDF_SCHEME_mst, marshalingTypes)),
828     { (UINT32) TPM_ALG_MGF1, (UINT32) TPM_ALG_KDF1_SP800_56A,
829       (UINT32) TPM_ALG_KDF2, (UINT32) TPM_ALG_KDF1_SP800_108,
830       (UINT32) TPM_ALG_NULL},
831     { (UINT16) (TPMS_KDF_SCHEME_MGF1 MARSHAL_REF),
832       (UINT16) (TPMS_KDF_SCHEME_KDF1_SP800_56A MARSHAL_REF),
833       (UINT16) (TPMS_KDF_SCHEME_KDF2 MARSHAL_REF),
834       (UINT16) (TPMS_KDF_SCHEME_KDF1_SP800_108 MARSHAL_REF),
835       (UINT16) (UINT0_MARSHAL_REF) }
836 },
837 // TPMT_KDF_SCHEME_DATA
838 {STRUCTURE_MTYPE, 2, {
839     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
840     TPMI_ALG_KDF MARSHAL_REF,
841     (UINT16) (offsetof(TPMT_KDF_SCHEME, scheme)),
842     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
843     TPMU_KDF_SCHEME MARSHAL_REF,
844     (UINT16) (offsetof(TPMT_KDF_SCHEME, details))}},
845 // TPMI_ALG_ASYM_SCHEME_DATA
846 {MIN_MAX_MTYPE, TWO_BYTES | TAKES_NULL | HAS_BITS, (UINT8) TPM_RC_VALUE,
847     { TPM_ALG_NULL,
848       RANGE(20, 29, UINT16),
849       (UINT32) ((ALG_RSASSA << 0) | (ALG_RSAES << 1) | (ALG_RSAPSS << 2) |
850                (ALG_OAEP << 3) | (ALG_ECDSA << 4) | (ALG_ECDH << 5) |
851                (ALG_ECDAA << 6) | (ALG_SM2 << 7) | (ALG_ECSCHNORR << 8) |
852                (ALG_ECMQV << 9))}},
853 // TPMU_ASYM_SCHEME_DATA
854 {11, 0, (UINT16) (offsetof(TPMU_ASYM_SCHEME_mst, marshalingTypes)),
855     { (UINT32) TPM_ALG_ECDH, (UINT32) TPM_ALG_ECMQV,
856       (UINT32) TPM_ALG_ECDAA, (UINT32) TPM_ALG_RSASSA,
857       (UINT32) TPM_ALG_RSAPSS, (UINT32) TPM_ALG_ECDSA,
858       (UINT32) TPM_ALG_SM2, (UINT32) TPM_ALG_ECSCHNORR,
859       (UINT32) TPM_ALG_RSAES, (UINT32) TPM_ALG_OAEP,
860       (UINT32) TPM_ALG_NULL},
861     { (UINT16) (TPMS_KEY_SCHEME_ECDH MARSHAL_REF),
862       (UINT16) (TPMS_KEY_SCHEME_ECMQV MARSHAL_REF),
863       (UINT16) (TPMS_SIG_SCHEME_ECDAA MARSHAL_REF),
864       (UINT16) (TPMS_SIG_SCHEME_RSASSA MARSHAL_REF),
865       (UINT16) (TPMS_SIG_SCHEME_RSAPSS MARSHAL_REF),
866       (UINT16) (TPMS_SIG_SCHEME_ECDSA MARSHAL_REF),
867       (UINT16) (TPMS_SIG_SCHEME_SM2 MARSHAL_REF),
868       (UINT16) (TPMS_SIG_SCHEME_ECSCHNORR MARSHAL_REF),
869       (UINT16) (TPMS_ENC_SCHEME_RSAES MARSHAL_REF),
870       (UINT16) (TPMS_ENC_SCHEME_OAEP MARSHAL_REF),
871       (UINT16) (UINT0_MARSHAL_REF) }
872 },
873 // TPMI_ALG_RSA_SCHEME_DATA
874 {MIN_MAX_MTYPE, TWO_BYTES | TAKES_NULL | HAS_BITS, (UINT8) TPM_RC_VALUE,
875     { TPM_ALG_NULL,
876       RANGE(20, 23, UINT16),
877       (UINT32) ((ALG_RSASSA << 0) | (ALG_RSAES << 1) | (ALG_RSAPSS << 2) | (ALG_OAEP << 3))}},
878 // TPMT_RSA_SCHEME_DATA
879 {STRUCTURE_MTYPE, 2, {
880     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
881     TPMI_ALG_RSA_SCHEME MARSHAL_REF,
882     (UINT16) (offsetof(TPMT_RSA_SCHEME, scheme)),

```

```

883     SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
884     TPMU_ASYM_SCHEME_MARSHAL_REF,
885     (UINT16)(offsetof(TPMT_RSA_SCHEME, details))}},
886 // TPMI_ALG_RSA_DECRYPT_DATA
887 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_VALUE,
888     {TPM_ALG_NULL,
889     RANGE(21, 23, UINT16),
890     (UINT32)((ALG_RSAES << 0)|(ALG_OAEP << 2))}},
891 // TPMT_RSA_DECRYPT_DATA
892 {STRUCTURE_MTYPE, 2, {
893     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
894     TPMI_ALG_RSA_DECRYPT_MARSHAL_REF,
895     (UINT16)(offsetof(TPMT_RSA_DECRYPT, scheme)),
896     SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
897     TPMU_ASYM_SCHEME_MARSHAL_REF,
898     (UINT16)(offsetof(TPMT_RSA_DECRYPT, details))}},
899 // TPM2B_PUBLIC_KEY_RSA_DATA
900 {TPM2B_MTYPE, Type33_MARSHAL_REF},
901 // TPMI_RSA_KEY_BITS_DATA
902 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
903     {3072*RSA_3072, 1024*RSA_1024, 2048*RSA_2048}},
904 // TPM2B_PRIVATE_KEY_RSA_DATA
905 {TPM2B_MTYPE, Type34_MARSHAL_REF},
906 // TPM2B_ECC_PARAMETER_DATA
907 {TPM2B_MTYPE, Type35_MARSHAL_REF},
908 // TPMS_ECC_POINT_DATA
909 {STRUCTURE_MTYPE, 2, {
910     SET_ELEMENT_TYPE(SIMPLE_STYPE),
911     TPM2B_ECC_PARAMETER_MARSHAL_REF,
912     (UINT16)(offsetof(TPMS_ECC_POINT, x)),
913     SET_ELEMENT_TYPE(SIMPLE_STYPE),
914     TPM2B_ECC_PARAMETER_MARSHAL_REF,
915     (UINT16)(offsetof(TPMS_ECC_POINT, y))}},
916 // TPM2B_ECC_POINT_DATA
917 {TPM2BS_MTYPE,
918     (UINT8)(offsetof(TPM2B_ECC_POINT, point))|SIZE_EQUAL,
919     UINT16_MARSHAL_REF,
920     TPMS_ECC_POINT_MARSHAL_REF},
921 // TPMI_ALG_ECC_SCHEME_DATA
922 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
923     {TPM_ALG_NULL,
924     RANGE(24, 29, UINT16),
925     (UINT32)((ALG_ECDSA << 0) | (ALG_ECDH << 1) | (ALG_ECDSA << 2) |
926     (ALG_SM2 << 3) | (ALG_ECSCHNORR << 4) | (ALG_ECMQV << 5))}},
927 // TPMI_ECC_CURVE_DATA
928 {MIN_MAX_MTYPE, TWO_BYTES|HAS_BITS, (UINT8)TPM_RC_CURVE,
929     {RANGE(1, 32, UINT16),
930     (UINT32)((ECC_NIST_P192 << 0) | (ECC_NIST_P224 << 1) | (ECC_NIST_P256 << 2) |
931     (ECC_NIST_P384 << 3) | (ECC_NIST_P521 << 4) | (ECC_BN_P256 << 15) |
932     (ECC_BN_P638 << 16) | (ECC_SM2_P256 << 31))}},
933 // TPMT_ECC_SCHEME_DATA
934 {STRUCTURE_MTYPE, 2, {
935     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
936     TPMI_ALG_ECC_SCHEME_MARSHAL_REF,
937     (UINT16)(offsetof(TPMT_ECC_SCHEME, scheme)),
938     SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
939     TPMU_ASYM_SCHEME_MARSHAL_REF,
940     (UINT16)(offsetof(TPMT_ECC_SCHEME, details))}},
941 // TPMS_ALGORITHM_DETAIL_ECC_DATA
942 {STRUCTURE_MTYPE, 11, {
943     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
944     TPM_ECC_CURVE_MARSHAL_REF,
945     (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, curveID)),
946     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
947     UINT16_MARSHAL_REF,
948     (UINT16)(offsetof(TPMS_ALGORITHM_DETAIL_ECC, keySize)),

```

```

949     SET_ELEMENT_TYPE(SIMPLE_TYPE),
950     TPMT_KDF_SCHEME_MARSHAL_REF|NULL_FLAG,
951     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, kdf)),
952     SET_ELEMENT_TYPE(SIMPLE_TYPE),
953     TPMT_ECC_SCHEME_MARSHAL_REF|NULL_FLAG,
954     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, sign)),
955     SET_ELEMENT_TYPE(SIMPLE_TYPE),
956     TPM2B_ECC_PARAMETER_MARSHAL_REF,
957     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, p)),
958     SET_ELEMENT_TYPE(SIMPLE_TYPE),
959     TPM2B_ECC_PARAMETER_MARSHAL_REF,
960     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, a)),
961     SET_ELEMENT_TYPE(SIMPLE_TYPE),
962     TPM2B_ECC_PARAMETER_MARSHAL_REF,
963     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, b)),
964     SET_ELEMENT_TYPE(SIMPLE_TYPE),
965     TPM2B_ECC_PARAMETER_MARSHAL_REF,
966     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, gX)),
967     SET_ELEMENT_TYPE(SIMPLE_TYPE),
968     TPM2B_ECC_PARAMETER_MARSHAL_REF,
969     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, gY)),
970     SET_ELEMENT_TYPE(SIMPLE_TYPE),
971     TPM2B_ECC_PARAMETER_MARSHAL_REF,
972     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, n)),
973     SET_ELEMENT_TYPE(SIMPLE_TYPE),
974     TPM2B_ECC_PARAMETER_MARSHAL_REF,
975     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, h))}},
976 // TPMS_SIGNATURE_RSA_DATA
977 {STRUCTURE_MTYPE, 2, {
978     SET_ELEMENT_TYPE(SIMPLE_TYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
979     TPMI_ALG_HASH_MARSHAL_REF,
980     (UINT16) (offsetof(TPMS_SIGNATURE_RSA, hash)),
981     SET_ELEMENT_TYPE(SIMPLE_TYPE),
982     TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF,
983     (UINT16) (offsetof(TPMS_SIGNATURE_RSA, sig))}},
984 // TPMS_SIGNATURE_ECC_DATA
985 {STRUCTURE_MTYPE, 3, {
986     SET_ELEMENT_TYPE(SIMPLE_TYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
987     TPMI_ALG_HASH_MARSHAL_REF,
988     (UINT16) (offsetof(TPMS_SIGNATURE_ECC, hash)),
989     SET_ELEMENT_TYPE(SIMPLE_TYPE),
990     TPM2B_ECC_PARAMETER_MARSHAL_REF,
991     (UINT16) (offsetof(TPMS_SIGNATURE_ECC, signatureR)),
992     SET_ELEMENT_TYPE(SIMPLE_TYPE),
993     TPM2B_ECC_PARAMETER_MARSHAL_REF,
994     (UINT16) (offsetof(TPMS_SIGNATURE_ECC, signatureS))}},
995 // TPMU_SIGNATURE_DATA
996 {8, 0, (UINT16) (offsetof(TPMU_SIGNATURE_mst, marshalingTypes)),
997     {(UINT32)TPM_ALG_ECDSA, (UINT32)TPM_ALG_RSASSA,
998     (UINT32)TPM_ALG_RSAPSS, (UINT32)TPM_ALG_ECDSA,
999     (UINT32)TPM_ALG_SM2, (UINT32)TPM_ALG_ECSCHNORR,
1000    (UINT32)TPM_ALG_HMAC, (UINT32)TPM_ALG_NULL},
1001     {(UINT16) (TPMS_SIGNATURE_ECDSA_MARSHAL_REF),
1002     (UINT16) (TPMS_SIGNATURE_RSASSA_MARSHAL_REF),
1003     (UINT16) (TPMS_SIGNATURE_RSAPSS_MARSHAL_REF),
1004     (UINT16) (TPMS_SIGNATURE_ECDSA_MARSHAL_REF),
1005     (UINT16) (TPMS_SIGNATURE_SM2_MARSHAL_REF),
1006     (UINT16) (TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF),
1007     (UINT16) (TPMT_HA_MARSHAL_REF),
1008     (UINT16) (UINT0_MARSHAL_REF)}},
1009 },
1010 // TPMT_SIGNATURE_DATA
1011 {STRUCTURE_MTYPE, 2, {
1012     SET_ELEMENT_TYPE(SIMPLE_TYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
1013     TPMI_ALG_SIG_SCHEME_MARSHAL_REF,
1014     (UINT16) (offsetof(TPMT_SIGNATURE, sigAlg)),

```



```

1015     SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
1016     TPMU_SIGNATURE_MARSHAL_REF,
1017     (UINT16)(offsetof(TPMT_SIGNATURE, signature))}},
1018 // TPMU_ENCRYPTED_SECRET_DATA
1019 {4, IS_ARRAY_UNION, (UINT16)(offsetof(TPMU_ENCRYPTED_SECRET_mst, marshalingTypes)),
1020  { (UINT32)TPM_ALG_ECC, (UINT32)TPM_ALG_RSA,
1021    (UINT32)TPM_ALG_SYMCIPHER, (UINT32)TPM_ALG_KEYEDHASH},
1022  { (UINT16)(sizeof(TPMS_ECC_POINT)), (UINT16)(MAX_RSA_KEY_BYTES),
1023    (UINT16)(sizeof(TPM2B_DIGEST)), (UINT16)(sizeof(TPM2B_DIGEST))}
1024 },
1025 // TPM2B_ENCRYPTED_SECRET_DATA
1026 {TPM2B_MTYPE, Type36_MARSHAL_REF},
1027 // TPMI_ALG_PUBLIC_DATA
1028 {MIN_MAX_MTYPE, TWO_BYTES|HAS_BITS, (UINT8)TPM_RC_TYPE,
1029  {RANGE(1, 37, UINT16),
1030   (UINT32)((ALG_RSA << 0)|(ALG_KEYEDHASH << 7)),
1031   (UINT32)((ALG_ECC << 2)|(ALG_SYMCIPHER << 4))}},
1032 // TPMU_PUBLIC_ID_DATA
1033 {4, 0, (UINT16)(offsetof(TPMU_PUBLIC_ID_mst, marshalingTypes)),
1034  { (UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER,
1035    (UINT32)TPM_ALG_RSA, (UINT32)TPM_ALG_ECC},
1036  { (UINT16)(TPM2B_DIGEST_MARSHAL_REF),
1037    (UINT16)(TPM2B_DIGEST_MARSHAL_REF),
1038    (UINT16)(TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF),
1039    (UINT16)(TPMS_ECC_POINT_MARSHAL_REF)}
1040 },
1041 // TPMS_KEYEDHASH_PARMS_DATA
1042 {STRUCTURE_MTYPE, 1, {
1043     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1044     TPMT_KEYEDHASH_SCHEME_MARSHAL_REF|NULL_FLAG,
1045     (UINT16)(offsetof(TPMS_KEYEDHASH_PARMS, scheme))}},
1046 // TPMS_RSA_PARMS_DATA
1047 {STRUCTURE_MTYPE, 4, {
1048     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1049     TPMT_SYM_DEF_OBJECT_MARSHAL_REF|NULL_FLAG,
1050     (UINT16)(offsetof(TPMS_RSA_PARMS, symmetric)),
1051     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1052     TPMT_RSA_SCHEME_MARSHAL_REF|NULL_FLAG,
1053     (UINT16)(offsetof(TPMS_RSA_PARMS, scheme)),
1054     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1055     TPMI_RSA_KEY_BITS_MARSHAL_REF,
1056     (UINT16)(offsetof(TPMS_RSA_PARMS, keyBits)),
1057     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1058     UINT32_MARSHAL_REF,
1059     (UINT16)(offsetof(TPMS_RSA_PARMS, exponent))}},
1060 // TPMS_ECC_PARMS_DATA
1061 {STRUCTURE_MTYPE, 4, {
1062     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1063     TPMT_SYM_DEF_OBJECT_MARSHAL_REF|NULL_FLAG,
1064     (UINT16)(offsetof(TPMS_ECC_PARMS, symmetric)),
1065     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1066     TPMT_ECC_SCHEME_MARSHAL_REF|NULL_FLAG,
1067     (UINT16)(offsetof(TPMS_ECC_PARMS, scheme)),
1068     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1069     TPMI_ECC_CURVE_MARSHAL_REF,
1070     (UINT16)(offsetof(TPMS_ECC_PARMS, curveID)),
1071     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1072     TPMT_KDF_SCHEME_MARSHAL_REF|NULL_FLAG,
1073     (UINT16)(offsetof(TPMS_ECC_PARMS, kdf))}},
1074 // TPMU_PUBLIC_PARMS_DATA
1075 {4, 0, (UINT16)(offsetof(TPMU_PUBLIC_PARMS_mst, marshalingTypes)),
1076  { (UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER,
1077    (UINT32)TPM_ALG_RSA, (UINT32)TPM_ALG_ECC},
1078  { (UINT16)(TPMS_KEYEDHASH_PARMS_MARSHAL_REF),
1079    (UINT16)(TPMS_SYMCIPHER_PARMS_MARSHAL_REF),
1080    (UINT16)(TPMS_RSA_PARMS_MARSHAL_REF),

```

```

1081         (UINT16) (TPMS_ECC_PARMS_MARSHAL_REF) }
1082     },
1083     // TPMT_PUBLIC_PARMS_DATA
1084     {STRUCTURE_MTYPE, 2, {
1085         SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1086         TPMI_ALG_PUBLIC_MARSHAL_REF,
1087         (UINT16) (offsetof(TPMT_PUBLIC_PARMS, type)),
1088         SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
1089         TPMU_PUBLIC_PARMS_MARSHAL_REF,
1090         (UINT16) (offsetof(TPMT_PUBLIC_PARMS, parameters))}},
1091     // TPMT_PUBLIC_DATA
1092     {STRUCTURE_MTYPE, 6, {
1093         SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1094         TPMI_ALG_PUBLIC_MARSHAL_REF,
1095         (UINT16) (offsetof(TPMT_PUBLIC, type)),
1096         SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
1097         TPMI_ALG_HASH_MARSHAL_REF,
1098         (UINT16) (offsetof(TPMT_PUBLIC, nameAlg)),
1099         SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1100         TPMA_OBJECT_MARSHAL_REF,
1101         (UINT16) (offsetof(TPMT_PUBLIC, objectAttributes)),
1102         SET_ELEMENT_TYPE(SIMPLE_STYPE),
1103         TPM2B_DIGEST_MARSHAL_REF,
1104         (UINT16) (offsetof(TPMT_PUBLIC, authPolicy)),
1105         SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
1106         TPMU_PUBLIC_PARMS_MARSHAL_REF,
1107         (UINT16) (offsetof(TPMT_PUBLIC, parameters)),
1108         SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
1109         TPMU_PUBLIC_ID_MARSHAL_REF,
1110         (UINT16) (offsetof(TPMT_PUBLIC, unique))}},
1111     // TPM2B_PUBLIC_DATA
1112     {TPM2BS_MTYPE,
1113         (UINT8) (offsetof(TPM2B_PUBLIC, publicArea)) | SIZE_EQUAL | ELEMENT_PROPAGATE,
1114         UINT16_MARSHAL_REF,
1115         TPMT_PUBLIC_MARSHAL_REF},
1116     // TPM2B_TEMPLATE_DATA
1117     {TPM2B_MTYPE, Type37_MARSHAL_REF},
1118     // TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA
1119     {TPM2B_MTYPE, Type38_MARSHAL_REF},
1120     // TPMU_SENSITIVE_COMPOSITE_DATA
1121     {4, 0, (UINT16) (offsetof(TPMU_SENSITIVE_COMPOSITE_mst, marshalingTypes)),
1122         {(UINT32) TPM_ALG_RSA, (UINT32) TPM_ALG_ECC,
1123          (UINT32) TPM_ALG_KEYEDHASH, (UINT32) TPM_ALG_SYMCIPHER},
1124         {(UINT16) (TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF),
1125          (UINT16) (TPM2B_ECC_PARAMETER_MARSHAL_REF),
1126          (UINT16) (TPM2B_SENSITIVE_DATA_MARSHAL_REF),
1127          (UINT16) (TPM2B_SYM_KEY_MARSHAL_REF)}},
1128     },
1129     // TPMT_SENSITIVE_DATA
1130     {STRUCTURE_MTYPE, 4, {
1131         SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1132         TPMI_ALG_PUBLIC_MARSHAL_REF,
1133         (UINT16) (offsetof(TPMT_SENSITIVE, sensitiveType)),
1134         SET_ELEMENT_TYPE(SIMPLE_STYPE),
1135         TPM2B_AUTH_MARSHAL_REF,
1136         (UINT16) (offsetof(TPMT_SENSITIVE, authValue)),
1137         SET_ELEMENT_TYPE(SIMPLE_STYPE),
1138         TPM2B_DIGEST_MARSHAL_REF,
1139         (UINT16) (offsetof(TPMT_SENSITIVE, seedValue)),
1140         SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
1141         TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF,
1142         (UINT16) (offsetof(TPMT_SENSITIVE, sensitive))}},
1143     // TPM2B_SENSITIVE_DATA
1144     {TPM2BS_MTYPE,
1145         (UINT8) (offsetof(TPM2B_SENSITIVE, sensitiveArea)),
1146         UINT16_MARSHAL_REF,

```



```

1147     TPMT_SENSITIVE_MARSHAL_REF},
1148 // TPM2B_PRIVATE_DATA
1149 {TPM2B_MTYPE, Type39_MARSHAL_REF},
1150 // TPM2B_ID_OBJECT_DATA
1151 {TPM2B_MTYPE, Type40_MARSHAL_REF},
1152 // TPMS_NV_PIN_COUNTER_PARAMETERS_DATA
1153 {STRUCTURE_MTYPE, 2, {
1154     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1155     UINT32_MARSHAL_REF,
1156     (UINT16)(offsetof(TPMS_NV_PIN_COUNTER_PARAMETERS, pinCount)),
1157     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1158     UINT32_MARSHAL_REF,
1159     (UINT16)(offsetof(TPMS_NV_PIN_COUNTER_PARAMETERS, pinLimit))}},
1160 // TPMA_NV_DATA
1161 {ATTRIBUTES_MTYPE, FOUR_BYTES, 0x01F00300},
1162 // TPMS_NV_PUBLIC_DATA
1163 {STRUCTURE_MTYPE, 5, {
1164     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1165     TPMI_RH_NV_INDEX_MARSHAL_REF,
1166     (UINT16)(offsetof(TPMS_NV_PUBLIC, nvIndex)),
1167     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1168     TPMI_ALG_HASH_MARSHAL_REF,
1169     (UINT16)(offsetof(TPMS_NV_PUBLIC, nameAlg)),
1170     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1171     TPMA_NV_MARSHAL_REF,
1172     (UINT16)(offsetof(TPMS_NV_PUBLIC, attributes)),
1173     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1174     TPM2B_DIGEST_MARSHAL_REF,
1175     (UINT16)(offsetof(TPMS_NV_PUBLIC, authPolicy)),
1176     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
1177     Type41_MARSHAL_REF,
1178     (UINT16)(offsetof(TPMS_NV_PUBLIC, dataSize))}},
1179 // TPM2B_NV_PUBLIC_DATA
1180 {TPM2BS_MTYPE,
1181     (UINT8)(offsetof(TPM2B_NV_PUBLIC, nvPublic))|SIZE_EQUAL,
1182     UINT16_MARSHAL_REF,
1183     TPMS_NV_PUBLIC_MARSHAL_REF},
1184 // TPM2B_CONTEXT_SENSITIVE_DATA
1185 {TPM2B_MTYPE, Type42_MARSHAL_REF},
1186 // TPMS_CONTEXT_DATA_DATA
1187 {STRUCTURE_MTYPE, 2, {
1188     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1189     TPM2B_DIGEST_MARSHAL_REF,
1190     (UINT16)(offsetof(TPMS_CONTEXT_DATA, integrity)),
1191     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1192     TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF,
1193     (UINT16)(offsetof(TPMS_CONTEXT_DATA, encrypted))}},
1194 // TPM2B_CONTEXT_DATA_DATA
1195 {TPM2B_MTYPE, Type43_MARSHAL_REF},
1196 // TPMS_CONTEXT_DATA
1197 {STRUCTURE_MTYPE, 4, {
1198     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(EIGHT_BYTES),
1199     UINT64_MARSHAL_REF,
1200     (UINT16)(offsetof(TPMS_CONTEXT, sequence)),
1201     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1202     TPMI_DH_SAVED_MARSHAL_REF,
1203     (UINT16)(offsetof(TPMS_CONTEXT, savedHandle)),
1204     SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
1205     TPMI_RH_HIERARCHY_MARSHAL_REF|NULL_FLAG,
1206     (UINT16)(offsetof(TPMS_CONTEXT, hierarchy)),
1207     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1208     TPM2B_CONTEXT_DATA_MARSHAL_REF,
1209     (UINT16)(offsetof(TPMS_CONTEXT, contextBlob))}},
1210 // TPMS_CREATION_DATA_DATA
1211 {STRUCTURE_MTYPE, 7, {
1212     SET_ELEMENT_TYPE(SIMPLE_STYPE),

```

```

1213         TPML_PCR_SELECTION_MARSHAL_REF,
1214         (UINT16) (offsetof(TPMS_CREATION_DATA, pcrSelect)),
1215     SET_ELEMENT_TYPE(SIMPLE_TYPE),
1216     TPM2B_DIGEST_MARSHAL_REF,
1217     (UINT16) (offsetof(TPMS_CREATION_DATA, pcrDigest)),
1218     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
1219     TPMA_LOCALITY_MARSHAL_REF,
1220     (UINT16) (offsetof(TPMS_CREATION_DATA, locality)),
1221     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1222     TPM_ALG_ID_MARSHAL_REF,
1223     (UINT16) (offsetof(TPMS_CREATION_DATA, parentNameAlg)),
1224     SET_ELEMENT_TYPE(SIMPLE_TYPE),
1225     TPM2B_NAME_MARSHAL_REF,
1226     (UINT16) (offsetof(TPMS_CREATION_DATA, parentName)),
1227     SET_ELEMENT_TYPE(SIMPLE_TYPE),
1228     TPM2B_NAME_MARSHAL_REF,
1229     (UINT16) (offsetof(TPMS_CREATION_DATA, parentQualifiedName)),
1230     SET_ELEMENT_TYPE(SIMPLE_TYPE),
1231     TPM2B_DATA_MARSHAL_REF,
1232     (UINT16) (offsetof(TPMS_CREATION_DATA, outsideInfo))},
1233 // TPM2B_CREATION_DATA_DATA
1234 {TPM2BS_MTYPE,
1235     (UINT8) (offsetof(TPM2B_CREATION_DATA, creationData)) | SIZE_EQUAL,
1236     UINT16_MARSHAL_REF,
1237     TPMS_CREATION_DATA_MARSHAL_REF},
1238 // TPM_AT_DATA
1239 {TABLE_MTYPE, FOUR_BYTES, (UINT8) TPM_RC_VALUE, 4,
1240     {TPM_AT_ANY, TPM_AT_ERROR, TPM_AT_PV1, TPM_AT_VEND}},
1241 // TPMS_AC_OUTPUT_DATA
1242 {STRUCTURE_MTYPE, 2, {
1243     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1244     TPM_AT_MARSHAL_REF,
1245     (UINT16) (offsetof(TPMS_AC_OUTPUT, tag)),
1246     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1247     UINT32_MARSHAL_REF,
1248     (UINT16) (offsetof(TPMS_AC_OUTPUT, data))}},
1249 // TPML_AC_CAPABILITIES_DATA
1250 {LIST_MTYPE,
1251     (UINT8) (offsetof(TPML_AC_CAPABILITIES, acCapabilities)),
1252     Type44_MARSHAL_REF,
1253     TPMS_AC_OUTPUT_ARRAY_MARSHAL_INDEX},
1254 // Type00_DATA
1255 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1256     {RANGE(0, sizeof(TPMU_HA), UINT16)}},
1257 // Type01_DATA
1258 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1259     {RANGE(0, sizeof(TPMT_HA), UINT16)}},
1260 // Type02_DATA
1261 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1262     {RANGE(0, 1024, UINT16)}},
1263 // Type03_DATA
1264 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1265     {RANGE(0, MAX_DIGEST_BUFFER, UINT16)}},
1266 // Type04_DATA
1267 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1268     {RANGE(0, MAX_NV_BUFFER_SIZE, UINT16)}},
1269 // Type05_DATA
1270 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1271     {RANGE(0, sizeof(UINT64), UINT16)}},
1272 // Type06_DATA
1273 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1274     {RANGE(0, MAX_SYM_BLOCK_SIZE, UINT16)}},
1275 // Type07_DATA
1276 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1277     {RANGE(0, sizeof(TPMU_NAME), UINT16)}},
1278 // Type08_DATA

```

```

1279 {MIN_MAX_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE,
1280 {RANGE(PCR_SELECT_MIN, PCR_SELECT_MAX, UINT8)}}},
1281 // Type10_DATA
1282 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1283 {TPM_ST_CREATION}}},
1284 // Type11_DATA
1285 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1286 {TPM_ST_VERIFIED}}},
1287 // Type12_DATA
1288 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 2,
1289 {TPM_ST_AUTH_SECRET, TPM_ST_AUTH_SIGNED}}},
1290 // Type13_DATA
1291 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1292 {TPM_ST_HASHCHECK}}},
1293 // Type15_DATA
1294 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1295 {RANGE(0, MAX_CAP_CC, UINT32)}}},
1296 // Type17_DATA
1297 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1298 {RANGE(0, MAX_ALG_LIST_SIZE, UINT32)}}},
1299 // Type18_DATA
1300 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1301 {RANGE(0, MAX_CAP_HANDLES, UINT32)}}},
1302 // Type19_DATA
1303 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1304 {RANGE(2, 8, UINT32)}}},
1305 // Type20_DATA
1306 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1307 {RANGE(0, HASH_COUNT, UINT32)}}},
1308 // Type22_DATA
1309 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1310 {RANGE(0, MAX_CAP_ALGS, UINT32)}}},
1311 // Type23_DATA
1312 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1313 {RANGE(0, MAX_TPM_PROPERTIES, UINT32)}}},
1314 // Type24_DATA
1315 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1316 {RANGE(0, MAX_PCR_PROPERTIES, UINT32)}}},
1317 // Type25_DATA
1318 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1319 {RANGE(0, MAX_ECC_CURVES, UINT32)}}},
1320 // Type26_DATA
1321 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1322 {RANGE(0, MAX_TAGGED_POLICIES, UINT32)}}},
1323 // Type27_DATA
1324 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1325 {RANGE(0, MAX_ACT_DATA, UINT32)}}},
1326 // Type28_DATA
1327 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1328 {RANGE(0, sizeof(TPMS_ATTEST), UINT16)}}},
1329 // Type29_DATA
1330 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1331 {RANGE(0, MAX_SYM_KEY_BYTES, UINT16)}}},
1332 // Type30_DATA
1333 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1334 {RANGE(0, LABEL_MAX_BUFFER, UINT16)}}},
1335 // Type31_DATA
1336 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1337 {RANGE(0, sizeof(TPMS_DERIVE), UINT16)}}},
1338 // Type32_DATA
1339 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1340 {RANGE(0, sizeof(TPMU_SENSITIVE_CREATE), UINT16)}}},
1341 // Type33_DATA
1342 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1343 {RANGE(0, MAX_RSA_KEY_BYTES, UINT16)}}},
1344 // Type34_DATA

```

```

1345 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1346   {RANGE(0, RSA_PRIVATE_SIZE, UINT16)}}},
1347 // Type35_DATA
1348 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1349   {RANGE(0, MAX_ECC_KEY_BYTES, UINT16)}}},
1350 // Type36_DATA
1351 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1352   {RANGE(0, sizeof(TPMU_ENCRYPTED_SECRET), UINT16)}}},
1353 // Type37_DATA
1354 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1355   {RANGE(0, sizeof(TPMT_PUBLIC), UINT16)}}},
1356 // Type38_DATA
1357 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1358   {RANGE(0, PRIVATE_VENDOR_SPECIFIC_BYTES, UINT16)}}},
1359 // Type39_DATA
1360 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1361   {RANGE(0, sizeof(_PRIVATE), UINT16)}}},
1362 // Type40_DATA
1363 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1364   {RANGE(0, sizeof(TPMS_ID_OBJECT), UINT16)}}},
1365 // Type41_DATA
1366 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1367   {RANGE(0, MAX_NV_INDEX_SIZE, UINT16)}}},
1368 // Type42_DATA
1369 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1370   {RANGE(0, MAX_CONTEXT_SIZE, UINT16)}}},
1371 // Type43_DATA
1372 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1373   {RANGE(0, sizeof(TPMS_CONTEXT_DATA), UINT16)}}},
1374 // Type44_DATA
1375 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1376   {RANGE(0, MAX_AC_CAPABILITIES, UINT32)}}},
1377 };
1378 #endif // TABLE_DRIVEN_MARSHAL
1379 // clang-format on

```

7.198 /tpm/src/support/TpmFail.c

```

1  /** Includes, Defines, and Types
2  #define TPM_FAIL_C
3  #include "Tpm.h"
4
5  // On MS C compiler, can save the alignment state and set the alignment to 1 for
6  // the duration of the TpmTypes.h include. This will avoid a lot of alignment
7  // warnings from the compiler for the unaligned structures. The alignment of the
8  // structures is not important as this function does not use any of the structures
9  // in TpmTypes.h and only include it for the #defines of the capabilities,
10 // properties, and command code values.
11 #include "TpmTypes.h"
12
13 /** Typedefs
14 // These defines are used primarily for sizing of the local response buffer.
15 typedef struct
16 {
17     TPM_ST tag;
18     UINT32 size;
19     TPM_RC code;
20 } HEADER;
21
22 typedef struct
23 {
24     BYTE tag[sizeof(TPM_ST)];
25     BYTE size[sizeof(UINT32)];
26     BYTE code[sizeof(TPM_RC)];
27 } PACKED_HEADER;

```

```

28
29 typedef struct
30 {
31     BYTE size[sizeof(UINT16)];
32     struct
33     {
34         BYTE function[sizeof(UINT32)];
35         BYTE line[sizeof(UINT32)];
36         BYTE code[sizeof(UINT32)];
37     } values;
38     BYTE returnCode[sizeof(TPM_RC)];
39 } GET_TEST_RESULT_PARAMETERS;
40
41 typedef struct
42 {
43     BYTE moreData[sizeof(TPMI_YES_NO)];
44     BYTE capability[sizeof(TPM_CAP)]; // Always TPM_CAP_TPM_PROPERTIES
45     BYTE tpmProperty[sizeof(TPML_TAGGED_TPM_PROPERTY)];
46 } GET_CAPABILITY_PARAMETERS;
47
48 typedef struct
49 {
50     BYTE header[sizeof(PACKED_HEADER)];
51     BYTE getTestResult[sizeof(GET_TEST_RESULT_PARAMETERS)];
52 } TEST_RESPONSE;
53
54 typedef struct
55 {
56     BYTE header[sizeof(PACKED_HEADER)];
57     BYTE getCap[sizeof(GET_CAPABILITY_PARAMETERS)];
58 } CAPABILITY_RESPONSE;
59
60 typedef union
61 {
62     BYTE test[sizeof(TEST_RESPONSE)];
63     BYTE cap[sizeof(CAPABILITY_RESPONSE)];
64 } RESPONSES;
65
66 // Buffer to hold the responses. This may be a little larger than
67 // required due to padding that a compiler might add.
68 // Note: This is not in Global.c because of the specialized data definitions above.
69 // Since the data contained in this structure is not relevant outside of the
70 // execution of a single command (when the TPM is in failure mode. There is no
71 // compelling reason to move all the typedefs to Global.h and this structure
72 // to Global.c.
73 #ifndef __IGNORE_STATE__ // Don't define this value
74 static BYTE response[sizeof(RESPONSES)];
75 #endif
76
77 /** Local Functions
78
79 **** MarshalUint16()
80 // Function to marshal a 16 bit value to the output buffer.
81 static INT32 MarshalUint16(UINT16 integer, BYTE** buffer)
82 {
83     UINT16_TO_BYTE_ARRAY(integer, *buffer);
84     *buffer += 2;
85     return 2;
86 }
87
88 **** MarshalUint32()
89 // Function to marshal a 32 bit value to the output buffer.
90 static INT32 MarshalUint32(UINT32 integer, BYTE** buffer)
91 {
92     UINT32_TO_BYTE_ARRAY(integer, *buffer);
93     *buffer += 4;

```

```

94     return 4;
95 }
96
97 /***Unmarshal32()
98 static BOOL Unmarshal32(UINT32* target, BYTE** buffer, INT32* size)
99 {
100     if((*size -= 4) < 0)
101         return FALSE;
102     *target = BYTE_ARRAY_TO_UINT32(*buffer);
103     *buffer += 4;
104     return TRUE;
105 }
106
107 /***Unmarshal16()
108 static BOOL Unmarshal16(UINT16* target, BYTE** buffer, INT32* size)
109 {
110     if((*size -= 2) < 0)
111         return FALSE;
112     *target = BYTE_ARRAY_TO_UINT16(*buffer);
113     *buffer += 2;
114     return TRUE;
115 }
116
117 /*** Public Functions
118
119 /*** SetForceFailureMode()
120 // This function is called by the simulator to enable failure mode testing.
121 #if ALLOW_FORCE_FAILURE_MODE
122 LIB_EXPORT void SetForceFailureMode(void)
123 {
124     g_forceFailureMode = TRUE;
125     return;
126 }
127 #endif // ALLOW_FORCE_FAILURE_MODE
128
129 /*** TpmFail()
130 // This function is called by TPM.lib when a failure occurs. It will set up the
131 // failure values to be returned on TPM2_GetTestResult().
132 NORETURN void TpmFail(
133 #if FAIL_TRACE
134     const char* function,
135     int line,
136 #else
137     uint64_t locationCode,
138 #endif
139     int failureCode)
140 {
141     // Save the values that indicate where the error occurred.
142     // On a 64-bit machine, this may truncate the address of the string
143     // of the function name where the error occurred.
144     #if FAIL_TRACE
145         s_failFunctionName = function;
146         s_failFunction      = (UINT32) (ptrdiff_t) function;
147         s_failLine          = line;
148     #else
149         s_failFunction = (UINT32) (locationCode >> 32);
150         s_failLine     = (UINT32) (locationCode);
151     #endif
152     s_failCode = failureCode;
153
154     // We are in failure mode
155     g_inFailureMode = TRUE;
156
157     // Notify the platform that we hit a failure.
158     //
159     // In the LONGJMP case, the reference platform code is expected to long-jmp

```



```

160     // back to the ExecuteCommand call and output a failure response.
161     //
162     // In the NO_LONGJMP case, this is a notification to the platform, and the
163     // platform may take any (implementation-defined) behavior, including no-op,
164     // debugging, or whatever. The core library is expected to surface the failure
165     // back to ExecuteCommand through error propagation and return an appropriate
166     // failure reply.
167     _plat_Fail();
168 }
169
170 /*** TpmFailureMode(
171 // This function is called by the interface code when the platform is in failure
172 // mode.
173 void TpmFailureMode(uint32_t      inRequestSize,    // IN: command buffer size
174                     unsigned char* inRequest,      // IN: command buffer
175                     uint32_t*      outResponseSize, // OUT: response buffer size
176                     unsigned char** outResponse    // OUT: response buffer
177 )
178 {
179     UINT32 marshalSize;
180     UINT32 capability;
181     HEADER header; // unmarshaled command header
182     UINT32 pt;     // unmarshaled property type
183     UINT32 count;  // unmarshaled property count
184     UINT8* buffer = inRequest;
185     INT32 size = inRequestSize;
186
187     // If there is no command buffer, then just return TPM_RC_FAILURE
188     if(inRequestSize == 0 || inRequest == NULL)
189         goto FailureModeReturn;
190     // If the header is not correct for TPM2_GetCapability() or
191     // TPM2_GetTestResult() then just return the in failure mode response;
192     if(!(Unmarshall16(&header.tag, &buffer, &size)
193         && Unmarshal32(&header.size, &buffer, &size)
194         && Unmarshal32(&header.code, &buffer, &size)))
195         goto FailureModeReturn;
196     if(header.tag != TPM_ST_NO_SESSIONS || header.size < 10)
197         goto FailureModeReturn;
198     switch(header.code)
199     {
200     case TPM_CC_GetTestResult:
201         // make sure that the command size is correct
202         if(header.size != 10)
203             goto FailureModeReturn;
204         buffer = &response[10];
205         marshalSize = MarshalUint16(3 * sizeof(UINT32), &buffer);
206         marshalSize += MarshalUint32(s_failFunction, &buffer);
207         marshalSize += MarshalUint32(s_failLine, &buffer);
208         marshalSize += MarshalUint32(s_failCode, &buffer);
209         if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
210             marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
211         else
212             marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
213         break;
214     case TPM_CC_GetCapability:
215         // make sure that the size of the command is exactly the size
216         // returned for the capability, property, and count
217         if(header.size != (10 + (3 * sizeof(UINT32))))
218             // also verify that this is requesting TPM properties
219             || !Unmarshal32(&capability, &buffer, &size)
220             || capability != TPM_CAP_TPM_PROPERTIES
221             || !Unmarshal32(&pt, &buffer, &size)
222             || !Unmarshal32(&count, &buffer, &size))
223             goto FailureModeReturn;
224
225     if(count > 0)

```

```

226         count = 1;
227     else if(pt > TPM_PT_FIRMWARE_VERSION_2)
228         count = 0;
229     if(pt < TPM_PT_MANUFACTURER)
230         pt = TPM_PT_MANUFACTURER;
231     // set up for return
232     buffer = &response[10];
233     // if the request was for a PT less than the last one
234     // then we indicate more, otherwise, not.
235     if(pt < TPM_PT_FIRMWARE_VERSION_2)
236         *buffer++ = YES;
237     else
238         *buffer++ = NO;
239     marshalSize = 1;
240
241     // indicate the capability type
242     marshalSize += MarshalUint32(capability, &buffer);
243     // indicate the number of values that are being returned (0 or 1)
244     marshalSize += MarshalUint32(count, &buffer);
245     // indicate the property
246     marshalSize += MarshalUint32(pt, &buffer);
247
248     if(count > 0)
249         switch(pt)
250         {
251             case TPM_PT_MANUFACTURER:
252                 // the vendor ID unique to each TPM manufacturer
253                 pt = _plat__GetManufacturerCapabilityCode();
254                 break;
255
256             case TPM_PT_VENDOR_STRING_1:
257                 // the first four characters of the vendor ID string
258                 pt = _plat__GetVendorCapabilityCode(1);
259                 break;
260
261             case TPM_PT_VENDOR_STRING_2:
262                 // the second four characters of the vendor ID string
263                 pt = _plat__GetVendorCapabilityCode(2);
264                 break;
265
266             case TPM_PT_VENDOR_STRING_3:
267                 // the third four characters of the vendor ID string
268                 pt = _plat__GetVendorCapabilityCode(3);
269                 break;
270
271             case TPM_PT_VENDOR_STRING_4:
272                 // the fourth four characters of the vendor ID string
273                 pt = _plat__GetVendorCapabilityCode(4);
274                 break;
275
276             case TPM_PT_VENDOR_TPM_TYPE:
277                 // vendor-defined value indicating the TPM model
278                 // We just make up a number here
279                 pt = _plat__GetTpmType();
280                 break;
281
282             case TPM_PT_FIRMWARE_VERSION_1:
283                 // the more significant 32-bits of a vendor-specific value
284                 // indicating the version of the firmware
285                 pt = _plat__GetTpmFirmwareVersionHigh();
286                 break;
287
288             default: // TPM_PT_FIRMWARE_VERSION_2:
289                 // the less significant 32-bits of a vendor-specific value
290                 // indicating the version of the firmware
291                 pt = _plat__GetTpmFirmwareVersionLow();

```

```

292         break;
293     }
294     marshalSize += MarshalUint32(pt, &buffer);
295     break;
296     default: // default for switch (cc)
297         goto FailureModeReturn;
298 }
299 // Now do the header
300 buffer = response;
301 marshalSize = marshalSize + 10; // Add the header size to the
302                                // stuff already marshaled
303 MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
304 MarshalUint32(marshalSize, &buffer); // responseSize
305 MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code
306
307 *outResponseSize = marshalSize;
308 *outResponse = (unsigned char*)&response;
309 return;
310 FailureModeReturn:
311 buffer = response;
312 marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
313 marshalSize += MarshalUint32(10, &buffer);
314 marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
315 *outResponseSize = marshalSize;
316 *outResponse = (unsigned char*)response;
317 return;
318 }
319
320 /** UnmarshalFail()
321 // This is a stub that is used to catch an attempt to unmarshal an entry
322 // that is not defined. Don't ever expect this to be called but...
323 void UnmarshalFail(void* type, BYTE** buffer, INT32* size)
324 {
325     NOT_REFERENCED(type);
326     NOT_REFERENCED(buffer);
327     NOT_REFERENCED(size);
328     FAIL(FATAL_ERROR_INTERNAL);
329 }

```

7.199 /tpm/src/support/TpmSizeChecks.c

```

1  /** Includes, Defines, and Types
2  #include "Tpm.h"
3  #include <stdio.h>
4  #include <assert.h>
5  #include "Marshal.h"
6
7  #if RUNTIME_SIZE_CHECKS
8
9  # if DEBUG
10 static int once = 0;
11 # endif
12
13 /** TpmSizeChecks()
14 // This function is used during the development process to make sure that the
15 // vendor-specific values result in a consistent implementation. When possible,
16 // the code contains #if to do compile-time checks. However, in some cases, the
17 // values require the use of "sizeof()" and that can't be used in an #if.
18 BOOL TpmSizeChecks(void)
19 {
20     BOOL PASS = TRUE;
21
22     # if DEBUG
23     //
24     if(once++ != 0)

```

```

25         return 1;
26
27 #   if ALG_ECC
28 {
29     // This is just to allow simple access to the ecc curve data during debug
30     const TPM_ECC_CURVE_METADATA* ecc = CryptEccGetParametersByCurveId(3);
31     if(ecc == NULL)
32         ecc = NULL;
33 }
34 #   endif // ALG_ECC
35 {
36     UINT32 maxAsymSecurityStrength = MAX_ASYM_SECURITY_STRENGTH;
37     UINT32 maxHashSecurityStrength = MAX_HASH_SECURITY_STRENGTH;
38     UINT32 maxSymSecurityStrength = MAX_SYM_SECURITY_STRENGTH;
39     UINT32 maxSecurityStrengthBits = MAX_SECURITY_STRENGTH_BITS;
40     UINT32 proofSize = PROOF_SIZE;
41     UINT32 compliantProofSize = COMPLIANT_PROOF_SIZE;
42     UINT32 compliantPrimarySeedSize = COMPLIANT_PRIMARY_SEED_SIZE;
43     UINT32 primarySeedSize = PRIMARY_SEED_SIZE;
44
45     UINT32 cmacState = sizeof(tpmCmacState_t);
46     UINT32 hashState = sizeof(HASH_STATE);
47     UINT32 keyScheduleSize = sizeof(tpmCryptKeySchedule_t);
48     //
49     NOT_REFERENCED(cmacState);
50     NOT_REFERENCED(hashState);
51     NOT_REFERENCED(keyScheduleSize);
52     NOT_REFERENCED(maxAsymSecurityStrength);
53     NOT_REFERENCED(maxHashSecurityStrength);
54     NOT_REFERENCED(maxSymSecurityStrength);
55     NOT_REFERENCED(maxSecurityStrengthBits);
56     NOT_REFERENCED(proofSize);
57     NOT_REFERENCED(compliantProofSize);
58     NOT_REFERENCED(compliantPrimarySeedSize);
59     NOT_REFERENCED(primarySeedSize);
60
61 #   if ALG_RSA
62 {
63     TPMT_SENSITIVE* p;
64     // This assignment keeps compiler from complaining about a conditional
65     // comparison being between two constants
66     UINT16 max_rsa_key_bytes = MAX_RSA_KEY_BYTES;
67     if((max_rsa_key_bytes / 2) != (sizeof(p->sensitive.rsa.t.buffer) / 5))
68     {
69         printf("Sensitive part of TPMT_SENSITIVE is undersized. May be "
70             "caused"
71             " by use of wrong version of Part 2.\n");
72         PASS = FALSE;
73     }
74 }
75 #   endif // ALG_RSA
76 #   if TABLE_DRIVEN_MARSHAL
77     printf("sizeof(MarshalData) = %zu\n", sizeof(MarshalData_st));
78 #   endif
79
80     printf("Size of OBJECT = %zu\n", sizeof(OBJECT));
81     printf("Size of components in TPMT_SENSITIVE = %zu\n",
82         sizeof(TPMT_SENSITIVE));
83     printf("    TPMI_ALG_PUBLIC %zu\n", sizeof(TPMI_ALG_PUBLIC));
84     printf("    TPM2B_AUTH %zu\n", sizeof(TPM2B_AUTH));
85     printf("    TPM2B_DIGEST %zu\n", sizeof(TPM2B_DIGEST));
86     printf("    TPMU_SENSITIVE_COMPOSITE %zu\n",
87         sizeof(TPMU_SENSITIVE_COMPOSITE));
88 }
89 // Make sure that the size of the context blob is large enough for the largest
90 // context

```

```

91 // TPMS_CONTEXT_DATA contains two TPM2B values. That is not how this is
92 // implemented. Rather, the size field of the TPM2B_CONTEXT_DATA is used to
93 // determine the amount of data in the encrypted data. That part is not
94 // independently sized. This makes the actual size 2 bytes smaller than
95 // calculated using Part 2. Since this is opaque to the caller, it is not
96 // necessary to fix. The actual size is returned by TPM2_GetCapabilities().
97
98 // Initialize output handle. At the end of command action, the output
99 // handle of an object will be replaced, while the output handle
100 // for a session will be the same as input
101
102 // Get the size of fingerprint in context blob. The sequence value in
103 // TPMS_CONTEXT structure is used as the fingerprint
104 {
105     UINT32 fingerprintSize = sizeof(UINT64);
106     UINT32 integritySize =
107         sizeof(UINT16) + CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
108     UINT32 biggestObject =
109         MAX(MAX(sizeof(HASH_OBJECT), sizeof(OBJECT)), sizeof(SESSION));
110     UINT32 biggestContext = fingerprintSize + integritySize + biggestObject;
111
112     // round required size up to nearest 8 byte boundary.
113     biggestContext = 8 * ((biggestContext + 7) / 8);
114
115     if(MAX_CONTEXT_SIZE < biggestContext)
116     {
117         printf("MAX_CONTEXT_SIZE needs to be increased to at least %d (%d)\n",
118             biggestContext,
119             MAX_CONTEXT_SIZE);
120         PASS = FALSE;
121     }
122     else if(MAX_CONTEXT_SIZE > biggestContext)
123     {
124         printf("MAX_CONTEXT_SIZE can be reduced to %d (%d)\n",
125             biggestContext,
126             MAX_CONTEXT_SIZE);
127     }
128 }
129 {
130     union u
131     {
132         TPMA_OBJECT attributes;
133         UINT32 uint32Value;
134     } u;
135     // these are defined so that compiler doesn't complain about conditional
136     // expressions comparing two constants.
137     int aSize = sizeof(u.attributes);
138     int uSize = sizeof(u.uint32Value);
139     u.uint32Value = 0;
140     SET_ATTRIBUTE(u.attributes, TPMA_OBJECT, fixedTPM);
141     if(u.uint32Value != 2)
142     {
143         printf("The bit allocation in a TPMA_OBJECT is not as expected");
144         PASS = FALSE;
145     }
146     if(aSize != uSize) // comparison of two sizeof() values annoys compiler
147     {
148         printf("A TPMA_OBJECT is not the expected size.");
149         PASS = FALSE;
150     }
151 }
152 # if ACT_SUPPORT
153 // Check that the platform implements each of the ACT that the TPM thinks are
154 // present
155 {
156     uint32_t act;

```

```

157     for(act = 0; act < 16; act++)
158     {
159         switch(act)
160         {
161             FOR_EACH_ACT(CASE_ACT_NUMBER)
162             if(!_plat__ACT_GetImplemented(act))
163             {
164                 printf("TPM_RH_ACT_%1X is not implemented by platform\n", act);
165                 PASS = FALSE;
166             }
167             default:
168                 break;
169         }
170     }
171 }
172 # endif // ACT_SUPPORT
173 {
174     // Had a problem with the macros coming up with some bad values. Make sure
175     // the size is rational
176     int t = MAX_DIGEST_SIZE;
177     if(t < 20)
178     {
179         printf("Check the MAX_DIGEST_SIZE computation (%d)", MAX_DIGEST_SIZE);
180         PASS = FALSE;
181     }
182 }
183 # endif // DEBUG
184 return (PASS);
185 }
186
187 #endif // RUNTIME_SIZE_CHECKS

```

7.200 /tpm/src/X509/TpmASN1.c

```

1  /** Includes
2  #include "Tpm.h"
3  #define _OIDS_
4  #include "OIDS.h"
5  #include "TpmASN1.h"
6  #include "TpmASN1_fp.h"
7
8  #if CC_CertifyX509
9
10 /** Unmarshaling Functions
11
12 /*** ASN1UnmarshalContextInitialize()
13 // Function does standard initialization of a context.
14 // Return Type: BOOL
15 //     TRUE(1)    success
16 //     FALSE(0)   failure
17 BOOL ASN1UnmarshalContextInitialize(
18     ASN1UnmarshalContext* ctx, INT16 size, BYTE* buffer)
19 {
20     GOTO_ERROR_UNLESS(buffer != NULL);
21     GOTO_ERROR_UNLESS(size > 0);
22     ctx->buffer = buffer;
23     ctx->size = size;
24     ctx->offset = 0;
25     ctx->tag = 0xFF;
26     return TRUE;
27 Error:
28     return FALSE;
29 }
30
31 /***ASN1DecodeLength()

```



```

32 // This function extracts the length of an element from 'buffer' starting at 'offset'.
33 // Return Type: UINT16
34 //     >=0      the extracted length
35 //     <0      an error
36 INT16
37 ASN1DecodeLength(ASN1UnmarshalContext* ctx)
38 {
39     BYTE first; // Next octet in buffer
40     INT16 value;
41     //
42     GOTO_ERROR_UNLESS(ctx->offset < ctx->size);
43     first = NEXT_OCTET(ctx);
44     // If the number of octets of the entity is larger than 127, then the first octet
45     // is the number of octets in the length specifier.
46     if(first >= 0x80)
47     {
48         // Make sure that this length field is contained with the structure being
49         // parsed
50         CHECK_SIZE(ctx, (first & 0x7F));
51         if(first == 0x82)
52         {
53             // Two octets of size
54             // get the next value
55             value = (INT16)NEXT_OCTET(ctx);
56             // Make sure that the result will fit in an INT16
57             GOTO_ERROR_UNLESS(value < 0x0080);
58             // Shift up and add next octet
59             value = (value << 8) + NEXT_OCTET(ctx);
60         }
61         else if(first == 0x81)
62             value = NEXT_OCTET(ctx);
63         // Sizes larger than will fit in a INT16 are an error
64         else
65             goto Error;
66     }
67     else
68         value = first;
69     // Make sure that the size defined something within the current context
70     CHECK_SIZE(ctx, value);
71     return value;
72 Error:
73     ctx->size = -1; // Makes everything fail from now on.
74     return -1;
75 }
76
77 /**ASN1NextTag()
78 // This function extracts the next type from 'buffer' starting at 'offset'.
79 // It advances 'offset' as it parses the type and the length of the type. It returns
80 // the length of the type. On return, the 'length' octets starting at 'offset' are the
81 // octets of the type.
82 // Return Type: UINT
83 //     >=0      the number of octets in 'type'
84 //     <0      an error
85 INT16
86 ASN1NextTag(ASN1UnmarshalContext* ctx)
87 {
88     // A tag to get?
89     GOTO_ERROR_UNLESS(ctx->offset < ctx->size);
90     // Get it
91     ctx->tag = NEXT_OCTET(ctx);
92     // Make sure that it is not an extended tag
93     GOTO_ERROR_UNLESS((ctx->tag & 0x1F) != 0x1F);
94     // Get the length field and return that
95     return ASN1DecodeLength(ctx);
96
97 Error:

```

```

98     // Attempt to read beyond the end of the context or an illegal tag
99     ctx->size = -1; // Persistent failure
100    ctx->tag = 0xFF;
101    return -1;
102 }
103
104 /*** ASN1GetBitStringValue()
105 // Try to parse a bit string of up to 32 bits from a value that is expected to be
106 // a bit string. The bit string is left justified so that the MSb of the input is
107 // the MSb of the returned value.
108 // If there is a general parsing error, the context->size is set to -1.
109 // Return Type: BOOL
110 //     TRUE(1)      success
111 //     FALSE(0)     failure
112 BOOL ASN1GetBitStringValue(ASN1UnmarshalContext* ctx, UINT32* val)
113 {
114     int      shift;
115     INT16    length;
116     UINT32   value = 0;
117     int      inputBits;
118     //
119     length = ASN1NextTag(ctx);
120     GOTO_ERROR_UNLESS(length >= 1);
121     GOTO_ERROR_UNLESS(ctx->tag == ASN1_BITSTRING);
122     // Get the shift value for the bit field (how many bits to lop off of the end)
123     shift = NEXT_OCTET(ctx);
124     length--;
125     // Get the number of bits in the input
126     inputBits = (8 * length) - shift;
127     // the shift count has to make sense
128     GOTO_ERROR_UNLESS((shift < 8) && ((length > 0) || (shift == 0)));
129     // if there are any bytes left
130     for(; length > 1; length--)
131     {
132         // for all but the last octet, just shift and add the new octet
133         GOTO_ERROR_UNLESS((value & 0xFF000000) == 0); // can't loose significant bits
134         value = (value << 8) + NEXT_OCTET(ctx);
135     }
136     if(length == 1)
137     {
138         // for the last octet, just shift the accumulated value enough to
139         // accept the significant bits in the last octet and shift the last
140         // octet down
141         GOTO_ERROR_UNLESS(((value & (0xFF000000 << (8 - shift)))) == 0);
142         value = (value << (8 - shift)) + (NEXT_OCTET(ctx) >> shift);
143     }
144     // 'Left justify' the result
145     if(inputBits > 0)
146         value <<= (32 - inputBits);
147     *val = value;
148     return TRUE;
149 }
150 Error:
151     ctx->size = -1;
152     return FALSE;
153 }
154
155 /*** Marshaling Functions
156
157 /*** Introduction
158 // Marshaling of an ASN.1 structure is accomplished from the bottom up. That is,
159 // the things that will be at the end of the structure are added last. To manage the
160 // collecting of the relative sizes, start a context for the outermost container, if
161 // there is one, and then placing items in from the bottom up. If the bottom-most

```

```

164 // item is also within a structure, create a nested context by calling
165 // ASN1StartMarshalingContext().
166 //
167 // The context control structure contains a 'buffer' pointer, an 'offset', an 'end'
168 // and a stack. 'offset' is the offset from the start of the buffer of the last added
169 // byte. When 'offset' reaches 0, the buffer is full. 'offset' is a signed value so
170 // that, when it becomes negative, there is an overflow. Only two functions are
171 // allowed to move bytes into the buffer: ASN1PushByte() and ASN1PushBytes(). These
172 // functions make sure that no data is written beyond the end of the buffer.
173 //
174 // When a new context is started, the current value of 'end' is pushed
175 // on the stack and 'end' is set to 'offset'. As bytes are added, offset gets smaller.
176 // At any time, the count of bytes in the current context is simply 'end' - 'offset'.
177 //
178 // Since starting a new context involves setting 'end' = 'offset', the number of bytes
179 // in the context starts at 0. The nominal way of ending a context is to use
180 // 'end' - 'offset' to set the length value, and then a tag is added to the buffer.
181 // Then the previous 'end' value is popped meaning that the context just ended
182 // becomes a member of the now current context.
183 //
184 // The nominal strategy for building a completed ASN.1 structure is to push everything
185 // into the buffer and then move everything to the start of the buffer. The move is
186 // simple as the size of the move is the initial 'end' value minus the final 'offset'
187 // value. The destination is 'buffer' and the source is 'buffer' + 'offset'. As Skippy
188 // would say "Easy peasy, Joe."
189 //
190 // It is not necessary to provide a buffer into which the data is placed. If no buffer
191 // is provided, then the marshaling process will return values needed for marshaling.
192 // On strategy for filling the buffer would be to execute the process for building
193 // the structure without using a buffer. This would return the overall size of the
194 // structure. Then that amount of data could be allocated for the buffer and the fill
195 // process executed again with the data going into the buffer. At the end, the data
196 // would be in its final resting place.
197
198 /*** ASN1InitialializeMarshalContext()
199 // This creates a structure for handling marshaling of an ASN.1 formatted data
200 // structure.
201 void ASN1InitialializeMarshalContext(
202     ASN1MarshalContext* ctx, INT16 length, BYTE* buffer)
203 {
204     ctx->buffer = buffer;
205     if(buffer)
206         ctx->offset = length;
207     else
208         ctx->offset = INT16_MAX;
209     ctx->end = ctx->offset;
210     ctx->depth = -1;
211 }
212
213 /*** ASN1StartMarshalContext()
214 // This starts a new constructed element. It is constructed on 'top' of the value
215 // that was previously placed in the structure.
216 void ASN1StartMarshalContext(ASN1MarshalContext* ctx)
217 {
218     pAssert((ctx->depth + 1) < MAX_DEPTH);
219     ctx->depth++;
220     ctx->ends[ctx->depth] = ctx->end;
221     ctx->end = ctx->offset;
222 }
223
224 /*** ASN1EndMarshalContext()
225 // This function restores the end pointer for an encapsulating structure.
226 // Return Type: INT16
227 //     > 0         the size of the encapsulated structure that was just ended
228 //     <= 0        an error
229 INT16

```

```

230 ASN1EndMarshalContext(ASN1MarshalContext* ctx)
231 {
232     INT16 length;
233     pAssert(ctx->depth >= 0);
234     length = ctx->end - ctx->offset;
235     ctx->end = ctx->ends[ctx->depth--];
236     return length;
237 }
238
239 /**ASN1EndEncapsulation()
240 // This function puts a tag and length in the buffer. In this function, an embedded
241 // BIT_STRING is assumed to be a collection of octets. To indicate that all bits
242 // are used, a byte of zero is prepended. If a raw bit-string is needed, a new
243 // function like ASN1PushInteger() would be needed.
244 // Return Type: INT16
245 //      > 0      number of octets in the encapsulation
246 //      == 0      failure
247 UINT16
248 ASN1EndEncapsulation(ASN1MarshalContext* ctx, BYTE tag)
249 {
250     // only add a leading zero for an encapsulated BIT STRING
251     if(tag == ASN1_BITSTRING)
252         ASN1PushByte(ctx, 0);
253     ASN1PushTagAndLength(ctx, tag, ctx->end - ctx->offset);
254     return ASN1EndMarshalContext(ctx);
255 }
256
257 /**ASN1PushByte()
258 BOOL ASN1PushByte(ASN1MarshalContext* ctx, BYTE b)
259 {
260     if(ctx->offset > 0)
261     {
262         ctx->offset -= 1;
263         if(ctx->buffer)
264             ctx->buffer[ctx->offset] = b;
265         return TRUE;
266     }
267     ctx->offset = -1;
268     return FALSE;
269 }
270
271 /**ASN1PushBytes()
272 // Push some raw bytes onto the buffer. 'count' cannot be zero.
273 // Return Type: INT16
274 //      > 0      count bytes
275 //      == 0      failure unless count was zero
276 INT16
277 ASN1PushBytes(ASN1MarshalContext* ctx, INT16 count, const BYTE* buffer)
278 {
279     // make sure that count is not negative which would mess up the math; and that
280     // if there is a count, there is a buffer
281     GOTO_ERROR_UNLESS((count >= 0) && ((buffer != NULL) || (count == 0)));
282     // back up the offset to determine where the new octets will get pushed
283     ctx->offset -= count;
284     // can't go negative
285     GOTO_ERROR_UNLESS(ctx->offset >= 0);
286     // if there are buffers, move the data, otherwise, assume that this is just a
287     // test.
288     if(count && buffer && ctx->buffer)
289         MemoryCopy(&ctx->buffer[ctx->offset], buffer, count);
290     return count;
291 Error:
292     ctx->offset = -1;
293     return 0;
294 }
295

```

```

296  /*** ASN1PushNull()
297  // Return Type: INT16
298  //      > 0      count bytes
299  //      == 0      failure unless count was zero
300  INT16
301  ASN1PushNull(ASN1MarshalContext* ctx)
302  {
303      ASN1PushByte(ctx, 0);
304      ASN1PushByte(ctx, ASN1_NULL);
305      return (ctx->offset >= 0) ? 2 : 0;
306  }
307
308  /*** ASN1PushLength()
309  // Push a length value. This will only handle length values that fit in an INT16.
310  // Return Type: UINT16
311  //      > 0      number of bytes added
312  //      == 0      failure
313  INT16
314  ASN1PushLength(ASN1MarshalContext* ctx, INT16 len)
315  {
316      UINT16 start = ctx->offset;
317      GOTO_ERROR_UNLESS(len >= 0);
318      if(len <= 127)
319          ASN1PushByte(ctx, (BYTE)len);
320      else
321      {
322          ASN1PushByte(ctx, (BYTE)(len & 0xFF));
323          len >>= 8;
324          if(len == 0)
325              ASN1PushByte(ctx, 0x81);
326          else
327          {
328              ASN1PushByte(ctx, (BYTE)(len));
329              ASN1PushByte(ctx, 0x82);
330          }
331      }
332      goto Exit;
333  Error:
334      ctx->offset = -1;
335  Exit:
336      return (ctx->offset > 0) ? start - ctx->offset : 0;
337  }
338
339  /*** ASN1PushTagAndLength()
340  // Return Type: INT16
341  //      > 0      number of bytes added
342  //      == 0      failure
343  INT16
344  ASN1PushTagAndLength(ASN1MarshalContext* ctx, BYTE tag, INT16 length)
345  {
346      INT16 bytes;
347      bytes = ASN1PushLength(ctx, length);
348      bytes += (INT16)ASN1PushByte(ctx, tag);
349      return (ctx->offset < 0) ? 0 : bytes;
350  }
351
352  /*** ASN1PushTaggedOctetString()
353  // This function will push a random octet string.
354  // Return Type: INT16
355  //      > 0      number of bytes added
356  //      == 0      failure
357  INT16
358  ASN1PushTaggedOctetString(
359      ASN1MarshalContext* ctx, INT16 size, const BYTE* string, BYTE tag)
360  {
361      ASN1PushBytes(ctx, size, string);

```

```

362     // PushTagAndLenght just tells how many octets it added so the total size of this
363     // element is the sum of those octets and input size.
364     size += ASN1PushTagAndLength(ctx, tag, size);
365     return size;
366 }
367
368 /*** ASN1PushUINT()
369 // This function pushes an native-endian integer value. This just changes a
370 // native-endian integer into a big-endian byte string and calls ASN1PushInteger().
371 // That function will remove leading zeros and make sure that the number is positive.
372 // Return Type: INT16
373 //     > 0           count bytes
374 //     == 0           failure unless count was zero
375 INT16
376 ASN1PushUINT(ASN1MarshalContext* ctx, UINT32 integer)
377 {
378     BYTE marshaled[4];
379     UINT32_TO_BYTE_ARRAY(integer, marshaled);
380     return ASN1PushInteger(ctx, 4, marshaled);
381 }
382
383 /*** ASN1PushInteger
384 // Push a big-endian integer on the end of the buffer
385 // Return Type: UINT16
386 //     > 0           the number of bytes marshaled for the integer
387 //     == 0           failure
388 INT16
389 ASN1PushInteger(ASN1MarshalContext* ctx,          // IN/OUT: buffer context
390                 INT16 iLen,                      // IN: octets of the integer
391                 BYTE* integer                    // IN: big-endian integer
392 )
393 {
394     // no leading 0's
395     while((*integer == 0) && (--iLen > 0))
396         integer++;
397     // Move the bytes to the buffer
398     ASN1PushBytes(ctx, iLen, integer);
399     // if needed, add a leading byte of 0 to make the number positive
400     if(*integer & 0x80)
401         iLen += (INT16)ASN1PushByte(ctx, 0);
402     // PushTagAndLenght just tells how many octets it added so the total size of this
403     // element is the sum of those octets and the adjusted input size.
404     iLen += ASN1PushTagAndLength(ctx, ASN1_INTEGER, iLen);
405     return iLen;
406 }
407
408 /*** ASN1PushOID()
409 // This function is used to add an OID. An OID is 0x06 followed by a byte of size
410 // followed by size bytes. This is used to avoid having to do anything special in the
411 // definition of an OID.
412 // Return Type: UINT16
413 //     > 0           the number of bytes marshaled for the integer
414 //     == 0           failure
415 INT16
416 ASN1PushOID(ASN1MarshalContext* ctx, const BYTE* OID)
417 {
418     if((*OID == ASN1_OBJECT_IDENTIFIER) && ((OID[1] & 0x80) == 0))
419     {
420         return ASN1PushBytes(ctx, OID[1] + 2, OID);
421     }
422     ctx->offset = -1;
423     return 0;
424 }
425
426 #endif // CC_CertifyX509

```


7.201 /tpm/src/X509/X509_ECC.c

```

1  /** Includes
2  #include "Tpm.h"
3  #include "X509.h"
4  #include "OIDs.h"
5  #include "TpmASN1_fp.h"
6  #include "X509_ECC_fp.h"
7  #include "X509_spt_fp.h"
8  #include "CryptHash_fp.h"
9
10 /** Functions
11
12 /*** X509PushPoint()
13 // This seems like it might be used more than once so...
14 // Return Type: INT16
15 // > 0      number of bytes added
16 // == 0      failure
17 INT16
18 X509PushPoint(ASN1MarshalContext* ctx, TPMS_ECC_POINT* p)
19 {
20     // Push a bit string containing the public key. For now, push the x, and y
21     // coordinates of the public point, bottom up
22     ASN1StartMarshalContext(ctx); // BIT STRING
23     {
24         ASN1PushBytes(ctx, p->y.t.size, p->y.t.buffer);
25         ASN1PushBytes(ctx, p->x.t.size, p->x.t.buffer);
26         ASN1PushByte(ctx, 0x04);
27     }
28     return ASN1EndEncapsulation(ctx, ASN1_BITSTRING); // Ends BIT STRING
29 }
30
31 /*** X509AddSigningAlgorithmECC()
32 // This creates the signing algorithm data.
33 // Return Type: INT16
34 // > 0      number of bytes added
35 // == 0      failure
36 INT16
37 X509AddSigningAlgorithmECC(
38     OBJECT* signKey, TPMT_SIG_SCHEME* scheme, ASN1MarshalContext* ctx)
39 {
40     PHASH_DEF hashDef = CryptGetHashDef(scheme->details.any.hashAlg);
41     //
42     NOT_REFERENCED(signKey);
43     // If the desired hashAlg definition wasn't found...
44     if(hashDef->hashAlg != scheme->details.any.hashAlg)
45         return 0;
46
47     switch(scheme->scheme)
48     {
49 #if ALG_ECDSA
50         case TPM_ALG_ECDSA:
51             // Make sure that we have an OID for this hash and ECC
52             if((hashDef->ECDSA)[0] != ASN1_OBJECT_IDENTIFIER)
53                 break;
54             // if this is just an implementation check, indicate that this
55             // combination is supported
56             if(!ctx)
57                 return 1;
58             ASN1StartMarshalContext(ctx);
59             ASN1PushOID(ctx, hashDef->ECDSA);
60             return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
61 #endif // ALG_ECDSA
62         default:
63             break;
64     }

```

```

65     return 0;
66 }
67
68 /*** X509AddPublicECC()
69 // This function will add the publicKey description to the DER data. If ctx is
70 // NULL, then no data is transferred and this function will indicate if the TPM
71 // has the values for DER-encoding of the public key.
72 // Return Type: INT16
73 //     > 0         number of bytes added
74 //     == 0         failure
75 INT16
76 X509AddPublicECC(OBJECT* object, ASN1MarshalContext* ctx)
77 {
78     const BYTE* curveOid =
79         CryptEccGetOID(object->publicArea.parameters.eccDetail.curveID);
80     if((curveOid == NULL) || (*curveOid != ASN1_OBJECT_IDENTIFIER))
81         return 0;
82     //
83     //
84     // SEQUENCE (2 elem) 1st
85     // SEQUENCE (2 elem) 2nd
86     // OBJECT IDENTIFIER 1.2.840.10045.2.1 ecPublicKey (ANSI X9.62 public key
type)
87     // OBJECT IDENTIFIER 1.2.840.10045.3.1.7 prime256v1 (ANSI X9.62 named curve)
88     // BIT STRING (520 bit)
000001001010000111010101010111001001101101000100000010...
89     //
90     // If this is a check to see if the key can be encoded, it can.
91     // Need to mark the end sequence
92     if(ctx == NULL)
93         return 1;
94     ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
95     {
96         X509PushPoint(ctx, &object->publicArea.unique.ecc); // BIT STRING
97         ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 2nd
98         {
99             ASN1PushOID(ctx, curveOid); // curve dependent
100             ASN1PushOID(ctx, OID_ECC_PUBLIC); // (1.2.840.10045.2.1)
101         }
102         ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 2nd
103     }
104     return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 1st
105 }

```

7.202 /tpm/src/X509/X509_RSA.c

```

1  /*** Includes
2  #include "Tpm.h"
3  #include "X509.h"
4  #include "TpmASN1_fp.h"
5  #include "X509_RSA_fp.h"
6  #include "X509_spt_fp.h"
7  #include "CryptHash_fp.h"
8  #include "CryptRsa_fp.h"
9
10 /*** Functions
11
12 #if ALG_RSA
13
14 /*** X509AddSigningAlgorithmRSA()
15 // This creates the signing algorithm data.
16 // Return Type: INT16
17 //     > 0         number of bytes added
18 //     == 0         failure
19 INT16

```

```

20 X509AddSigningAlgorithmRSA(
21     OBJECT* signKey, TPMT_SIG_SCHEME* scheme, ASN1MarshalContext* ctx)
22 {
23     TPM_ALG_ID hashAlg = scheme->details.any.hashAlg;
24     PHASH_DEF hashDef = CryptGetHashDef(hashAlg);
25     //
26     NOT_REFERENCED(signKey);
27     // return failure if hash isn't implemented
28     if(hashDef->hashAlg != hashAlg)
29         return 0;
30     switch(scheme->scheme)
31     {
32     case TPM_ALG_RSASSA:
33     {
34         // if the hash is implemented but there is no PKCS1 OID defined
35         // then this is not a valid signing combination.
36         if(hashDef->PKCS1[0] != ASN1_OBJECT_IDENTIFIER)
37             break;
38         if(ctx == NULL)
39             return 1;
40         return X509PushAlgorithmIdentifierSequence(ctx, hashDef->PKCS1);
41     }
42     case TPM_ALG_RSAPSS:
43         // leave if this is just an implementation check
44         if(ctx == NULL)
45             return 1;
46         // In the case of SHA1, everything is default and RFC4055 says that
47         // implementations that do signature generation MUST omit the parameter
48         // when defaults are used. )-:
49         if(hashDef->hashAlg == TPM_ALG_SHA1)
50         {
51             return X509PushAlgorithmIdentifierSequence(ctx, OID_RSAPSS);
52         }
53         else
54         {
55             // Going to build something that looks like:
56             // SEQUENCE (2 elem)
57             //   OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
58             //   SEQUENCE (3 elem)
59             //     [0] (1 elem)
60             //       SEQUENCE (2 elem)
61             //         OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
62             //         NULL
63             //       [1] (1 elem)
64             //         SEQUENCE (2 elem)
65             //           OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
66             //           SEQUENCE (2 elem)
67             //             OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
68             //             NULL
69             //           [2] (1 elem) salt length
70             //           INTEGER 32
71
72             // The indentation is just to keep track of where we are in the
73             // structure
74             ASN1StartMarshalContext(ctx); // SEQUENCE (2 elements)
75             {
76                 ASN1StartMarshalContext(ctx); // SEQUENCE (3 elements)
77                 {
78                     // [2] (1 elem) salt length
79                     //   INTEGER 32
80                     ASN1StartMarshalContext(ctx);
81                     {
82                         INT16 saltSize = CryptRsaPssSaltSize(
83                             (INT16)hashDef->digestSize,
84                             (INT16)signKey->publicArea.unique.rsa.t.size);
85                         ASN1PushUINT(ctx, saltSize);

```

```

86     }
87     ASN1EndEncapsulation(ctx, ASN1_APPLICATION_SPECIFIC + 2);
88
89     // Add the mask generation algorithm
90     // [1] (1 elem)
91     //     SEQUENCE (2 elem) 1st
92     //     OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
93     //     SEQUENCE (2 elem) 2nd
94     //     OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
95     //     NULL
96     ASN1StartMarshalContext(ctx); // mask context [1] (1 elem)
97     {
98         ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
99         // Handle the 2nd Sequence (sequence (object, null))
100         {
101             // This adds a NULL, then an OID and a SEQUENCE
102             // wrapper.
103             X509PushAlgorithmIdentifierSequence(ctx,
104                                                 hashDef->OID);
105             // add the pkcs1-MGF OID
106             ASN1PushOID(ctx, OID_MGF1);
107         }
108         // End outer sequence
109         ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
110     }
111     // End the [1]
112     ASN1EndEncapsulation(ctx, ASN1_APPLICATION_SPECIFIC + 1);
113
114     // Add the hash algorithm
115     // [0] (1 elem)
116     //     SEQUENCE (2 elem) (done by
117     //     X509PushAlgorithmIdentifierSequence)
118     //     OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256 (NIST)
119     //     NULL
120     ASN1StartMarshalContext(ctx); // [0] (1 elem)
121     {
122         X509PushAlgorithmIdentifierSequence(ctx, hashDef->OID);
123     }
124     ASN1EndEncapsulation(ctx, (ASN1_APPLICATION_SPECIFIC + 0));
125 }
126 // SEQUENCE (3 elements) end
127 ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
128
129 // RSA PSS OID
130 // OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
131 ASN1PushOID(ctx, OID_RSAPSS);
132 }
133 // End Sequence (2 elements)
134 return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
135 }
136 default:
137     break;
138 }
139 return 0;
140 }
141
142 /*** X509AddPublicRSA()
143 // This function will add the publicKey description to the DER data. If fillPtr is
144 // NULL, then no data is transferred and this function will indicate if the TPM
145 // has the values for DER-encoding of the public key.
146 // Return Type: INT16
147 //     > 0      number of bytes added
148 //     == 0      failure
149 INT16
150 X509AddPublicRSA(OBJECT* object, ASN1MarshalContext* ctx)
151 {

```

```

152     UINT32 exp = object->publicArea.parameters.rsaDetail.exponent;
153     //
154     /*
155     SEQUENCE (2 elem) 1st
156     SEQUENCE (2 elem) 2nd
157     OBJECT IDENTIFIER 1.2.840.113549.1.1.1 rsaEncryption (PKCS #1)
158     NULL
159     BIT STRING (1 elem)
160     SEQUENCE (2 elem) 3rd
161     INTEGER (2048 bit) 2197304513741227955725834199357401
162     INTEGER 65537
163 */
164     // If this is a check to see if the key can be encoded, it can.
165     // Need to mark the end sequence
166     if(ctx == NULL)
167         return 1;
168     ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
169     ASN1StartMarshalContext(ctx); // BIT STRING
170     ASN1StartMarshalContext(ctx); // SEQUENCE *(2 elem) 3rd
171
172     // Get public exponent in big-endian byte order.
173     if(exp == 0)
174         exp = RSA_DEFAULT_PUBLIC_EXPONENT;
175
176     // Push a 4 byte integer. This might get reduced if there are leading zeros or
177     // extended if the high order byte is negative.
178     ASN1PushUINT(ctx, exp);
179     // Push the public key as an integer
180     ASN1PushInteger(ctx,
181                     object->publicArea.unique.rsa.t.size,
182                     object->publicArea.unique.rsa.t.buffer);
183     // Embed this in a SEQUENCE tag and length in for the key, exponent sequence
184     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // SEQUENCE (3rd)
185
186     // Embed this in a BIT STRING
187     ASN1EndEncapsulation(ctx, ASN1_BITSTRING);
188
189     // Now add the formatted SEQUENCE for the RSA public key OID. This is a
190     // fully constructed value so it doesn't need to have a context started
191     X509PushAlgorithmIdentifierSequence(ctx, OID_PKCS1_PUB);
192
193     return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
194 }
195
196 #endif // ALG_RSA

```

7.203 /tpm/src/X509/X509_spt.c

```

1  /** Includes
2  #include "Tpm.h"
3  #include "TpmASN1.h"
4  #include "TpmASN1_fp.h"
5  #define _X509_SPT_
6  #include "X509.h"
7  #include "X509_spt_fp.h"
8  #if ALG_RSA
9  # include "X509_RSA_fp.h"
10 #endif // ALG_RSA
11 #if ALG_ECC
12 # include "X509_ECC_fp.h"
13 #endif // ALG_ECC
14 #if ALG_SM2
15 //# include "X509_SM2_fp.h"
16 #endif // ALG_RSA
17

```

```

18  #if CC_CertifyX509
19
20  /** Unmarshaling Functions
21
22  /** X509FindExtensionByOID()
23  // This will search a list of X509 extensions to find an extension with the
24  // requested OID. If the extension is found, the output context ('ctx') is set up
25  // to point to the OID in the extension.
26  // Return Type: BOOL
27  //     TRUE(1)          success
28  //     FALSE(0)         failure (could be catastrophic)
29  BOOL X509FindExtensionByOID(ASN1UnmarshalContext* ctxIn, // IN: the context to search
30                             ASN1UnmarshalContext* ctx,  // OUT: the extension context
31                             const BYTE* OID             // IN: oid to search for
32 )
33 {
34     INT16 length;
35     //
36     pAssert(ctxIn != NULL);
37     // Make the search non-destructive of the input if ctx provided. Otherwise, use
38     // the provided context.
39     if(ctx == NULL)
40         ctx = ctxIn;
41     // if the provided search context is different from the context of the extension,
42     // then copy the search context to the search context.
43     else if(ctx != ctxIn)
44         *ctx = *ctxIn;
45     // Now, search in the extension context
46     for(; ctx->size > ctx->offset; ctx->offset += length)
47     {
48         GOTO_ERROR_UNLESS((length = ASN1NextTag(ctx)) >= 0);
49         // If this is not a constructed sequence, then it doesn't belong
50         // in the extensions.
51         GOTO_ERROR_UNLESS(ctx->tag == ASN1_CONSTRUCTED_SEQUENCE);
52         // Make sure that this entry could hold the OID
53         if(length >= OID_SIZE(OID))
54         {
55             // See if this is a match for the provided object identifier.
56             if(MemoryEqual(OID, &(ctx->buffer[ctx->offset]), OID_SIZE(OID)))
57             {
58                 // Return with ' ctx' set to point to the start of the OID with the
59                 size
60                 // set to be the size of the SEQUENCE
61                 ctx->buffer += ctx->offset;
62                 ctx->offset = 0;
63                 ctx->size = length;
64                 return TRUE;
65             }
66         }
67         GOTO_ERROR_UNLESS(ctx->offset == ctx->size);
68         return FALSE;
69     Error:
70         ctxIn->size = -1;
71         ctx->size = -1;
72         return FALSE;
73 }
74
75 /** X509GetExtensionBits()
76 // This function will extract a bit field from an extension. If the extension doesn't
77 // contain a bit string, it will fail.
78 // Return Type: BOOL
79 //     TRUE(1)          success
80 //     FALSE(0)         failure
81 UINT32
82 X509GetExtensionBits(ASN1UnmarshalContext* ctx, UINT32* value)

```



```

83 {
84     INT16 length;
85     //
86     while(((length = ASN1NextTag(ctx)) > 0) && (ctx->size > ctx->offset))
87     {
88         // Since this is an extension, the extension value will be in an OCTET STRING
89         if(ctx->tag == ASN1_OCTET_STRING)
90         {
91             return ASN1GetBitStringValue(ctx, value);
92         }
93         ctx->offset += length;
94     }
95     ctx->size = -1;
96     return FALSE;
97 }
98
99 /**X509ProcessExtensions()
100 // This function is used to process the TPMA_OBJECT and KeyUsage extensions. It is not
101 // in the CertifyX509.c code because it makes the code harder to follow.
102 // Return Type: TPM_RC
103 //     TPM_RCS_ATTRIBUTES    the attributes of object are not consistent with
104 //                             the extension setting
105 //     TPM_RC_VALUE          problem parsing the extensions
106 TPM_RC
107 X509ProcessExtensions(
108     OBJECT* object,          // IN: The object with the attributes to
109                             //     check
110     stringRef* extension    // IN: The start and length of the extensions
111 )
112 {
113     ASN1UnmarshalContext ctx;
114     ASN1UnmarshalContext extensionCtx;
115     INT16 length;
116     UINT32 value;
117     TPMA_OBJECT attributes = object->publicArea.objectAttributes;
118     //
119     if(!ASN1UnmarshalContextInitialize(&ctx, extension->len, extension->buf)
120        || ((length = ASN1NextTag(&ctx)) < 0) || (ctx.tag != X509_EXTENSIONS))
121         return TPM_RCS_VALUE;
122     if(((length = ASN1NextTag(&ctx)) < 0) || (ctx.tag != (ASN1_CONSTRUCTED_SEQUENCE)))
123         return TPM_RCS_VALUE;
124
125     // Get the extension for the TPMA_OBJECT if there is one
126     if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_TCG_TPMA_OBJECT)
127        && X509GetExtensionBits(&extensionCtx, &value))
128     {
129         // If an keyAttributes extension was found, it must be exactly the same as the
130         // attributes of the object.
131         // NOTE: MemoryEqual() is used rather than a simple UINT32 compare to avoid
132         // type-punned pointer warning/error.
133         if(!MemoryEqual(&value, &attributes, sizeof(value)))
134             return TPM_RCS_ATTRIBUTES;
135     }
136     // Make sure the failure to find the value wasn't because of a fatal error
137     else if(extensionCtx.size < 0)
138         return TPM_RCS_VALUE;
139
140     // Get the keyUsage extension. This one is required
141     if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_KEY_USAGE_EXTENSION)
142        && X509GetExtensionBits(&extensionCtx, &value))
143     {
144         x509KeyUsageUnion keyUsage;
145         BOOL badSign;
146         BOOL badDecrypt;
147         BOOL badFixedTPM;
148         BOOL badRestricted;

```

```

149
150     //
151     keyUsage.integer = value;
152     // For KeyUsage:
153     // 1) 'sign' is SET if Key Usage includes signing
154     badSign = ((KEY_USAGE_SIGN.integer & keyUsage.integer) != 0)
155               && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign);
156     // 2) 'decrypt' is SET if Key Usage includes decryption uses
157     badDecrypt = ((KEY_USAGE_DECRYPT.integer & keyUsage.integer) != 0)
158                 && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt);
159     // 3) 'fixedTPM' is SET if Key Usage is non-repudiation
160     badFixedTPM = IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE, nonrepudiation)
161                  && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM);
162     // 4) 'restricted' is SET if Key Usage is for key encipherment.
163     badRestricted =
164         IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE, keyEncipherment)
165         && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted);
166     if(badSign || badDecrypt || badFixedTPM || badRestricted)
167         return TPM_RCS_VALUE;
168 }
169 else
170     // The KeyUsage extension is required
171     return TPM_RCS_VALUE;
172
173 return TPM_RC_SUCCESS;
174 }
175
176 /** Marshaling Functions
177
178 **** X509AddSigningAlgorithm()
179 // This creates the signing algorithm data.
180 // Return Type: INT16
181 // > 0          number of octets added
182 // <= 0          failure
183 INT16
184 X509AddSigningAlgorithm(
185     ASN1MarshalContext* ctx, OBJECT* signKey, TPMT_SIG_SCHEME* scheme)
186 {
187     switch(signKey->publicArea.type)
188     {
189 # if ALG_RSA
190         case TPM_ALG_RSA:
191             return X509AddSigningAlgorithmRSA(signKey, scheme, ctx);
192 # endif // ALG_RSA
193 # if ALG_ECC
194         case TPM_ALG_ECC:
195             return X509AddSigningAlgorithmECC(signKey, scheme, ctx);
196 # endif // ALG_ECC
197 # if ALG_SM2
198         case TPM_ALG_SM2:
199             break; // no signing algorithm for SM2 yet
200 //             return X509AddSigningAlgorithmSM2(signKey, scheme, ctx);
201 # endif // ALG_SM2
202         default:
203             break;
204     }
205     return 0;
206 }
207
208 **** X509AddPublicKey()
209 // This function will add the publicKey description to the DER data. If fillPtr is
210 // NULL, then no data is transferred and this function will indicate if the TPM
211 // has the values for DER-encoding of the public key.
212 // Return Type: INT16
213 // > 0          number of octets added
214 // == 0          failure

```

```

215 INT16
216 X509AddPublicKey(ASN1MarshalContext* ctx, OBJECT* object)
217 {
218     switch(object->publicArea.type)
219     {
220     # if ALG_RSA
221         case TPM_ALG_RSA:
222             return X509AddPublicRSA(object, ctx);
223     # endif
224     # if ALG_ECC
225         case TPM_ALG_ECC:
226             return X509AddPublicECC(object, ctx);
227     # endif
228     # if ALG_SM2
229         case TPM_ALG_SM2:
230             break;
231     # endif
232         default:
233             break;
234     }
235     return FALSE;
236 }
237
238 /*** X509PushAlgorithmIdentifierSequence()
239 // The function adds the algorithm identifier sequence.
240 // Return Type: INT16
241 // > 0      number of bytes added
242 // == 0      failure
243 INT16
244 X509PushAlgorithmIdentifierSequence(ASN1MarshalContext* ctx, const BYTE* OID)
245 {
246     // An algorithm ID sequence is:
247     // SEQUENCE
248     //   OID
249     //   NULL
250     ASN1StartMarshalContext(ctx); // hash algorithm
251     ASN1PushNull(ctx);
252     ASN1PushOID(ctx, OID);
253     return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
254 }
255
256 #endif // CC_CertifyX509
257

```

Annex A (informative) Implementation Dependent

A.1 Introduction

These files contains definitions that are used to define a TPM profile. The values are chosen by the manufacturer. The values here are chosen to represent a full featured TPM so that all of the TPM's capabilities can be simulated and tested. This file would change based on the implementation.

The file listed below was generated by an automated tool using three documents as inputs. They are:

- 1) The TCG Algorithm Registry,
- 2) Part 2 of this specification, and
- 3) A purpose-built document that contains vendor-specific information in tables.
- 4) All of the values in this file have `#ifdef 'guards'` so that they may be defined in a command line. Additionally, `TpmBuildSwitches.h` allows an additional file to be specified in the compiler command line and preset any of these values.

A.1.1. /TpmConfiguration/TpmConfiguration/TpmBuildSwitches.h

```

1  // This file contains the build switches. This contains switches for multiple
2  // versions of the crypto-library so some may not apply to your environment.
3  // Each switch has an accompanying description below.
4  //
5  // clang-format off
6  #ifndef _TPM_BUILD_SWITCHES_H_
7  #define _TPM_BUILD_SWITCHES_H_
8
9  #if defined(YES) || defined(NO)
10 # error YES and NO should be defined in TpmBuildSwitches.h
11 #endif
12 #if defined(SET) || defined(CLEAR)
13 # error SET and CLEAR should be defined in TpmBuildSwitches.h
14 #endif
15
16 #define YES 1
17 #define SET 1
18 #define NO 0
19 #define CLEAR 0
20
21 // TRUE/FALSE may be coming from system headers, but if not, provide them.
22 #ifndef TRUE
23 # define TRUE 1
24 #endif
25 #ifndef FALSE
26 # define FALSE 0
27 #endif
28
29 // Need an unambiguous definition for DEBUG. Do not change this
30 #ifndef DEBUG
31 # ifdef NDEBUG
32 #   define DEBUG NO
33 # else
34 #   define DEBUG YES
35 # endif
36 #elif (DEBUG != NO) && (DEBUG != YES)
37 # error DEBUG should be 0 or 1
38 #endif
39
```

```

40  //////////////////////////////////////
41  // DEBUG OPTIONS
42  //////////////////////////////////////
43
44  // The SIMULATION switch allows certain other macros to be enabled. The things that
45  // can be enabled in a simulation include key caching, reproducible "random"
46  // sequences, instrumentation of the RSA key generation process, and certain other
47  // debug code. SIMULATION Needs to be defined as either YES or NO. This grouping of
48  // macros will make sure that it is set correctly. A simulated TPM would include a
49  // Virtual TPM. The interfaces for a Virtual TPM should be modified from the standard
50  // ones in the Simulator project.
51  #define SIMULATION                YES
52
53  // If doing debug, can set the DRBG to print out the intermediate test values.
54  // Before enabling this, make sure that the dbgDumpMemBlock() function
55  // has been added somewhere (preferably, somewhere in CryptRand.c)
56  #define DRBG_DEBUG_PRINT          (NO * DEBUG)
57
58  // This define is used to control the debug for the CertifyX509 command.
59  #define CERTIFYX509_DEBUG        (YES * DEBUG)
60
61  // This provides fixed seeding of the RNG when doing debug on a simulator. This
62  // should allow consistent results on test runs as long as the input parameters
63  // to the functions remains the same.
64  #define USE_DEBUG_RNG            (NO * DEBUG)
65
66  //////////////////////////////////////
67  // RSA DEBUG OPTIONS
68  //////////////////////////////////////
69
70  // Enable the instrumentation of the sieve process. This is used to tune the sieve
71  // variables.
72  #define RSA_INSTRUMENT           (NO * DEBUG)
73
74  // Enables use of the key cache. Default is YES
75  #define USE_RSA_KEY_CACHE        (NO * DEBUG)
76
77  // Enables use of a file to store the key cache values so that the TPM will start
78  // faster during debug. Default for this is YES
79  #define USE_KEY_CACHE_FILE       (NO * DEBUG)
80
81  //////////////////////////////////////
82  // TEST OPTIONS
83  //////////////////////////////////////
84  // The SIMULATION flag can enable test crypto behaviors and caching that
85  // significantly change the behavior of the code. This flag controls only the
86  // g_forceFailureMode flag in the TPM library while leaving the rest of the TPM
87  // behavior alone. Useful for testing when the full set of options controlled by
88  // SIMULATION may not be desired.
89  #define ALLOW_FORCE_FAILURE_MODE YES
90
91  //////////////////////////////////////
92  // Internal checks
93  //////////////////////////////////////
94
95  // Define this to run the function that checks the compatibility between the
96  // chosen big number math library and the TPM code. Not all ports use this.
97  #define LIBRARY_COMPATIBILITY_CHECK YES
98
99  // In some cases, the relationship between two values may be dependent on things that
100  // change based on various selections like the chosen cryptographic libraries. It is
101  // possible that these selections will result in incompatible settings. These are
102  // often
103  // detectable by the compiler but it is not always possible to do the check in the
104  // preprocessor code. For example, when the check requires use of 'sizeof()' then the
105  // preprocessor can't do the comparison. For these cases, we include a special macro

```

```

105 // that, depending on the compiler will generate a warning to indicate if the check
106 // always passes or always fails because it involves fixed constants.
107 //
108 // In modern compilers this is now commonly known as a static_assert, but the precise
109 // implementation varies by compiler. CompilerDependencies.h defines MUST_BE as a
110 // macro
111 // that abstracts out the differences, and COMPILER_CHECKS can remove the checks where
112 // the current compiler doesn't support it. COMPILER_CHECKS should be enabled if the
113 // compiler supports some form of static_assert.
114 // See the CompilerDependencies_*.h files for specific implementations per compiler.
115 #define COMPILER_CHECKS YES
116
117 // Some of the values (such as sizes) are the result of different options set in
118 // TpmProfile.h. The combination might not be consistent. A function is defined
119 // (TpmSizeChecks()) that is used to verify the sizes at run time. To enable the
120 // function, define this parameter.
121 #define RUNTIME_SIZE_CHECKS YES
122
123 // Compliance options
124
125
126 // Enable extra behaviors to meet FIPS compliance requirements
127 #define FIPS_COMPLIANT YES
128
129 // Indicates if the implementation is to compute the sizes of the proof and primary
130 // seed size values based on the implemented algorithms.
131 #define USE_SPEC_COMPLIANT_PROOFS YES
132
133 // Set this to allow compile to continue even though the chosen proof values
134 // do not match the compliant values. This is written so that someone would
135 // have to proactively ignore errors.
136 #define SKIP_PROOF_ERRORS NO
137
138 // Implementation alternatives - don't change external behavior
139
140
141 // Define TABLE_DRIVEN_DISPATCH to use tables rather than case statements
142 // for command dispatch and handle unmarshaling
143 #define TABLE_DRIVEN_DISPATCH YES
144
145 // This define is used to enable the new table-driven marshaling code.
146 #define TABLE_DRIVEN_MARSHAL NO
147
148 // This switch allows use of #defines in place of pass-through marshaling or
149 // unmarshaling code. A pass-through function just calls another function to do
150 // the required function and does no parameter checking of its own. The
151 // table-driven dispatcher calls directly to the lowest level
152 // marshaling/unmarshaling code and by-passes any pass-through functions.
153 #define USE_MARSHALING_DEFINES YES
154
155 // Switch added to support packed lists that leave out space associated with
156 // unimplemented commands. Comment this out to use linear lists.
157 // Note: if vendor specific commands are present, the associated list is always
158 // in compressed form.
159 #define COMPRESSED_LISTS YES
160
161 // This define is used to eliminate the use of bit-fields. It can be enabled for big-
162 // or little-endian machines. For big-endian architectures that numbers bits in
163 // registers from left to right (MSb0) this must be enabled. Little-endian machines
164 // number from right to left with the least significant bit having assigned a bit
165 // number of 0. These are LSB0 machines (they are also little-endian so they are also
166 // least-significant byte 0 (LSB0) machines. Big-endian (MSB0) machines may number in
167 // either direction (MSb0 or LSB0). For an MSB0+MSb0 machine this value is required to
168 // be 'NO'
169

```



```

170 #define USE_BIT_FIELD_STRUCTURES    NO
171
172 // Enable the generation of RSA primes using a sieve.
173 #define RSA_KEY_SIEVE                YES
174
175 ///////////////////////////////////////////////////////////////////
176 // Implementation alternatives - changes external behavior
177 ///////////////////////////////////////////////////////////////////
178
179 // This switch enables the RNG state save and restore
180 #define _DRBG_STATE_SAVE              YES
181
182 // Definition to allow alternate behavior for non-orderly startup. If there is a
183 // chance that the TPM could not update 'failedTries'
184 #define USE_DA_USED                    YES
185
186 // This switch is used to enable the self-test capability in AlgorithmTests.c
187 #define SELF_TEST                      YES
188
189 // This switch indicates where clock epoch value should be stored. If this value
190 // defined, then it is assumed that the timer will change at any time so the
191 // nonce should be a random number kept in RAM. When it is not defined, then the
192 // timer only stops during power outages.
193 #define CLOCK_STOPS                    NO
194
195 // Indicate if the implementation is going to give lockout time credit for time up to
196 // the last orderly shutdown.
197 #define ACCUMULATE_SELF_HEAL_TIMER     YES
198
199 // If an assertion event is not going to produce any trace information (function and
200 // line number) then make FAIL_TRACE == NO
201 #define FAIL_TRACE                      YES
202
203 // TODO_RENAME_INC_FOLDER: public refers to the TPM_CoreLib public headers
204 #include <public/CompilerDependencies.h>
205
206 #endif // _TPM_BUILD_SWITCHES_H_

```

A.1.2. /TpmConfiguration/TpmConfiguration/TpmProfile.h

```

1 // The primary configuration file that collects all configuration options for a
2 // TPM build.
3 #ifndef _TPM_PROFILE_H_
4 #define _TPM_PROFILE_H_
5
6 #include <TpmConfiguration/TpmBuildSwitches.h>
7 #include <TpmConfiguration/TpmProfile_Common.h>
8 #include <TpmConfiguration/TpmProfile_CommandList.h>
9 #include <TpmConfiguration/TpmProfile_Misc.h>
10 #include <TpmConfiguration/TpmProfile_ErrorCodes.h>
11 #include <TpmConfiguration/VendorInfo.h>
12
13 #endif // _TPM_PROFILE_H_

```

A.1.3. /TpmConfiguration/TpmConfiguration/TpmProfile_CommandList.h

```

1 // this file defines the desired command list that should be built into the
2 // Tpm Core Lib.
3
4 #ifndef _TPM_PROFILE_COMMAND_LIST_H_
5 #define _TPM_PROFILE_COMMAND_LIST_H_
6
7 #if (YES != 1 || NO != 0)
8 # error YES and NO must be correctly set before including TpmProfile_CommandList.h

```

```

9  #endif
10 #if defined(CC_YES) || defined(CC_NO)
11 # error CC_YES and CC_NO should be defined by the command line file, not before
12 #endif
13
14 #define CC_YES YES
15 #define CC_NO NO
16
17 //
18 // Defines for Implemented Commands
19 //
20
21 // Commands that are defined in the spec, but not implemented for various
22 // reasons:
23
24 // The TPM reference implementation does not implement attached-component
25 // features, and the Compliance test suite has no test cases.
26 #define CC_AC_GetCapability CC_NO
27 #define CC_AC_Send          CC_NO
28
29 // The TPM reference implementation does not implement firmware upgrade.
30 #define CC_FieldUpgradeData CC_NO
31 #define CC_FieldUpgradeStart CC_NO
32 #define CC_FirmwareRead     CC_NO
33
34 // A prototype of CertifyX509 is provided here for informative purposes only.
35 // While all of the TPM reference implementation is provided "AS IS" without any
36 // warranty, the current design and implementation of CertifyX509 are considered
37 // to be especially unsuitable for product use.
38 #define CC_CertifyX509 CC_NO
39
40 // Normal commands:
41
42 #define CC_ACT_SetTimeout          (CC_YES && ACT_SUPPORT)
43 #define CC_ActivateCredential      CC_YES
44 #define CC_Certify                 CC_YES
45 #define CC_CertifyCreation         CC_YES
46 #define CC_ChangeEPS              CC_YES
47 #define CC_ChangePPS              CC_YES
48 #define CC_Clear                  CC_YES
49 #define CC_ClearControl            CC_YES
50 #define CC_ClockRateAdjust        CC_YES
51 #define CC_ClockSet               CC_YES
52 #define CC_Commit                 (CC_YES && ALG_ECC)
53 #define CC_ContextLoad            CC_YES
54 #define CC_ContextSave            CC_YES
55 #define CC_Create                 CC_YES
56 #define CC_CreateLoaded           CC_YES
57 #define CC_CreatePrimary          CC_YES
58 #define CC_DictionaryAttackLockReset CC_YES
59 #define CC_DictionaryAttackParameters CC_YES
60 #define CC_Duplicate              CC_YES
61 #define CC_ECC_Decrypt            (CC_YES && ALG_ECC)
62 #define CC_ECC_Encrypt            (CC_YES && ALG_ECC)
63 #define CC_ECC_Parameters         (CC_YES && ALG_ECC)
64 #define CC_ECDH_KeyGen            (CC_YES && ALG_ECC)
65 #define CC_ECDH_ZGen              (CC_YES && ALG_ECC)
66 #define CC_EC_Ephemeral           (CC_YES && ALG_ECC)
67 #define CC_EncryptDecrypt         CC_YES
68 #define CC_EncryptDecrypt2        CC_YES
69 #define CC_EventSequenceComplete CC_YES
70 #define CC_EvictControl           CC_YES
71 #define CC_FlushContext           CC_YES
72 #define CC_GetCapability          CC_YES
73 #define CC_GetCommandAuditDigest CC_YES
74 #define CC_GetRandom              CC_YES

```

```

75 #define CC_GetSessionAuditDigest CC_YES
76 #define CC_GetTestResult CC_YES
77 #define CC_GetTime CC_YES
78 #define CC_HMAC (CC_YES && !ALG_CMIC)
79 #define CC_HMAC_Start (CC_YES && !ALG_CMIC)
80 #define CC_Hash CC_YES
81 #define CC_HashSequenceStart CC_YES
82 #define CC_HierarchyChangeAuth CC_YES
83 #define CC_HierarchyControl CC_YES
84 #define CC_Import CC_YES
85 #define CC_IncrementalSelfTest CC_YES
86 #define CC_Load CC_YES
87 #define CC_LoadExternal CC_YES
88 #define CC_MAC (CC_YES && ALG_CMIC)
89 #define CC_MAC_Start (CC_YES && ALG_CMIC)
90 #define CC_MakeCredential CC_YES
91 #define CC_NV_Certify CC_YES
92 #define CC_NV_ChangeAuth CC_YES
93 #define CC_NV_DefineSpace CC_YES
94 #define CC_NV_Extend CC_YES
95 #define CC_NV_GlobalWriteLock CC_YES
96 #define CC_NV_Increment CC_YES
97 #define CC_NV_Read CC_YES
98 #define CC_NV_ReadLock CC_YES
99 #define CC_NV_ReadPublic CC_YES
100 #define CC_NV_SetBits CC_YES
101 #define CC_NV_UndefineSpace CC_YES
102 #define CC_NV_UndefineSpaceSpecial CC_YES
103 #define CC_NV_Write CC_YES
104 #define CC_NV_WriteLock CC_YES
105 #define CC_ObjectChangeAuth CC_YES
106 #define CC_PCR_Allocate CC_YES
107 #define CC_PCR_Event CC_YES
108 #define CC_PCR_Extend CC_YES
109 #define CC_PCR_Read CC_YES
110 #define CC_PCR_Reset CC_YES
111 #define CC_PCR_SetAuthPolicy CC_YES
112 #define CC_PCR_SetAuthValue CC_YES
113 #define CC_PP_Commands CC_YES
114 #define CC_PolicyAuthValue CC_YES
115 #define CC_PolicyAuthorize CC_YES
116 #define CC_PolicyAuthorizeNV CC_YES
117 #define CC_PolicyCapability CC_YES
118 #define CC_PolicyCommandCode CC_YES
119 #define CC_PolicyCounterTimer CC_YES
120 #define CC_PolicyCpHash CC_YES
121 #define CC_PolicyDuplicationSelect CC_YES
122 #define CC_PolicyGetDigest CC_YES
123 #define CC_PolicyLocality CC_YES
124 #define CC_PolicyNV CC_YES
125 #define CC_PolicyNameHash CC_YES
126 #define CC_PolicyNvWritten CC_YES
127 #define CC_PolicyOR CC_YES
128 #define CC_PolicyPCR CC_YES
129 #define CC_PolicyPassword CC_YES
130 #define CC_PolicyParameters CC_YES
131 #define CC_PolicyPhysicalPresence CC_YES
132 #define CC_PolicyRestart CC_YES
133 #define CC_PolicySecret CC_YES
134 #define CC_PolicySigned CC_YES
135 #define CC_PolicyTemplate CC_YES
136 #define CC_PolicyTicket CC_YES
137 #define CC_Policy_AC_SendSelect CC_YES
138 #define CC_Quote CC_YES
139 #define CC_RSA_Decrypt (CC_YES && ALG_RSA)
140 #define CC_RSA_Encrypt (CC_YES && ALG_RSA)

```

```

141 #define CC_ReadClock          CC_YES
142 #define CC_ReadPublic         CC_YES
143 #define CC_Rewrap              CC_YES
144 #define CC_SelfTest            CC_YES
145 #define CC_SequenceComplete    CC_YES
146 #define CC_SequenceUpdate      CC_YES
147 #define CC_SetAlgorithmSet     CC_YES
148 #define CC_SetCommandCodeAuditStatus CC_YES
149 #define CC_SetPrimaryPolicy     CC_YES
150 #define CC_Shutdown            CC_YES
151 #define CC_Sign                 CC_YES
152 #define CC_StartAuthSession     CC_YES
153 #define CC_Startup              CC_YES
154 #define CC_StirRandom           CC_YES
155 #define CC_TestParms            CC_YES
156 #define CC_Unseal               CC_YES
157 #define CC_Vendor_TCG_Test      CC_YES
158 #define CC_VerifySignature       CC_YES
159 #define CC_ZGen_2Phase          (CC_YES && ALG_ECC)
160 #define CC_NV_DefineSpace2       CC_YES
161 #define CC_NV_ReadPublic2        CC_YES
162 #define CC_SetCapability         CC_NO
163
164 #endif // _TPM_PROFILE_COMMAND_LIST_H_

```

A.1.4. /TpmConfiguration/TpmConfiguration/TpmProfile_Common.h

```

1 // clang-format off
2 // clang-format off to preserve define alignment breaking sections.
3
4 // this file defines the common optional selections for the TPM library build
5 // Requires basic YES/NO defines are already set (by TpmBuildSwitches.h)
6 // Less frequently changed items are in other TpmProfile Headers.
7
8 #ifndef TPM_PROFILE_COMMON_H
9 #define TPM_PROFILE_COMMON_H
10 // YES & NO defined by TpmBuildSwitches.h
11 #if (YES != 1 || NO != 0)
12 # error YES or NO incorrectly set
13 #endif
14 #if defined(ALG_YES) || defined(ALG_NO)
15 # error ALG_YES and ALG_NO should only be defined by the TpmProfile_Common.h file
16 #endif
17
18 // Change these definitions to turn all algorithms ON or OFF. That is, to turn
19 // all algorithms on, set ALG_NO to YES. This is intended as a debug feature.
20 #define ALG_YES YES
21 #define ALG_NO NO
22
23 // Defines according to the processor being built for.
24 // Are building for a BIG_ENDIAN processor?
25 #define BIG_ENDIAN_TPM NO
26 #define LITTLE_ENDIAN_TPM !BIG_ENDIAN_TPM
27 // Does the processor put the most-significant bit at bit position 0?
28 #define MOST_SIGNIFICANT_BIT_0 NO
29 #define LEAST_SIGNIFICANT_BIT_0 !MOST_SIGNIFICANT_BIT_0
30 // Does processor support Auto align?
31 #define AUTO_ALIGN NO
32
33 //*****
34 // Defines for Symmetric Algorithms
35 //*****
36
37 #define ALG_AES ALG_YES
38

```

```

39 #define      AES_128                (YES * ALG_AES)
40 #define      AES_192                (NO * ALG_AES)
41 #define      AES_256                (YES * ALG_AES)
42
43 #define ALG_SM4                      ALG_NO
44
45 #define      SM4_128                (NO * ALG_SM4)
46
47 #define ALG_CAMELLIA                ALG_YES
48
49 #define      CAMELLIA_128           (YES * ALG_CAMELLIA)
50 #define      CAMELLIA_192           (NO * ALG_CAMELLIA)
51 #define      CAMELLIA_256           (YES * ALG_CAMELLIA)
52
53 // must be yes if any above are yes.
54 #define ALG_SYMCIPHER                (ALG_AES || ALG_SM4 || ALG_CAMELLIA)
55 #define ALG_CMAC                    (YES * ALG_SYMCIPHER)
56
57 // block cipher modes
58 #define ALG_CTR                      ALG_YES
59 #define ALG_OFB                      ALG_YES
60 #define ALG_CBC                      ALG_YES
61 #define ALG_CFB                      ALG_YES
62 #define ALG_ECB                      ALG_YES
63
64 //*****
65 // Defines for RSA Asymmetric Algorithms
66 //*****
67 #define ALG_RSA                      ALG_YES
68 #define      RSA_1024                (YES * ALG_RSA)
69 #define      RSA_2048                (YES * ALG_RSA)
70 #define      RSA_3072                (YES * ALG_RSA)
71 #define      RSA_4096                (YES * ALG_RSA)
72 #define      RSA_16384               (NO * ALG_RSA)
73
74 #define      ALG_RSASSA               (YES * ALG_RSA)
75 #define      ALG_RSAES               (YES * ALG_RSA)
76 #define      ALG_RSAPSS              (YES * ALG_RSA)
77 #define      ALG_OAEP                (YES * ALG_RSA)
78
79 // RSA Implementation Styles
80 // use Chinese Remainder Theorem (5 prime) format for private key ?
81 #define CRT_FORMAT_RSA              YES
82 #define RSA_DEFAULT_PUBLIC_EXPONENT 0x00010001
83
84 //*****
85 // Defines for ECC Asymmetric Algorithms
86 //*****
87 #define ALG_ECC                      ALG_YES
88 #define      ALG_ECDH                (YES * ALG_ECC)
89 #define      ALG_ECDSA               (YES * ALG_ECC)
90 #define      ALG_ECDAA               (YES * ALG_ECC)
91 #define      ALG_SM2                 (YES * ALG_ECC)
92 #define      ALG_ECSCNORR            (YES * ALG_ECC)
93 #define      ALG_ECMQV               (YES * ALG_ECC)
94 #define      ALG_KDF1_SP800_56A      (YES * ALG_ECC)
95 #define      ALG_EDDSA               (NO * ALG_ECC)
96 #define      ALG_EDDSA_PH            (NO * ALG_ECC)
97
98 #define      ECC_NIST_P192            (YES * ALG_ECC)
99 #define      ECC_NIST_P224            (YES * ALG_ECC)
100 #define      ECC_NIST_P256            (YES * ALG_ECC)
101 #define      ECC_NIST_P384            (YES * ALG_ECC)
102 #define      ECC_NIST_P521            (YES * ALG_ECC)
103 #define      ECC_BN_P256              (YES * ALG_ECC)
104 #define      ECC_BN_P638              (YES * ALG_ECC)

```

```

105 #define      ECC_SM2_P256                      (YES * ALG_ECC)
106
107 #define      ECC_BP_P256_R1                    (NO * ALG_ECC)
108 #define      ECC_BP_P384_R1                    (NO * ALG_ECC)
109 #define      ECC_BP_P512_R1                    (NO * ALG_ECC)
110 #define      ECC_CURVE_25519                   (NO * ALG_ECC)
111 #define      ECC_CURVE_448                     (NO * ALG_ECC)
112
113 //*****
114 // Defines for Hash/XOF Algorithms
115 //*****
116 #define ALG_MGF1                                ALG_YES
117 #define ALG_SHA1                                ALG_YES
118 #define ALG_SHA256                              ALG_YES
119 #define ALG_SHA256_192                          ALG_NO
120 #define ALG_SHA384                              ALG_YES
121 #define ALG_SHA512                              ALG_NO
122
123 #define ALG_SHA3_256                            ALG_NO
124 #define ALG_SHA3_384                            ALG_NO
125 #define ALG_SHA3_512                            ALG_NO
126
127 #define ALG_SM3_256                              ALG_NO
128
129 #define ALG_SHAKE256_192                        ALG_NO
130 #define ALG_SHAKE256_256                      ALG_NO
131 #define ALG_SHAKE256_512                      ALG_NO
132
133 //*****
134 // Defines for Stateful Signature Algorithms
135 //*****
136 #define ALG_LMS                                ALG_NO
137 #define ALG_XMSS                                ALG_NO
138
139 //*****
140 // Defines for Keyed Hashes
141 //*****
142 #define ALG_KEYEDHASH                          ALG_YES
143 #define ALG_HMAC                              ALG_YES
144
145 //*****
146 // Defines for KDFs
147 //*****
148 #define ALG_KDF2                                ALG_YES
149 #define ALG_KDF1_SP800_108                    ALG_YES
150
151 //*****
152 // Defines for Obscuration/MISC/compatibility
153 //*****
154 #define ALG_XOR                                ALG_YES
155
156 //*****
157 // Defines controlling ACT
158 //*****
159 #define ACT_SUPPORT                            YES
160 #define RH_ACT_0                                (YES * ACT_SUPPORT)
161 #define RH_ACT_1                                ( NO * ACT_SUPPORT)
162 #define RH_ACT_2                                ( NO * ACT_SUPPORT)
163 #define RH_ACT_3                                ( NO * ACT_SUPPORT)
164 #define RH_ACT_4                                ( NO * ACT_SUPPORT)
165 #define RH_ACT_5                                ( NO * ACT_SUPPORT)
166 #define RH_ACT_6                                ( NO * ACT_SUPPORT)
167 #define RH_ACT_7                                ( NO * ACT_SUPPORT)
168 #define RH_ACT_8                                ( NO * ACT_SUPPORT)
169 #define RH_ACT_9                                ( NO * ACT_SUPPORT)
170 #define RH_ACT_A                                (YES * ACT_SUPPORT)

```



```

171 #define RH_ACT_B ( NO * ACT_SUPPORT)
172 #define RH_ACT_C ( NO * ACT_SUPPORT)
173 #define RH_ACT_D ( NO * ACT_SUPPORT)
174 #define RH_ACT_E ( NO * ACT_SUPPORT)
175 #define RH_ACT_F ( NO * ACT_SUPPORT)
176
177
178 //*****
179 // Enable VENDOR_PERMANENT_AUTH_HANDLE?
180 //*****
181 #define VENDOR_PERMANENT_AUTH_ENABLED NO
182 // if YES, this must be valid per Part2 (TPM_RH_AUTH_00 - TPM_RH_AUTH_FF)
183 // if NO, this must be #undef
184 #undef VENDOR_PERMANENT_AUTH_HANDLE
185
186 //*****
187 // Defines controlling optional implementation
188 //*****
189 #define FIELD_UPGRADE_IMPLEMENTED NO
190
191 //*****
192 // Buffer Sizes based on implementation
193 //*****
194 // When using PC CRB, the page size for both commands and
195 // control registers is 4k. The command buffer starts at
196 // offset 0x80, so the net size available is:
197 #define MAX_COMMAND_SIZE (4096-0x80)
198 #define MAX_RESPONSE_SIZE (4096-0x80)
199
200 //*****
201 // Vendor Info
202 //*****
203 // max buffer for vendor commands
204 // Max data buffer leaving space for TPM2B size prefix
205 #define VENDOR_COMMAND_COUNT 0
206 #define MAX_VENDOR_BUFFER_SIZE (MAX_RESPONSE_SIZE-2)
207 #define PRIVATE_VENDOR_SPECIFIC_BYTES RSA_PRIVATE_SIZE
208
209 //*****
210 // Defines controlling Firmware- and SVN-limited objects
211 //*****
212 #define FW_LIMITED_SUPPORT YES
213 #define SVN_LIMITED_SUPPORT YES
214
215 //*****
216 // Defines controlling External NV
217 //*****
218 // This is a software reference implementation of the TPM: there is no
219 // "external NV" as such. This #define configures the TPM to implement
220 // "external NV" that is stored in the same place as "internal NV."
221 // NOTE: enabling this doesn't necessarily mean that the expanded
222 // (external-NV-specific) attributes are supported.
223 #define EXTERNAL_NV YES
224
225 #endif // _TPM_PROFILE_COMMON_H

```

A.1.5. /TpmConfiguration/TpmConfiguration/TpmProfile_ErrorCodes.h

```

1 /** Introduction
2 // This file defines error codes used in failure macros in the TPM Core Library.
3 // This file is part of TpmConfiguration because the Platform library can add error
4 // codes of it's own, and ultimately the specific error codes are a vendor decision
5 // because TPM2_GetTestResult returns manufacturer-defined data in failure mode.
6 // The only thing in this file that must be consistent with a vendor's implementation
7 // are the _names_ of error codes used by the core library. Even the values can

```

```

8 // change and are only a suggestion.
9
10 #ifndef _TPMPROFILE_ERRORCODES_H
11 #define _TPMPROFILE_ERRORCODES_H
12
13 // turn off clang-format because alignment doesn't persist across comments
14 // with current settings
15 // clang-format off
16
17 #define FATAL_ERROR_ALLOCATION (1)
18 #define FATAL_ERROR_DIVIDE_ZERO (2)
19 #define FATAL_ERROR_INTERNAL (3)
20 #define FATAL_ERROR_PARAMETER (4)
21 #define FATAL_ERROR_ENTROPY (5)
22 #define FATAL_ERROR_SELF_TEST (6)
23 #define FATAL_ERROR_CRYPT (7)
24 #define FATAL_ERROR_NV_UNRECOVERABLE (8)
25
26 // indicates that the TPM has been re-manufactured after an
27 // unrecoverable NV error
28 #define FATAL_ERROR_REMANUFACTURED (9)
29 #define FATAL_ERROR_DRBG (10)
30 #define FATAL_ERROR_MOVE_SIZE (11)
31 #define FATAL_ERROR_COUNTER_OVERFLOW (12)
32 #define FATAL_ERROR_SUBTRACT (13)
33 #define FATAL_ERROR_MATHLIBRARY (14)
34 // end of codes defined through v1.52
35
36 // leave space for numbers that may have been used by vendors or platforms.
37 // Ultimately this file and these ranges are only a suggestion because
38 // TPM2_GetTestResult returns manufacturer-defined data in failure mode.
39 // Reserve 15-499
40 #define FATAL_ERROR_RESERVED_START (15)
41 #define FATAL_ERROR_RESERVED_END (499)
42
43 // Additional error codes defined by TPM library:
44 #define FATAL_ERROR_ASSERT (500)
45 // Platform library violated interface contract.
46 #define FATAL_ERROR_PLATFORM (600)
47
48 // Test/Simulator errors 1000+
49 #define FATAL_ERROR_FORCED (1000)
50
51 #endif // _TPMPROFILE_ERRORCODES_H

```

A.1.6. /TpmConfiguration/TpmConfiguration/TpmProfile_Misc.h

```

1 // Misc profile settings that don't currently have a better home.
2 // These are rarely changed, but available for vendor customization.
3
4 #ifndef _TPM_PROFILE_MISC_H
5 #define _TPM_PROFILE_MISC_H
6
7 // YES & NO defined by TpmBuildSwitches.h
8 #if (YES != 1 || NO != 0)
9 # error YES or NO incorrectly set
10 #endif
11
12 // clang-format off
13 // clang-format off to preserve horizontal spacing
14 #define IMPLEMENTATION_PCR 24
15 #define PLATFORM_PCR 24
16 #define DRTM_PCR 17
17 #define HCRTM_PCR 0
18 #define NUM_LOCALITIES 5

```

```

19 #define MAX_HANDLE_NUM 3
20 #define MAX_ACTIVE_SESSIONS 64
21 #define MAX_LOADED_SESSIONS 3
22 #define MAX_SESSION_NUM 3
23 #define MAX_LOADED_OBJECTS 3
24 #define MIN_EVICT_OBJECTS 2
25 #define NUM_POLICY_PCR_GROUP 1
26 #define NUM_AUTHVALUE_PCR_GROUP 1
27 #define MAX_CONTEXT_SIZE 2168
28 #define MAX_DIGEST_BUFFER 1024
29 #define MAX_NV_INDEX_SIZE 2048
30 #define MAX_NV_BUFFER_SIZE 1024
31 #define MAX_CAP_BUFFER 1024
32 #define NV_MEMORY_SIZE 16384
33 #define MIN_COUNTER_INDICES 8
34 #define NUM_STATIC_PCR 16
35 #define MAX_ALG_LIST_SIZE 64
36 #define PRIMARY_SEED_SIZE 32
37 #define CONTEXT_ENCRYPT_ALGORITHM AES
38 #define NV_CLOCK_UPDATE_INTERVAL 22
39 #define NUM_POLICY_PCR 1
40
41 #define ORDERLY_BITS 8
42 #define MAX_SYM_DATA 128
43 #define MAX_RNG_ENTROPY_SIZE 64
44 #define RAM_INDEX_SPACE 512
45 #define ENABLE_PCR_NO_INCREMENT YES
46
47 #define SIZE_OF_X509_SERIAL_NUMBER 20
48
49 // amount of space the platform can provide in PERSISTENT_DATA during
50 // manufacture
51 #define PERSISTENT_DATA_PLATFORM_SPACE 16
52
53 // structure padding space for these structures. Used if a
54 // particular configuration needs them to be aligned to a
55 // specific size
56 #define ORDERLY_DATA_PADDING 0
57 #define STATE_CLEAR_DATA_PADDING 0
58 #define STATE_RESET_DATA_PADDING 0
59
60 // configuration values that may vary by SIMULATION/DEBUG
61 #if SIMULATION && DEBUG
62 // This forces the use of a smaller context slot size. This reduction reduces the
63 // range of the epoch allowing the tester to force the epoch to occur faster than
64 // the normal production size
65 # define CONTEXT_SLOT UINT8
66 #else
67 # define CONTEXT_SLOT UINT16
68 #endif
69
70 #endif // _TPM_PROFILE_MISC_H_

```

A.1.7. /TpmConfiguration/TpmConfiguration/VendorInfo.h

```

1 #ifndef _VENDORINFO_H
2 #define _VENDORINFO_H
3
4 // Define the TPM specification-specific capability values.
5 #define TPM_SPEC_FAMILY (0x322E3000)
6 #define TPM_SPEC_LEVEL (00)
7 #define TPM_SPEC_VERSION (181)
8 #define TPM_SPEC_YEAR (2023)
9 #define TPM_SPEC_DAY_OF_YEAR (333)
10 #define MAX_VENDOR_PROPERTY (1)

```

```
11
12 // Define the platform specification-specific capability values.
13 #define PLATFORM_FAMILY      (0)
14 #define PLATFORM_LEVEL      (0)
15 #define PLATFORM_VERSION    (0)
16 #define PLATFORM_YEAR       (0)
17 #define PLATFORM_DAY_OF_YEAR (0)
18
19 #endif
20
```

Annex B (informative) Library-Specific

B.1 Introduction

This clause contains the files that are specific to a cryptographic library used by the TPM code.

Three categories are defined for cryptographic functions:

- 1) big number math (asymmetric cryptography),
- 2) symmetric ciphers, and
- 3) hash functions.

The code is structured to make it possible to use different libraries for different categories. For example, one might choose to use OpenSSL for its math library, but use a different library for hashing and symmetric cryptography. Since OpenSSL supports all three categories, it might be more typical to combine libraries of specific functions; that is, one library might only contain block ciphers while another supports big number math.

B.2 Common Cryptographic Functionality

B.2.1. /tpm/cryptolib/./common/include/MathLibraryInterface.h

```

1  /** Introduction
2  //
3  // This file contains the function prototypes for the functions that need to be
4  // present in the selected math library. For each function listed, there should
5  // be a small stub function. That stub provides the interface between the TPM
6  // code and the support library. In most cases, the stub function will only need
7  // to do a format conversion between the Crypt_* formats to the internal support
8  // library format. Since the external library also provides the buffer macros
9  // for the underlying types, this is typically just a cast from the TPM type to
10 // the internal type.
11 //
12 // Arithmetic operations return a BOOL to indicate if the operation completed
13 // successfully or not.
14
15 #ifndef MATH_LIBRARY_INTERFACE_H
16 #define MATH_LIBRARY_INTERFACE_H
17
18 // Types
19 #include "MathLibraryInterfaceTypes.h"
20
21 // *****
22 // Library Level Functions
23 // *****
24
25 /** ExtMath_LibInit()
26 // This function is called by CryptInit() so that necessary initializations can be
27 // performed on the cryptographic library.
28 LIB_EXPORT int ExtMath_LibInit(void);
29
30 /** MathLibraryCompatibilityCheck()
31 // This function is only used during development to make sure that the library
32 // that is being referenced is using the same size of data structures as the TPM.
33 LIB_EXPORT BOOL ExtMath_Debug_CompatibilityCheck(void);
34
35 // *****
36 // Integer/Number Functions (non-ECC)

```

```

37 // *****
38
39 // #####
40 // type initializers
41 // #####
42
43 /** ExtMath_Initialize_Int()
44 // Initialize* functions tells the Crypt_Int types how large of a value it can
45 // contain which is a compile time constant
46 LIB_EXPORT Crypt_Int* ExtMath_Initialize_Int(Crypt_Int* buffer, NUMBYTES bits);
47
48 // #####
49 // Buffer Converters
50 // #####
51 // convert TPM2B byte data into the private format. The Crypt_Int must already be
52 // initialized with it's maximum size. Byte-based Initializers must be MSB first
53 // (TPM external format).
54 LIB_EXPORT Crypt_Int* ExtMath_IntFromBytes(
55     Crypt_Int* buffer, const BYTE* input, NUMBYTES byteCount);
56 // Convert Crypt_Int into external format as a byte array.
57 LIB_EXPORT BOOL ExtMath_IntToBytes(
58     const Crypt_Int* value, BYTE* output, NUMBYTES* pByteCount);
59 // Set Crypt_Int to a given small value. Words are native format.
60 LIB_EXPORT Crypt_Int* ExtMath_SetWord(Crypt_Int* buffer, crypt_ushort_t word);
61
62 // #####
63 // Copy Functions
64 // #####
65
66 /** ExtMath_Copy()
67 // Function to copy a bignum_t. If the output is NULL, then
68 // nothing happens. If the input is NULL, the output is set to zero.
69 LIB_EXPORT BOOL ExtMath_Copy(Crypt_Int* out, const Crypt_Int* in);
70
71 // #####
72 // Ordinary Arithmetic, writ large
73 // #####
74
75 /** ExtMath_Multiply()
76 // Multiplies two numbers and returns the result
77 LIB_EXPORT BOOL ExtMath_Multiply(
78     Crypt_Int* result, const Crypt_Int* multiplicand, const Crypt_Int* multiplier);
79
80 /** ExtMath_Divide()
81 // This function divides two Crypt_Int* values. The function returns FALSE if there is
82 // an error in the operation. Quotient may be null, in which case this function
83 // returns
84 // only the remainder.
85 LIB_EXPORT BOOL ExtMath_Divide(Crypt_Int* quotient,
86                               Crypt_Int* remainder,
87                               const Crypt_Int* dividend,
88                               const Crypt_Int* divisor);
89
90 /** ExtMath_GCD()
91 // Get the greatest common divisor of two numbers. This function is only needed
92 // when the TPM implements RSA.
93 LIB_EXPORT BOOL ExtMath_GCD(
94     Crypt_Int* gcd, const Crypt_Int* number1, const Crypt_Int* number2);
95
96 /** ExtMath_Add()
97 // This function adds two Crypt_Int* values. This function always returns TRUE.
98 LIB_EXPORT BOOL ExtMath_Add(
99     Crypt_Int* result, const Crypt_Int* op1, const Crypt_Int* op2);
100
101 /** ExtMath_AddWord()
102 // This function adds a word value to a Crypt_Int*. This function always returns TRUE.

```



```

102 LIB_EXPORT BOOL ExtMath_AddWord(
103     Crypt_Int* result, const Crypt_Int* op, crypt_ushort_t word);
104
105 /*** ExtMath_Subtract()
106 // This function does subtraction of two Crypt_Int* values and returns result = op1 -
107 // op2
108 // when op1 is greater than op2. If op2 is greater than op1, then a fault is
109 // generated. This function always returns TRUE.
110 LIB_EXPORT BOOL ExtMath_Subtract(
111     Crypt_Int* result, const Crypt_Int* op1, const Crypt_Int* op2);
112
113 /*** ExtMath_SubtractWord()
114 // This function subtracts a word value from a Crypt_Int*. This function always
115 // returns TRUE.
116 LIB_EXPORT BOOL ExtMath_SubtractWord(
117     Crypt_Int* result, const Crypt_Int* op, crypt_ushort_t word);
118
119 // #####
120 // Modular Arithmetic, writ large
121 // #####
122
123 /*** ExtMath_Mod()
124 // compute valueAndResult = valueAndResult mod modulus
125 // This function divides two Crypt_Int* values and returns only the remainder,
126 // replacing the original dividend. The function returns FALSE if there is an
127 // error in the operation.
128 LIB_EXPORT BOOL ExtMath_Mod(Crypt_Int* valueAndResult, const Crypt_Int* modulus);
129
130 /*** ExtMath_ModMult()
131 // Compute result = (op1 * op2) mod modulus
132 LIB_EXPORT BOOL ExtMath_ModMult(Crypt_Int* result,
133     const Crypt_Int* op1,
134     const Crypt_Int* op2,
135     const Crypt_Int* modulus);
136
137 /*** ExtMath_ModExp()
138 // Compute result = (number ^ exponent) mod modulus
139 // where ^ indicates exponentiation.
140 // This function is only needed when the TPM implements RSA.
141 LIB_EXPORT BOOL ExtMath_ModExp(Crypt_Int* result,
142     const Crypt_Int* number,
143     const Crypt_Int* exponent,
144     const Crypt_Int* modulus);
145
146 /*** ExtMath_ModInverse()
147 // Compute the modular multiplicative inverse.
148 // result = (number ^ -1) mod modulus
149 // This function is only needed when the TPM implements RSA.
150 LIB_EXPORT BOOL ExtMath_ModInverse(
151     Crypt_Int* result, const Crypt_Int* number, const Crypt_Int* modulus);
152
153 /*** ExtMath_ModInversePrime()
154 // Compute the modular multiplicative inverse. This is an optimized function for
155 // the case where the modulus is known to be prime.
156 //
157 // CAUTION: Depending on the library implementation this may be much faster than
158 // the normal ModInverse, and therefore is subject to exposing the fact the
159 // modulus is prime via a timing side-channel. In many cases (e.g. ECC primes),
160 // the prime is not sensitive and this optimized route can be used.
161 LIB_EXPORT BOOL ExtMath_ModInversePrime(
162     Crypt_Int* result, const Crypt_Int* number, const Crypt_Int* primeModulus);
163
164 /*** ExtMath_ModWord()
165 // compute numerator
166 // This function does modular division of a big number when the modulus is a
167 // word value.

```

```

167 LIB_EXPORT crypt_word_t ExtMath_ModWord(const Crypt_Int* numerator,
168                                         crypt_word_t      modulus);
169
170 // #####
171 // Queries
172 // #####
173
174 /*** ExtMath_UnsignedCmp()
175 // This function performs a comparison of op1 to op2. The compare is approximately
176 // constant time if the size of the values used in the compare is consistent
177 // across calls (from the same line in the calling code).
178 // Return Type: int
179 //      < 0          op1 is less than op2
180 //      0            op1 is equal to op2
181 //      > 0          op1 is greater than op2
182 LIB_EXPORT int ExtMath_UnsignedCmp(const Crypt_Int* op1, const Crypt_Int* op2);
183
184 /*** ExtMath_UnsignedCmpWord()
185 // Compare a Crypt_Int* to a crypt_uword_t.
186 // Return Type: int
187 //      -1           op1 is less than word
188 //      0            op1 is equal to word
189 //      1            op1 is greater than word
190 LIB_EXPORT int ExtMath_UnsignedCmpWord(const Crypt_Int* op1, crypt_uword_t word);
191
192 /*** ExtMath_IsEqualWord()
193 // Compare a Crypt_Int* to a crypt_uword_t for equality
194 // Return Type: BOOL
195 LIB_EXPORT BOOL ExtMath_IsEqualWord(const Crypt_Int* bn, crypt_uword_t word);
196
197 /*** ExtMath_IsZero()
198 // Compare a Crypt_Int* to zero, expected to be O(1) time.
199 // Return Type: BOOL
200 LIB_EXPORT BOOL ExtMath_IsZero(const Crypt_Int* op1);
201
202 /*** ExtMath_MostSigBitNum()
203 //
204 // This function returns the zero-based number of the MSb (Most significant bit)
205 // of a Crypt_Int* value.
206 //
207 // Return Type: int
208 //
209 //      -1           the word was zero or 'bn' was NULL
210 //      n            the bit number of the most significant bit in the word
211 LIB_EXPORT int ExtMath_MostSigBitNum(const Crypt_Int* bn);
212
213 /*** ExtMath_GetLeastSignificant32bits()
214 //
215 // This function returns the least significant 32-bits of an integer value
216 // Return Type: uint32_t
217 LIB_EXPORT uint32_t ExtMath_GetLeastSignificant32bits(const Crypt_Int* bn);
218
219 /*** ExtMath_SizeInBits()
220 //
221 // This function returns the number of bits required to hold a number. It is one
222 // greater than the MSb. This function is expected to be side channel safe, and
223 // may be O(size) or O(1) where 'size' is the allocated (not actual) size of the
224 // value.
225 LIB_EXPORT unsigned ExtMath_SizeInBits(const Crypt_Int* n);
226
227 // #####
228 // Bitwise Operations
229 // #####
230
231 /*** ExtMath_SetBit()
232 //

```

```

233 // This function will SET a bit in a Crypt_Int*. Bit 0 is the least-significant
234 // bit in the 0th digit_t. The function returns TRUE if the bitNum is within the
235 // range valid for the given number. If bitNum is too large, the function
236 // should return FALSE, and the TPM will enter failure mode.
237 // Return Type: BOOL
238 LIB_EXPORT BOOL ExtMath_SetBit(Crypt_Int* bn, // IN/OUT: big number to modify
239                               unsigned int bitNum // IN: Bit number to SET
240 );
241
242 /** ExtMath_TestBit()
243 // This function is used to check to see if a bit is SET in a bignum_t. The 0th bit
244 // is the LSb of d[0].
245 // Return Type: BOOL
246 // TRUE(1) the bit is set
247 // FALSE(0) the bit is not set or the number is out of range
248 LIB_EXPORT BOOL ExtMath_TestBit(Crypt_Int* bn, // IN: number to check
249                               unsigned int bitNum // IN: bit to test
250 );
251
252 /** ExtMath_MaskBits()
253 // This function is used to mask off high order bits of a big number.
254 // The returned value will have no more than 'maskBit' bits
255 // set.
256 // Note: There is a requirement that unused words of a bignum_t are set to zero.
257 // Return Type: BOOL
258 // TRUE(1) result masked
259 // FALSE(0) the input was not as large as the mask
260 LIB_EXPORT BOOL ExtMath_MaskBits(
261     Crypt_Int* bn, // IN/OUT: number to mask
262     crypt_uword_t maskBit // IN: the bit number for the mask.
263 );
264
265 /** ExtMath_ShiftRight()
266 // This function will shift a Crypt_Int* to the right by the shiftAmount.
267 // This function always returns TRUE.
268 LIB_EXPORT BOOL ExtMath_ShiftRight(
269     Crypt_Int* result, const Crypt_Int* toShift, uint32_t shiftAmount);
270
271 // *****
272 // ECC Functions
273 // *****
274 // #####
275 // type initializers
276 // #####
277
278 /** initialize point structure given memory size of each coordinate
279 LIB_EXPORT Crypt_Point* ExtEcc_Initialize_Point(Crypt_Point* buffer,
280     NUMBYTES bitsPerCoord);
281
282 /** ExtEcc_CurveInitialize()
283 // This function is used to initialize a Crypt_EccCurve structure. The
284 // structure is a set of pointers to Crypt_Int* values. The curve-dependent values are
285 // set by a different function. This function is only needed
286 // if the TPM supports ECC.
287 LIB_EXPORT const Crypt_EccCurve* ExtEcc_CurveInitialize(Crypt_EccCurve* E,
288     TPM_ECC_CURVE curveId);
289
290 // #####
291 // DESTRUCTOR - See Warning
292 // #####
293
294 /** ExtEcc_CurveFree()
295 // This function will free the allocated components of the curve and end the
296 // frame in which the curve data exists.
297 // WARNING: Not guaranteed to be called in presence of LONGJMP.
298 LIB_EXPORT void ExtEcc_CurveFree(const Crypt_EccCurve* E);

```

```

299
300 // #####
301 // Buffer Converters
302 // #####
303 /** point structure to/from raw coordinate buffers.
304 LIB_EXPORT Crypt_Point* ExtEcc_PointFromBytes(Crypt_Point* buffer,
305                                               const BYTE* x,
306                                               NUMBYTES nBytesX,
307                                               const BYTE* y,
308                                               NUMBYTES nBytesY);
309
310 LIB_EXPORT BOOL ExtEcc_PointToBytes(
311     const Crypt_Point* point, BYTE* x, NUMBYTES* nBytesX, BYTE* y, NUMBYTES*
nBytesY);
312
313 // #####
314 // ECC Point Operations
315 // #####
316
317 /** ExtEcc_PointMultiply()
318 // This function does a point multiply of the form  $R = [d]S$ . A return of FALSE
319 // indicates that the result was the point at infinity. This function is only needed
320 // if the TPM supports ECC.
321 LIB_EXPORT BOOL ExtEcc_PointMultiply(Crypt_Point* R,
322                                     const Crypt_Point* S,
323                                     const Crypt_Int* d,
324                                     const Crypt_EccCurve* E);
325
326 /** ExtEcc_PointMultiplyAndAdd()
327 // This function does a point multiply of the form  $R = [d]S + [u]Q$ . A return of
328 // FALSE indicates that the result was the point at infinity. This function is only
329 // needed if the TPM supports ECC.
330 LIB_EXPORT BOOL ExtEcc_PointMultiplyAndAdd(Crypt_Point* R,
331                                           const Crypt_Point* S,
332                                           const Crypt_Int* d,
333                                           const Crypt_Point* Q,
334                                           const Crypt_Int* u,
335                                           const Crypt_EccCurve* E);
336
337 /** ExtEcc_PointAdd()
338 // This function does a point add  $R = S + Q$ . A return of FALSE
339 // indicates that the result was the point at infinity. This function is only needed
340 // if the TPM supports ECC.
341 LIB_EXPORT BOOL ExtEcc_PointAdd(Crypt_Point* R,
342                                const Crypt_Point* S,
343                                const Crypt_Point* Q,
344                                const Crypt_EccCurve* E);
345
346 // #####
347 // ECC Point Information
348 // #####
349 LIB_EXPORT BOOL ExtEcc_IsPointOnCurve(const Crypt_Point* Q, const Crypt_EccCurve* E);
350 LIB_EXPORT BOOL ExtEcc_IsInfinityPoint(const Crypt_Point* pt);
351 // extract the X-Coordinate of a point
352 LIB_EXPORT const Crypt_Int* ExtEcc_PointX(const Crypt_Point* pt);
353
354 // extract the Y-Coordinate of a point
355 // (no current use case for the Y coordinate alone, signatures use X)
356 // LIB_EXPORT const Crypt_Int* ExtEcc_PointY(const Crypt_Point* pt);
357
358 // #####
359 // ECC Curve Information
360 // #####
361 // These functions are expected to be fast, returning pre-built constants without
362 // allocation or copying.
363 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetPrime(TPM_ECC_CURVE curveId);

```

```

364 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetOrder(TPM_ECC_CURVE curveId);
365 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetCofactor(TPM_ECC_CURVE curveId);
366 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGet_a(TPM_ECC_CURVE curveId);
367 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGet_b(TPM_ECC_CURVE curveId);
368 LIB_EXPORT const Crypt_Point* ExtEcc_CurveGetG(TPM_ECC_CURVE curveId);
369 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetGx(TPM_ECC_CURVE curveId);
370 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetGy(TPM_ECC_CURVE curveId);
371 LIB_EXPORT TPM_ECC_CURVE ExtEcc_CurveGetCurveId(const Crypt_EccCurve* E);
372
373 #endif

```

B.2.2. /tpm/cryptolib/common/include/MathLibraryInterfaceTypes.h

```

1  /** Introduction
2  // This file contains the declaration and initialization macros for
3  // low-level cryptographic buffer types. This requires the underlying
4  // Crypto library to have already defined the CRYPT_INT_BUF family of
5  // macros. See tpm_crypto_lib.md for details.
6
7  #ifndef MATH_LIBRARY_INTERFACE_TYPES_H
8  #define MATH_LIBRARY_INTERFACE_TYPES_H
9
10 #ifndef CRYPT_INT_BUF
11 # error CRYPT_INT_BUF must be defined before including this file.
12 #endif
13 #ifndef CRYPT_POINT_BUF
14 # error CRYPT_POINT_BUF must be defined before including this file.
15 #endif
16 #ifndef CRYPT_CURVE_BUF
17 # error CRYPT_CURVE_BUF must be defined before including this file.
18 #endif
19
20 // Crypt_Int underlying types Crypt_Int is an abstract type that is used as a
21 // pointer. The underlying math library is expected to be able to find the
22 // actual allocated size for a given Crypt_Int object given a pointer to it, and
23 // therefore we typedefed here to a size 1 (smallest possible).
24 typedef CRYPT_INT_BUF(one, 1) Crypt_Int;
25 typedef CRYPT_POINT_BUF(pointone, 1) Crypt_Point;
26 typedef CRYPT_CURVE_BUF(curvebuf, MAX_ECC_KEY_BITS) Crypt_EccCurve;
27
28 // produces bare typedef ci_<typename>_t
29 #define CRYPT_INT_TYPE(typename, bits) \
30     typedef CRYPT_INT_BUF(ci_##typename##_buf_t, bits) ci_##typename##_t
31
32 // produces allocated `Crypt_Int* varname` backed by a
33 // stack buffer named `<varname>_buf`. Initialization at the discretion of the
34 // ExtMath library.
35 #define CRYPT_INT_VAR(varname, bits) \
36     CRYPT_INT_BUF(ci_##varname##_buf_t, bits) varname##_buf; \
37     Crypt_Int* varname = ExtMath_Initialize_Int((Crypt_Int*)&(varname##_buf), bits);
38
39 // produces initialized `Crypt_Int* varname = (TPM2B) initializer` backed by a
40 // stack buffer named `<varname>_buf`
41 #define CRYPT_INT_INITIALIZED(varname, bits, initializer) \
42     CRYPT_INT_BUF(ci_##varname##_buf_t, bits) varname##_buf; \
43     Crypt_Int* varname = \
44         TpmMath_IntFrom2B(ExtMath_Initialize_Int((Crypt_Int*)&(varname##_buf), bits), \
45             (TPM2B*)initializer);
46
47 // convenience variants of above:
48 // largest supported integer
49 #define CRYPT_INT_MAX(varname) CRYPT_INT_VAR(varname, LARGEST_NUMBER_BITS)
50
51 #define CRYPT_INT_MAX_INITIALIZED(name, initializer) \
52     CRYPT_INT_INITIALIZED(name, LARGEST_NUMBER_BITS, initializer)

```



```

53
54 // A single RADIX_BITS value.
55 #define CRYPT_INT_WORD(name) CRYPT_INT_VAR(name, RADIX_BITS)
56
57 #define CRYPT_INT_WORD_INITIALIZED(varname, initializer) \
58     CRYPT_INT_BUF(cibuf##varname, RADIX_BITS) varname##_buf; \
59     Crypt_Int* varname = ExtMath_SetWord( \
60         ExtMath_Initialize_Int((Crypt_Int*)&(varname##_buf), RADIX_BITS), \
61         initializer);
62
63 // Crypt_EccCurve underlying types
64 #define CRYPT_CURVE_INITIALIZED(varname, initializer) \
65     CRYPT_CURVE_BUF(cv##varname, MAX_ECC_KEY_BITS) varname##_buf; \
66     const Crypt_EccCurve* varname = \
67         ExtEcc_CurveInitialize(&(varname##_buf), initializer)
68
69 /* no guarantee free will be called in the presence of longjmp */
70 #define CRYPT_CURVE_FREE(varname) ExtEcc_CurveFree(varname)
71
72 // Crypt_Point underlying types
73 #define CRYPT_POINT_VAR(varname) \
74     CRYPT_POINT_BUF(cp##varname##_buf_t, MAX_ECC_KEY_BITS) varname##_buf; \
75     Crypt_Point* varname = \
76         ExtEcc_Initialize_Point((Crypt_Point*)&(varname##_buf), MAX_ECC_KEY_BITS);
77
78 #define CRYPT_POINT_INITIALIZED(varname, initValue) \
79     CRYPT_POINT_BUF(cp##varname##_buf_t, MAX_ECC_KEY_BITS) varname##_buf; \
80     Crypt_Point* varname = TpmEcc_PointFrom2B( \
81         ExtEcc_Initialize_Point((Crypt_Point*)&(varname##_buf), MAX_ECC_KEY_BITS), \
82         initValue);
83
84 #endif //MATH_LIBRARY_INTERFACE_TYPES_H
85

```

B.3 TpmBigNum

B.3.1. /tpm/cryptolib/TpmBigNum/BnConvert.c

```

1  /** Introduction
2  // This file contains the basic conversion functions that will convert TPM2B
3  // to/from the internal format. The internal format is a bigNum,
4  //
5
6  /** Includes
7
8  #include "TpmBigNum.h"
9
10 /** Functions
11
12 /** BnFromBytes()
13 // This function will convert a big-endian byte array to the internal number
14 // format. If bn is NULL, then the output is NULL. If bytes is null or the
15 // required size is 0, then the output is set to zero
16 LIB_EXPORT bigNum BnFromBytes(bigNum bn, const BYTE* bytes, NUMBYTES nBytes)
17 {
18     const BYTE* pFrom; // 'p' points to the least significant bytes of source
19     BYTE* pTo; // points to least significant bytes of destination
20     crypt_uword_t size;
21     //
22
23     size = (bytes != NULL) ? BYTES_TO_CRYPT_WORDS(nBytes) : 0;
24
25     // If nothing in, nothing out
26     if(bn == NULL)

```



```

27         return NULL;
28
29     // make sure things fit
30     pAssert(BnGetAllocated(bn) >= size);
31
32     if(size > 0)
33     {
34         // Clear the topmost word in case it is not filled with data
35         bn->d[size - 1] = 0;
36         // Moving the input bytes from the end of the list (LSB) end
37         pFrom = bytes + nBytes - 1;
38         // To the LS0 of the LSW of the bigNum.
39         pTo = (BYTE*)bn->d;
40         for(; nBytes != 0; nBytes--)
41             *pTo++ = *pFrom--;
42         // For a little-endian machine, the conversion is a straight byte
43         // reversal. For a big-endian machine, we have to put the words in
44         // big-endian byte order
45 #if BIG_ENDIAN_TPM
46     {
47         crypt_word_t t;
48         for(t = (crypt_word_t)size - 1; t >= 0; t--)
49             bn->d[t] = SWAP_CRYPT_WORD(bn->d[t]);
50     }
51 #endif
52     }
53     BnSetTop(bn, size);
54     return bn;
55 }
56
57 /*** BnFrom2B()
58 // Convert an TPM2B to a BIG_NUM.
59 // If the input value does not exist, or the output does not exist, or the input
60 // will not fit into the output the function returns NULL
61 LIB_EXPORT bigNum BnFrom2B(bigNum bn, // OUT:
62                             const TPM2B* a2B // IN: number to convert
63 )
64 {
65     if(a2B != NULL)
66         return BnFromBytes(bn, a2B->buffer, a2B->size);
67     // Make sure that the number has an initialized value rather than whatever
68     // was there before
69     BnSetTop(bn, 0); // Function accepts NULL
70     return NULL;
71 }
72
73 /*** BnToBytes()
74 // This function converts a BIG_NUM to a byte array. It converts the bigNum to a
75 // big-endian byte string and sets 'size' to the normalized value. If 'size' is an
76 // input 0, then the receiving buffer is guaranteed to be large enough for the result
77 // and the size will be set to the size required for bigNum (leading zeros
78 // suppressed).
79 //
80 // The conversion for a little-endian machine simply requires that all significant
81 // bytes of the bigNum be reversed. For a big-endian machine, rather than
82 // unpack each word individually, the bigNum is converted to little-endian words,
83 // copied, and then converted back to big-endian.
84 LIB_EXPORT BOOL BnToBytes(bigConst bn,
85                           BYTE* buffer,
86                           NUMBYTES* size // This the number of bytes that are
87                                           // available in the buffer. The result
88                                           // should be this big.
89 )
90 {
91     crypt_uword_t requiredSize;
92     BYTE* pFrom;

```

```

93     BYTE*      pTo;
94     crypt_ushort_t count;
95     //
96     // validate inputs
97     pAssert(bn && buffer && size);
98
99     requiredSize = (BnSizeInBits(bn) + 7) / 8;
100    if(requiredSize == 0)
101    {
102        // If the input value is 0, return a byte of zero
103        *size = 1;
104        *buffer = 0;
105    }
106    else
107    {
108        #if BIG_ENDIAN_TPM
109            // Copy the constant input value into a modifiable value
110            BN_VAR(bnL, LARGEST_NUMBER_BITS * 2);
111            BnCopy(bnL, bn);
112            // byte swap the words in the local value to make them little-endian
113            for(count = 0; count < bnL->size; count++)
114                bnL->d[count] = SWAP_CRYPT_WORD(bnL->d[count]);
115            bn = (bigConst)bnL;
116        #endif
117        if(*size == 0)
118            *size = (NUMBYTES)requiredSize;
119        pAssert(requiredSize <= *size);
120        // Byte swap the number (not words but the whole value)
121        count = *size;
122        // Start from the least significant word and offset to the most significant
123        // byte which is in some high word
124        pFrom = (BYTE*)(&bn->d[0]) + requiredSize - 1;
125        pTo = buffer;
126
127        // If the number of output bytes is larger than the number bytes required
128        // for the input number, pad with zeros
129        for(count = *size; count > requiredSize; count--)
130            *pTo++ = 0;
131        // Move the most significant byte at the end of the BigNum to the next most
132        // significant byte position of the 2B and repeat for all significant bytes.
133        for(; requiredSize > 0; requiredSize--)
134            *pTo++ = *pFrom--;
135    }
136    return TRUE;
137 }
138
139 /*** BnTo2B()
140  // Function to convert a BIG_NUM to TPM2B.
141  // The TPM2B size is set to the requested 'size' which may require padding.
142  // If 'size' is non-zero and less than required by the value in 'bn' then an error
143  // is returned. If 'size' is zero, then the TPM2B is assumed to be large enough
144  // for the data and a2b->size will be adjusted accordingly.
145  LIB_EXPORT BOOL BnTo2B(bigConst bn, // IN:
146                        TPM2B* a2B, // OUT:
147                        NUMBYTES size // IN: the desired size
148  )
149  {
150      // Set the output size
151      if(bn && a2B)
152      {
153          a2B->size = size;
154          return BnToBytes(bn, a2B->buffer, &a2B->size);
155      }
156      return FALSE;
157  }
158

```

```

159 #if ALG_ECC
160
161 /*** BnPointFromBytes()
162 // Function to create a BIG_POINT structure from a byte buffer in big-endian order.
163 // A point is going to be two ECC values in the same buffer. The values are going
164 // to be the size of the modulus. They are in modular form.
165 LIB_EXPORT bn_point_t* BnPointFromBytes(
166     bigPoint    ecP, // OUT: the preallocated point structure
167     const BYTE* x,
168     NUMBYTES    nBytesX,
169     const BYTE* y,
170     NUMBYTES    nBytesY)
171 {
172     if(x == NULL || y == NULL)
173         return NULL;
174
175     if(NULL != ecP)
176     {
177         BnFromBytes(ecP->x, x, nBytesX);
178         BnFromBytes(ecP->y, y, nBytesY);
179         BnSetWord(ecP->z, 1);
180     }
181     return ecP;
182 }
183
184 /*** BnPointToBytes()
185 // This function extracts coordinates from a BIG_POINT into
186 // most-significant-byte-first memory buffers (the native format of
187 // a TPMS_ECC_POINT.)
188 // on input the NUMBYTES* parameters indicate the maximum buffer size.
189 // on output, they represent the amount of significant data in that buffer.
190 LIB_EXPORT BOOL BnPointToBytes(
191     pointConst ecP, // OUT: the preallocated point structure
192     BYTE*      x,
193     NUMBYTES*  pBytesX,
194     BYTE*      y,
195     NUMBYTES*  pBytesY)
196 {
197     pAssert(ecP && x && y && pBytesX && pBytesY);
198     pAssert(BnEqualWord(ecP->z, 1));
199     BOOL result = BnToBytes(ecP->x, x, pBytesX);
200     result      = result && BnToBytes(ecP->y, y, pBytesY);
201     // TODO: zeroize on error?
202     return result;
203 }
204
205 #endif // ALG_ECC

```

B.3.2. /tpm/cryptolib/TpmBigNum/BnEccConstants.c

```

1  /*(Auto-generated)
2  *   Created by TpmStructures; Version 4.4 Mar 26, 2019
3  *   Date: Aug 30, 2019   Time: 02:11:52PM
4  */
5  #include "TpmBigNum.h"
6  // #include "Tpm.h"
7  // TODO_RENAME_INC_FOLDER:private refers to the TPM_CoreLib private headers
8  #include <private/OIDs.h>
9
10 #if ALG_ECC
11
12 // define macros expected by EccConstantData to convert the data to BigNum format
13
14 # define TO_ECC_64                                TO_CRYPT_WORD_64
15 # define TO_ECC_56(a, b, c, d, e, f, g) TO_ECC_64(0, a, b, c, d, e, f, g)

```

```

16 # define TO_ECC_48(a, b, c, d, e, f)    TO_ECC_64(0, 0, a, b, c, d, e, f)
17 # define TO_ECC_40(a, b, c, d, e)      TO_ECC_64(0, 0, 0, a, b, c, d, e)
18 # if RADIX_BITS > 32
19 #   define TO_ECC_32(a, b, c, d) TO_ECC_64(0, 0, 0, 0, a, b, c, d)
20 #   define TO_ECC_24(a, b, c)    TO_ECC_64(0, 0, 0, 0, 0, a, b, c)
21 #   define TO_ECC_16(a, b)       TO_ECC_64(0, 0, 0, 0, 0, 0, a, b)
22 #   define TO_ECC_8(a)           TO_ECC_64(0, 0, 0, 0, 0, 0, 0, a)
23 # else // RADIX_BITS == 32
24 #   define TO_ECC_32             BIG_ENDIAN_BYTES_TO_UINT32
25 #   define TO_ECC_24(a, b, c)    TO_ECC_32(0, a, b, c)
26 #   define TO_ECC_16(a, b)       TO_ECC_32(0, 0, a, b)
27 #   define TO_ECC_8(a)           TO_ECC_32(0, 0, 0, a)
28 # endif
29 # define TO_ECC_192(a, b, c)      c, b, a
30 # define TO_ECC_224(a, b, c, d)   d, c, b, a
31 # define TO_ECC_256(a, b, c, d)   d, c, b, a
32 # define TO_ECC_384(a, b, c, d, e, f) f, e, d, c, b, a
33 # define TO_ECC_528(a, b, c, d, e, f, g, h, i) i, h, g, f, e, d, c, b, a
34 # define TO_ECC_640(a, b, c, d, e, f, g, h, i, j) j, i, h, g, f, e, d, c, b, a
35
36 # define BN_MIN_ALLOC(bytes) \
37     (BYTES_TO_CRYPT_WORDS(bytes) == 0) ? 1 : BYTES_TO_CRYPT_WORDS(bytes)
38 # define ECC_CONST(NAME, bytes, initializer) \
39     const struct \
40     { \
41         crypt_uword_t allocate, size, d[BN_MIN_ALLOC(bytes)]; \
42     } NAME = {BN_MIN_ALLOC(bytes), BYTES_TO_CRYPT_WORDS(bytes), {initializer}}
43
44 // This file contains the raw data for ECC curve constants. The data is wrapped
45 // in macros so this file can be included in other files that format the data in
46 // a memory format desired by the user. This file itself is never used alone.
47 # include <EccConstantData.inl>
48
49 // now define the TPMBN_ECC_CURVE_CONSTANTS objects for the known curves
50
51 # if ECC_NIST_P192
52 const TPMBN_ECC_CURVE_CONSTANTS NIST_P192 = {TPM_ECC_NIST_P192,
53     (bigNum) &NIST_P192_p,
54     (bigNum) &NIST_P192_n,
55     (bigNum) &NIST_P192_h,
56     (bigNum) &NIST_P192_a,
57     (bigNum) &NIST_P192_b,
58     {(bigNum) &NIST_P192_gX,
59      (bigNum) &NIST_P192_gY,
60      (bigNum) &NIST_P192_gZ}};
61 # endif // ECC_NIST_P192
62
63 # if ECC_NIST_P224
64 const TPMBN_ECC_CURVE_CONSTANTS NIST_P224 = {TPM_ECC_NIST_P224,
65     (bigNum) &NIST_P224_p,
66     (bigNum) &NIST_P224_n,
67     (bigNum) &NIST_P224_h,
68     (bigNum) &NIST_P224_a,
69     (bigNum) &NIST_P224_b,
70     {(bigNum) &NIST_P224_gX,
71      (bigNum) &NIST_P224_gY,
72      (bigNum) &NIST_P224_gZ}};
73 # endif // ECC_NIST_P224
74
75 # if ECC_NIST_P256
76 const TPMBN_ECC_CURVE_CONSTANTS NIST_P256 = {TPM_ECC_NIST_P256,
77     (bigNum) &NIST_P256_p,
78     (bigNum) &NIST_P256_n,
79     (bigNum) &NIST_P256_h,
80     (bigNum) &NIST_P256_a,
81     (bigNum) &NIST_P256_b,

```

```

82                                     { (bigNum) &NIST_P256_gX,
83                                     (bigNum) &NIST_P256_gY,
84                                     (bigNum) &NIST_P256_gZ}};
85 # endif // ECC_NIST_P256
86
87 # if ECC_NIST_P384
88 const TPMBN_ECC_CURVE_CONSTANTS NIST_P384 = {TPM_ECC_NIST_P384,
89                                     (bigNum) &NIST_P384_p,
90                                     (bigNum) &NIST_P384_n,
91                                     (bigNum) &NIST_P384_h,
92                                     (bigNum) &NIST_P384_a,
93                                     (bigNum) &NIST_P384_b,
94                                     { (bigNum) &NIST_P384_gX,
95                                     (bigNum) &NIST_P384_gY,
96                                     (bigNum) &NIST_P384_gZ}};
97 # endif // ECC_NIST_P384
98
99 # if ECC_NIST_P521
100 const TPMBN_ECC_CURVE_CONSTANTS NIST_P521 = {TPM_ECC_NIST_P521,
101                                     (bigNum) &NIST_P521_p,
102                                     (bigNum) &NIST_P521_n,
103                                     (bigNum) &NIST_P521_h,
104                                     (bigNum) &NIST_P521_a,
105                                     (bigNum) &NIST_P521_b,
106                                     { (bigNum) &NIST_P521_gX,
107                                     (bigNum) &NIST_P521_gY,
108                                     (bigNum) &NIST_P521_gZ}};
109 # endif // ECC_NIST_P521
110
111 # if ECC_BN_P256
112 const TPMBN_ECC_CURVE_CONSTANTS BN_P256 = {TPM_ECC_BN_P256,
113                                     (bigNum) &BN_P256_p,
114                                     (bigNum) &BN_P256_n,
115                                     (bigNum) &BN_P256_h,
116                                     (bigNum) &BN_P256_a,
117                                     (bigNum) &BN_P256_b,
118                                     { (bigNum) &BN_P256_gX,
119                                     (bigNum) &BN_P256_gY,
120                                     (bigNum) &BN_P256_gZ}};
121 # endif // ECC_BN_P256
122
123 # if ECC_BN_P638
124 const TPMBN_ECC_CURVE_CONSTANTS BN_P638 = {TPM_ECC_BN_P638,
125                                     (bigNum) &BN_P638_p,
126                                     (bigNum) &BN_P638_n,
127                                     (bigNum) &BN_P638_h,
128                                     (bigNum) &BN_P638_a,
129                                     (bigNum) &BN_P638_b,
130                                     { (bigNum) &BN_P638_gX,
131                                     (bigNum) &BN_P638_gY,
132                                     (bigNum) &BN_P638_gZ}};
133 # endif // ECC_BN_P638
134
135 # if ECC_SM2_P256
136 const TPMBN_ECC_CURVE_CONSTANTS SM2_P256 = {TPM_ECC_SM2_P256,
137                                     (bigNum) &SM2_P256_p,
138                                     (bigNum) &SM2_P256_n,
139                                     (bigNum) &SM2_P256_h,
140                                     (bigNum) &SM2_P256_a,
141                                     (bigNum) &SM2_P256_b,
142                                     { (bigNum) &SM2_P256_gX,
143                                     (bigNum) &SM2_P256_gY,
144                                     (bigNum) &SM2_P256_gZ}};
145 # endif // ECC_SM2_P256
146
147 # define comma

```

```

148 const TPMBN_ECC_CURVE_CONSTANTS* bnEccCurveData[] = {
149 # if ECC_NIST_P192
150     &NIST_P192,
151 # endif
152 # if ECC_NIST_P224
153     &NIST_P224,
154 # endif
155 # if ECC_NIST_P256
156     &NIST_P256,
157 # endif
158 # if ECC_NIST_P384
159     &NIST_P384,
160 # endif
161 # if ECC_NIST_P521
162     &NIST_P521,
163 # endif
164 # if ECC_BN_P256
165     &BN_P256,
166 # endif
167 # if ECC_BN_P638
168     &BN_P638,
169 # endif
170 # if ECC_SM2_P256
171     &SM2_P256,
172 # endif
173 };
174
175 MUST_BE((sizeof(bnEccCurveData) / sizeof(bnEccCurveData[0])) == (ECC_CURVE_COUNT));
176
177 /** BnGetCurveData()
178 // This function returns the pointer for the constant parameter data
179 // associated with a curve.
180 const TPMBN_ECC_CURVE_CONSTANTS* BnGetCurveData(TPM_ECC_CURVE curveId)
181 {
182     for(int i = 0; i < ECC_CURVE_COUNT; i++)
183     {
184         if(bnEccCurveData[i]->curveId == curveId)
185             return bnEccCurveData[i];
186     }
187     return NULL;
188 }
189
190 #endif // TPM_ALG_ECC

```

B.3.3. /tpm/cryptolib/TpmBigNum/BnMath.c

```

1  /** Introduction
2  // The simulator code uses the canonical form whenever possible in order to make
3  // the code in Part 3 more accessible. The canonical data formats are simple and
4  // not well suited for complex big number computations. When operating on big
5  // numbers, the data format is changed for easier manipulation. The format is native
6  // words in little-endian format. As the magnitude of the number decreases, the
7  // length of the array containing the number decreases but the starting address
8  // doesn't change.
9  //
10 // The functions in this file perform simple operations on these big numbers. Only
11 // the more complex operations are passed to the underlying support library.
12 // Although the support library would have most of these functions, the interface
13 // code to convert the format for the values is greater than the size of the
14 // code to implement the functions here. So, rather than incur the overhead of
15 // conversion, they are done here.
16 //
17 // If an implementer would prefer, the underlying library can be used simply by
18 // making code substitutions here.
19 //

```



```

20 // NOTE: There is an intention to continue to augment these functions so that there
21 // would be no need to use an external big number library.
22 //
23 // Many of these functions have no error returns and will always return TRUE. This
24 // is to allow them to be used in "guarded" sequences. That is:
25 //     OK = OK || BnSomething(s);
26 // where the BnSomething() function should not be called if OK isn't true.
27
28 /** Includes
29 #include "TpmBigNum.h"
30 extern BOOL g_inFailureMode; // can't use global.h because we can't use tpm.h
31
32 // A constant value of zero as a stand in for NULL bigNum values
33 const bignum_t BnConstZero = {1, 0, {0}};
34
35 /** Functions
36
37 /*** AddSame()
38 // Adds two values that are the same size. This function allows 'result' to be
39 // the same as either of the addends. This is a nice function to put into assembly
40 // because handling the carry for multi-precision stuff is not as easy in C
41 // (unless there is a REALLY smart compiler). It would be nice if there were idioms
42 // in a language that a compiler could recognize what is going on and optimize
43 // loops like this.
44 // Return Type: int
45 //     0      no carry out
46 //     1      carry out
47 static BOOL AddSame(crypt_ushort_t* result,
48                    const crypt_ushort_t* op1,
49                    const crypt_ushort_t* op2,
50                    int count)
51 {
52     int carry = 0;
53     int i;
54
55     for(i = 0; i < count; i++)
56     {
57         crypt_ushort_t a = op1[i];
58         crypt_ushort_t sum = a + op2[i];
59         result[i] = sum + carry;
60         // generate a carry if the sum is less than either of the inputs
61         // propagate a carry if there was a carry and the sum + carry is zero
62         // do this using bit operations rather than logical operations so that
63         // the time is about the same.
64         // propagate term | generate term
65         carry = ((result[i] == 0) & carry) | (sum < a);
66     }
67     return carry;
68 }
69
70 /*** CarryProp()
71 // Propagate a carry
72 static int CarryProp(
73     crypt_ushort_t* result, const crypt_ushort_t* op, int count, int carry)
74 {
75     for(; count; count--)
76         carry = ((*result++ = *op++ + carry) == 0) & carry;
77     return carry;
78 }
79
80 static void CarryResolve(bignum result, int stop, int carry)
81 {
82     if(carry)
83     {
84         pAssert((unsigned)stop < result->allocated);
85         result->d[stop++] = 1;

```

```

86     }
87     BnSetTop(result, stop);
88 }
89
90 /*** BnAdd()
91 // This function adds two bigNum values. This function always returns TRUE.
92 LIB_EXPORT BOOL BnAdd(bigNum result, bigConst op1, bigConst op2)
93 {
94     crypt_ushort_t    stop;
95     int               carry;
96     const bignum_t* n1 = op1;
97     const bignum_t* n2 = op2;
98
99     //
100    if(n2->size > n1->size)
101    {
102        n1 = op2;
103        n2 = op1;
104    }
105    pAssert(result->allocated >= n1->size);
106    stop = MIN(n1->size, n2->allocated);
107    carry = (int)AddSame(result->d, n1->d, n2->d, (int)stop);
108    if(n1->size > stop)
109        carry =
110            CarryProp(&result->d[stop], &n1->d[stop], (int)(n1->size - stop), carry);
111    CarryResolve(result, (int)n1->size, carry);
112    return TRUE;
113 }
114
115 /*** BnAddWord()
116 // This function adds a word value to a bigNum. This function always returns TRUE.
117 LIB_EXPORT BOOL BnAddWord(bigNum result, bigConst op, crypt_ushort_t word)
118 {
119     int carry;
120     //
121     carry = (result->d[0] = op->d[0] + word) < word;
122     carry = CarryProp(&result->d[1], &op->d[1], (int)(op->size - 1), carry);
123     CarryResolve(result, (int)op->size, carry);
124     return TRUE;
125 }
126
127 /*** SubSame()
128 // This function subtracts two values that have the same size.
129 static int SubSame(crypt_ushort_t* result,
130                  const crypt_ushort_t* op1,
131                  const crypt_ushort_t* op2,
132                  int count)
133 {
134     int borrow = 0;
135     int i;
136     for(i = 0; i < count; i++)
137     {
138         crypt_ushort_t a = op1[i];
139         crypt_ushort_t diff = a - op2[i];
140         result[i] = diff - borrow;
141         // generate | propagate
142         borrow = (diff > a) | ((diff == 0) & borrow);
143     }
144     return borrow;
145 }
146
147 /*** BorrowProp()
148 // This propagates a borrow. If borrow is true when the end
149 // of the array is reached, then it means that op2 was larger than
150 // op1 and we don't handle that case so an assert is generated.
151 // This design choice was made because our only bigNum computations

```

```

152 // are on large positive numbers (primes) or on fields.
153 // Propagate a borrow.
154 static int BorrowProp(
155     crypt_uword_t* result, const crypt_uword_t* op, int size, int borrow)
156 {
157     for(; size > 0; size--)
158         borrow = ((*result++ = *op++ - borrow) == MAX_CRYPT_UWORD) && borrow;
159     return borrow;
160 }
161
162 /*** BnSub()
163 // This function does subtraction of two bigNum values and returns result = op1 - op2
164 // when op1 is greater than op2. If op2 is greater than op1, then a fault is
165 // generated. This function always returns TRUE.
166 LIB_EXPORT BOOL BnSub(bigNum result, bigConst op1, bigConst op2)
167 {
168     int borrow;
169     int stop = (int)MIN(op1->size, op2->allocated);
170     //
171     // Make sure that op2 is not obviously larger than op1
172     pAssert(op1->size >= op2->size);
173     borrow = SubSame(result->d, op1->d, op2->d, stop);
174     if(op1->size > (crypt_uword_t)stop)
175         borrow = BorrowProp(
176             &result->d[stop], &op1->d[stop], (int)(op1->size - stop), borrow);
177     pAssert(!borrow);
178     BnSetTop(result, op1->size);
179     return TRUE;
180 }
181
182 /*** BnSubWord()
183 // This function subtracts a word value from a bigNum. This function always
184 // returns TRUE.
185 LIB_EXPORT BOOL BnSubWord(bigNum result, bigConst op, crypt_uword_t word)
186 {
187     int borrow;
188     //
189     pAssert(op->size > 1 || word <= op->d[0]);
190     borrow = word > op->d[0];
191     result->d[0] = op->d[0] - word;
192     borrow = BorrowProp(&result->d[1], &op->d[1], (int)(op->size - 1), borrow);
193     pAssert(!borrow);
194     BnSetTop(result, op->size);
195     return TRUE;
196 }
197
198 /*** BnUnsignedCmp()
199 // This function performs a comparison of op1 to op2. The compare is approximately
200 // constant time if the size of the values used in the compare is consistent
201 // across calls (from the same line in the calling code).
202 // Return Type: int
203 //     < 0      op1 is less than op2
204 //     0        op1 is equal to op2
205 //     > 0      op1 is greater than op2
206 LIB_EXPORT int BnUnsignedCmp(bigConst op1, bigConst op2)
207 {
208     int retVal;
209     int diff;
210     int i;
211     //
212     pAssert((op1 != NULL) && (op2 != NULL));
213     retVal = (int)(op1->size - op2->size);
214     if(retVal == 0)
215     {
216         for(i = (int)(op1->size - 1); i >= 0; i--)
217             {

```

```

218         diff = (op1->d[i] < op2->d[i]) ? -1 : (op1->d[i] != op2->d[i]);
219         retVal = retVal == 0 ? diff : retVal;
220     }
221 }
222 else
223     retVal = (retVal < 0) ? -1 : 1;
224 return retVal;
225 }
226
227 /*** BnUnsignedCmpWord()
228 // Compare a bigNum to a crypt_uword_t.
229 // Return Type: int
230 //     -1          op1 is less than word
231 //     0           op1 is equal to word
232 //     1           op1 is greater than word
233 LIB_EXPORT int BnUnsignedCmpWord(bigConst op1, crypt_uword_t word)
234 {
235     if(op1->size > 1)
236         return 1;
237     else if(op1->size == 1)
238         return (op1->d[0] < word) ? -1 : (op1->d[0] > word);
239     else // op1 is zero
240         // equal if word is zero
241         return (word == 0) ? 0 : -1;
242 }
243
244 /*** BnModWord()
245 // This function does modular division of a big number when the modulus is a
246 // word value.
247 LIB_EXPORT crypt_word_t BnModWord(bigConst numerator, crypt_word_t modulus)
248 {
249     BN_MAX(remainder);
250     BN_VAR(mod, RADIX_BITS);
251     //
252     mod->d[0] = modulus;
253     mod->size = (modulus != 0);
254     BnDiv(NULL, remainder, numerator, mod);
255     return remainder->d[0];
256 }
257
258 /*** Msb()
259 // This function returns the bit number of the most significant bit of a
260 // crypt_uword_t. The number for the least significant bit of any bigNum value is 0.
261 // The maximum return value is RADIX_BITS - 1,
262 // Return Type: int
263 //     -1          the word was zero
264 //     n           the bit number of the most significant bit in the word
265 static int Msb(crypt_uword_t word)
266 {
267     int retVal = -1;
268     //
269     #if RADIX_BITS == 64
270     if(word & 0xffffffff00000000)
271     {
272         retVal += 32;
273         word >>= 32;
274     }
275     #endif
276     if(word & 0xffff0000)
277     {
278         retVal += 16;
279         word >>= 16;
280     }
281     if(word & 0x0000ff00)
282     {
283         retVal += 8;

```

```

284     word >>= 8;
285 }
286 if(word & 0x000000f0)
287 {
288     retVal += 4;
289     word >>= 4;
290 }
291 if(word & 0x0000000c)
292 {
293     retVal += 2;
294     word >>= 2;
295 }
296 if(word & 0x00000002)
297 {
298     retVal += 1;
299     word >>= 1;
300 }
301 return retVal + (int)word;
302 }
303
304 /*** BnMsb()
305 // This function returns the number of the MSb of a bigNum value.
306 // Return Type: int
307 //     -1           the word was zero or 'bn' was NULL
308 //     n           the bit number of the most significant bit in the word
309 LIB_EXPORT int BnMsb(bigConst bn)
310 {
311     // If the value is NULL, or the size is zero then treat as zero and return -1
312     if(bn != NULL && bn->size > 0)
313     {
314         int retVal = Msb(bn->d[bn->size - 1]);
315         retVal += (int)(bn->size - 1) * RADIX_BITS;
316         return retVal;
317     }
318     else
319         return -1;
320 }
321
322 /*** BnSizeInBits()
323 // This function returns the number of bits required to hold a number. It is one
324 // greater than the Msb.
325 //
326 LIB_EXPORT unsigned BnSizeInBits(bigConst n)
327 {
328     int bits = BnMsb(n) + 1;
329     //
330     return bits < 0 ? 0 : (unsigned)bits;
331 }
332
333 /*** BnSetWord()
334 // Change the value of a bignum_t to a word value.
335 LIB_EXPORT bigNum BnSetWord(bigNum n, crypt_uword_t w)
336 {
337     if(n != NULL)
338     {
339         pAssert(n->allocated > 1);
340         n->d[0] = w;
341         BnSetTop(n, (w != 0) ? 1 : 0);
342     }
343     return n;
344 }
345
346 /*** BnSetBit()
347 // This function will SET a bit in a bigNum. Bit 0 is the least-significant bit in
348 // the 0th digit_t. The function will return FALSE if the bitNum is invalid, else
349 TRUE.

```

```

349 LIB_EXPORT BOOL BnSetBit(bigNum      bn,      // IN/OUT: big number to modify
350                          unsigned int bitNum // IN: Bit number to SET
351 )
352 {
353     crypt_ushort_t offset = bitNum / RADIX_BITS;
354     if(bitNum > bn->allocated * RADIX_BITS)
355     {
356         // out of range
357         return FALSE;
358     }
359     // Grow the number if necessary to set the bit.
360     while(bn->size <= offset)
361         bn->d[bn->size++] = 0;
362     bn->d[offset] |= ((crypt_ushort_t)1 << RADIX_MOD(bitNum));
363     return TRUE;
364 }
365
366 /*** BnTestBit()
367 // This function is used to check to see if a bit is SET in a bignum_t. The 0th bit
368 // is the LSB of d[0].
369 // Return Type: BOOL
370 //     TRUE(1)         the bit is set
371 //     FALSE(0)        the bit is not set or the number is out of range
372 LIB_EXPORT BOOL BnTestBit(bigNum      bn,      // IN: number to check
373                          unsigned int bitNum // IN: bit to test
374 )
375 {
376     crypt_ushort_t offset = RADIX_DIV(bitNum);
377     //
378     if(bn->size > offset)
379         return ((bn->d[offset] & (((crypt_ushort_t)1) << RADIX_MOD(bitNum))) != 0);
380     else
381         return FALSE;
382 }
383
384 /*** BnMaskBits()
385 // This function is used to mask off high order bits of a big number.
386 // The returned value will have no more than 'maskBit' bits
387 // set.
388 // Note: There is a requirement that unused words of a bignum_t are set to zero.
389 // Return Type: BOOL
390 //     TRUE(1)         result masked
391 //     FALSE(0)        the input was not as large as the mask
392 LIB_EXPORT BOOL BnMaskBits(bigNum      bn,      // IN/OUT: number to mask
393                          crypt_ushort_t maskBit // IN: the bit number for the mask.
394 )
395 {
396     crypt_ushort_t finalSize;
397     BOOL          retVal;
398
399     finalSize = BITS_TO_CRYPT_WORDS(maskBit);
400     retVal    = (finalSize <= bn->allocated);
401     if(retVal && (finalSize > 0))
402     {
403         crypt_ushort_t mask;
404         mask = ~((crypt_ushort_t)0) >> RADIX_MOD(maskBit);
405         bn->d[finalSize - 1] &= mask;
406     }
407     BnSetTop(bn, finalSize);
408     return retVal;
409 }
410
411 /*** BnShiftRight()
412 // This function will shift a bigNum to the right by the shiftAmount.
413 // This function always returns TRUE.
414 LIB_EXPORT BOOL BnShiftRight(bigNum result, bigConst toShift, uint32_t shiftAmount)

```



```

415 {
416     uint32_t      offset = (shiftAmount >> RADIX_LOG2);
417     uint32_t      i;
418     uint32_t      shiftIn;
419     crypt_uword_t finalSize;
420     //
421     shiftAmount = shiftAmount & RADIX_MASK;
422     shiftIn      = RADIX_BITS - shiftAmount;
423
424     // The end size is toShift->size - offset less one additional
425     // word if the shiftAmount would make the upper word == 0
426     if(toShift->size > offset)
427     {
428         finalSize = toShift->size - offset;
429         finalSize -= (toShift->d[toShift->size - 1] >> shiftAmount) == 0 ? 1 : 0;
430     }
431     else
432         finalSize = 0;
433
434     pAssert(finalSize <= result->allocated);
435     if(finalSize != 0)
436     {
437         for(i = 0; i < finalSize; i++)
438         {
439             result->d[i] = (toShift->d[i + offset] >> shiftAmount)
440                 | (toShift->d[i + offset + 1] << shiftIn);
441         }
442         if(offset == 0)
443             result->d[i] = toShift->d[i] >> shiftAmount;
444     }
445     BnSetTop(result, finalSize);
446     return TRUE;
447 }
448
449 /*** BnIsPointOnCurve()
450 // This function checks if a point is on the curve.
451 BOOL BnIsPointOnCurve(pointConst Q, const TPMBN_ECC_CURVE_CONSTANTS* C)
452 {
453     BN_VAR(right, (MAX_ECC_KEY_BITS * 3));
454     BN_VAR(left, (MAX_ECC_KEY_BITS * 2));
455     bigConst prime = BnCurveGetPrime(C);
456     //
457     // Show that point is on the curve  $y^2 = x^3 + ax + b$ ;
458     // Or  $y^2 = x(x^2 + a) + b$ 
459     //  $y^2$ 
460     BnMult(left, Q->y, Q->y);
461
462     BnMod(left, prime);
463     //  $x^2$ 
464     BnMult(right, Q->x, Q->x);
465
466     //  $x^2 + a$ 
467     BnAdd(right, right, BnCurveGet_a(C));
468
469     // ExtMath_Mod(right, CurveGetPrime(C));
470     //  $x(x^2 + a)$ 
471     BnMult(right, right, Q->x);
472
473     //  $x(x^2 + a) + b$ 
474     BnAdd(right, right, BnCurveGet_b(C));
475
476     BnMod(right, prime);
477     if(BnUnsignedCmp(left, right) == 0)
478         return TRUE;
479     else
480         return FALSE;

```

481 }

B.3.4. /tpm/cryptolib/TpmBigNum/BnMemory.c

```

1  /*** Introduction
2  // This file contains the memory setup functions used by the bigNum functions
3  // in CryptoEngine
4
5  /*** Includes
6  #include "TpmBigNum.h"
7
8  /*** Functions
9
10 /*** BnSetTop()
11 // This function is used when the size of a bignum_t is changed. It
12 // makes sure that the unused words are set to zero and that any significant
13 // words of zeros are eliminated from the used size indicator.
14 LIB_EXPORT bigNum BnSetTop(bigNum bn, // IN/OUT: number to clean
15                             crypt_ushort_t top // IN: the new top
16 )
17 {
18     if(bn != NULL)
19     {
20         pAssert(top <= bn->allocated);
21         // If forcing the size to be decreased, make sure that the words being
22         // discarded are being set to 0
23         while(bn->size > top)
24             bn->d[--bn->size] = 0;
25         bn->size = top;
26         // Now make sure that the words that are left are 'normalized' (no high-order
27         // words of zero.
28         while((bn->size > 0) && (bn->d[bn->size - 1] == 0))
29             bn->size -= 1;
30     }
31     return bn;
32 }
33
34 /*** BnClearTop()
35 // This function will make sure that all unused words are zero.
36 LIB_EXPORT bigNum BnClearTop(bigNum bn)
37 {
38     crypt_ushort_t i;
39     //
40     if(bn != NULL)
41     {
42         for(i = bn->size; i < bn->allocated; i++)
43             bn->d[i] = 0;
44         while((bn->size > 0) && (bn->d[bn->size] == 0))
45             bn->size -= 1;
46     }
47     return bn;
48 }
49
50 /*** BnInitializeWord()
51 // This function is used to initialize an allocated bigNum with a word value. The
52 // bigNum does not have to be allocated with a single word.
53 LIB_EXPORT bigNum BnInitializeWord(bigNum bn, // IN:
54                                     crypt_ushort_t allocated, // IN:
55                                     crypt_ushort_t word // IN:
56 )
57 {
58     bn->allocated = allocated;
59     bn->size = (word != 0);
60     bn->d[0] = word;
61     while(allocated > 1)

```

```

62     bn->d[--allocated] = 0;
63     return bn;
64 }
65
66 /*** BnInit()
67 // This function initializes a stack allocated bignum_t. It initializes
68 // 'allocated' and 'size' and zeros the words of 'd'.
69 LIB_EXPORT bigNum BnInit(bigNum bn, crypt_ushort_t allocated)
70 {
71     if(bn != NULL)
72     {
73         bn->allocated = allocated;
74         bn->size      = 0;
75         while(allocated != 0)
76             bn->d[--allocated] = 0;
77     }
78     return bn;
79 }
80
81 /*** BnCopy()
82 // Function to copy a bignum_t. If the output is NULL, then
83 // nothing happens. If the input is NULL, the output is set
84 // to zero.
85 LIB_EXPORT BOOL BnCopy(bigNum out, bigConst in)
86 {
87     if(in == out)
88         BnSetTop(out, BnGetSize(out));
89     else if(out != NULL)
90     {
91         if(in != NULL)
92         {
93             unsigned int i;
94             pAssert(BnGetAllocated(out) >= BnGetSize(in));
95             for(i = 0; i < BnGetSize(in); i++)
96                 out->d[i] = in->d[i];
97             BnSetTop(out, BnGetSize(in));
98         }
99         else
100             BnSetTop(out, 0);
101     }
102     return TRUE;
103 }
104
105 #if ALG_ECC
106
107 /*** BnPointCopy()
108 // Function to copy a bn point.
109 LIB_EXPORT BOOL BnPointCopy(bigPoint pOut, pointConst pIn)
110 {
111     return BnCopy(pOut->x, pIn->x) && BnCopy(pOut->y, pIn->y)
112           && BnCopy(pOut->z, pIn->z);
113 }
114
115 /*** BnInitializePoint()
116 // This function is used to initialize a point structure with the addresses
117 // of the coordinates.
118 LIB_EXPORT bn_point_t* BnInitializePoint(
119     bigPoint p, // OUT: structure to receive pointers
120     bigNum x,   // IN: x coordinate
121     bigNum y,   // IN: y coordinate
122     bigNum z    // IN: x coordinate
123 )
124 {
125     p->x = x;
126     p->y = y;
127     p->z = z;

```

```

128     BnSetWord(z, 1);
129     return p;
130 }
131
132 #endif // ALG_ECC

```

B.3.5. /tpm/cryptolib/TpmBigNum/BnUtil.c

```

1  /** Introduction
2  // Utility functions to support TpmBigNum library
3
4  #include "TpmBigNum.h"

```

B.3.6. /tpm/cryptolib/TpmBigNum/TpmBigNum.h

```

1  /** Introduction
2  // This file contains the headers necessary to build the tpm big num library.
3  // TODO_RENAME_INC_FOLDER: public refers to the TPM_CoreLib public headers
4  #include <public/tpm_public.h>
5  #include <public/prototypes/TpmFail_fp.h>
6  // TODO_RENAME_INC_FOLDER: private refers to the TPM_CoreLib private(protected)
   headers
7  #include <public/TpmAlgorithmDefines.h>
8  #include <public/GpMacros.h> // required for TpmFail_fp.h
9  #include <public/Capabilities.h>
10 #include <public/TpmTypes.h> // requires capabilities & GpMacros
11 #include <TpmBigNum/TpmToTpmBigNumMath.h>
12 #include "BnSupport_Interface.h"
13 #include "BnConvert_fp.h"
14 #include "BnMemory_fp.h"
15 #include "BnMath_fp.h"
16 #include "BnUtil_fp.h"
17 #include <MathLibraryInterface.h>

```

B.3.7. /tpm/cryptolib/TpmBigNum/TpmBigNumThunks.c

```

1  /** Introduction
2  // This file contains BN Thunks between the MathInterfaceLibrary types and the
3  // bignum_t types.
4
5  #include "TpmBigNum.h"
6
7  // Note - these were moved out of TPM_INLINE to build correctly on GCC. On MSVC
8  // link time code generation correctly handles the inline versions, but
9  // it isn't portable to GCC.
10
11 // *****
12 // Library Level Functions
13 // *****
14
15 // Called when system is initializing to allow math libraries to perform
16 // startup actions.
17 LIB_EXPORT int ExtMath_LibInit(void)
18 {
19     return BnSupportLibInit();
20 }
21
22 /** MathLibraryCompatibilityCheck()
23 // This function is only used during development to make sure that the library
24 // that is being referenced is using the same size of data structures as the TPM.
25 LIB_EXPORT BOOL ExtMath_Debug_CompatibilityCheck(void)
26 {
27     return BnMathLibraryCompatibilityCheck();

```

```

28 }
29
30 // *****
31 // Integer/Number Functions (non-ECC)
32 // *****
33 // #####
34 // type initializers
35 // #####
36 LIB_EXPORT Crypt_Int* ExtMath_Initialize_Int(Crypt_Int* var, NUMBYTES bitCount)
37 {
38     return (Crypt_Int*)BnInit((bigNum)var, BN_STRUCT_ALLOCATION(bitCount));
39 }
40
41 // #####
42 // Buffer Converters
43 // #####
44 LIB_EXPORT Crypt_Int* ExtMath_IntFromBytes(
45     Crypt_Int* buffer, const BYTE* input, NUMBYTES byteCount)
46 {
47     return (Crypt_Int*)BnFromBytes((bigNum)buffer, input, byteCount);
48 }
49
50 LIB_EXPORT BOOL ExtMath_IntToBytes(
51     const Crypt_Int* value, BYTE* output, NUMBYTES* pByteCount)
52 {
53     return BnToBytes((bigConst)value, output, pByteCount);
54 }
55
56 LIB_EXPORT Crypt_Int* ExtMath_SetWord(Crypt_Int* n, crypt_uword_t w)
57 {
58     return (Crypt_Int*)BnSetWord((bigNum)n, w);
59 }
60 // #####
61 // Copy Functions
62 // #####
63 LIB_EXPORT BOOL ExtMath_Copy(Crypt_Int* out, const Crypt_Int* in)
64 {
65     return BnCopy((bigNum)out, (bigConst)in);
66 }
67
68 // #####
69 // Ordinary Arithmetic, writ large
70 // #####
71
72 /** ExtMath_Multiply()
73  * Multiplies two numbers and returns the result
74  */
75 LIB_EXPORT BOOL ExtMath_Multiply(
76     Crypt_Int* result, const Crypt_Int* multiplicand, const Crypt_Int* multiplier)
77 {
78     return BnMult((bigNum)result, (bigConst)multiplicand, (bigConst)multiplier);
79 }
80
81 /** ExtMath_Divide()
82  * This function divides two Crypt_Int* values. The function returns FALSE if there is
83  * an error in the operation. Quotient may be null, in which case this function
84  * returns
85  * only the remainder.
86  */
87 LIB_EXPORT BOOL ExtMath_Divide(Crypt_Int* quotient,
88     Crypt_Int* remainder,
89     const Crypt_Int* dividend,
90     const Crypt_Int* divisor)
91 {
92     return BnDiv(
93         (bigNum)quotient, (bigNum)remainder, (bigConst)dividend, (bigConst)divisor);
94 }

```

```

93  #if ALG_RSA
94  /** ExtMath_GCD()
95  // Get the greatest common divisor of two numbers. This function is only needed
96  // when the TPM implements RSA.
97  LIB_EXPORT BOOL ExtMath_GCD(
98      Crypt_Int* gcd, const Crypt_Int* number1, const Crypt_Int* number2)
99  {
100     return BnGcd((bigNum)gcd, (bigConst)number1, (bigConst)number2);
101 }
102 #endif // ALG_RSA
103
104 /** ExtMath_Add()
105 // This function adds two Crypt_Int* values. This function always returns TRUE.
106 LIB_EXPORT BOOL ExtMath_Add(
107     Crypt_Int* result, const Crypt_Int* op1, const Crypt_Int* op2)
108 {
109     return BnAdd((bigNum)result, (bigConst)op1, (bigConst)op2);
110 }
111
112 /** ExtMath_AddWord()
113 // This function adds a word value to a Crypt_Int*. This function always returns TRUE.
114 LIB_EXPORT BOOL ExtMath_AddWord(
115     Crypt_Int* result, const Crypt_Int* op, crypt_ushort_t word)
116 {
117     return BnAddWord((bigNum)result, (bigConst)op, word);
118 }
119
120 /** ExtMath_Subtract()
121 // This function does subtraction of two Crypt_Int* values and returns result = op1 -
122 // op2
123 // when op1 is greater than op2. If op2 is greater than op1, then a fault is
124 // generated. This function always returns TRUE.
125 LIB_EXPORT BOOL ExtMath_Subtract(
126     Crypt_Int* result, const Crypt_Int* op1, const Crypt_Int* op2)
127 {
128     return BnSub((bigNum)result, (bigConst)op1, (bigConst)op2);
129 }
130
131 /** ExtMath_SubtractWord()
132 // This function subtracts a word value from a Crypt_Int*. This function always
133 // returns TRUE.
134 LIB_EXPORT BOOL ExtMath_SubtractWord(
135     Crypt_Int* result, const Crypt_Int* op, crypt_ushort_t word)
136 {
137     return BnSubWord((bigNum)result, (bigConst)op, word);
138 }
139
140 // #####
141 // Modular Arithmetic, writ large
142 // #####
143 // define Mod in terms of Divide
144 LIB_EXPORT BOOL ExtMath_Mod(Crypt_Int* valueAndResult, const Crypt_Int* modulus)
145 {
146     return ExtMath_Divide(NULL, valueAndResult, valueAndResult, modulus);
147 }
148
149 /** ExtMath_ModMult()
150 // Does 'op1' * 'op2' and divide by 'modulus' returning the remainder of the divide.
151 LIB_EXPORT BOOL ExtMath_ModMult(Crypt_Int* result,
152     const Crypt_Int* op1,
153     const Crypt_Int* op2,
154     const Crypt_Int* modulus)
155 {
156     return BnModMult((bigNum)result, (bigConst)op1, (bigConst)op2, (bigConst)modulus);
157 }

```



```

158 #if ALG_RSA
159 /** ExtMath_ModExp()
160 // Do modular exponentiation using Crypt_Int* values. This function is only needed
161 // when the TPM implements RSA.
162 LIB_EXPORT BOOL ExtMath_ModExp(Crypt_Int*      result,
163                               const Crypt_Int* number,
164                               const Crypt_Int* exponent,
165                               const Crypt_Int* modulus)
166 {
167     return BnModExp(
168         (bigNum)result, (bigConst)number, (bigConst)exponent, (bigConst)modulus);
169 }
170 #endif // ALG_RSA
171
172 /** ExtMath_ModInverse()
173 // Modular multiplicative inverse.
174 LIB_EXPORT BOOL ExtMath_ModInverse(
175     Crypt_Int* result, const Crypt_Int* number, const Crypt_Int* modulus)
176 {
177     return BnModInverse((bigNum)result, (bigConst)number, (bigConst)modulus);
178 }
179
180 /** ExtMath_ModWord()
181 // This function does modular division of a big number when the modulus is a
182 // word value.
183 LIB_EXPORT crypt_word_t ExtMath_ModWord(const Crypt_Int* numerator,
184                                         crypt_word_t      modulus)
185 {
186     return BnModWord((bigConst)numerator, modulus);
187 }
188
189 // #####
190 // Queries
191 // #####
192
193 /** ExtMath_UnsignedCmp()
194 // This function performs a comparison of op1 to op2. The compare is approximately
195 // constant time if the size of the values used in the compare is consistent
196 // across calls (from the same line in the calling code).
197 // Return Type: int
198 //     < 0      op1 is less than op2
199 //     0        op1 is equal to op2
200 //     > 0      op1 is greater than op2
201 LIB_EXPORT int ExtMath_UnsignedCmp(const Crypt_Int* op1, const Crypt_Int* op2)
202 {
203     return BnUnsignedCmp((bigConst)op1, (bigConst)op2);
204 }
205
206 /** ExtMath_UnsignedCmpWord()
207 // Compare a Crypt_Int* to a crypt_uword_t.
208 // Return Type: int
209 //     -1      op1 is less than word
210 //     0       op1 is equal to word
211 //     1       op1 is greater than word
212 LIB_EXPORT int ExtMath_UnsignedCmpWord(const Crypt_Int* op1, crypt_uword_t word)
213 {
214     return BnUnsignedCmpWord((bigConst)op1, word);
215 }
216
217 LIB_EXPORT BOOL ExtMath_IsEqualWord(const Crypt_Int* bn, crypt_uword_t word)
218 {
219     return BnEqualWord((bigConst)bn, word);
220 }
221
222 LIB_EXPORT BOOL ExtMath_IsZero(const Crypt_Int* op1)
223 {

```

```

224     return BnEqualZero((bigConst)op1);
225 }
226
227 /*** ExtMath_MostSigBitNum()
228 // This function returns the number of the MSb of a Crypt_Int* value.
229 // Return Type: int
230 //     -1           the word was zero or 'bn' was NULL
231 //     n           the bit number of the most significant bit in the word
232 LIB_EXPORT int ExtMath_MostSigBitNum(const Crypt_Int* bn)
233 {
234     return BnMsb((bigConst)bn);
235 }
236
237 LIB_EXPORT uint32_t ExtMath_GetLeastSignificant32bits(const Crypt_Int* bn)
238 {
239     MUST_BE(RADIX_BITS >= 32);
240     #if RADIX_BITS == 32
241         return BnGetWord(bn, 0);
242     #else
243         // RADIX_BITS must be > 32 by MUST_BE above.
244         return (uint32_t)(BnGetWord(bn, 0) & 0xFFFFFFFF);
245     #endif
246 }
247
248 /*** ExtMath_SizeInBits()
249 // This function returns the number of bits required to hold a number. It is one
250 // greater than the Msb.
251 LIB_EXPORT unsigned ExtMath_SizeInBits(const Crypt_Int* n)
252 {
253     return BnSizeInBits((bigConst)n);
254 }
255
256 // #####
257 // Bitwise Operations
258 // #####
259
260 LIB_EXPORT BOOL ExtMath_SetBit(Crypt_Int* bn, unsigned int bitNum)
261 {
262     return BnSetBit((bigNum)bn, bitNum);
263 }
264
265 // This function is used to check to see if a bit is SET in a bigNum_t. The 0th bit
266 /*** ExtMath_TestBit()
267 // is the LSB of d[0].
268 // Return Type: BOOL
269 //     TRUE(1)       the bit is set
270 //     FALSE(0)      the bit is not set or the number is out of range
271 LIB_EXPORT BOOL ExtMath_TestBit(Crypt_Int* bn, // IN: number to check
272                                unsigned int bitNum // IN: bit to test
273 )
274 {
275     return BnTestBit((bigNum)bn, bitNum);
276 }
277
278 /***ExtMath_MaskBits()
279 // This function is used to mask off high order bits of a big number.
280 // The returned value will have no more than 'maskBit' bits
281 // set.
282 // Note: There is a requirement that unused words of a bigNum_t are set to zero.
283 // Return Type: BOOL
284 //     TRUE(1)       result masked
285 //     FALSE(0)      the input was not as large as the mask
286 LIB_EXPORT BOOL ExtMath_MaskBits(
287     Crypt_Int* bn, // IN/OUT: number to mask
288     crypt_uword_t maskBit // IN: the bit number for the mask.
289 )

```

```

290 {
291     return BnMaskBits((bigNum)bn, maskBit);
292 }
293
294 /*** ExtMath_ShiftRight()
295 // This function will shift a Crypt_Int* to the right by the shiftAmount.
296 // This function always returns TRUE.
297 LIB_EXPORT BOOL ExtMath_ShiftRight(
298     Crypt_Int* result, const Crypt_Int* toShift, uint32_t shiftAmount)
299 {
300     return BnShiftRight((bigNum)result, (bigConst)toShift, shiftAmount);
301 }
302
303 // *****
304 // ECC Functions
305 // *****
306 // #####
307 // Point initializers
308 // #####
309 LIB_EXPORT Crypt_Point* ExtEcc_Initialize_Point(Crypt_Point* point, NUMBYTES bitCount)
310 {
311     // Since we define the structure, we know that BN_POINT_BUFs are a bn_point_t
312     // followed by bignums.
313     // and that the size is always the MAX_ECC_KEY_SIZE
314     // tell the individual bignums how large they are:
315     bn_fullpoint_t* pBuf = (bn_fullpoint_t*)point;
316     BnInit((bigNum) & (pBuf->x), BN_STRUCT_ALLOCATION(bitCount));
317     BnInit((bigNum) & (pBuf->y), BN_STRUCT_ALLOCATION(bitCount));
318     BnInit((bigNum) & (pBuf->z), BN_STRUCT_ALLOCATION(bitCount));
319
320     // now feed the addresses of those coordinates to the bn_point_t structure
321     bn_point_t* bnPoint = (bn_point_t*)point;
322     BnInitializePoint(
323         bnPoint, (bigNum) & (pBuf->x), (bigNum) & (pBuf->y), (bigNum) & (pBuf->z));
324     return point;
325 }
326 // #####
327 // Curve initializers
328 // #####
329 LIB_EXPORT const Crypt_EccCurve* ExtEcc_CurveInitialize(Crypt_EccCurve* E,
330                                                         TPM_ECC_CURVE curveId)
331 {
332     return BnCurveInitialize((bigCurveData*)E, curveId);
333 }
334
335 // #####
336 // Curve DESTRUCTOR
337 // #####
338 // WARNING: Not guaranteed to be called in presence of LONGJMP.
339 LIB_EXPORT void ExtEcc_CurveFree(const Crypt_EccCurve* E)
340 {
341     BnCurveFree((bigCurveData*)E);
342 }
343
344 // #####
345 // Buffer Converters
346 // #####
347 /*** BnPointFromBytes()
348 // Function to create a BIG_POINT structure from a 2B point.
349 // A point is going to be two ECC values in the same buffer. The values are going
350 // to be the size of the modulus. They are in modular form.
351 LIB_EXPORT Crypt_Point* ExtEcc_PointFromBytes(Crypt_Point* point,
352                                               const BYTE* x,
353                                               NUMBYTES nBytesX,
354                                               const BYTE* y,

```

```

355                                     NUMBYTES      nBytesY)
356 {
357     return (Crypt_Point*)BnPointFromBytes((bigPoint)point, x, nBytesX, y, nBytesY);
358 }
359
360 LIB_EXPORT BOOL ExtEcc_PointToBytes(
361     const Crypt_Point* point, BYTE* x, NUMBYTES* pBytesX, BYTE* y, NUMBYTES* pBytesY)
362 {
363     return BnPointToBytes((pointConst)point, x, pBytesX, y, pBytesY);
364 }
365
366 // #####
367 // ECC Point Operations
368 // #####
369 /** ExtEcc_PointMultiply()
370 // This function does a point multiply of the form R = [d]S. A return of FALSE
371 // indicates that the result was the point at infinity. This function is only needed
372 // if the TPM supports ECC.
373 LIB_EXPORT BOOL ExtEcc_PointMultiply(
374     Crypt_Point* R, const Crypt_Point* S, const Crypt_Int* d, const Crypt_EccCurve* E)
375 {
376     return BnEccModMult((bigPoint)R, (pointConst)S, (bigConst)d, (bigCurveData*)E);
377 }
378
379 /** ExtEcc_PointMultiplyAndAdd()
380 // This function does a point multiply of the form R = [d]S + [u]Q. A return of
381 // FALSE indicates that the result was the point at infinity. This function is only
382 // needed if the TPM supports ECC.
383 LIB_EXPORT BOOL ExtEcc_PointMultiplyAndAdd(Crypt_Point*      R,
384     const Crypt_Point* S,
385     const Crypt_Int*   d,
386     const Crypt_Point* Q,
387     const Crypt_Int*   u,
388     const Crypt_EccCurve* E)
389 {
390     return BnEccModMult2((bigPoint)R,
391         (pointConst)S,
392         (bigConst)d,
393         (pointConst)Q,
394         (bigConst)u,
395         (bigCurveData*)E);
396 }
397
398 LIB_EXPORT BOOL ExtEcc_PointAdd(Crypt_Point*      R,
399     const Crypt_Point* S,
400     const Crypt_Point* Q,
401     const Crypt_EccCurve* E)
402 {
403     return BnEccAdd((bigPoint)R, (pointConst)S, (pointConst)Q, (bigCurveData*)E);
404 }
405
406 // #####
407 // ECC Point Information
408 // #####
409 LIB_EXPORT BOOL ExtEcc_IsPointOnCurve(const Crypt_Point* Q, const Crypt_EccCurve* E)
410 {
411     return BnIsPointOnCurve((pointConst)Q, AccessCurveConstants(E));
412 }
413
414 LIB_EXPORT const Crypt_Int* ExtEcc_PointX(const Crypt_Point* point)
415 {
416     return (const Crypt_Int*)((pointConst)point->x);
417 }
418
419 LIB_EXPORT BOOL ExtEcc_IsInfinityPoint(const Crypt_Point* point)
420 {

```

```

421     return BnEqualZero((pointConst)point)->z);
422 }
423
424 // #####
425 // ECC Curve Information
426 // #####
427 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetPrime(TPM_ECC_CURVE curveId)
428 {
429     return (const Crypt_Int*)BnCurveGetPrime(BnGetCurveData(curveId));
430 }
431
432 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetOrder(TPM_ECC_CURVE curveId)
433 {
434     return (const Crypt_Int*)BnCurveGetOrder(BnGetCurveData(curveId));
435 }
436
437 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetCofactor(TPM_ECC_CURVE curveId)
438 {
439     return (const Crypt_Int*)BnCurveGetCofactor(BnGetCurveData(curveId));
440 }
441
442 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGet_a(TPM_ECC_CURVE curveId)
443 {
444     return (const Crypt_Int*)BnCurveGet_a(BnGetCurveData(curveId));
445 }
446
447 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGet_b(TPM_ECC_CURVE curveId)
448 {
449     return (const Crypt_Int*)BnCurveGet_b(BnGetCurveData(curveId));
450 }
451
452 LIB_EXPORT const Crypt_Point* ExtEcc_CurveGetG(TPM_ECC_CURVE curveId)
453 {
454     return (const Crypt_Point*)BnCurveGetG(BnGetCurveData(curveId));
455 }
456
457 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetGx(TPM_ECC_CURVE curveId)
458 {
459     return (const Crypt_Int*)BnCurveGetGx(BnGetCurveData(curveId));
460 }
461
462 LIB_EXPORT const Crypt_Int* ExtEcc_CurveGetGy(TPM_ECC_CURVE curveId)
463 {
464     return (const Crypt_Int*)BnCurveGetGy(BnGetCurveData(curveId));
465 }
466
467 LIB_EXPORT TPM_ECC_CURVE ExtEcc_CurveGetCurveId(const Crypt_EccCurve* E)
468 {
469     return BnCurveGetCurveId(AccessCurveConstants(E));
470 }

```

B.3.8. /tpm/cryptolib/TpmBigNum/include/BnConvert_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Mar 28, 2019   Time: 08:25:18PM
4  */
5
6  #ifndef _BN_CONVERT_FP_H_
7  #define _BN_CONVERT_FP_H_
8
9  /*** BnFromBytes()
10 // This function will convert a big-endian byte array to the internal number
11 // format. If bn is NULL, then the output is NULL. If bytes is null or the
12 // required size is 0, then the output is set to zero

```

```

13  LIB_EXPORT bigNum BnFromBytes(bigNum bn, const BYTE* bytes, NUMBYTES nBytes);
14
15  /*** BnFrom2B()
16  // Convert an TPM2B to a BIG_NUM.
17  // If the input value does not exist, or the output does not exist, or the input
18  // will not fit into the output the function returns NULL
19  LIB_EXPORT bigNum BnFrom2B(bigNum bn, // OUT:
20                             const TPM2B* a2B // IN: number to convert
21  );
22
23  /*** BnToBytes()
24  // This function converts a BIG_NUM to a byte array. It converts the bigNum to a
25  // big-endian byte string and sets 'size' to the normalized value. If 'size' is an
26  // input 0, then the receiving buffer is guaranteed to be large enough for the result
27  // and the size will be set to the size required for bigNum (leading zeros
28  // suppressed).
29  //
30  // The conversion for a little-endian machine simply requires that all significant
31  // bytes of the bigNum be reversed. For a big-endian machine, rather than
32  // unpack each word individually, the bigNum is converted to little-endian words,
33  // copied, and then converted back to big-endian.
34  LIB_EXPORT BOOL BnToBytes(bigConst bn,
35                           BYTE* buffer,
36                           NUMBYTES* size // This the number of bytes that are
37                                           // available in the buffer. The result
38                                           // should be this big.
39  );
40
41  /*** BnTo2B()
42  // Function to convert a BIG_NUM to TPM2B.
43  // The TPM2B size is set to the requested 'size' which may require padding.
44  // If 'size' is non-zero and less than required by the value in 'bn' then an error
45  // is returned. If 'size' is zero, then the TPM2B is assumed to be large enough
46  // for the data and a2b->size will be adjusted accordingly.
47  LIB_EXPORT BOOL BnTo2B(bigConst bn, // IN:
48                        TPM2B* a2B, // OUT:
49                        NUMBYTES size // IN: the desired size
50  );
51  #if ALG_ECC
52
53  /*** BnPointFromBytes()
54  // Function to create a BIG_POINT structure from a byte buffer in big-endian order.
55  // A point is going to be two ECC values in the same buffer. The values are going
56  // to be the size of the modulus. They are in modular form.
57  LIB_EXPORT bn_point_t* BnPointFromBytes(
58      bigPoint ecP, // OUT: the preallocated point structure
59      const BYTE* x,
60      NUMBYTES nBytesX,
61      const BYTE* y,
62      NUMBYTES nBytesY);
63
64  /*** BnPointToBytes()
65  // This function converts a BIG_POINT into a TPMS_ECC_POINT. A TPMS_ECC_POINT
66  // contains two TPM2B_ECC_PARAMETER values. The maximum size of the parameters
67  // is dependent on the maximum EC key size used in an implementation.
68  // The presumption is that the TPMS_ECC_POINT is large enough to hold 2 TPM2B
69  // values, each as large as a MAX_ECC_PARAMETER_BYTES
70  LIB_EXPORT BOOL BnPointToBytes(
71      pointConst ecP, // OUT: the preallocated point structure
72      BYTE* x,
73      NUMBYTES* pBytesX,
74      BYTE* y,
75      NUMBYTES* pBytesY);
76  #endif // ALG_ECC
77
78  #endif // _BN_CONVERT_FP_H_

```


B.3.9. /tpm/cryptolib/TpmBigNum/include/BnMath_fp.h

```

1  /* (Auto-generated)
2  *   Created by TpmPrototypes; Version 3.0 July 18, 2017
3  *   Date: Aug 30, 2019   Time: 02:11:54PM
4  */
5
6  #ifndef _BN_MATH_FP_H
7  #define _BN_MATH_FP_H
8
9  /*** BnAdd()
10 // This function adds two bigNum values. This function always returns TRUE.
11 LIB_EXPORT BOOL BnAdd(bigNum result, bigConst op1, bigConst op2);
12
13 /*** BnAddWord()
14 // This function adds a word value to a bigNum. This function always returns TRUE.
15 LIB_EXPORT BOOL BnAddWord(bigNum result, bigConst op, crypt_ushort_t word);
16
17 /*** BnSub()
18 // This function does subtraction of two bigNum values and returns result = op1 - op2
19 // when op1 is greater than op2. If op2 is greater than op1, then a fault is
20 // generated. This function always returns TRUE.
21 LIB_EXPORT BOOL BnSub(bigNum result, bigConst op1, bigConst op2);
22
23 /*** BnSubWord()
24 // This function subtracts a word value from a bigNum. This function always
25 // returns TRUE.
26 LIB_EXPORT BOOL BnSubWord(bigNum result, bigConst op, crypt_ushort_t word);
27
28 /*** BnUnsignedCmp()
29 // This function performs a comparison of op1 to op2. The compare is approximately
30 // constant time if the size of the values used in the compare is consistent
31 // across calls (from the same line in the calling code).
32 // Return Type: int
33 //   < 0      op1 is less than op2
34 //   0        op1 is equal to op2
35 //   > 0      op1 is greater than op2
36 LIB_EXPORT int BnUnsignedCmp(bigConst op1, bigConst op2);
37
38 /*** BnUnsignedCmpWord()
39 // Compare a bigNum to a crypt_ushort_t.
40 // Return Type: int
41 //   -1       op1 is less than word
42 //   0        op1 is equal to word
43 //   1        op1 is greater than word
44 LIB_EXPORT int BnUnsignedCmpWord(bigConst op1, crypt_ushort_t word);
45
46 /*** BnModWord()
47 // This function does modular division of a big number when the modulus is a
48 // word value.
49 LIB_EXPORT crypt_ushort_t BnModWord(bigConst numerator, crypt_ushort_t modulus);
50
51 /*** BnMsb()
52 // This function returns the number of the MSb of a bigNum value.
53 // Return Type: int
54 //   -1       the word was zero or 'bn' was NULL
55 //   n        the bit number of the most significant bit in the word
56 LIB_EXPORT int BnMsb(bigConst bn);
57
58 /*** BnSizeInBits()
59 // This function returns the number of bits required to hold a number. It is one
60 // greater than the Msb.
61 //
62 LIB_EXPORT unsigned BnSizeInBits(bigConst n);
63
64 /*** BnSetWord()

```

```

65 // Change the value of a bignum_t to a word value.
66 LIB_EXPORT bigNum BnSetWord(bigNum n, crypt_uword_t w);
67
68 /*** BnSetBit()
69 // This function will SET a bit in a bigNum. Bit 0 is the least-significant bit in
70 // the 0th digit_t. The function always return TRUE
71 LIB_EXPORT BOOL BnSetBit(bigNum bn, // IN/OUT: big number to modify
72                          unsigned int bitNum // IN: Bit number to SET
73 );
74
75 /*** BnTestBit()
76 // This function is used to check to see if a bit is SET in a bignum_t. The 0th bit
77 // is the LSb of d[0].
78 // Return Type: BOOL
79 // TRUE(1) the bit is set
80 // FALSE(0) the bit is not set or the number is out of range
81 LIB_EXPORT BOOL BnTestBit(bigNum bn, // IN: number to check
82                          unsigned int bitNum // IN: bit to test
83 );
84
85 /***BnMaskBits()
86 // This function is used to mask off high order bits of a big number.
87 // The returned value will have no more than 'maskBit' bits
88 // set.
89 // Note: There is a requirement that unused words of a bignum_t are set to zero.
90 // Return Type: BOOL
91 // TRUE(1) result masked
92 // FALSE(0) the input was not as large as the mask
93 LIB_EXPORT BOOL BnMaskBits(bigNum bn, // IN/OUT: number to mask
94                          crypt_uword_t maskBit // IN: the bit number for the mask.
95 );
96
97 /*** BnShiftRight()
98 // This function will shift a bigNum to the right by the shiftAmount.
99 // This function always returns TRUE.
100 LIB_EXPORT BOOL BnShiftRight(bigNum result, bigConst toShift, uint32_t shiftAmount);
101
102 /*** BnGetCurveData()
103 // This function returns the pointer for the parameter data
104 // associated with a curve.
105 const TPMBN_ECC_CURVE_CONSTANTS* BnGetCurveData(TPM_ECC_CURVE curveId);
106
107 /*** BnIsPointOnCurve()
108 // This function checks if a point is on the curve.
109 BOOL BnIsPointOnCurve(pointConst Q, const TPMBN_ECC_CURVE_CONSTANTS* C);
110
111 #endif // _BN_MATH_FP_H

```

B.3.10. /tpm/cryptolib/TpmBigNum/include/BnMemory_fp.h

```

1 /*(Auto-generated)
2 * Created by TpmPrototypes; Version 3.0 July 18, 2017
3 * Date: Mar 28, 2019 Time: 08:25:18PM
4 */
5
6 #ifndef _BN_MEMORY_FP_H
7 #define _BN_MEMORY_FP_H
8
9 /*** BnSetTop()
10 // This function is used when the size of a bignum_t is changed. It
11 // makes sure that the unused words are set to zero and that any significant
12 // words of zeros are eliminated from the used size indicator.
13 LIB_EXPORT bigNum BnSetTop(bigNum bn, // IN/OUT: number to clean
14                          crypt_uword_t top // IN: the new top
15 );

```

```

16
17  /*** BnClearTop()
18  // This function will make sure that all unused words are zero.
19  LIB_EXPORT bigNum BnClearTop(bigNum bn);
20
21  /*** BnInitializeWord()
22  // This function is used to initialize an allocated bigNum with a word value. The
23  // bigNum does not have to be allocated with a single word.
24  LIB_EXPORT bigNum BnInitializeWord(bigNum bn, // IN:
25                                     crypt_ushort_t allocated, // IN:
26                                     crypt_ushort_t word // IN:
27  );
28
29  /*** BnInit()
30  // This function initializes a stack allocated bignum_t. It initializes
31  // 'allocated' and 'size' and zeros the words of 'd'.
32  LIB_EXPORT bigNum BnInit(bigNum bn, crypt_ushort_t allocated);
33
34  /*** BnCopy()
35  // Function to copy a bignum_t. If the output is NULL, then
36  // nothing happens. If the input is NULL, the output is set
37  // to zero.
38  LIB_EXPORT BOOL BnCopy(bigNum out, bigConst in);
39  #if ALG_ECC
40
41  /*** BnPointCopy()
42  // Function to copy a bn point.
43  LIB_EXPORT BOOL BnPointCopy(bigPoint pOut, pointConst pIn);
44
45  /*** BnInitializePoint()
46  // This function is used to initialize a point structure with the addresses
47  // of the coordinates.
48  LIB_EXPORT bn_point_t* BnInitializePoint(
49      bigPoint p, // OUT: structure to receive pointers
50      bigNum x, // IN: x coordinate
51      bigNum y, // IN: y coordinate
52      bigNum z // IN: x coordinate
53  );
54  #endif // ALG_ECC
55
56  #endif // _BN_MEMORY_FP_H

```

B.3.11. /tpm/cryptolib/TpmBigNum/include/BnSupport_Interface.h

```

1  /*** Introduction
2  // Prototypes for functions the bignum library requires
3  // from a bignum-based math support library.
4  // Functions contained in the MathInterface but not listed here are provided by
5  // the TpmBigNum library itself.
6  //
7  // This file contains the function prototypes for the functions that need to be
8  // present in the selected math library. For each function listed, there should
9  // be a small stub function. That stub provides the interface between the TPM
10 // code and the support library. In most cases, the stub function will only need
11 // to do a format conversion between the TPM big number and the support library
12 // big number. The TPM big number format was chosen to make this relatively
13 // simple and fast.
14 //
15 // Arithmetic operations return a BOOL to indicate if the operation completed
16 // successfully or not.
17
18 #ifndef BN_SUPPORT_INTERFACE_H
19 #define BN_SUPPORT_INTERFACE_H
20 // TODO_RENAME_INC_FOLDER:private refers to the TPM_CoreLib private headers
21 #include <public/GpMacros.h>

```

```

22  #include <BnValues.h>
23
24  /** BnSupportLibInit()
25   // This function is called by CryptInit() so that necessary initializations can be
26   // performed on the cryptographic library.
27  LIB_EXPORT
28  int BnSupportLibInit(void);
29
30  /** MathLibraryCompatibilityCheck()
31   // This function is only used during development to make sure that the library
32   // that is being referenced is using the same size of data structures as the TPM.
33  BOOL BnMathLibraryCompatibilityCheck(void);
34
35  /** BnModMult()
36   // Does 'op1' * 'op2' and divide by 'modulus' returning the remainder of the divide.
37  LIB_EXPORT BOOL BnModMult(
38      bigNum result, bigConst op1, bigConst op2, bigConst modulus);
39
40  /** BnMult()
41   // Multiplies two numbers and returns the result
42  LIB_EXPORT BOOL BnMult(bigNum result, bigConst multiplicand, bigConst multiplier);
43
44  /** BnDiv()
45   // This function divides two bigNum values. The function returns FALSE if there is
46   // an error in the operation.
47  LIB_EXPORT BOOL BnDiv(
48      bigNum quotient, bigNum remainder, bigConst dividend, bigConst divisor);
49  /** BnMod()
50  #define BnMod(a, b) BnDiv(NULL, (a), (a), (b))
51
52  #if ALG_RSA
53  /** BnGcd()
54   // Get the greatest common divisor of two numbers. This function is only needed
55   // when the TPM implements RSA.
56  LIB_EXPORT BOOL BnGcd(bigNum gcd, bigConst number1, bigConst number2);
57
58  /** BnModExp()
59   // Do modular exponentiation using bigNum values. This function is only needed
60   // when the TPM implements RSA.
61  LIB_EXPORT BOOL BnModExp(
62      bigNum result, bigConst number, bigConst exponent, bigConst modulus);
63  #endif // ALG_RSA
64
65  /** BnModInverse()
66   // Modular multiplicative inverse.
67  LIB_EXPORT BOOL BnModInverse(bigNum result, bigConst number, bigConst modulus);
68
69  #if ALG_ECC
70
71  /** BnCurveInitialize()
72   // This function is used to initialize the pointers of a bigCurveData structure. The
73   // structure is a set of pointers to bigNum values. The curve-dependent values are
74   // set by a different function. This function is only needed
75   // if the TPM supports ECC.
76  LIB_EXPORT bigCurveData* BnCurveInitialize(bigCurveData* E, TPM_ECC_CURVE curveId);
77
78  /** BnCurveFree()
79   // This function will free the allocated components of the curve and end the
80   // frame in which the curve data exists
81  LIB_EXPORT void BnCurveFree(bigCurveData* E);
82
83  /** BnEccModMult()
84   // This function does a point multiply of the form R = [d]S. A return of FALSE
85   // indicates that the result was the point at infinity. This function is only needed
86   // if the TPM supports ECC.
87  LIB_EXPORT BOOL BnEccModMult(

```

```

88     bigPoint R, pointConst S, bigConst d, const bigCurveData* E);
89
90 /** BnEccModMult2()
91 // This function does a point multiply of the form R = [d]S + [u]Q. A return of
92 // FALSE indicates that the result was the point at infinity. This function is only
93 // needed if the TPM supports ECC.
94 LIB_EXPORT BOOL BnEccModMult2(bigPoint      R,
95                               pointConst    S,
96                               bigConst      d,
97                               pointConst    Q,
98                               bigConst      u,
99                               const bigCurveData* E);
100
101 /** BnEccAdd()
102 // This function does a point add R = S + Q. A return of FALSE
103 // indicates that the result was the point at infinity. This function is only needed
104 // if the TPM supports ECC.
105 LIB_EXPORT BOOL BnEccAdd(
106     bigPoint R, pointConst S, pointConst Q, const bigCurveData* E);
107
108 #endif // ALG_ECC
109
110 #endif //BN_SUPPORT_INTERFACE_H

```

B.3.12. /tpm/cryptolib/TpmBigNum/include/BnUtil_fp.h

```

1  /** Introduction
2  // Utility functions to support TpmBigNum library
3  #ifndef BNUTIL_FP_H
4  #define BNUTIL_FP_H
5
6  #endif // _BNUTIL_FP_H

```

B.3.13. /tpm/cryptolib/TpmBigNum/include/BnValues.h

```

1  /** Introduction
2
3  // This file contains the definitions needed for defining the internal bigNum
4  // structure.
5
6  // A bigNum is a pointer to a structure. The structure has three fields. The
7  // last field is an array (d) of crypt_ushort_t. Each word is in machine format
8  // (big- or little-endian) with the words in ascending significance (i.e. words
9  // in little-endian order). This is the order that seems to be used in every
10 // big number library in the worlds, so...
11 //
12 // The first field in the structure (allocated) is the number of words in 'd'.
13 // This is the upper limit on the size of the number that can be held in the
14 // structure. This differs from libraries like OpenSSL as this is not intended
15 // to deal with numbers of arbitrary size; just numbers that are needed to deal
16 // with the algorithms that are defined in the TPM implementation.
17 //
18 // The second field in the structure (size) is the number of significant words
19 // in 'n'. When this number is zero, the number is zero. The word at used-1 should
20 // never be zero. All words between d[size] and d[allocated-1] should be zero.
21
22 /** Defines
23
24 #ifndef BN_NUMBERS_H
25 #define BN_NUMBERS_H
26 // TODO_RENAME_INC_FOLDER:private refers to the TPM_CoreLib private headers
27 #include <public/TpmAlgorithmDefines.h>
28 #include <public/GpMacros.h> // required for TpmFail_fp.h
29 #include <public/Capabilities.h>

```

```

30 #include <public/TpmTypes.h> // requires capabilities & GpMacros
31
32 // These are the basic big number formats. This is convertible to the library-
33 // specific format without too much difficulty. For the math performed using
34 // these numbers, the value is always positive.
35 #define BN_STRUCT_DEF(struct_type, count) \
36     struct st_##struct_type##_t          \
37     {                                     \
38         crypt_uword_t allocated;          \
39         crypt_uword_t size;               \
40         crypt_uword_t d[count];          \
41     }
42
43 typedef BN_STRUCT_DEF(bnroot, 1) bignum_t;
44
45 #ifndef bigNum
46 typedef bignum_t*      bigNum;
47 typedef const bignum_t* bigConst;
48 #endif //bigNum
49
50 extern const bignum_t BnConstZero;
51
52 // The Functions to access the properties of a big number.
53 // Get number of allocated words
54 #define BnGetAllocated(x) ((x)->allocated)
55
56 // Get number of words used
57 #define BnGetSize(x) ((x)->size)
58
59 // Get a pointer to the data array
60 #define BnGetArray(x) ((crypt_uword_t*)&((x)->d[0]))
61
62 // Get the nth word of a bigNum (zero-based)
63 #define BnGetWord(x, i) (crypt_uword_t)((x)->d[i])
64
65 // Some things that are done often.
66
67 // Test to see if a bignum_t is equal to zero
68 #define BnEqualZero(bn) (BnGetSize(bn) == 0)
69
70 // Test to see if a bignum_t is equal to a word type
71 #define BnEqualWord(bn, word) \
72     ((BnGetSize(bn) == 1) && (BnGetWord(bn, 0) == (crypt_uword_t)word))
73
74 // Determine if a bigNum is even. A zero is even. Although the
75 // indication that a number is zero is that its size is zero,
76 // all words of the number are 0 so this test works on zero.
77 #define BnIsEven(n) ((BnGetWord(n, 0) & 1) == 0)
78
79 // The macros below are used to define bigNum values of the required
80 // size. The values are allocated on the stack so they can be
81 // treated like simple local values.
82
83 // This will call the initialization function for a defined bignum_t.
84 // This sets the allocated and used fields and clears the words of 'n'.
85 #define BN_INIT(name) \
86     (bigNum) BnInit((bigNum) & (name), BYTES_TO_CRYPT_WORDS(sizeof(name.d)))
87
88 #define CRYPT_WORDS(bytes) BYTES_TO_CRYPT_WORDS(bytes)
89 #define MIN_ALLOC(bytes) (CRYPT_WORDS(bytes) < 1 ? 1 : CRYPT_WORDS(bytes))
90 #define BN_CONST(name, bytes, initializer) \
91     typedef const struct name##_type      \
92     {                                     \
93         crypt_uword_t allocated;          \
94         crypt_uword_t size;               \
95         crypt_uword_t d[MIN_ALLOC(bytes)];

```



```

96     } name##_type;
97     name##_type name = {MIN_ALLOC(bytes), CRYPT_WORDS(bytes), {initializer}};
98
99     #define BN_STRUCT_ALLOCATION(bits) (BITS_TO_CRYPT_WORDS(bits) + 1)
100
101     // Create a structure of the correct size.
102     #define BN_STRUCT(struct_type, bits) \
103         BN_STRUCT_DEF(struct_type, BN_STRUCT_ALLOCATION(bits))
104
105     // Define a bigNum type with a specific allocation
106     #define BN_TYPE(name, bits) typedef BN_STRUCT(name, bits) bn_##name##_t
107
108     // This creates a local bigNum variable of a specific size and
109     // initializes it from a TPM2B input parameter.
110     #define BN_INITIALIZED(name, bits, initializer) \
111         BN_STRUCT(name, bits) name##_; \
112         bigNum name = TpmMath_IntFrom2B(BN_INIT(name##_), (const TPM2B*)initializer)
113
114     // Create a local variable that can hold a number with 'bits'
115     #define BN_VAR(name, bits) \
116         BN_STRUCT(name, bits) _##name; \
117         bigNum name = BN_INIT(_##name)
118
119     // Create a type that can hold the largest number defined by the
120     // implementation.
121     #define BN_MAX(name) BN_VAR(name, LARGEST_NUMBER_BITS)
122     #define BN_MAX_INITIALIZED(name, initializer) \
123         BN_INITIALIZED(name, LARGEST_NUMBER_BITS, initializer)
124
125     // A word size value is useful
126     #define BN_WORD(name) BN_VAR(name, RADIX_BITS)
127
128     // This is used to create a word-size bigNum and initialize it with
129     // an input parameter to a function.
130     #define BN_WORD_INITIALIZED(name, initial) \
131         BN_STRUCT(RADIX_BITS) name##_; \
132         bigNum name = \
133             BnInitializeWord((bigNum)&name##_, BN_STRUCT_ALLOCATION(RADIX_BITS), initial)
134
135     // ECC-Specific Values
136
137     // This is the format for a point. It is always in affine format. The Z value is
138     // carried as part of the point, primarily to simplify the interface to the support
139     // library. Rather than have the interface layer have to create space for the
140     // point each time it is used...
141     // The x, y, and z values are pointers to bigNum values and not in-line versions of
142     // the numbers. This is a relic of the days when there was no standard TPM format
143     // for the numbers
144     typedef struct _bn_point_t
145     {
146         bigNum x;
147         bigNum y;
148         bigNum z;
149     } bn_point_t;
150
151     typedef bn_point_t*      bigPoint;
152     typedef const bn_point_t* pointConst;
153
154     typedef struct constant_point_t
155     {
156         bigConst x;
157         bigConst y;
158         bigConst z;
159     } constant_point_t;
160
161     // coords points into x,y,z

```

```

162 // a bigPoint is a pointer to one of these structures, and
163 // therefore a pointer to bn_point_t (a coords).
164 // so bigPoint->coords->x->size is the size of x, and
165 // all 3 components are the same size.
166 #define BN_POINT_BUF(typename, bits) \
167     struct bnpt_st_##typename##_t \
168     { \
169         bn_point_t coords; \
170         BN_STRUCT(typename##_x, MAX_ECC_KEY_BITS) x; \
171         BN_STRUCT(typename##_y, MAX_ECC_KEY_BITS) y; \
172         BN_STRUCT(typename##_z, MAX_ECC_KEY_BITS) z; \
173     }
174
175 typedef BN_POINT_BUF(fullpoint, MAX_ECC_KEY_BITS) bn_fullpoint_t;
176
177 // TPMBN_ECC_CURVE_CONSTANTS
178 // =====
179 // A cryptographic elliptic curve is a mathematical set (Group) of points that
180 // satisfy the group equation and are generated by linear multiples of some
181 // initial "generator" point (Gx,Gy).
182 //
183 // The TPM code supports ECC Curves that satisfy equations of the following
184 // form:
185 //
186 //  $(y^2 = x^3 + a*x + b) \bmod p$ 
187 //
188 // A particular cryptographic curve is fully described by the following
189 // parameters:
190 //
191 // | Name      | Meaning
192 // | :----- | :-----
193 // | p        | curve prime
194 // | a, b      | equation coefficients
195 // | (Gx,Gy)   | X and Y coordinates of the generator point.
196 // | n         | the order (size) of the generated group. n must be prime.
197 // | h         | the cofactor of the group size to the full set of points for a
198 //               | particular equation. |
199 //
200 // The group of constants to describe a particular ECC Curve (such as NIST P256
201 // or P384) are contained in TPMBN_ECC_CURVE_CONSTANTS objects. In the
202 // TpmBigNum library these constants are always stored in TPM's internal BN
203 // (bigNum) format.
204 //
205 // Other math libraries are expected to provide these as compile time constants
206 // in a format they can efficiently consume at runtime.
207 //
208 // Structure for the curve parameters. This is an analog to the
209 // TPMS_ALGORITHM_DETAIL_ECC
210 typedef struct
211 {
212     TPM_ECC_CURVE    curveId; // TPM Algorithm ID for this data
213     bigConst         prime;    // a prime number
214     bigConst         order;    // the order of the curve
215     bigConst         h;        // cofactor
216     bigConst         a;        // linear coefficient
217     bigConst         b;        // constant term
218     constant_point_t base;     // base point
219 } TPMBN_ECC_CURVE_CONSTANTS;

```

```

220 // Access macros for the TPMBN_ECC_CURVE_CONSTANTS structure. The parameter 'C' is a
221 // pointer
222 // to an TPMBN_ECC_CURVE_CONSTANTS structure. In some libraries, the curve structure E
223 // contains
224 // a pointer to an TPMBN_ECC_CURVE_CONSTANTS structure as well as some other bits. For
225 // those
226 // cases, the AccessCurveConstants function is used in the code to first get the
227 // pointer
228 // to the TPMBN_ECC_CURVE_CONSTANTS for access. In some cases, the function does
229 // nothing.
230 // AccessCurveConstants and these functions are all defined as inline so they can be
231 // optimized
232 // away in cases where they are no-ops.
233 TPM_INLINE bigConst BnCurveGetPrime(const TPMBN_ECC_CURVE_CONSTANTS* C)
234 {
235     return C->prime;
236 }
237 TPM_INLINE bigConst BnCurveGetOrder(const TPMBN_ECC_CURVE_CONSTANTS* C)
238 {
239     return C->order;
240 }
241 TPM_INLINE bigConst BnCurveGetCofactor(const TPMBN_ECC_CURVE_CONSTANTS* C)
242 {
243     return C->h;
244 }
245 TPM_INLINE bigConst BnCurveGet_a(const TPMBN_ECC_CURVE_CONSTANTS* C)
246 {
247     return C->a;
248 }
249 TPM_INLINE bigConst BnCurveGet_b(const TPMBN_ECC_CURVE_CONSTANTS* C)
250 {
251     return C->b;
252 }
253 TPM_INLINE pointConst BnCurveGetG(const TPMBN_ECC_CURVE_CONSTANTS* C)
254 {
255     return (pointConst) & (C->base);
256 }
257 TPM_INLINE bigConst BnCurveGetGx(const TPMBN_ECC_CURVE_CONSTANTS* C)
258 {
259     return C->base.x;
260 }
261 TPM_INLINE bigConst BnCurveGetGy(const TPMBN_ECC_CURVE_CONSTANTS* C)
262 {
263     return C->base.y;
264 }
265 TPM_INLINE TPM_ECC_CURVE BnCurveGetCurveId(const TPMBN_ECC_CURVE_CONSTANTS* C)
266 {
267     return C->curveId;
268 }
269
270 // Convert bytes in initializers
271 // This is used for CryptEccData.c.
272 #define BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d) \
273     (((UINT32)(a) << 24) + ((UINT32)(b) << 16) + ((UINT32)(c) << 8) + ((UINT32)(d)))
274
275 #define BIG_ENDIAN_BYTES_TO_UINT64(a, b, c, d, e, f, g, h) \
276     (((UINT64)(a) << 56) + ((UINT64)(b) << 48) + ((UINT64)(c) << 40) \
277     + ((UINT64)(d) << 32) + ((UINT64)(e) << 24) + ((UINT64)(f) << 16) \
278     + ((UINT64)(g) << 8) + ((UINT64)(h)))
279
280 // These macros are used for data initialization of big number ECC constants
281 // These two macros combine a macro for data definition with a macro for
282 // structure initialization. The 'a' parameter is a macro that gives numbers to
283 // each of the bytes of the initializer and defines where each of the numberd
284 // bytes will show up in the final structure. The 'b' value is a structure that
285 // contains the requisite number of bytes in big endian order. S, the MJOIN

```

```

280 // and JOIND macros will combine a macro defining a data layout with a macro defining
281 // the data to be places. Generally, these macros will only need expansion when
282 // CryptEccData.c gets compiled.
283 #define JOINED(a, b) a b
284 #define MJOIN(a, b) a b
285
286 #if RADIX_BYTES == 64
287 # define B8_TO_BN(a, b, c, d, e, f, g, h) \
288     (((((((((UINT64)a) << 8) | (UINT64)b) << 8) | (UINT64)c) << 8) | \
289         | (UINT64)d) \
290         << 8) \
291         | (UINT64)e) \
292         << 8) \
293         | (UINT64)f) \
294         << 8) \
295         | (UINT64)g) \
296         << 8) \
297         | (UINT64)h)
298 # define B1_TO_BN(a) B8_TO_BN(0, 0, 0, 0, 0, 0, 0, a)
299 # define B2_TO_BN(a, b) B8_TO_BN(0, 0, 0, 0, 0, 0, a, b)
300 # define B3_TO_BN(a, b, c) B8_TO_BN(0, 0, 0, 0, 0, a, b, c)
301 # define B4_TO_BN(a, b, c, d) B8_TO_BN(0, 0, 0, 0, a, b, c, d)
302 # define B5_TO_BN(a, b, c, d, e) B8_TO_BN(0, 0, 0, a, b, c, d, e)
303 # define B6_TO_BN(a, b, c, d, e, f) B8_TO_BN(0, 0, a, b, c, d, e, f)
304 # define B7_TO_BN(a, b, c, d, e, f, g) B8_TO_BN(0, a, b, c, d, e, f, g)
305 #else
306 # define B1_TO_BN(a) B4_TO_BN(0, 0, 0, a)
307 # define B2_TO_BN(a, b) B4_TO_BN(0, 0, a, b)
308 # define B3_TO_BN(a, b, c) B4_TO_BN(0, a, b, c)
309 # define B4_TO_BN(a, b, c, d) \
310     ((((((UINT32)a << 8) | (UINT32)b) << 8) | (UINT32)c) << 8) | (UINT32)d)
311 # define B5_TO_BN(a, b, c, d, e) B4_TO_BN(b, c, d, e), B1_TO_BN(a)
312 # define B6_TO_BN(a, b, c, d, e, f) B4_TO_BN(c, d, e, f), B2_TO_BN(a, b)
313 # define B7_TO_BN(a, b, c, d, e, f, g) B4_TO_BN(d, e, f, g), B3_TO_BN(a, b, c)
314 # define B8_TO_BN(a, b, c, d, e, f, g, h) B4_TO_BN(e, f, g, h), B4_TO_BN(a, b, c, d)
315
316 #endif
317
318 #endif // _BN_NUMBERS_H

```

B.3.14. /tpm/cryptolib/TpmBigNum/include/TpmBigNum/TpmToTpmBigNumMath.h

```

1 /** Introduction
2 // This file contains OpenSSL specific functions called by TpmBigNum library to
3 // provide
4 // the TpmBigNum + OpenSSL math support.
5
6 #ifndef TPM_TO_TPMBIGNUM_MATH_H
7 #define TPM_TO_TPMBIGNUM_MATH_H
8
9 #ifdef MATH_LIB_DEFINED
10 # error only one primary math library allowed
11 #endif
12 #define MATH_LIB_DEFINED
13
14 // indicate the TPMBIGNUM library is active
15 #define MATH_LIB_TPMBIGNUM
16
17 // TODO_RENAME_INC_FOLDER: private refers to the TPM_CoreLib private headers
18 #include <public/GpMacros.h> // required for TpmFail_fp.h
19 #include <public/Capabilities.h>
20 #include <public/TpmTypes.h> // requires capabilities & GpMacros
21 #include "BnValues.h"
22
23 #ifndef LIB_INCLUDE

```

```

23  # error include ordering error, LIB_INCLUDE not defined
24  #endif
25  #ifndef BN_MATH_LIB
26  # error BN_MATH_LIB not defined, required to provide BN library functions.
27  #endif
28
29  #if defined(CRYPT_CURVE_INITIALIZED) || defined(CRYPT_CURVE_FREE)
30  #error include ordering error, expected CRYPT_CURVE_INITIALIZED & CRYPT_CURVE_FREE to
    be undefined.
31  #endif
32
33  // Add support library dependent definitions.
34  // For TpmBigNum, we expect bigCurveData to be a defined type.
35  #include LIB_INCLUDE(BnTo, BN_MATH_LIB, Math)
36
37  #include "BnConvert_fp.h"
38  #include "BnMath_fp.h"
39  #include "BnMemory_fp.h"
40  #include "BnSupport_Interface.h"
41
42  // Define macros and types necessary for the math library abstraction layer
43  // Create a data object backing a Crypt_Int big enough for the given number of
44  // data bits
45  #define CRYPT_INT_BUF(buftypepname, bits) BN_STRUCT(buftypepname, bits)
46
47  // Create a data object backing a Crypt_Point big enough for the given number of
48  // data bits, per coordinate
49  #define CRYPT_POINT_BUF(buftypepname, bits) BN_POINT_BUF(buftypepname, bits)
50
51  // Create an instance of a data object underlying Crypt_EccCurve on the stack
52  // sufficient for given bit size. In our case, all are the same size.
53  #define CRYPT_CURVE_BUF(buftypepname, max_size_in_bits) bigCurveData
54
55  // now include the math library functional interface and instantiate the
56  // Crypt_Int & related types
57  // TODO_RENAME_INC_FOLDER: This should have a Tpm_Cryptolib_Common component prefix.
58  #include <MathLibraryInterface.h>
59
60  #endif // _TPM_TO_TPMBIGNUM_MATH_H_
61

```

B.4 OpenSSL-Specific Files

B.4.1. Introduction

The following files are specific to a port that uses the OpenSSL library for cryptographic functions.

B.4.1.1. /tpm/cryptolib/Ossl/BnToOsslMath.c

```

1  /** Introduction
2  // The functions in this file provide the low-level interface between the TPM code
3  // and the big number and elliptic curve math routines in OpenSSL.
4  //
5  // Most math on big numbers require a context. The context contains the memory in
6  // which OpenSSL creates and manages the big number values. When a OpenSSL math
7  // function will be called that modifies a BIGNUM value, that value must be created in
8  // an OpenSSL context. The first line of code in such a function must be:
9  // OSSL_ENTER(); and the last operation before returning must be OSSL_LEAVE().
10 // OpenSSL variables can then be created with BnNewVariable(). Constant values to be
11 // used by OpenSSL are created from the bigNum values passed to the functions in this
12 // file. Space for the BIGNUM control block is allocated in the stack of the
13 // function and then it is initialized by calling BigInitialized(). That function
14 // sets up the values in the BIGNUM structure and sets the data pointer to point to

```

```

15 // the data in the bignum_t. This is only used when the value is known to be a
16 // constant in the called function.
17 //
18 // Because the allocations of constants is on the local stack and the
19 // OSSL_ENTER()/OSSL_LEAVE() pair flushes everything created in OpenSSL memory, there
20 // should be no chance of a memory leak.
21
22 /** Includes and Defines
23 #include "Tpm.h"
24 #include "BnOssl.h"
25
26 #ifndef MATH_LIB_OSSL
27 # include <Ossl/BnToOsslMath_fp.h>
28
29 /** Functions
30
31 /*** OsslToTpmBn()
32 // This function converts an OpenSSL BIGNUM to a TPM bigNum. In this implementation
33 // it is assumed that OpenSSL uses a different control structure but the same data
34 // layout -- an array of native-endian words in little-endian order.
35 // Return Type: BOOL
36 //      TRUE(1)          success
37 //      FALSE(0)         failure because value will not fit or OpenSSL variable doesn't
38 //                        exist
39 BOOL OsslToTpmBn(bigNum bn, BIGNUM* osslBn)
40 {
41     GOTO_ERROR_UNLESS(osslBn != NULL);
42     // If the bn is NULL, it means that an output value pointer was NULL meaning that
43     // the results is simply to be discarded.
44     if(bn != NULL)
45     {
46         int i;
47         //
48         GOTO_ERROR_UNLESS((unsigned)osslBn->top <= BnGetAllocated(bn));
49         for(i = 0; i < osslBn->top; i++)
50             bn->d[i] = osslBn->d[i];
51         BnSetTop(bn, osslBn->top);
52     }
53     return TRUE;
54 Error:
55     return FALSE;
56 }
57
58 /*** BigInitialized()
59 // This function initializes an OSSL BIGNUM from a TPM bigConst. Do not use this for
60 // values that are passed to OpenSSL when they are not declared as const in the
61 // function prototype. Instead, use BnNewVariable().
62 BIGNUM* BigInitialized(BIGNUM* toInit, bigConst initializer)
63 {
64     if(initializer == NULL)
65         FAIL(FATAL_ERROR_PARAMETER);
66     if(toInit == NULL || initializer == NULL)
67         return NULL;
68     toInit->d      = (BN_ULONG*)&initializer->d[0];
69     toInit->dmax    = (int)initializer->allocated;
70     toInit->top     = (int)initializer->size;
71     toInit->neg     = 0;
72     toInit->flags   = 0;
73     return toInit;
74 }
75
76 # ifndef OSSL_DEBUG
77 #   define BIGNUM_PRINT(label, bn, eol)
78 #   define DEBUG_PRINT(x)
79 # else
80 #   define DEBUG_PRINT(x)          printf("%s", x)

```



```

81 #   define BIGNUM_PRINT(label, bn, eol) BIGNUM_print((label), (bn), (eol))
82
83 /*** BIGNUM_print()
84 static void BIGNUM_print(const char* label, const BIGNUM* a, BOOL eol)
85 {
86     BN_ULONG* d;
87     int        i;
88     int        notZero = FALSE;
89
90     if(label != NULL)
91         printf("%s", label);
92     if(a == NULL)
93     {
94         printf("NULL");
95         goto done;
96     }
97     if(a->neg)
98         printf("-");
99     for(i = a->top, d = &a->d[i - 1]; i > 0; i--)
100     {
101         int        j;
102         BN_ULONG l = *d--;
103         for(j = BN_BITS2 - 8; j >= 0; j -= 8)
104         {
105             BYTE b = (BYTE)((l >> j) & 0xFF);
106             notZero = notZero || (b != 0);
107             if(notZero)
108                 printf("%02x", b);
109         }
110         if(!notZero)
111             printf("0");
112     }
113 done:
114     if(eol)
115         printf("\n");
116     return;
117 }
118 #   endif
119
120 /*** BnNewVariable()
121 // This function allocates a new variable in the provided context. If the context
122 // does not exist or the allocation fails, it is a catastrophic failure.
123 static BIGNUM* BnNewVariable(BN_CTX* CTX)
124 {
125     BIGNUM* new;
126     //
127     // This check is intended to protect against calling this function without
128     // having initialized the CTX.
129     if((CTX == NULL) || ((new = BN_CTX_get(CTX)) == NULL))
130         FAIL(FATAL_ERROR_ALLOCATION);
131     return new;
132 }
133
134 #   if LIBRARY_COMPATIBILITY_CHECK
135
136 /*** MathLibraryCompatibilityCheck()
137 BOOL BnMathLibraryCompatibilityCheck(void)
138 {
139     OSSL_ENTER();
140     BIGNUM*      osslTemp = BnNewVariable(CTX);
141     crypt_ushort t i;
142     BYTE test[] = {0x1F, 0x1E, 0x1D, 0x1C, 0x1B, 0x1A, 0x19, 0x18, 0x17, 0x16, 0x15,
143                   0x14, 0x13, 0x12, 0x11, 0x10, 0x0F, 0x0E, 0x0D, 0x0C, 0x0B, 0x0A,
144                   0x09, 0x08, 0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00};
145     BN_VAR(tpmTemp, sizeof(test) * 8); // allocate some space for a test value
146     //

```

```

147     // Convert the test data to a bigNum
148     BnFromBytes(tpmTemp, test, sizeof(test));
149     // Convert the test data to an OpenSSL BIGNUM
150     BN_bin2bn(test, sizeof(test), osslTemp);
151     // Make sure the values are consistent
152     GOTO_ERROR_UNLESS(osslTemp->top == (int)tpmTemp->size);
153     for(i = 0; i < tpmTemp->size; i++)
154         GOTO_ERROR_UNLESS(osslTemp->d[i] == tpmTemp->d[i]);
155     OSSL_LEAVE();
156     return 1;
157 Error:
158     return 0;
159 }
160 # endif
161
162 /*** BnModMult()
163 // This function does a modular multiply. It first does a multiply and then a divide
164 // and returns the remainder of the divide.
165 // Return Type: BOOL
166 //     TRUE(1)          success
167 //     FALSE(0)         failure in operation
168 LIB_EXPORT BOOL BnModMult(bigNum result, bigConst op1, bigConst op2, bigConst modulus)
169 {
170     OSSL_ENTER();
171     BOOL OK = TRUE;
172     BIGNUM* bnResult = BN_NEW();
173     BIGNUM* bnTemp = BN_NEW();
174     BIG_INITIALIZED(bnOp1, op1);
175     BIG_INITIALIZED(bnOp2, op2);
176     BIG_INITIALIZED(bnMod, modulus);
177     //
178     GOTO_ERROR_UNLESS(BN_mul(bnTemp, bnOp1, bnOp2, CTX));
179     GOTO_ERROR_UNLESS(BN_div(NULL, bnResult, bnTemp, bnMod, CTX));
180     GOTO_ERROR_UNLESS(OsslToTpmBn(result, bnResult));
181     goto Exit;
182 Error:
183     OK = FALSE;
184 Exit:
185     OSSL_LEAVE();
186     return OK;
187 }
188
189 /*** BnMult()
190 // Multiplies two numbers
191 // Return Type: BOOL
192 //     TRUE(1)          success
193 //     FALSE(0)         failure in operation
194 LIB_EXPORT BOOL BnMult(bigNum result, bigConst multiplicand, bigConst multiplier)
195 {
196     OSSL_ENTER();
197     BIGNUM* bnTemp = BN_NEW();
198     BOOL OK = TRUE;
199     BIG_INITIALIZED(bnA, multiplicand);
200     BIG_INITIALIZED(bnB, multiplier);
201     //
202     GOTO_ERROR_UNLESS(BN_mul(bnTemp, bnA, bnB, CTX));
203     GOTO_ERROR_UNLESS(OsslToTpmBn(result, bnTemp));
204     goto Exit;
205 Error:
206     OK = FALSE;
207 Exit:
208     OSSL_LEAVE();
209     return OK;
210 }
211
212 /*** BnDiv()

```

```

213 // This function divides two bigNum values. The function returns FALSE if
214 // there is an error in the operation.
215 // Return Type: BOOL
216 //     TRUE(1)          success
217 //     FALSE(0)         failure in operation
218 LIB_EXPORT BOOL BnDiv(
219     bigNum quotient, bigNum remainder, bigConst dividend, bigConst divisor)
220 {
221     OSSL_ENTER();
222     BIGNUM* bnQ = BN_NEW();
223     BIGNUM* bnR = BN_NEW();
224     BOOL OK = TRUE;
225     BIG_INITIALIZED(bnDend, dividend);
226     BIG_INITIALIZED(bnSor, divisor);
227     //
228     if(BnEqualZero(divisor))
229         FAIL(FATAL_ERROR_DIVIDE_ZERO);
230     GOTO_ERROR_UNLESS(BN_div(bnQ, bnR, bnDend, bnSor, CTX));
231     GOTO_ERROR_UNLESS(OsslToTpmBn(quotient, bnQ));
232     GOTO_ERROR_UNLESS(OsslToTpmBn(remainder, bnR));
233     DEBUG_PRINT("In BnDiv:\n");
234     BIGNUM_PRINT("    bnDividend: ", bnDend, TRUE);
235     BIGNUM_PRINT("    bnDivisor: ", bnSor, TRUE);
236     BIGNUM_PRINT("    bnQuotient: ", bnQ, TRUE);
237     BIGNUM_PRINT("    bnRemainder: ", bnR, TRUE);
238     goto Exit;
239 Error:
240     OK = FALSE;
241 Exit:
242     OSSL_LEAVE();
243     return OK;
244 }
245
246 # if ALG_RSA
247 /*** BnGcd()
248 // Get the greatest common divisor of two numbers
249 // Return Type: BOOL
250 //     TRUE(1)          success
251 //     FALSE(0)         failure in operation
252 LIB_EXPORT BOOL BnGcd(bigNum gcd, // OUT: the common divisor
253     bigConst number1, // IN:
254     bigConst number2 // IN:
255 )
256 {
257     OSSL_ENTER();
258     BIGNUM* bnGcd = BN_NEW();
259     BOOL OK = TRUE;
260     BIG_INITIALIZED(bn1, number1);
261     BIG_INITIALIZED(bn2, number2);
262     //
263     GOTO_ERROR_UNLESS(BN_gcd(bnGcd, bn1, bn2, CTX));
264     GOTO_ERROR_UNLESS(OsslToTpmBn(gcd, bnGcd));
265     goto Exit;
266 Error:
267     OK = FALSE;
268 Exit:
269     OSSL_LEAVE();
270     return OK;
271 }
272
273 /***BnModExp()
274 // Do modular exponentiation using bigNum values. The conversion from a bignum_t to
275 // a bigNum is trivial as they are based on the same structure
276 // Return Type: BOOL
277 //     TRUE(1)          success
278 //     FALSE(0)         failure in operation

```

```

279 LIB_EXPORT BOOL BnModExp(bigNum result, // OUT: the result
280                          bigConst number, // IN: number to exponentiate
281                          bigConst exponent, // IN:
282                          bigConst modulus // IN:
283 )
284 {
285     OSSL_ENTER();
286     BIGNUM* bnResult = BN_NEW();
287     BOOL OK = TRUE;
288     BIG_INITIALIZED(bnN, number);
289     BIG_INITIALIZED(bnE, exponent);
290     BIG_INITIALIZED(bnM, modulus);
291     //
292     GOTO_ERROR_UNLESS(BN_mod_exp(bnResult, bnN, bnE, bnM, CTX));
293     GOTO_ERROR_UNLESS(OsslToTpmBn(result, bnResult));
294     goto Exit;
295 Error:
296     OK = FALSE;
297 Exit:
298     OSSL_LEAVE();
299     return OK;
300 }
301 # endif // ALG_RSA
302
303 /*** BnModInverse()
304 // Modular multiplicative inverse
305 // Return Type: BOOL
306 // TRUE(1) success
307 // FALSE(0) failure in operation
308 LIB_EXPORT BOOL BnModInverse(bigNum result, bigConst number, bigConst modulus)
309 {
310     OSSL_ENTER();
311     BIGNUM* bnResult = BN_NEW();
312     BOOL OK = TRUE;
313     BIG_INITIALIZED(bnN, number);
314     BIG_INITIALIZED(bnM, modulus);
315     //
316     GOTO_ERROR_UNLESS(BN_mod_inverse(bnResult, bnN, bnM, CTX) != NULL);
317     GOTO_ERROR_UNLESS(OsslToTpmBn(result, bnResult));
318     goto Exit;
319 Error:
320     OK = FALSE;
321 Exit:
322     OSSL_LEAVE();
323     return OK;
324 }
325
326 # if ALG_ECC
327
328 /*** PointFromOssl()
329 // Function to copy the point result from an OSSL function to a bigNum
330 // Return Type: BOOL
331 // TRUE(1) success
332 // FALSE(0) failure in operation
333 static BOOL PointFromOssl(bigPoint pOut, // OUT: resulting point
334                          EC_POINT* pIn, // IN: the point to return
335                          const bigCurveData* E // IN: the curve
336 )
337 {
338     BIGNUM* x = NULL;
339     BIGNUM* y = NULL;
340     BOOL OK;
341     BN_CTX_start(E->CTX);
342     //
343     x = BN_CTX_get(E->CTX);
344     y = BN_CTX_get(E->CTX);

```

```

345
346     if(y == NULL)
347         FAIL(FATAL_ERROR_ALLOCATION);
348     // If this returns false, then the point is at infinity
349     OK = EC_POINT_get_affine_coordinates_GFp(E->G, pIn, x, y, E->CTX);
350     if(OK)
351     {
352         OsslToTpmBn(pOut->x, x);
353         OsslToTpmBn(pOut->y, y);
354         BnSetWord(pOut->z, 1);
355     }
356     else
357         BnSetWord(pOut->z, 0);
358     BN_CTX_end(E->CTX);
359     return OK;
360 }
361
362 /*** EcPointInitialized()
363 // Allocate and initialize a point.
364 static EC_POINT* EcPointInitialized(pointConst initializer, const bigCurveData* E)
365 {
366     EC_POINT* P = NULL;
367
368     if(initializer != NULL)
369     {
370         BIG_INITIALIZED(bnX, initializer->x);
371         BIG_INITIALIZED(bnY, initializer->y);
372         if(E == NULL)
373             FAIL(FATAL_ERROR_ALLOCATION);
374         P = EC_POINT_new(E->G);
375         if(!EC_POINT_set_affine_coordinates_GFp(E->G, P, bnX, bnY, E->CTX))
376             P = NULL;
377     }
378     return P;
379 }
380
381 /*** BnCurveInitialize()
382 // This function initializes the OpenSSL curve information structure. This
383 // structure points to the TPM-defined values for the curve, to the context for the
384 // number values in the frame, and to the OpenSSL-defined group values.
385 // Return Type: bigCurveData*
386 // NULL the TPM_ECC_CURVE is not valid or there was a problem in
387 // in initializing the curve data
388 // non-NULL points to 'E'
389 LIB_EXPORT bigCurveData* BnCurveInitialize(
390     bigCurveData* E, // IN: curve structure to initialize
391     TPM_ECC_CURVE curveId // IN: curve identifier
392 )
393 {
394     const TPMBN_ECC_CURVE_CONSTANTS* C = BnGetCurveData(curveId);
395     if(C == NULL)
396         E = NULL;
397     if(E != NULL)
398     {
399         // This creates the OpenSSL memory context that stays in effect as long as the
400         // curve (E) is defined.
401         OSSL_ENTER(); // if the allocation fails, the TPM fails
402         EC_POINT* P = NULL;
403         BIG_INITIALIZED(bnP, C->prime);
404         BIG_INITIALIZED(bnA, C->a);
405         BIG_INITIALIZED(bnB, C->b);
406         BIG_INITIALIZED(bnX, C->base.x);
407         BIG_INITIALIZED(bnY, C->base.y);
408         BIG_INITIALIZED(bnN, C->order);
409         BIG_INITIALIZED(bnH, C->h);
410         //

```

```

411     E->C = C;
412     E->CTX = CTX;
413
414     // initialize EC group, associate a generator point and initialize the point
415     // from the parameter data
416     // Create a group structure
417     E->G = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, CTX);
418     GOTO_ERROR_UNLESS(E->G != NULL);
419
420     // Allocate a point in the group that will be used in setting the
421     // generator. This is not needed after the generator is set.
422     P = EC_POINT_new(E->G);
423     GOTO_ERROR_UNLESS(P != NULL);
424
425     // Need to use this in case Montgomery method is being used
426     GOTO_ERROR_UNLESS(
427         EC_POINT_set_affine_coordinates_GFp(E->G, P, bnX, bnY, CTX));
428     // Now set the generator
429     GOTO_ERROR_UNLESS(EC_GROUP_set_generator(E->G, P, bnN, bnH));
430
431     EC_POINT_free(P);
432     goto Exit;
433 Error:
434     EC_POINT_free(P);
435     BnCurveFree(E);
436     E = NULL;
437 }
438 Exit:
439     return E;
440 }
441
442 /*** BnCurveFree()
443 // This function will free the allocated components of the curve and end the
444 // frame in which the curve data exists
445 LIB_EXPORT void BnCurveFree(bigCurveData* E)
446 {
447     if(E)
448     {
449         EC_GROUP_free(E->G);
450         OsslContextLeave(E->CTX);
451     }
452 }
453
454 /*** BnEccModMult()
455 // This function does a point multiply of the form R = [d]S
456 // Return Type: BOOL
457 //     TRUE(1)      success
458 //     FALSE(0)     failure in operation; treat as result being point at infinity
459 LIB_EXPORT BOOL BnEccModMult(bigPoint R, // OUT: computed point
460                             pointConst S, // IN: point to multiply by 'd' (optional)
461                             bigConst d, // IN: scalar for [d]S
462                             const bigCurveData* E)
463 {
464     EC_POINT* pR = EC_POINT_new(E->G);
465     EC_POINT* pS = EcPointInitialized(S, E);
466     BIG_INITIALIZED(bnD, d);
467
468     if(S == NULL)
469         EC_POINT_mul(E->G, pR, bnD, NULL, NULL, E->CTX);
470     else
471         EC_POINT_mul(E->G, pR, NULL, pS, bnD, E->CTX);
472     PointFromOssl(R, pR, E);
473     EC_POINT_free(pR);
474     EC_POINT_free(pS);
475     return !BnEqualZero(R->z);
476 }

```



```

477
478 /*** BnEccModMult2()
479 // This function does a point multiply of the form R = [d]G + [u]Q
480 // Return Type: BOOL
481 //     TRUE(1)          success
482 //     FALSE(0)         failure in operation; treat as result being point at infinity
483 LIB_EXPORT BOOL BnEccModMult2(bigPoint      R, // OUT: computed point
484                               pointConst    S, // IN: optional point
485                               bigConst      d, // IN: scalar for [d]S or [d]G
486                               pointConst    Q, // IN: second point
487                               bigConst      u, // IN: second scalar
488                               const bigCurveData* E // IN: curve
489 )
490 {
491     EC_POINT* pR = EC_POINT_new(E->G);
492     EC_POINT* pS = EcPointInitialized(S, E);
493     BIG_INITIALIZED(bnD, d);
494     EC_POINT* pQ = EcPointInitialized(Q, E);
495     BIG_INITIALIZED(bnU, u);
496
497     if(S == NULL || S == (pointConst) & (AccessCurveConstants(E)->base))
498         EC_POINT_mul(E->G, pR, bnD, pQ, bnU, E->CTX);
499     else
500     {
501         const EC_POINT* points[2];
502         const BIGNUM*   scalars[2];
503         points[0] = pS;
504         points[1] = pQ;
505         scalars[0] = bnD;
506         scalars[1] = bnU;
507         EC_POINTs_mul(E->G, pR, NULL, 2, points, scalars, E->CTX);
508     }
509     PointFromOssl(R, pR, E);
510     EC_POINT_free(pR);
511     EC_POINT_free(pS);
512     EC_POINT_free(pQ);
513     return !BnEqualZero(R->z);
514 }
515
516 /*** BnEccAdd()
517 // This function does addition of two points.
518 // Return Type: BOOL
519 //     TRUE(1)          success
520 //     FALSE(0)         failure in operation; treat as result being point at infinity
521 LIB_EXPORT BOOL BnEccAdd(bigPoint      R, // OUT: computed point
522                          pointConst    S, // IN: first point to add
523                          pointConst    Q, // IN: second point
524                          const bigCurveData* E // IN: curve
525 )
526 {
527     EC_POINT* pR = EC_POINT_new(E->G);
528     EC_POINT* pS = EcPointInitialized(S, E);
529     EC_POINT* pQ = EcPointInitialized(Q, E);
530     //
531     EC_POINT_add(E->G, pR, pS, pQ, E->CTX);
532
533     PointFromOssl(R, pR, E);
534     EC_POINT_free(pR);
535     EC_POINT_free(pS);
536     EC_POINT_free(pQ);
537     return !BnEqualZero(R->z);
538 }
539
540 # endif // ALG_ECC
541
542 #endif // MATHLIB_OSSL

```

B.4.1.2. /tpm/cryptolib/Ossl/TpmToOsslSupport.c

```

1  /** Introduction
2  //
3  // The functions in this file are used for initialization of the interface to the
4  // OpenSSL library.
5
6  /** Defines and Includes
7
8  #include "BnOssl.h"
9
10 #if defined(HASH_LIB_OSSL) || defined(MATH_LIB_OSSL) || defined(SYM_LIB_OSSL)
11 // Used to pass the pointers to the correct sub-keys
12 typedef const BYTE* desKeyPointers[3];
13
14 /*** BnSupportLibInit()
15 // This does any initialization required by the support library.
16 LIB_EXPORT int BnSupportLibInit(void)
17 {
18     return TRUE;
19 }
20
21 /*** OsslContextEnter()
22 // This function is used to initialize an OpenSSL context at the start of a function
23 // that will call to an OpenSSL math function.
24 BN_CTX* OsslContextEnter(void)
25 {
26     BN_CTX* CTX = BN_CTX_new();
27     //
28     return OsslPushContext(CTX);
29 }
30
31 /*** OsslContextLeave()
32 // This is the companion function to OsslContextEnter().
33 void OsslContextLeave(BN_CTX* CTX)
34 {
35     OsslPopContext(CTX);
36     BN_CTX_free(CTX);
37 }
38
39 /*** OsslPushContext()
40 // This function is used to create a frame in a context. All values allocated within
41 // this context after the frame is started will be automatically freed when the
42 // context (OsslPopContext())
43 BN_CTX* OsslPushContext(BN_CTX* CTX)
44 {
45     if(CTX == NULL)
46         FAIL(FATAL_ERROR_ALLOCATION);
47     BN_CTX_start(CTX);
48     return CTX;
49 }
50
51 /*** OsslPopContext()
52 // This is the companion function to OsslPushContext().
53 void OsslPopContext(BN_CTX* CTX)
54 {
55     // BN_CTX_end can't be called with NULL. It will blow up.
56     if(CTX != NULL)
57         BN_CTX_end(CTX);
58 }
59
60 #endif // HASH_LIB_OSSL || MATH_LIB_OSSL || SYM_LIB_OSSL

```

B.4.1.3. /tpm/cryptolib/Ossl/include/BnOssl.h

```

1  /** Introduction
2  // This file contains the headers necessary to build the Open SSL support for
3  // the TpmBigNum library.
4  #ifndef _BNOSSL_H_
5  #define _BNOSSL_H_
6  // TODO_RENAME_INC_FOLDER: public refers to the TPM_CoreLib public headers
7  #include <public/tpm_public.h>
8  #include <public/prototypes/TpmFail_fp.h>
9  #include <Ossl/BnToOsslMath.h>
10 // TODO_RENAME_INC_FOLDER: these refer to TpmBigNum protected headers
11 #include <BnSupport_Interface.h>
12 #include <BnUtil_fp.h>
13 #include <BnMemory_fp.h>
14 #include <BnMath_fp.h>
15 #include <BnConvert_fp.h>
16 #endif

```

B.4.1.4. /tpm/cryptolib/Ossl/include/Ossl/BnToOsslMath.h

```

1  /** Introduction
2  // This file contains OpenSSL specific functions called by TpmBigNum library to
3  // provide
4  // the TpmBigNum + OpenSSL math support.
5
6  #ifndef _BN_TO_OSSL_MATH_H_
7  #define _BN_TO_OSSL_MATH_H_
8
9  #define MATH_LIB_OSSL
10
11 // Require TPM Big Num types
12 #if !defined(MATH_LIB_TPMBIGNUM) && !defined(_BNOSSL_H_)
13 # error this OpenSSL Interface expects to be used from TpmBigNum
14 #endif
15
16 #include <BnValues.h>
17 #include <openssl/evp.h>
18 #include <openssl/ec.h>
19 #include <openssl/bn.h>
20
21 #if OPENSSL_VERSION_NUMBER >= 0x30100000L
22 // Check the bignum_st definition against the one below and either update the
23 // version check or provide the new definition for this version.
24 # error Untested OpenSSL version
25 #elif OPENSSL_VERSION_NUMBER >= 0x10100000L
26 // from crypto/bn/bn_lcl.h (OpenSSL 1.x) or crypto/bn/bn_local.h (OpenSSL 3.0)
27 struct bignum_st
28 {
29     BN_ULONG* d; /* Pointer to an array of 'BN_BITS2' bit
30                  * chunks. */
31     int top; /* Index of last used d +1. */
32     /* The next are internal book keeping for bn_expand. */
33     int dmax; /* Size of the d array. */
34     int neg; /* one if the number is negative */
35     int flags;
36 };
37 #else
38 # define EC_POINT_get_affine_coordinates EC_POINT_get_affine_coordinates_GFp
39 # define EC_POINT_set_affine_coordinates EC_POINT_set_affine_coordinates_GFp
40 #endif // OPENSSL_VERSION_NUMBER
41
42 /** Macros and Defines
43
44 // Make sure that the library is using the correct size for a crypt word

```

```

44  #if defined THIRTY_TWO_BIT && (RADIX_BITS != 32) \
45      || ((defined SIXTY_FOUR_BIT_LONG || defined SIXTY_FOUR_BIT) \
46          && (RADIX_BITS != 64))
47  # error Ossl library is using different radix
48  #endif
49
50  // Allocate a local BIGNUM value. For the allocation, a bigNum structure is created
51  // as is a local BIGNUM. The bigNum is initialized and then the BIGNUM is
52  // set to reference the local value.
53  #define BIG_VAR(name, bits) \
54      BN_VAR(name##Bn, (bits)); \
55      BIGNUM _##name; \
56      BIGNUM* name = BigInitialized( \
57          &_##name, BnInit(name##Bn, BYTES_TO_CRYPT_WORDS(sizeof(_##name##Bn.d)))
58
59  // Allocate a BIGNUM and initialize with the values in a bigNum initializer
60  #define BIG_INITIALIZED(name, initializer) \
61      BIGNUM _##name; \
62      BIGNUM* name = BigInitialized(&_##name, initializer)
63
64  typedef struct
65  {
66      const TPMBN_ECC_CURVE_CONSTANTS* C; // the TPM curve values
67      EC_GROUP* G; // group parameters
68      BN_CTX* CTX; // the context for the math (this might not be
69                  // the context in which the curve was created);
70  } OSSL_CURVE_DATA;
71
72  // Define the curve data type expected by the TpmBigNum library:
73  typedef OSSL_CURVE_DATA bigCurveData;
74
75  TPM_INLINE const TPMBN_ECC_CURVE_CONSTANTS* AccessCurveConstants(
76      const bigCurveData* E)
77  {
78      return E->C;
79  }
80
81  #include "TpmToOsslSupport_fp.h"
82
83  // Start and end a context within which the OpenSSL memory management works
84  #define OSSL_ENTER() BN_CTX* CTX = OsslContextEnter()
85  #define OSSL_LEAVE() OsslContextLeave(CTX)
86
87  // Start and end a local stack frame within the context of the curve frame
88  #define ECC_ENTER() BN_CTX* CTX = OsslPushContext(E->CTX)
89  #define ECC_LEAVE() OsslPopContext(CTX)
90
91  #define BN_NEW() BnNewVariable(CTX)
92
93  // This definition would change if there were something to report
94  #define MathLibSimulationEnd()
95
96  #endif // _BN_TO_OSSL_MATH_H_

```

B.4.1.5. /tpm/cryptolib/Ossl/include/Ossl/BnToOsslMath_fp.h

```

1  /* (Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Oct 24, 2019 Time: 11:37:07AM
4   */
5
6  #ifndef _BN_TO_OSSL_MATH_FP_H_
7  #define _BN_TO_OSSL_MATH_FP_H_
8
9  #ifdef MATH_LIB_OSSL

```

```

10
11  /*** OsslToTpmBn()
12  // This function converts an OpenSSL BIGNUM to a TPM bigNum. In this implementation
13  // it is assumed that OpenSSL uses a different control structure but the same data
14  // layout -- an array of native-endian words in little-endian order.
15  // Return Type: BOOL
16  //      TRUE(1)          success
17  //      FALSE(0)        failure because value will not fit or OpenSSL variable doesn't
18  //                      exist
19  BOOL OsslToTpmBn(bigNum bn, BIGNUM* osslBn);
20
21  /*** BigInitialized()
22  // This function initializes an OSSL BIGNUM from a TPM bigConst. Do not use this for
23  // values that are passed to OpenSSL when they are not declared as const in the
24  // function prototype. Instead, use BnNewVariable().
25  BIGNUM* BigInitialized(BIGNUM* toInit, bigConst initializer);
26  #endif // MATHLIB_OSSL
27
28  #endif // _TPM_TO_OSSL_MATH_FP_H_

```

B.4.1.6. /tpm/cryptolib/Ossl/include/Ossl/TpmToOsslHash.h

```

1  /*** Introduction
2  //
3  // This header file is used to 'splice' the OpenSSL hash code into the TPM code.
4  //
5  #ifndef HASH_LIB_DEFINED
6  #define HASH_LIB_DEFINED
7
8  #define HASH_LIB_OSSL
9
10 #include <openssl/evp.h>
11 #include <openssl/sha.h>
12
13 #if ALG_SM3_256
14 # if defined(OPENSSL_NO_SM3) || OPENSSL_VERSION_NUMBER < 0x10101010L
15 #   error "Current version of OpenSSL doesn't support SM3"
16 # elif OPENSSL_VERSION_NUMBER >= 0x10200000L
17 #   include <openssl/sm3.h>
18 # else
19 // OpenSSL 1.1.1 keeps smX.h headers in the include/crypto directory,
20 // and they do not get installed as part of the libssl package
21 #   define SM3_LBLOCK (64 / 4)
22
23 typedef struct SM3state_st
24 {
25     unsigned int A, B, C, D, E, F, G, H;
26     unsigned int N1, Nh;
27     unsigned int data[SM3_LBLOCK];
28     unsigned int num;
29 } SM3_CTX;
30
31 int sm3_init(SM3_CTX* c);
32 int sm3_update(SM3_CTX* c, const void* data, size_t len);
33 int sm3_final(unsigned char* md, SM3_CTX* c);
34 # endif // OpenSSL < 1.2
35 #endif // ALG_SM3_256
36
37 #include <openssl/osslib_tpm.h>
38
39 /*** Links to the OpenSSL HASH code
40 /*** Links to the OpenSSL HASH code
41 /*** Links to the OpenSSL HASH code
42
43 // Redefine the internal name used for each of the hash state structures to the

```

```

44 // name used by the library.
45 // These defines need to be known in all parts of the TPM so that the structure
46 // sizes can be properly computed when needed.
47 #define tpmHashStateSHA1_t      SHA_CTX
48 #define tpmHashStateSHA256_t   SHA256_CTX
49 #define tpmHashStateSHA384_t   SHA512_CTX
50 #define tpmHashStateSHA512_t   SHA512_CTX
51 #define tpmHashStateSM3_256_t  SM3_CTX
52
53 // The defines below are only needed when compiling CryptHash.c or CryptSmac.c.
54 // This isolation is primarily to avoid name space collision. However, if there
55 // is a real collision, it will likely show up when the linker tries to put things
56 // together.
57
58 #ifdef _CRYPT_HASH_C_
59
60 typedef BYTE*      PBYTE;
61 typedef const BYTE* PCBYTE;
62
63 // Define the interface between CryptHash.c to the functions provided by the
64 // library. For each method, define the calling parameters of the method and then
65 // define how the method is invoked in CryptHash.c.
66 //
67 // All hashes are required to have the same calling sequence. If they don't, create
68 // a simple adaptation function that converts from the "standard" form of the call
69 // to the form used by the specific hash (and then send a nasty letter to the
70 // person who wrote the hash function for the library).
71 //
72 // The macro that calls the method also defines how the
73 // parameters get swizzled between the default form (in CryptHash.c) and the
74 // library form.
75 //
76 // Initialize the hash context
77 # define HASH_START_METHOD_DEF void(HASH_START_METHOD)(PANY_HASH_STATE state)
78 # define HASH_START(hashState) ((hashState)->def->method.start)(&(hashState)->state);
79
80 // Add data to the hash
81 # define HASH_DATA_METHOD_DEF \
82     void(HASH_DATA_METHOD)(PANY_HASH_STATE state, PCBYTE buffer, size_t size)
83 # define HASH_DATA(hashState, dInSize, dIn) \
84     ((hashState)->def->method.data)(&(hashState)->state, dIn, dInSize)
85
86 // Finalize the hash and get the digest
87 # define HASH_END_METHOD_DEF \
88     void(HASH_END_METHOD)(BYTE * buffer, PANY_HASH_STATE state)
89 # define HASH_END(hashState, buffer) \
90     ((hashState)->def->method.end)(buffer, &(hashState)->state)
91
92 // Copy the hash context
93 // Note: For import, export, and copy, memcpy() is used since there is no
94 // reformatting necessary between the internal and external forms.
95 # define HASH_STATE_COPY_METHOD_DEF \
96     void(HASH_STATE_COPY_METHOD)( \
97         PANY_HASH_STATE to, PCANY_HASH_STATE from, size_t size)
98 # define HASH_STATE_COPY(hashStateOut, hashStateIn) \
99     ((hashStateIn)->def->method.copy)(&(hashStateOut)->state, \
100                                     &(hashStateIn)->state, \
101                                     (hashStateIn)->def->contextSize)
102
103 // Copy (with reformatting when necessary) an internal hash structure to an
104 // external blob
105 # define HASH_STATE_EXPORT_METHOD_DEF \
106     void(HASH_STATE_EXPORT_METHOD)(BYTE * to, PCANY_HASH_STATE from, size_t size)
107 # define HASH_STATE_EXPORT(to, hashStateFrom) \
108     ((hashStateFrom)->def->method.copyOut)( \
109         &(((BYTE*)(to))[offsetof(HASH_STATE, state)]), \

```



```

110         &(hashStateFrom)->state, \
111         (hashStateFrom)->def->contextSize)
112
113 // Copy from an external blob to an internal formate (with reformatting when
114 // necessary
115 # define HASH_STATE_IMPORT_METHOD_DEF \
116     void(HASH_STATE_IMPORT_METHOD)(PANY_HASH_STATE to, const BYTE* from, size_t size)
117 # define HASH_STATE_IMPORT(hashStateTo, from) \
118     ((hashStateTo)->def->method.copyIn) ( \
119         &(hashStateTo)->state, \
120         &(((const BYTE*)(from))[offsetof(HASH_STATE, state)]), \
121         (hashStateTo)->def->contextSize)
122
123 // Function aliases. The code in CryptHash.c uses the internal designation for the
124 // functions. These need to be translated to the function names of the library.
125 # define tpmHashStart_SHA1          SHA1_Init
126 # define tpmHashData_SHA1          SHA1_Update
127 # define tpmHashEnd_SHA1           SHA1_Final
128 # define tpmHashStateCopy_SHA1     memcpy
129 # define tpmHashStateExport_SHA1   memcpy
130 # define tpmHashStateImport_SHA1   memcpy
131 # define tpmHashStart_SHA256       SHA256_Init
132 # define tpmHashData_SHA256       SHA256_Update
133 # define tpmHashEnd_SHA256        SHA256_Final
134 # define tpmHashStateCopy_SHA256   memcpy
135 # define tpmHashStateExport_SHA256 memcpy
136 # define tpmHashStateImport_SHA256 memcpy
137 # define tpmHashStart_SHA384       SHA384_Init
138 # define tpmHashData_SHA384       SHA384_Update
139 # define tpmHashEnd_SHA384        SHA384_Final
140 # define tpmHashStateCopy_SHA384   memcpy
141 # define tpmHashStateExport_SHA384 memcpy
142 # define tpmHashStateImport_SHA384 memcpy
143 # define tpmHashStart_SHA512       SHA512_Init
144 # define tpmHashData_SHA512       SHA512_Update
145 # define tpmHashEnd_SHA512        SHA512_Final
146 # define tpmHashStateCopy_SHA512   memcpy
147 # define tpmHashStateExport_SHA512 memcpy
148 # define tpmHashStateImport_SHA512 memcpy
149 # define tpmHashStart_SM3_256      sm3_init
150 # define tpmHashData_SM3_256      sm3_update
151 # define tpmHashEnd_SM3_256       sm3_final
152 # define tpmHashStateCopy_SM3_256  memcpy
153 # define tpmHashStateExport_SM3_256 memcpy
154 # define tpmHashStateImport_SM3_256 memcpy
155
156 #endif // _CRYPT_HASH_C_
157
158 #define LibHashInit()
159 // This definition would change if there were something to report
160 #define HashLibSimulationEnd()
161
162 #endif // HASH_LIB_DEFINED

```

B.4.1.7. /tpm/cryptolib/Ossl/include/Ossl/TpmToOsslSupport_fp.h

```

1 /*(Auto-generated)
2  * Created by TpmPrototypes; Version 3.0 July 18, 2017
3  * Date: Mar 28, 2019 Time: 08:25:19PM
4  */
5
6 #ifndef _TPM_TO_OSSL_SUPPORT_FP_H_
7 #define _TPM_TO_OSSL_SUPPORT_FP_H_
8
9 #if defined(HASH_LIB_OSSL) || defined(MATH_LIB_OSSL) || defined(SYM_LIB_OSSL)

```

```

10
11  /*** BnSupportLibInit()
12  // This does any initialization required by the support library.
13  LIB_EXPORT int BnSupportLibInit(void);
14
15  /*** OsslContextEnter()
16  // This function is used to initialize an OpenSSL context at the start of a function
17  // that will call to an OpenSSL math function.
18  BN_CTX* OsslContextEnter(void);
19
20  /*** OsslContextLeave()
21  // This is the companion function to OsslContextEnter().
22  void OsslContextLeave(BN_CTX* CTX);
23
24  /*** OsslPushContext()
25  // This function is used to create a frame in a context. All values allocated within
26  // this context after the frame is started will be automatically freed when the
27  // context (OsslPopContext())
28  BN_CTX* OsslPushContext(BN_CTX* CTX);
29
30  /*** OsslPopContext()
31  // This is the companion function to OsslPushContext().
32  void OsslPopContext(BN_CTX* CTX);
33  #endif // HASH_LIB_OSSL || MATH_LIB_OSSL || SYM_LIB_OSSL
34
35  #endif // _TPM_TO_OSSL_SUPPORT_FP_H_

```

B.4.1.8. /tpm/cryptolib/Ossl/include/Ossl/TpmToOsslSym.h

```

1  /*** Introduction
2  //
3  // This header file is used to 'splice' the OpenSSL library into the TPM code.
4  //
5  // The support required of a library are a hash module, a block cipher module and
6  // portions of a big number library.
7
8  // All of the library-dependent headers should have the same guard to that only the
9  // first one gets defined.
10 #ifndef SYM_LIB_DEFINED
11 #define SYM_LIB_DEFINED
12
13 #define SYM_LIB_OSSL
14
15 #include <openssl/aes.h>
16
17 #if ALG_SM4
18 # if defined(OPENSSL_NO_SM4) || OPENSSL_VERSION_NUMBER < 0x10101010L
19 #   error "Current version of OpenSSL doesn't support SM4"
20 # elif OPENSSL_VERSION_NUMBER >= 0x10200000L
21 #   include <openssl/sm4.h>
22 # else
23 // OpenSSL 1.1.1 keeps smX.h headers in the include/crypto directory,
24 // and they do not get installed as part of the libssl package
25
26 #   define SM4_KEY_SCHEDULE 32
27
28 typedef struct SM4_KEY_st
29 {
30     uint32_t rk[SM4_KEY_SCHEDULE];
31 } SM4_KEY;
32
33 int SM4_set_key(const uint8_t* key, SM4_KEY* ks);
34 void SM4_encrypt(const uint8_t* in, uint8_t* out, const SM4_KEY* ks);
35 void SM4_decrypt(const uint8_t* in, uint8_t* out, const SM4_KEY* ks);
36 # endif // OpenSSL < 1.2

```

```

37  #endif    // ALG_SM4
38
39  #if ALG_CAMELLIA
40  # include <openssl/camellia.h>
41  #endif
42
43  #include <openssl/bn.h>
44  #include <openssl/ssl_typ.h>
45
46  /** Links to the OpenSSL symmetric algorithms.
47  /** Links to the OpenSSL symmetric algorithms.
48  /** Links to the OpenSSL symmetric algorithms.
49
50  // The Crypt functions that call the block encryption function use the parameters
51  // in the order:
52  // 1) keySchedule
53  // 2) in buffer
54  // 3) out buffer
55  // Since open SSL uses the order in encryptoCall_t above, need to swizzle the
56  // values to the order required by the library.
57  #define SWIZZLE(keySchedule, in, out) \
58      (const BYTE*)(in), (BYTE*)(out), (void*)(keySchedule)
59
60  // Define the order of parameters to the library functions that do block encryption
61  // and decryption.
62  typedef void (*TpmCryptSetSymKeyCall_t)(const BYTE* in, BYTE* out, void* keySchedule);
63
64  /** Links to the OpenSSL AES code
65  /** Links to the OpenSSL AES code
66  /** Links to the OpenSSL AES code
67  // Macros to set up the encryption/decryption key schedules
68  //
69  // AES:
70  #define TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule) \
71      AES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES*)(schedule))
72  #define TpmCryptSetDecryptKeyAES(key, keySizeInBits, schedule) \
73      AES_set_decrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES*)(schedule))
74
75  // Macros to alias encryption calls to specific algorithms. This should be used
76  // sparingly. Currently, only used by CryptSym.c and CryptRand.c
77  //
78  // When using these calls, to call the AES block encryption code, the caller
79  // should use:
80  // TpmCryptEncryptAES(SWIZZLE(keySchedule, in, out));
81  #define TpmCryptEncryptAES AES_encrypt
82  #define TpmCryptDecryptAES AES_decrypt
83  #define tpmKeyScheduleAES AES_KEY
84
85  /** Links to the OpenSSL SM4 code
86  /** Links to the OpenSSL SM4 code
87  /** Links to the OpenSSL SM4 code
88  // Macros to set up the encryption/decryption key schedules
89  #define TpmCryptSetEncryptKeySM4(key, keySizeInBits, schedule) \
90      SM4_set_key((key), (tpmKeyScheduleSM4*)(schedule))
91  #define TpmCryptSetDecryptKeySM4(key, keySizeInBits, schedule) \
92      SM4_set_key((key), (tpmKeyScheduleSM4*)(schedule))
93
94  // Macros to alias encryption calls to specific algorithms. This should be used
95  // sparingly.
96  #define TpmCryptEncryptSM4 SM4_encrypt
97  #define TpmCryptDecryptSM4 SM4_decrypt
98  #define tpmKeyScheduleSM4 SM4_KEY
99
100  /** Links to the OpenSSL CAMELLIA code
101  /** Links to the OpenSSL CAMELLIA code
102  /** Links to the OpenSSL CAMELLIA code

```

```
103 // Macros to set up the encryption/decryption key schedules
104 #define TpmCryptSetEncryptKeyCAMELLIA(key, keySizeInBits, schedule) \
105     Camellia_set_key((key), (keySizeInBits), (tpmKeyScheduleCAMELLIA*)(schedule))
106 #define TpmCryptSetDecryptKeyCAMELLIA(key, keySizeInBits, schedule) \
107     Camellia_set_key((key), (keySizeInBits), (tpmKeyScheduleCAMELLIA*)(schedule))
108
109 // Macros to alias encryption calls to specific algorithms. This should be used
110 // sparingly.
111 #define TpmCryptEncryptCAMELLIA Camellia_encrypt
112 #define TpmCryptDecryptCAMELLIA Camellia_decrypt
113 #define tpmKeyScheduleCAMELLIA CAMELLIA_KEY
114
115 // Forward reference
116
117 typedef union tpmCryptKeySchedule_t tpmCryptKeySchedule_t;
118
119 // This definition would change if there were something to report
120 #define SymLibSimulationEnd()
121
122 #endif // SYM_LIB_DEFINED
123
```

Annex C (informative) Simulation Environment

C.1 Introduction

These files are used to simulate some of the implementation-dependent hardware of a TPM. These files are provided to allow creation of a simulation environment for the TPM. These files are not expected to be part of a hardware TPM implementation.

C.1.1. /Platform/include/Platform.h

```

1  #ifndef _PLATFORM_H_
2  #define _PLATFORM_H_
3
4  #include <TpmConfiguration/TpmBuildSwitches.h>
5  #include <TpmConfiguration/TpmProfile.h>
6  // TODO_RENAME_INC_FOLDER: public refers to the TPM_CoreLib public headers
7  #include <public/BaseTypes.h>
8  #include <public/TPMB.h>
9  #include <public/MinMax.h>
10
11 #include "PlatformACT.h"
12 #include "PlatformClock.h"
13 #include "PlatformData.h"
14 #include "prototypes/platform_public_interface.h"
15 // TODO_RENAME_INC_FOLDER:platform_interface refers to the TPM_CoreLib platform
16 // interface
17 #include <platform_interface/tpm_to_platform_interface.h>
18 #include <platform_interface/platform_to_tpm_interface.h>
19
20 #define GLOBAL_C
21 #define NV_C
22 #include <platform_interface/pcrstruct.h>
23 #include <platform_interface/prototypes/platform_pcr_fp.h>
24
25 #endif // _PLATFORM_H_

```

C.1.2. /Platform/include/PlatformACT.h

```

1  // This file contains the definitions for the ACT macros and data types used in the
2  // ACT implementation.
3
4  #ifndef _PLATFORM_ACT_H_
5  #define _PLATFORM_ACT_H_
6
7  typedef struct ACT_DATA
8  {
9      uint32_t remaining;
10     uint32_t newValue;
11     uint8_t signaled;
12     uint8_t pending;
13     uint8_t number;
14 } ACT_DATA, *P_ACT_DATA;
15
16 #if !(defined RH_ACT_0) || (RH_ACT_0 != YES)
17 #   undef RH_ACT_0
18 #   define RH_ACT_0 NO
19 #   define IF_ACT_0_IMPLEMENTED(op)
20 #else
21 #   define IF_ACT_0_IMPLEMENTED(op) op(0)

```

```
22 #endif
23 #if !(defined RH_ACT_1) || (RH_ACT_1 != YES)
24 # undef RH_ACT_1
25 # define RH_ACT_1 NO
26 # define IF_ACT_1_IMPLEMENTED(op)
27 #else
28 # define IF_ACT_1_IMPLEMENTED(op) op(1)
29 #endif
30 #if !(defined RH_ACT_2) || (RH_ACT_2 != YES)
31 # undef RH_ACT_2
32 # define RH_ACT_2 NO
33 # define IF_ACT_2_IMPLEMENTED(op)
34 #else
35 # define IF_ACT_2_IMPLEMENTED(op) op(2)
36 #endif
37 #if !(defined RH_ACT_3) || (RH_ACT_3 != YES)
38 # undef RH_ACT_3
39 # define RH_ACT_3 NO
40 # define IF_ACT_3_IMPLEMENTED(op)
41 #else
42 # define IF_ACT_3_IMPLEMENTED(op) op(3)
43 #endif
44 #if !(defined RH_ACT_4) || (RH_ACT_4 != YES)
45 # undef RH_ACT_4
46 # define RH_ACT_4 NO
47 # define IF_ACT_4_IMPLEMENTED(op)
48 #else
49 # define IF_ACT_4_IMPLEMENTED(op) op(4)
50 #endif
51 #if !(defined RH_ACT_5) || (RH_ACT_5 != YES)
52 # undef RH_ACT_5
53 # define RH_ACT_5 NO
54 # define IF_ACT_5_IMPLEMENTED(op)
55 #else
56 # define IF_ACT_5_IMPLEMENTED(op) op(5)
57 #endif
58 #if !(defined RH_ACT_6) || (RH_ACT_6 != YES)
59 # undef RH_ACT_6
60 # define RH_ACT_6 NO
61 # define IF_ACT_6_IMPLEMENTED(op)
62 #else
63 # define IF_ACT_6_IMPLEMENTED(op) op(6)
64 #endif
65 #if !(defined RH_ACT_7) || (RH_ACT_7 != YES)
66 # undef RH_ACT_7
67 # define RH_ACT_7 NO
68 # define IF_ACT_7_IMPLEMENTED(op)
69 #else
70 # define IF_ACT_7_IMPLEMENTED(op) op(7)
71 #endif
72 #if !(defined RH_ACT_8) || (RH_ACT_8 != YES)
73 # undef RH_ACT_8
74 # define RH_ACT_8 NO
75 # define IF_ACT_8_IMPLEMENTED(op)
76 #else
77 # define IF_ACT_8_IMPLEMENTED(op) op(8)
78 #endif
79 #if !(defined RH_ACT_9) || (RH_ACT_9 != YES)
80 # undef RH_ACT_9
81 # define RH_ACT_9 NO
82 # define IF_ACT_9_IMPLEMENTED(op)
83 #else
84 # define IF_ACT_9_IMPLEMENTED(op) op(9)
85 #endif
86 #if !(defined RH_ACT_A) || (RH_ACT_A != YES)
87 # undef RH_ACT_A
```



```

88  # define RH_ACT_A NO
89  # define IF_ACT_A_IMPLEMENTED(op)
90  #else
91  # define IF_ACT_A_IMPLEMENTED(op) op(A)
92  #endif
93  #if !(defined RH_ACT_B) || (RH_ACT_B != YES)
94  # undef RH_ACT_B
95  # define RH_ACT_B NO
96  # define IF_ACT_B_IMPLEMENTED(op)
97  #else
98  # define IF_ACT_B_IMPLEMENTED(op) op(B)
99  #endif
100 #if !(defined RH_ACT_C) || (RH_ACT_C != YES)
101 # undef RH_ACT_C
102 # define RH_ACT_C NO
103 # define IF_ACT_C_IMPLEMENTED(op)
104 #else
105 # define IF_ACT_C_IMPLEMENTED(op) op(C)
106 #endif
107 #if !(defined RH_ACT_D) || (RH_ACT_D != YES)
108 # undef RH_ACT_D
109 # define RH_ACT_D NO
110 # define IF_ACT_D_IMPLEMENTED(op)
111 #else
112 # define IF_ACT_D_IMPLEMENTED(op) op(D)
113 #endif
114 #if !(defined RH_ACT_E) || (RH_ACT_E != YES)
115 # undef RH_ACT_E
116 # define RH_ACT_E NO
117 # define IF_ACT_E_IMPLEMENTED(op)
118 #else
119 # define IF_ACT_E_IMPLEMENTED(op) op(E)
120 #endif
121 #if !(defined RH_ACT_F) || (RH_ACT_F != YES)
122 # undef RH_ACT_F
123 # define RH_ACT_F NO
124 # define IF_ACT_F_IMPLEMENTED(op)
125 #else
126 # define IF_ACT_F_IMPLEMENTED(op) op(F)
127 #endif
128
129 #define FOR_EACH_ACT(op) \
130     IF_ACT_0_IMPLEMENTED(op) \
131     IF_ACT_1_IMPLEMENTED(op) \
132     IF_ACT_2_IMPLEMENTED(op) \
133     IF_ACT_3_IMPLEMENTED(op) \
134     IF_ACT_4_IMPLEMENTED(op) \
135     IF_ACT_5_IMPLEMENTED(op) \
136     IF_ACT_6_IMPLEMENTED(op) \
137     IF_ACT_7_IMPLEMENTED(op) \
138     IF_ACT_8_IMPLEMENTED(op) \
139     IF_ACT_9_IMPLEMENTED(op) \
140     IF_ACT_A_IMPLEMENTED(op) \
141     IF_ACT_B_IMPLEMENTED(op) \
142     IF_ACT_C_IMPLEMENTED(op) \
143     IF_ACT_D_IMPLEMENTED(op) \
144     IF_ACT_E_IMPLEMENTED(op) \
145     IF_ACT_F_IMPLEMENTED(op)
146
147 #endif // _PLATFORM_ACT_H_

```

C.1.3. /Platform/include/PlatformClock.h

```

1  // This file contains the instance data for the Platform module. It is collected
2  // in this file so that the state of the module is easier to manage.

```

```

3
4 #ifndef _PLATFORM_CLOCK_H_
5 #define _PLATFORM_CLOCK_H_
6
7 #ifndef _ARM_
8 #   ifdef _MSC_VER
9 #       include <sys/types.h>
10 #       include <sys/timeb.h>
11 #   else
12 #       include <time.h>
13 #   endif
14 #endif
15
16 #endif // _PLATFORM_CLOCK_H_

```

C.1.4. /Platform/include/PlatformData.h

```

1 // This file contains the instance data for the Platform module. It is collected
2 // in this file so that the state of the module is easier to manage.
3
4 #ifndef _PLATFORM_DATA_H_
5 #define _PLATFORM_DATA_H_
6
7 #ifndef EXTERN
8 #   ifdef _PLATFORM_DATA_C_
9 #       define EXTERN
10 #   else
11 #       define EXTERN extern
12 #   endif // _PLATFORM_DATA_C_
13 #endif // EXTERN
14
15 // From Cancel.c
16 // Cancel flag. It is initialized as FALSE, which indicate the command is not
17 // being canceled
18 EXTERN int s_isCanceled;
19
20 #ifndef HARDWARE_CLOCK
21 typedef uint64_t clock64_t;
22 // This is the value returned the last time that the system clock was read. This
23 // is only relevant for a simulator or virtual TPM.
24 EXTERN clock64_t s_realTimePrevious;
25
26 // These values are used to try to synthesize a long lived version of clock().
27 EXTERN clock64_t s_lastSystemTime;
28 EXTERN clock64_t s_lastReportedTime;
29
30 // This is the rate adjusted value that is the equivalent of what would be read from
31 // a hardware register that produced rate adjusted time.
32 EXTERN clock64_t s_tpmTime;
33 #endif // HARDWARE_CLOCK
34
35 // This value indicates that the timer was reset
36 EXTERN int s_timerReset;
37 // This value indicates that the timer was stopped. It causes a clock discontinuity.
38 EXTERN int s_timerStopped;
39
40 // This variable records the time when plat_TimerReset is called. This mechanism
41 // allow us to subtract the time when TPM is power off from the total
42 // time reported by clock() function
43 EXTERN uint64_t s_initClock;
44
45 // This variable records the timer adjustment factor.
46 EXTERN unsigned int s_adjustRate;
47
48 // For LocalityPlat.c

```

```

49 // Locality of current command
50 EXTERN unsigned char s_locality;
51
52 // For NVMem.c
53 // Choose if the NV memory should be backed by RAM or by file.
54 // If this macro is defined, then a file is used as NV. If it is not defined,
55 // then RAM is used to back NV memory. Comment out to use RAM.
56
57 #if(!defined VTPM) || ((VTPM != NO) && (VTPM != YES))
58 # undef VTPM
59 # define VTPM YES // Default: Either YES or NO
60 #endif
61
62 // For a simulation, use a file to back up the NV
63 #if(!defined FILE_BACKED_NV) || ((FILE_BACKED_NV != NO) && (FILE_BACKED_NV != YES))
64 # undef FILE_BACKED_NV
65 # define FILE_BACKED_NV (VTPM && YES) // Default: Either YES or NO
66 #endif
67
68 #if SIMULATION
69 # undef FILE_BACKED_NV
70 # define FILE_BACKED_NV YES
71 #endif // SIMULATION
72
73 EXTERN unsigned char s_NV[NV_MEMORY_SIZE];
74 EXTERN int s_NvIsAvailable;
75 EXTERN int s_NV_unrecoverable;
76 EXTERN int s_NV_recoverable;
77
78 // For PPPlat.c
79 // Physical presence. It is initialized to FALSE
80 EXTERN int s_physicalPresence;
81
82 // From Power
83 EXTERN int s_powerLost;
84
85 // For Entropy.c
86 EXTERN uint32_t lastEntropy;
87
88 #define DEFINE_ACT(N) EXTERN ACT_DATA ACT_##N;
89 FOR_EACH_ACT(DEFINE_ACT)
90
91 EXTERN int actTicksAllowed;
92
93 #endif // _PLATFORM_DATA_H_

```

C.1.5. /Platform/include/prototypes/platform_public_interface.h

```

1 // This file contains the interface into the platform layer from external callers.
2 // External callers are expected to be implementation specific, and may be a simulator
3 // or some other implementation
4
5 #ifndef _PLATFORM_PUBLIC_INTERFACE_H_
6 #define _PLATFORM_PUBLIC_INTERFACE_H_
7
8 #include <stddef.h>
9
10 /** From Cancel.c
11
12 // Set cancel flag.
13 LIB_EXPORT void _plat__SetCancel(void);
14
15 /***_plat__ClearCancel()
16 // Clear cancel flag
17 LIB_EXPORT void _plat__ClearCancel(void);

```

```

18
19 /** From Clock.c
20
21 /***_plat_TimerReset()
22 // This function sets current system clock time as t0 for counting TPM time.
23 // This function is called at a power on event to reset the clock. When the clock
24 // is reset, the indication that the clock was stopped is also set.
25 LIB_EXPORT void _plat_TimerReset(void);
26
27 /***_plat_TimerRestart()
28 // This function should be called in order to simulate the restart of the timer
29 // should it be stopped while power is still applied.
30 LIB_EXPORT void _plat_TimerRestart(void);
31
32 /***_plat_RealTime()
33 // This is another, probably futile, attempt to define a portable function
34 // that will return a 64-bit clock value that has mSec resolution.
35 LIB_EXPORT uint64_t _plat_RealTime(void);
36
37 /** From LocalityPlat.c
38
39 /***_plat_LocalitySet()
40 // Set the most recent command locality in locality value form
41 LIB_EXPORT void _plat_LocalitySet(unsigned char locality);
42
43 /** From NVMem.c
44
45 /***_plat_NvErrors()
46 // This function is used by the simulator to set the error flags in the NV
47 // subsystem to simulate an error in the NV loading process
48 LIB_EXPORT void _plat_NvErrors(int recoverable, int unrecoverable);
49
50 /***_plat_NVDisable()
51 // Disable NV memory
52 LIB_EXPORT void _plat_NVDisable(
53     void* platParameter, // platform specific parameter
54     size_t paramSize     // size of parameter. If size == 0, then
55                          // parameter is a sizeof(void*) scalar and should
56                          // be cast to an integer (intptr_t), not dereferenced.
57 );
58
59 /***_plat_SetNvAvail()
60 // Set the current NV state to available. This function is for testing purpose
61 // only. It is not part of the platform NV logic
62 LIB_EXPORT void _plat_SetNvAvail(void);
63
64 /***_plat_ClearNvAvail()
65 // Set the current NV state to unavailable. This function is for testing purpose
66 // only. It is not part of the platform NV logic
67 LIB_EXPORT void _plat_ClearNvAvail(void);
68
69 /***_plat_NVNeedsManufacture()
70 // This function is used by the simulator to determine when the TPM's NV state
71 // needs to be manufactured.
72 LIB_EXPORT int _plat_NVNeedsManufacture(void);
73
74 /** From PlatformACT.c
75
76 /***_plat_ACT_GetPending()
77 LIB_EXPORT int _plat_ACT_GetPending(uint32_t act //IN: number of ACT to check
78 );
79
80 /***_plat_ACT_Tick()
81 // This processes the once-per-second clock tick from the hardware. This is set up
82 // for the simulator to use the control interface to send ticks to the TPM. These
83 // ticks do not have to be on a per second basis. They can be as slow or as fast as

```

```

84  // desired so that the simulation can be tested.
85  LIB_EXPORT void _plat__ACT_Tick(void);
86
87  /** From PowerPlat.c
88
89  /***_plat__Signal_PowerOn()
90  // Signal platform power on
91  LIB_EXPORT int _plat__Signal_PowerOn(void);
92
93  /***_plat__Signal_Reset()
94  // This a TPM reset without a power loss.
95  LIB_EXPORT int _plat__Signal_Reset(void);
96
97  /***_plat__Signal_PowerOff()
98  // Signal platform power off
99  LIB_EXPORT void _plat__Signal_PowerOff(void);
100
101  /** From PPPlat.c
102
103  /***_plat__Signal_PhysicalPresenceOn()
104  // Signal physical presence on
105  LIB_EXPORT void _plat__Signal_PhysicalPresenceOn(void);
106
107  /***_plat__Signal_PhysicalPresenceOff()
108  // Signal physical presence off
109  LIB_EXPORT void _plat__Signal_PhysicalPresenceOff(void);
110
111  /***_plat__SetTpmFirmwareHash()
112  // Called by the simulator to set the TPM Firmware hash used for
113  // firmware-bound hierarchies. Not a cryptographically-strong hash.
114  #if SIMULATION
115  LIB_EXPORT void _plat__SetTpmFirmwareHash(uint32_t hash);
116  #endif
117
118  /***_plat__SetTpmFirmwareSvn()
119  // Called by the simulator to set the TPM Firmware SVN reported by
120  // getCapability.
121  #if SIMULATION
122  LIB_EXPORT void _plat__SetTpmFirmwareSvn(uint16_t svn);
123  #endif
124
125  /** From RunCommand.c
126
127  /***_plat__RunCommand()
128  // This version of RunCommand will set up a jum buf and call ExecuteCommand(). If
129  // the command executes without failing, it will return and RunCommand will return.
130  // If there is a failure in the command, then _plat__Fail() is called and it will
131  // longjump back to RunCommand which will call ExecuteCommand again. However, this
132  // time, the TPM will be in failure mode so ExecuteCommand will simply build
133  // a failure response and return.
134  LIB_EXPORT void _plat__RunCommand(
135      uint32_t      requestSize,    // IN: command buffer size
136      unsigned char* request,      // IN: command buffer
137      uint32_t*     responseSize,   // IN/OUT: response buffer size
138      unsigned char** response     // IN/OUT: response buffer
139  );
140
141  #endif // _PLATFORM_PUBLIC_INTERFACE_H_

```

C.1.6. /Platform/src/Cancel.c

```

1  /*** Description
2  //
3  // This module simulates the cancel pins on the TPM.
4  //

```

```

5  /** Includes, Typedefs, Structures, and Defines
6  #include "Platform.h"
7
8  /** Functions
9
10 /***_plat_IsCanceled()
11 // Check if the cancel flag is set
12 // Return Type: int
13 //     TRUE(1)         if cancel flag is set
14 //     FALSE(0)        if cancel flag is not set
15 LIB_EXPORT int _plat_IsCanceled(void)
16 {
17     // return cancel flag
18     return s_isCanceled;
19 }
20
21 /***_plat_SetCancel()
22
23 // Set cancel flag.
24 LIB_EXPORT void _plat_SetCancel(void)
25 {
26     s_isCanceled = TRUE;
27     return;
28 }
29
30 /***_plat_ClearCancel()
31 // Clear cancel flag
32 LIB_EXPORT void _plat_ClearCancel(void)
33 {
34     s_isCanceled = FALSE;
35     return;
36 }

```

C.1.7. /Platform/src/Clock.c

```

1  /** Description
2  //
3  // This file contains the routines that are used by the simulator to mimic
4  // a hardware clock on a TPM.
5  //
6  // In this implementation, all the time values are measured in millisecond.
7  // However, the precision of the clock functions may be implementation dependent.
8
9  /** Includes and Data Definitions
10 #include <assert.h>
11 #include "Platform.h"
12
13 // CLOCK_NOMINAL is the number of hardware ticks per ms. A value of 30000 means
14 // that the nominal clock rate used to drive the hardware clock is 30 MHz. The
15 // adjustment rates are used to determine the conversion of the hardware ticks to
16 // internal hardware clock value. In practice, we would expect that there would be
17 // a hardware register will accumulated mS. It would be incremented by the output
18 // of a pre-scaler. The pre-scaler would divide the ticks from the clock by some
19 // value that would compensate for the difference between clock time and real time.
20 // The code in Clock does the emulation of this function.
21 #define CLOCK_NOMINAL 30000
22 // A 1% change in rate is 300 counts
23 #define CLOCK_ADJUST_COARSE 300
24 // A 0.1% change in rate is 30 counts
25 #define CLOCK_ADJUST_MEDIUM 30
26 // A minimum change in rate is 1 count
27 #define CLOCK_ADJUST_FINE 1
28 // The clock tolerance is +/-15% (4500 counts)
29 // Allow some guard band (16.7%)
30 #define CLOCK_ADJUST_LIMIT 5000

```



```

31
32 /** Simulator Functions
33 /*** Introduction
34 // This set of functions is intended to be called by the simulator environment in
35 // order to simulate hardware events.
36
37 /*** _plat_TimerReset()
38 // This function sets current system clock time as t0 for counting TPM time.
39 // This function is called at a power on event to reset the clock. When the clock
40 // is reset, the indication that the clock was stopped is also set.
41 LIB_EXPORT void _plat_TimerReset(void)
42 {
43     s_lastSystemTime = 0;
44     s_tpmTime        = 0;
45     s_adjustRate     = CLOCK_NOMINAL;
46     s_timerReset     = TRUE;
47     s_timerStopped   = TRUE;
48     return;
49 }
50
51 /*** _plat_TimerRestart()
52 // This function should be called in order to simulate the restart of the timer
53 // should it be stopped while power is still applied.
54 LIB_EXPORT void _plat_TimerRestart(void)
55 {
56     s_timerStopped = TRUE;
57     return;
58 }
59
60 /** Functions Used by TPM
61 /*** Introduction
62 // These functions are called by the TPM code. They should be replaced by
63 // appropriated hardware functions.
64
65 #include <time.h>
66 clock_t debugTime;
67
68 /*** _plat_RealTime()
69 // This is another, probably futile, attempt to define a portable function
70 // that will return a 64-bit clock value that has mSec resolution.
71 LIB_EXPORT uint64_t _plat_RealTime(void)
72 {
73     clock64_t time;
74 #ifdef _MSC_VER
75     struct _timeb sysTime;
76     //
77     _ftime_s(&sysTime);
78     time = (clock64_t)(sysTime.time) * 1000 + sysTime.millitm;
79     // set the time back by one hour if daylight savings
80     if(sysTime.dstflag)
81         time -= 1000 * 60 * 60; // mSec/sec * sec/min * min/hour = ms/hour
82 #else
83     // hopefully, this will work with most UNIX systems
84     struct timespec systime;
85     //
86     clock_gettime(CLOCK_MONOTONIC, &systime);
87     time = (clock64_t)systime.tv_sec * 1000 + (systime.tv_nsec / 1000000);
88 #endif
89     return time;
90 }
91
92 /*** _plat_TimerRead()
93 // This function provides access to the tick timer of the platform. The TPM code
94 // uses this value to drive the TPM Clock.
95 //
96 // The tick timer is supposed to run when power is applied to the device. This timer

```

```

97 // should not be reset by time events including _TPM_Init. It should only be reset
98 // when TPM power is re-applied.
99 //
100 // If the TPM is run in a protected environment, that environment may provide the
101 // tick time to the TPM as long as the time provided by the environment is not
102 // allowed to go backwards. If the time provided by the system can go backwards
103 // during a power discontinuity, then the _plat__Signal_PowerOn should call
104 // _plat__TimerReset().
105 LIB_EXPORT uint64_t _plat__TimerRead(void)
106 {
107 #ifdef HARDWARE_CLOCK
108 # error "need a definition for reading the hardware clock"
109     return HARDWARE_CLOCK
110 #else
111     clock64_t timeDiff;
112     clock64_t adjustedTimeDiff;
113     clock64_t timeNow;
114     clock64_t readjustedTimeDiff;
115
116     // This produces a timeNow that is basically locked to the system clock.
117     timeNow = _plat__RealTime();
118
119     // if this hasn't been initialized, initialize it
120     if(s_lastSystemTime == 0)
121     {
122         s_lastSystemTime = timeNow;
123         debugTime = clock();
124         s_lastReportedTime = 0;
125         s_realTimePrevious = 0;
126     }
127     // The system time can bounce around and that's OK as long as we don't allow
128     // time to go backwards. When the time does appear to go backwards, set
129     // lastSystemTime to be the new value and then update the reported time.
130     if(timeNow < s_lastReportedTime)
131         s_lastSystemTime = timeNow;
132     s_lastReportedTime = s_lastReportedTime + timeNow - s_lastSystemTime;
133     s_lastSystemTime = timeNow;
134     timeNow = s_lastReportedTime;
135
136     // The code above produces a timeNow that is similar to the value returned
137     // by Clock(). The difference is that timeNow does not max out, and it is
138     // at a ms. rate rather than at a CLOCKS_PER_SEC rate. The code below
139     // uses that value and does the rate adjustment on the time value.
140     // If there is no difference in time, then skip all the computations
141     if(s_realTimePrevious >= timeNow)
142         return s_tpmTime;
143     // Compute the amount of time since the last update of the system clock
144     timeDiff = timeNow - s_realTimePrevious;
145
146     // Do the time rate adjustment and conversion from CLOCKS_PER_SEC to mSec
147     adjustedTimeDiff = (timeDiff * CLOCK_NOMINAL) / ((uint64_t)s_adjustRate);
148
149     // update the TPM time with the adjusted timeDiff
150     s_tpmTime += (clock64_t)adjustedTimeDiff;
151
152     // Might have some rounding error that would loose CLOCKS. See what is not
153     // being used. As mentioned above, this could result in putting back more than
154     // is taken out. Here, we are trying to recreate timeDiff.
155     readjustedTimeDiff = (adjustedTimeDiff * (uint64_t)s_adjustRate) / CLOCK_NOMINAL;
156
157     // adjusted is now converted back to being the amount we should advance the
158     // previous sampled time. It should always be less than or equal to timeDiff.
159     // That is, we could not have use more time than we started with.
160     s_realTimePrevious = s_realTimePrevious + readjustedTimeDiff;
161
162 #ifdef DEBUGGING_TIME

```

```

163     // Put this in so that TPM time will pass much faster than real time when
164     // doing debug.
165     // A value of 1000 for DEBUG_TIME_MULTIPLIER will make each ms into a second
166     // A good value might be 100
167     return (s_tpmTime * DEBUG_TIME_MULTIPLIER);
168 # endif
169     return s_tpmTime;
170 #endif
171 }
172
173 /***_plat_TimerWasReset()
174 // This function is used to interrogate the flag indicating if the tick timer has
175 // been reset.
176 //
177 // If the resetFlag parameter is SET, then the flag will be CLEAR before the
178 // function returns.
179 LIB_EXPORT int _plat_TimerWasReset(void)
180 {
181     int retVal    = s_timerReset;
182     s_timerReset = FALSE;
183     return retVal;
184 }
185
186 /***_plat_TimerWasStopped()
187 // This function is used to interrogate the flag indicating if the tick timer has
188 // been stopped. If so, this is typically a reason to roll the nonce.
189 //
190 // This function will CLEAR the s_timerStopped flag before returning. This provides
191 // functionality that is similar to status register that is cleared when read. This
192 // is the model used here because it is the one that has the most impact on the TPM
193 // code as the flag can only be accessed by one entity in the TPM. Any other
194 // implementation of the hardware can be made to look like a read-once register.
195 LIB_EXPORT int _plat_TimerWasStopped(void)
196 {
197     int retVal    = s_timerStopped;
198     s_timerStopped = FALSE;
199     return retVal;
200 }
201
202 /***_plat_ClockAdjustRate()
203 // Adjust the clock rate
204 LIB_EXPORT void _plat_ClockRateAdjust(_plat_ClockAdjustStep adjust)
205 {
206     // We expect the caller should only use a fixed set of constant values to
207     // adjust the rate
208     switch(adjust)
209     {
210         // slower increases the divisor
211         case PLAT_TPM_CLOCK_ADJUST_COARSE_SLOWER:
212             s_adjustRate += CLOCK_ADJUST_COARSE;
213             break;
214         case PLAT_TPM_CLOCK_ADJUST_MEDIUM_SLOWER:
215             s_adjustRate += CLOCK_ADJUST_MEDIUM;
216             break;
217         case PLAT_TPM_CLOCK_ADJUST_FINE_SLOWER:
218             s_adjustRate += CLOCK_ADJUST_FINE;
219             break;
220         // faster decreases the divisor
221         case PLAT_TPM_CLOCK_ADJUST_FINE_FASTER:
222             s_adjustRate -= CLOCK_ADJUST_FINE;
223             break;
224         case PLAT_TPM_CLOCK_ADJUST_MEDIUM_FASTER:
225             s_adjustRate -= CLOCK_ADJUST_MEDIUM;
226             break;
227         case PLAT_TPM_CLOCK_ADJUST_COARSE_FASTER:
228             s_adjustRate -= CLOCK_ADJUST_COARSE;

```

```

229         break;
230     }
231
232     if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
233         s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
234     if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
235         s_adjustRate = CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT;
236
237     return;
238 }

```

C.1.8. /Platform/src/DebugHelpers.c

```

1  /** Description
2  //
3  //   This file contains the NV read and write access methods. This implementation
4  //   uses RAM/file and does not manage the RAM/file as NV blocks.
5  //   The implementation may become more sophisticated over time.
6  //
7
8  /** Includes and Local
9  #include <stdio.h>
10 #include <time.h>
11 #include "Platform.h"
12
13 #if CERTIFYX509_DEBUG
14
15 const char* debugFileName = "DebugFile.txt";
16
17 /*** fileOpen()
18 // This exists to allow use of the 'safe' version of fopen() with a MS runtime.
19 static FILE* fileOpen(const char* fn, const char* mode)
20 {
21     FILE* f;
22     # if defined _MSC_VER
23     if(fopen_s(&f, fn, mode) != 0)
24         f = NULL;
25     # else
26     f = fopen(fn, mode);
27     # endif
28     return f;
29 }
30
31 /*** DebugFileInit()
32 // This function initializes the file containing the debug data with the time of the
33 // file creation.
34 // Return Type: int
35 // 0 success
36 // != 0 error
37 int DebugFileInit(void)
38 {
39     FILE* f = NULL;
40     time_t t = time(NULL);
41     //
42     // Get current date and time.
43     # if defined _MSC_VER
44     char timeString[100];
45     ctime_s(timeString, (size_t)sizeof(timeString), &t);
46     # else
47     char* timeString;
48     timeString = ctime(&t);
49     # endif
50     // Try to open the debug file
51     f = fileOpen(debugFileName, "w");
52     if(f)

```

```

53     {
54         // Initialize the contents with the time.
55         fprintf(f, "%s\n", timeString);
56         fclose(f);
57         return 0;
58     }
59     return -1;
60 }
61
62 /** DebugDumpBuffer()
63 void DebugDumpBuffer(int size, unsigned char* buf, const char* identifier)
64 {
65     int i;
66     //
67     FILE* f = fopen(debugFileName, "a");
68     if(!f)
69         return;
70     if(identifier)
71         fprintf(f, "%s\n", identifier);
72     if(buf)
73     {
74         for(i = 0; i < size; i++)
75         {
76             if((i % 16) == 0) && (i))
77                 fprintf(f, "\n");
78             fprintf(f, " %02X", buf[i]);
79         }
80         if((size % 16) != 0)
81             fprintf(f, "\n");
82     }
83     fclose(f);
84 }
85
86 #endif // CERTIFYX509_DEBUG

```

C.1.9. /Platform/src/Entropy.c

```

1  /** Includes and Local Values
2
3  #define _CRT_RAND_S
4  #include <stdlib.h>
5  #include <memory.h>
6  #include <time.h>
7  #include "Platform.h"
8
9  #ifdef _MSC_VER
10 # include <process.h>
11 #else
12 # include <unistd.h>
13 #endif
14
15 // This is the last 32-bits of hardware entropy produced. We have to check to
16 // see that two consecutive 32-bit values are not the same because
17 // according to FIPS 140-2, annex C:
18 //
19 // "If each call to an RNG produces blocks of n bits (where n > 15), the first
20 // n-bit block generated after power-up, initialization, or reset shall not be
21 // used, but shall be saved for comparison with the next n-bit block to be
22 // generated. Each subsequent generation of an n-bit block shall be compared with
23 // the previously generated block. The test shall fail if any two compared n-bit
24 // blocks are equal."
25 extern uint32_t lastEntropy;
26
27 /** Functions
28

```

```

29  /*** rand32()
30  // Local function to get a 32-bit random number
31  static uint32_t rand32(void)
32  {
33      uint32_t rndNum = rand();
34      #if RAND_MAX < UINT16_MAX
35          // If the maximum value of the random number is a 15-bit number, then shift it up
36          // 15 bits, get 15 more bits, shift that up 2 and then XOR in another value to get
37          // a full 32 bits.
38          rndNum = (rndNum << 15) ^ rand();
39          rndNum = (rndNum << 2) ^ rand();
40      #elif RAND_MAX == UINT16_MAX
41          // If the maximum size is 16-bits, shift it and add another 16 bits
42          rndNum = (rndNum << 16) ^ rand();
43      #elif RAND_MAX < UINT32_MAX
44          // If 31 bits, then shift 1 and include another random value to get the extra bit
45          rndNum = (rndNum << 1) ^ rand();
46      #endif
47      return rndNum;
48  }
49
50  /*** _plat_GetEntropy()
51  // This function is used to get available hardware entropy. In a hardware
52  // implementation of this function, there would be no call to the system
53  // to get entropy.
54  // Return Type: int32_t
55  // < 0      hardware failure of the entropy generator, this is sticky
56  // >= 0      the returned amount of entropy (bytes)
57  //
58  LIB_EXPORT int32_t _plat_GetEntropy(unsigned char* entropy, // output buffer
59                                     uint32_t amount        // amount requested
60 )
61 {
62     uint32_t rndNum;
63     int32_t ret;
64     //
65     if(amount == 0)
66     {
67         // Seed the platform entropy source if the entropy source is software. There
68         // is no reason to put a guard macro (#if or #ifdef) around this code because
69         // this code would not be here if someone was changing it for a system with
70         // actual hardware.
71         //
72         // NOTE 1: The following command does not provide proper cryptographic
73         // entropy. Its primary purpose to make sure that different instances of the
74         // simulator, possibly started by a script on the same machine, are seeded
75         // differently. Vendors of the actual TPMs need to ensure availability of
76         // proper entropy using their platform-specific means.
77         //
78         // NOTE 2: In debug builds by default the reference implementation will seed
79         // its RNG deterministically (without using any platform provided randomness).
80         // See the USE_DEBUG_RNG macro and DRBG_GetEntropy() function.
81     #ifdef _MSC_VER
82         srand((unsigned)_plat_RealTime() ^ _getpid());
83     #else
84         srand((unsigned)_plat_RealTime() ^ getpid());
85     #endif
86     lastEntropy = rand32();
87     ret = 0;
88 }
89 else
90 {
91     rndNum = rand32();
92     if(rndNum == lastEntropy)
93     {
94         ret = -1;

```



```

95     }
96     else
97     {
98         lastEntropy = rndNum;
99         // Each process will have its random number generator initialized
100        // according to the process id and the initialization time. This is not a
101        // lot of entropy so, to add a bit more, XOR the current time value into
102        // the returned entropy value.
103        // NOTE: the reason for including the time here rather than have it in
104        // in the value assigned to lastEntropy is that rand() could be broken and
105        // using the time would in the lastEntropy value would hide this.
106        rndNum ^= (uint32_t)_plat__RealTime();
107
108        // Only provide entropy 32 bits at a time to test the ability
109        // of the caller to deal with partial results.
110        ret = MIN(amount, sizeof(rndNum));
111        memcpy(entropy, &rndNum, ret);
112    }
113 }
114 return ret;
115 }

```

C.1.10. /Platform/src/ExtraData.c

```

1  /** Description
2  //
3  // This file contains routines that are called by the core library to allow the
4  // platform to use the Core storage structures for small amounts of related data.
5  //
6  // In this implementation, the buffers are all just set to 0xFF
7
8  /** Includes and Data Definitions
9  #include <assert.h>
10 #include <stdio.h>
11 #include <string.h>
12 #include "Platform.h"
13
14 /** _plat__GetPlatformManufactureData
15
16 // This function allows the platform to provide a small amount of data to be
17 // stored as part of the TPM's PERSISTENT_DATA structure during manufacture. Of
18 // course the platform can store data separately as well, but this allows a
19 // simple platform implementation to store a few bytes of data without
20 // implementing a multi-layer storage system. This function is called on
21 // manufacture and CLEAR. The buffer will contain the last value provided
22 // to the Core library.
23 LIB_EXPORT void _plat__GetPlatformManufactureData(uint8_t* pPlatformPersistentData,
24                                                    uint32_t bufferSize)
25 {
26     if(bufferSize != 0)
27     {
28         memset((void*)pPlatformPersistentData, 0xFF, bufferSize);
29     }
30 }

```

C.1.11. /Platform/src/LocalityPlat.c

```

1  /** Includes
2  #include "Platform.h"
3
4  /** Functions
5
6  /** _plat__LocalityGet()
7  // Get the most recent command locality in locality value form.

```

```

8  // This is an integer value for locality and not a locality structure
9  // The locality can be 0-4 or 32-255. 5-31 is not allowed.
10 LIB_EXPORT unsigned char _plat__LocalityGet(void)
11 {
12     return s_locality;
13 }
14
15 /***_ plat__LocalitySet()
16 // Set the most recent command locality in locality value form
17 LIB_EXPORT void _plat__LocalitySet(unsigned char locality)
18 {
19     if(locality > 4 && locality < 32)
20         locality = 0;
21     s_locality = locality;
22     return;
23 }

```

C.1.12. /Platform/src/NVMem.c

```

1  /***_ Description
2  //
3  // This file contains the NV read and write access methods. This implementation
4  // uses RAM/file and does not manage the RAM/file as NV blocks.
5  // The implementation may become more sophisticated over time.
6  //
7
8  /***_ Includes and Local
9  #include <memory.h>
10 #include <string.h>
11 #include <assert.h>
12 #include "Platform.h"
13 #if FILE_BACKED_NV
14 # include <stdio.h>
15 static FILE* s_NvFile = NULL;
16 static int s_NeedsManufacture = FALSE;
17 #endif
18
19 /***_Functions
20
21 #if FILE_BACKED_NV
22 const char* s_NvFilePath = "NVChip";
23
24 /***_ NvFileOpen()
25 // This function opens the file used to hold the NV image.
26 // Return Type: int
27 // >= 0 success
28 // -1 error
29 static int NvFileOpen(const char* mode)
30 {
31     // Try to open an exist NVChip file for read/write
32     # if defined _MSC_VER && 1
33     if(fopen_s(&s_NvFile, s_NvFilePath, mode) != 0)
34     {
35         s_NvFile = NULL;
36     }
37     # else
38     s_NvFile = fopen(s_NvFilePath, mode);
39     # endif
40     return (s_NvFile == NULL) ? -1 : 0;
41 }
42
43 /***_ NvFileCommit()
44 // Write all of the contents of the NV image to a file.
45 // Return Type: int
46 // TRUE(1) success

```

```

47 //      FALSE(0)      failure
48 static int NvFileCommit(void)
49 {
50     int OK;
51     // If NV file is not available, return failure
52     if(s_NvFile == NULL)
53         return 1;
54     // Write RAM data to NV
55     fseek(s_NvFile, 0, SEEK_SET);
56     OK = (NV_MEMORY_SIZE == fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NvFile));
57     OK = OK && (0 == fflush(s_NvFile));
58     assert(OK);
59     return OK;
60 }
61
62 /*** NvFileSize()
63 // This function gets the size of the NV file and puts the file pointer where desired
64 // using the seek method values. SEEK SET => beginning; SEEK_CUR => current position
65 // and SEEK_END => to the end of the file.
66 static long NvFileSize(int leaveAt)
67 {
68     long fileSize;
69     long filePos = ftell(s_NvFile);
70     //
71     assert(NULL != s_NvFile);
72
73     int fseek_result = fseek(s_NvFile, 0, SEEK_END);
74     NOT_REFERENCED(fseek_result); // Fix compiler warning for NDEBUB
75     assert(fseek_result == 0);
76     fileSize = ftell(s_NvFile);
77     assert(fileSize >= 0);
78     switch(leaveAt)
79     {
80         case SEEK_SET:
81             filePos = 0;
82         case SEEK_CUR:
83             fseek(s_NvFile, filePos, SEEK_SET);
84             break;
85         case SEEK_END:
86             break;
87         default:
88             assert(FALSE);
89             break;
90     }
91     return fileSize;
92 }
93 #endif
94
95 /*** _plat_NvErrors()
96 // This function is used by the simulator to set the error flags in the NV
97 // subsystem to simulate an error in the NV loading process
98 LIB_EXPORT void _plat_NvErrors(int recoverable, int unrecoverable)
99 {
100     s_NV_unrecoverable = unrecoverable;
101     s_NV_recoverable   = recoverable;
102 }
103
104 /*** _plat_NVEnable()
105 // Enable NV memory.
106 //
107 // This version just pulls in data from a file. In a real TPM, with NV on chip,
108 // this function would verify the integrity of the saved context. If the NV
109 // memory was not on chip but was in something like RPMB, the NV state would be
110 // read in, decrypted and integrity checked.
111 //
112 // The recovery from an integrity failure depends on where the error occurred. It

```

```

113 // it was in the state that is discarded by TPM Reset, then the error is
114 // recoverable if the TPM is reset. Otherwise, the TPM must go into failure mode.
115 // Return Type: int
116 //      0          if success
117 //      > 0        if receive recoverable error
118 //      <0         if unrecoverable error
119 #define NV_ENABLE_SUCCESS 0
120 #define NV_ENABLE_FAILED (-1)
121 LIB_EXPORT int _plat_NVEnable(
122     void* platParameter, // platform specific parameter
123     size_t paramSize     // size of parameter. If size == 0, then
124                          // parameter is a sizeof(void*) scalar and should
125                          // be cast to an integer (intptr_t), not dereferenced.
126 )
127 {
128     NOT_REFERENCED(platParameter); // to keep compiler quiet
129     NOT_REFERENCED(paramSize);     // to keep compiler quiet
130
131     // Start assuming everything is OK
132     s_NV_unrecoverable = FALSE;
133     s_NV_recoverable   = FALSE;
134 #if FILE_BACKED_NV
135     if(s_NvFile != NULL)
136         return NV_ENABLE_SUCCESS;
137     // Initialize all the bytes in the ram copy of the NV
138     _plat_NvMemoryClear(0, NV_MEMORY_SIZE);
139
140     // If the file exists
141     if(NvFileOpen("r+b") >= 0)
142     {
143         long fileSize = NvFileSize(SEEK_SET); // get the file size and leave the
144                                              // file pointer at the start
145                                              //
146         // If the size is right, read the data
147         if(NV_MEMORY_SIZE == fileSize)
148         {
149             s_NeedsManufacture = fread(s_NV, 1, NV_MEMORY_SIZE, s_NvFile)
150                               != NV_MEMORY_SIZE;
151         }
152         else
153         {
154             NvFileCommit(); // for any other size, initialize it
155             s_NeedsManufacture = TRUE;
156         }
157     }
158     // If NVChip file does not exist, try to create it for read/write.
159     else if(NvFileOpen("w+b") >= 0)
160     {
161         NvFileCommit(); // Initialize the file
162         s_NeedsManufacture = TRUE;
163     }
164     assert(NULL != s_NvFile); // Just in case we are broken for some reason.
165 #endif
166     // NV contents have been initialized and the error checks have been performed. For
167     // simulation purposes, use the signaling interface to indicate if an error is
168     // to be simulated and the type of the error.
169     if(s_NV_unrecoverable)
170         return NV_ENABLE_FAILED;
171     s_NvIsAvailable = TRUE;
172     return s_NV_recoverable;
173 }
174
175 /***_plat_NVDisable()
176 // Disable NV memory
177 LIB_EXPORT void _plat_NVDisable(
178     void* platParameter, // platform specific parameter

```

```

179     size_t paramSize        // size of parameter. If size == 0, then
180                             // parameter is a sizeof(void*) scalar and should
181                             // be cast to an integer (intptr_t), not dereferenced.
182 )
183 {
184     NOT_REFERENCED(paramSize); // to keep compiler quiet
185     int delete =
186         (intptr_t)platParameter; // IN: If TRUE (!=0), delete the NV contents.
187
188 #if FILE_BACKED_NV
189     if(NULL != s_NvFile)
190     {
191         fclose(s_NvFile); // Close NV file
192         // Alternative to deleting the file is to set its size to 0. This will not
193         // match the NV size so the TPM will need to be remanufactured.
194         if(delete)
195         {
196             // Open for writing at the start. Sets the size to zero.
197             if(NvFileOpen("w") >= 0)
198             {
199                 fflush(s_NvFile);
200                 fclose(s_NvFile);
201             }
202         }
203     }
204     s_NvFile = NULL; // Set file handle to NULL
205 #endif
206     s_NvIsAvailable = FALSE;
207     return;
208 }
209
210 /***_plat_GetNvReadyState()
211 // Check if NV is available
212 // Return Type: int
213 //      0      NV is available
214 //      1      NV is not available due to write failure
215 //      2      NV is not available due to rate limit
216 LIB_EXPORT int _plat_GetNvReadyState(void)
217 {
218     int retVal = NV_READY;
219     if(!s_NvIsAvailable)
220         retVal = NV_WRITEFAILURE;
221 #if FILE_BACKED_NV
222     else
223         retVal = (s_NvFile == NULL);
224 #endif
225     return retVal;
226 }
227
228 /***_plat_NvMemoryRead()
229 // Function: Read a chunk of NV memory
230 // Return Type: int
231 //      TRUE(1)      offset and size is within available NV size
232 //      FALSE(0)     otherwise; also trigger failure mode
233 LIB_EXPORT int _plat_NvMemoryRead(unsigned int startOffset, // IN: read start
234                                   unsigned int size,         // IN: size of bytes to read
235                                   void* data                // OUT: data buffer
236 )
237 {
238     assert(startOffset + size <= NV_MEMORY_SIZE);
239     if(startOffset + size <= NV_MEMORY_SIZE)
240     {
241         memcpy(data, &s_NV[startOffset], size); // Copy data from RAM
242         return TRUE;
243     }
244     return FALSE;

```

```

245 }
246
247 /***_plat_NvGetChangedStatus()
248 // This function checks to see if the NV is different from the test value. This is
249 // so that NV will not be written if it has not changed.
250 // Return Type: int
251 //     NV_HAS_CHANGED(1)      the NV location is different from the test value
252 //     NV_IS_SAME(0)         the NV location is the same as the test value
253 //     NV_INVALID_LOCATION(-1) the NV location is invalid; also triggers failure mode
254 LIB_EXPORT int _plat_NvGetChangedStatus(
255     unsigned int startOffset, // IN: read start
256     unsigned int size,        // IN: size of bytes to read
257     void* data                // IN: data buffer
258 )
259 {
260     assert(startOffset + size <= NV_MEMORY_SIZE);
261     if(startOffset + size <= NV_MEMORY_SIZE)
262     {
263         return (memcmp(&s_NV[startOffset], data, size) != 0);
264     }
265     // the NV location is invalid; the assert above should have triggered failure
266     // mode
267     return NV_INVALID_LOCATION;
268 }
269
270 /***_plat_NvMemoryWrite()
271 // This function is used to update NV memory. The "write" is to a memory copy of
272 // NV. At the end of the current command, any changes are written to
273 // the actual NV memory.
274 // NOTE: A useful optimization would be for this code to compare the current
275 // contents of NV with the local copy and note the blocks that have changed. Then
276 // only write those blocks when _plat_NvCommit() is called.
277 // Return Type: int
278 //     TRUE(1)          offset and size is within available NV size
279 //     FALSE(0)         otherwise; also trigger failure mode
280 LIB_EXPORT int _plat_NvMemoryWrite(unsigned int startOffset, // IN: write start
281     unsigned int size, // IN: size of bytes to write
282     void* data // OUT: data buffer
283 )
284 {
285     assert(startOffset + size <= NV_MEMORY_SIZE);
286     if(startOffset + size <= NV_MEMORY_SIZE)
287     {
288         memcpy(&s_NV[startOffset], data, size); // Copy the data to the NV image
289         return TRUE;
290     }
291     return FALSE;
292 }
293
294 /***_plat_NvMemoryClear()
295 // Function is used to set a range of NV memory bytes to an implementation-dependent
296 // value. The value represents the erase state of the memory.
297 LIB_EXPORT int _plat_NvMemoryClear(unsigned int startOffset, // IN: clear start
298     unsigned int size // IN: number of bytes to clear
299 )
300 {
301     assert(startOffset + size <= NV_MEMORY_SIZE);
302     if(startOffset + size <= NV_MEMORY_SIZE)
303     {
304         // In this implementation, assume that the erase value for NV is all 1s
305         memset(&s_NV[startOffset], 0xff, size);
306         return TRUE;
307     }
308     return FALSE;
309 }
310

```



```

311  /***_plat__NvMemoryMove()
312  // Function: Move a chunk of NV memory from source to destination
313  //      This function should ensure that if there overlap, the original data is
314  //      copied before it is written
315  LIB_EXPORT int _plat__NvMemoryMove(unsigned int sourceOffset, // IN: source offset
316                                     unsigned int destOffset, // IN: destination offset
317                                     unsigned int size // IN: size of data being moved
318 )
319 {
320     assert(sourceOffset + size <= NV_MEMORY_SIZE);
321     assert(destOffset + size <= NV_MEMORY_SIZE);
322     if(sourceOffset + size <= NV_MEMORY_SIZE && destOffset + size <= NV_MEMORY_SIZE)
323     {
324         memmove(&s_NV[destOffset], &s_NV[sourceOffset], size); // Move data in RAM
325         return TRUE;
326     }
327     return FALSE;
328 }
329
330 /***_plat__NvCommit()
331 // This function writes the local copy of NV to NV for permanent store. It will write
332 // NV_MEMORY_SIZE bytes to NV. If a file is use, the entire file is written.
333 // Return Type: int
334 // 0          NV write success
335 // non-0      NV write fail
336 LIB_EXPORT int _plat__NvCommit(void)
337 {
338     #if FILE_BACKED_NV
339         return (NvFileCommit() ? 0 : 1);
340     #else
341         return 0;
342     #endif
343 }
344
345 /***_plat__TearDown
346 // notify platform that TPM_TearDown was called so platform can cleanup or
347 // zeroize anything in the Platform. This should zeroize NV as well.
348 LIB_EXPORT void _plat__TearDown()
349 {
350     #if FILE_BACKED_NV
351         // remove(s_NvFilePath);
352     #endif
353 }
354
355 /***_plat__SetNvAvail()
356 // Set the current NV state to available. This function is for testing purpose
357 // only. It is not part of the platform NV logic
358 LIB_EXPORT void _plat__SetNvAvail(void)
359 {
360     s_NvIsAvailable = TRUE;
361     return;
362 }
363
364 /***_plat__ClearNvAvail()
365 // Set the current NV state to unavailable. This function is for testing purpose
366 // only. It is not part of the platform NV logic
367 LIB_EXPORT void _plat__ClearNvAvail(void)
368 {
369     s_NvIsAvailable = FALSE;
370     return;
371 }
372
373 /***_plat__NVNeedsManufacture()
374 // This function is used by the simulator to determine when the TPM's NV state
375 // needs to be manufactured.
376 LIB_EXPORT int _plat__NVNeedsManufacture(void)

```

```

377 {
378 #if FILE_BACKED_NV
379     return s_NeedsManufacture;
380 #else
381     return FALSE;
382 #endif
383 }

```

C.1.13. /Platform/src/PlatformACT.c

```

1  /** Includes
2  #include "Platform.h"
3
4  /** Functions
5
6  #if ACT_SUPPORT
7
8  /** ActSignal()
9  // Function called when there is an ACT event to signal or unsignal
10 static void ActSignal(P_ACT_DATA actData, int on)
11 {
12     if(actData == NULL)
13         return;
14     // If this is to turn a signal on, don't do anything if it is already on. If this
15     // is to turn the signal off, do it anyway because this might be for
16     // initialization.
17     if(on && (actData->signaled == TRUE))
18         return;
19     actData->signaled = (uint8_t)on;
20
21     // If there is an action, then replace the "Do something" with the correct action.
22     // It should test 'on' to see if it is turning the signal on or off.
23     switch(actData->number)
24     {
25 # if RH_ACT_0
26         case 0: // Do something
27             return;
28 # endif
29 # if RH_ACT_1
30         case 1: // Do something
31             return;
32 # endif
33 # if RH_ACT_2
34         case 2: // Do something
35             return;
36 # endif
37 # if RH_ACT_3
38         case 3: // Do something
39             return;
40 # endif
41 # if RH_ACT_4
42         case 4: // Do something
43             return;
44 # endif
45 # if RH_ACT_5
46         case 5: // Do something
47             return;
48 # endif
49 # if RH_ACT_6
50         case 6: // Do something
51             return;
52 # endif
53 # if RH_ACT_7
54         case 7: // Do something
55             return;

```

```

56 # endif
57 # if RH_ACT_8
58     case 8: // Do something
59         return;
60 # endif
61 # if RH_ACT_9
62     case 9: // Do something
63         return;
64 # endif
65 # if RH_ACT_A
66     case 0xA: // Do something
67         return;
68 # endif
69 # if RH_ACT_B
70     case 0xB:
71         // Do something
72         return;
73 # endif
74 # if RH_ACT_C
75     case 0xC: // Do something
76         return;
77 # endif
78 # if RH_ACT_D
79     case 0xD: // Do something
80         return;
81 # endif
82 # if RH_ACT_E
83     case 0xE: // Do something
84         return;
85 # endif
86 # if RH_ACT_F
87     case 0xF: // Do something
88         return;
89 # endif
90     default:
91         return;
92 }
93 }
94
95 /*** ActGetDataPointer()
96 static P_ACT_DATA ActGetDataPointer(uint32_t act)
97 {
98
99 # define RETURN_ACT_POINTER(N) \
100     if(0x##N == act) \
101         return &ACT_##N;
102
103     FOR_EACH_ACT(RETURN_ACT_POINTER)
104
105     return (P_ACT_DATA)NULL;
106 }
107
108 /*** _plat__ACT_GetImplemented()
109 // This function tests to see if an ACT is implemented. It is a belt and suspenders
110 // function because the TPM should not be calling to manipulate an ACT that is not
111 // implemented. However, this could help the simulator code which doesn't necessarily
112 // know if an ACT is implemented or not.
113 LIB_EXPORT int _plat__ACT_GetImplemented(uint32_t act)
114 {
115     return (ActGetDataPointer(act) != NULL);
116 }
117
118 /*** _plat__ACT_GetRemaining()
119 // This function returns the remaining time. If an update is pending, 'newValue' is
120 // returned. Otherwise, the current counter value is returned. Note that since the
121 // timers keep running, the returned value can get stale immediately. The actual count

```

```

122 // value will be no greater than the returned value.
123 LIB_EXPORT uint32_t _plat__ACT_GetRemaining(uint32_t act //IN: the ACT selector
124 )
125 {
126     P_ACT_DATA actData = ActGetDataPointer(act);
127     uint32_t remain;
128     //
129     if(actData == NULL)
130         return 0;
131     remain = actData->remaining;
132     if(actData->pending)
133         remain = actData->newValue;
134     return remain;
135 }
136
137 /***_plat__ACT_GetSignaled()
138 LIB_EXPORT int _plat__ACT_GetSignaled(uint32_t act //IN: number of ACT to check
139 )
140 {
141     P_ACT_DATA actData = ActGetDataPointer(act);
142     //
143     if(actData == NULL)
144         return 0;
145     return (int)actData->signaled;
146 }
147
148 /***_plat__ACT_SetSignaled()
149 LIB_EXPORT void _plat__ACT_SetSignaled(uint32_t act, int on)
150 {
151     ActSignal(ActGetDataPointer(act), on);
152 }
153
154 /***_plat__ACT_GetPending()
155 LIB_EXPORT int _plat__ACT_GetPending(uint32_t act //IN: number of ACT to check
156 )
157 {
158     P_ACT_DATA actData = ActGetDataPointer(act);
159     //
160     if(actData == NULL)
161         return 0;
162     return (int)actData->pending;
163 }
164
165 /***_plat__ACT_UpdateCounter()
166 // This function is used to write the newValue for the counter. If an update is
167 // pending, then no update occurs and the function returns FALSE. If 'setSignaled'
168 // is TRUE, then the ACT signaled state is SET and if 'newValue' is 0, nothing
169 // is posted.
170 LIB_EXPORT int _plat__ACT_UpdateCounter(uint32_t act, // IN: ACT to update
171                                         uint32_t newValue // IN: the value to post
172 )
173 {
174     P_ACT_DATA actData = ActGetDataPointer(act);
175     //
176     if(actData == NULL)
177         // actData doesn't exist but pretend update is pending rather than indicate
178         // that a retry is necessary.
179         return TRUE;
180     // if an update is pending then return FALSE so that there will be a retry
181     if(actData->pending != 0)
182         return FALSE;
183     actData->newValue = newValue;
184     actData->pending = TRUE;
185
186     return TRUE;
187 }

```

```

188
189 /**_plat_ACT_EnableTicks()
190 // This enables and disables the processing of the once-per-second ticks. This should
191 // be turned off ('enable' = FALSE) by TPM_Init and turned on ('enable' = TRUE) by
192 // TPM2_Startup() after all the initializations have completed.
193 LIB_EXPORT void _plat_ACT_EnableTicks(int enable)
194 {
195     actTicksAllowed = enable;
196 }
197
198 /** ActDecrement()
199 // If 'newValue' is non-zero it is copied to 'remaining' and then 'newValue' is
200 // set to zero. Then 'remaining' is decremented by one if it is not already zero. If
201 // the value is decremented to zero, then the associated event is signaled. If setting
202 // 'remaining' causes it to be greater than 1, then the signal associated with the ACT
203 // is turned off.
204 static void ActDecrement(P_ACT_DATA actData)
205 {
206     // Check to see if there is an update pending
207     if(actData->pending)
208     {
209         // If this update will cause the count to go from non-zero to zero, set
210         // the newValue to 1 so that it will timeout when decremented below.
211         if((actData->newValue == 0) && (actData->remaining != 0))
212             actData->newValue = 1;
213         actData->remaining = actData->newValue;
214
215         // Update processed
216         actData->pending = 0;
217     }
218     // no update so countdown if the count is non-zero but not max
219     if((actData->remaining != 0) && (actData->remaining != UINT32_MAX))
220     {
221         // If this countdown causes the count to go to zero, then turn the signal for
222         // the ACT on.
223         if((actData->remaining -= 1) == 0)
224             ActSignal(actData, TRUE);
225     }
226     // If the current value of the counter is non-zero, then the signal should be
227     // off.
228     if(actData->signaled && (actData->remaining > 0))
229         ActSignal(actData, FALSE);
230 }
231
232 /** _plat_ACT_Tick()
233 // This processes the once-per-second clock tick from the hardware. This is set up
234 // for the simulator to use the control interface to send ticks to the TPM. These
235 // ticks do not have to be on a per second basis. They can be as slow or as fast as
236 // desired so that the simulation can be tested.
237 LIB_EXPORT void _plat_ACT_Tick(void)
238 {
239     // Ticks processing is turned off at certain times just to make sure that nothing
240     // strange is happening before pointers and things are
241     if(actTicksAllowed)
242     {
243         // Handle the update for each counter.
244         # define DECREMENT_COUNT(N) ActDecrement(&ACT_##N);
245
246         FOR_EACH_ACT(DECREMENT_COUNT)
247     }
248 }
249
250 /** ActZero()
251 // This function initializes a single ACT
252 static void ActZero(uint32_t act, P_ACT_DATA actData)
253 {

```

```

254     actData->remaining = 0;
255     actData->newValue  = 0;
256     actData->pending   = 0;
257     actData->number    = (uint8_t)act;
258     ActSignal(actData, FALSE);
259 }
260
261 /** plat_ACT_Initialize()
262  * This function initializes the ACT hardware and data structures
263  */
264 LIB_EXPORT int _plat_ACT_Initialize(void)
265 {
266     actTicksAllowed = 0;
267     # define ZERO_ACT(N) ActZero(0x##N, &ACT_##N);
268     FOR_EACH_ACT(ZERO_ACT)
269
270     return TRUE;
271 }
272 #endif // ACT_SUPPORT

```

C.1.14. /Platform/src/PlatformData.c

```

1  /** Description
2  * This file will instance the TPM variables that are not stack allocated. The
3  * descriptions for these variables are in Global.h for this project.
4
5  /** Includes
6  #define _PLATFORM_DATA_C_
7  #include "Platform.h"

```

C.1.15. /Platform/src/PlatformPcr.c

```

1  // PCR platform interface functions
2  #include "Platform.h"
3  #include <public/TpmAlgorithmDefines.h>
4
5  // use this as a convenient lookup for hash size for PCRs.
6  UINT16 CryptHashGetDigestSize(TPM_ALG_ID hashAlg // IN: hash algorithm to look up
7  );
8  void MemorySet(void* dest, int value, size_t size);
9
10 // The initial value of PCR attributes. The value of these fields should be
11 // consistent with PC Client specification. The bitfield meanings are defined by
12 // the TPM Reference code.
13 // In this implementation, we assume the total number of implemented PCR is 24.
14 static const PCR_Attributes s_initAttributes[] = {
15     //
16     // PCR 0 - 15, static RTM
17     // PCR[0]
18     {
19         1, // save state
20         0, // in the "do not increment the PcrCounter" group? (0 = increment the
21         // PcrCounter)
22         0, // supportsPolicyAuth group number? 0 = policyAuth not supported for this
23         // PCR.
24         0, // supportsAuthValue group number? 0 = AuthValue not supported for this
25         // PCR.
26         0, // 0 = reset localities (cannot reset)
27         0x1F // 0x1F = extendlocalities [0,4]
28     },
29     {1, 0, 0, 0, 0, 0, 0x1F}, // PCR 1-3
30     {1, 0, 0, 0, 0, 0, 0x1F},
31     {1, 0, 0, 0, 0, 0, 0x1F},
32     {1, 0, 0, 0, 0, 0, 0x1F}, // PCR 4-6

```



```

30     {1, 0, 0, 0, 0, 0x1F},
31     {1, 0, 0, 0, 0, 0x1F},
32     {1, 0, 0, 0, 0, 0x1F}, // PCR 7-9
33     {1, 0, 0, 0, 0, 0x1F},
34     {1, 0, 0, 0, 0, 0x1F},
35     {1, 0, 0, 0, 0, 0x1F}, // PCR 10-12
36     {1, 0, 0, 0, 0, 0x1F},
37     {1, 0, 0, 0, 0, 0x1F},
38     {1, 0, 0, 0, 0, 0x1F}, // PCR 13-15
39     {1, 0, 0, 0, 0, 0x1F},
40     {1, 0, 0, 0, 0, 0x1F},
41
42     // these PCRs are never saved
43     {0, 0, 0, 0, 0x0F, 0x1F}, // PCR 16, Debug, reset allowed, extend all
44     {0, 0, 0, 0, 0x10, 0x1C}, // PCR 17, Locality 4, extend loc 2+
45     {0, 0, 0, 0, 0x10, 0x1C}, // PCR 18, Locality 3, extend loc 2+
46     {0, 0, 0, 0, 0x10, 0x0C}, // PCR 19, Locality 2, extend loc 2, 3
47     // these three support doNotIncrement, PolicyAuth, and AuthValue.
48     // this is consistent with the existing behavior of the TPM Reference code
49     // but differs from the behavior of the PC client spec.
50     {0, 1, 1, 1, 0x14, 0x0E}, // PCR 20, Locality 1, extend loc 1, 2, 3
51     {0, 1, 1, 1, 0x14, 0x04}, // PCR 21, Dynamic OS, extend loc 2
52     {0, 1, 1, 1, 0x14, 0x04}, // PCR 22, Dynamic OS, extend loc 2
53     {0, 0, 0, 0, 0x0F, 0x1F}, // PCR 23, reset allowed, App specific, extend all
54 };
55
56 #ifndef ARRAYSIZE
57 # define ARRAYSIZE(a) (sizeof(a) / sizeof(a[0]))
58 #endif
59
60 MUST_BE(ARRAYSIZE(s_initAttributes) == IMPLEMENTATION_PCR);
61
62 #if ALG_SHA256 != YES && ALG_SHA384 != YES
63 # error No default PCR banks defined
64 #endif
65
66 static const TPM_ALG_ID DefaultActivePcrBanks[] = {
67 #if ALG_SHA256
68     TPM_ALG_SHA256
69 #endif
70 #if ALG_SHA384
71 # if ALG_SHA256
72     ,
73 # endif
74     TPM_ALG_SHA384
75 #endif
76 };
77
78 UINT32 _platPcr__NumberOfPcrs()
79 {
80     return ARRAYSIZE(s_initAttributes);
81 }
82
83 // return the initialization attributes of a given PCR.
84 // pcrNumber expected to be in [0, _platPcr__NumberOfPcrs)
85 // returns the attributes for PCR[0] if the requested pcrNumber is out of range.
86 PCR_Attributes _platPcr__GetPcrInitializationAttributes(UINT32 pcrNumber)
87 {
88     if(pcrNumber >= _platPcr__NumberOfPcrs())
89     {
90         pcrNumber = 0;
91     }
92     return s_initAttributes[pcrNumber];
93 }
94
95 // should the given PCR algorithm default to active in a new TPM?

```

```

96  BOOL _platPcr_IsPcrBankDefaultActive(TPM_ALG_ID pcrAlg)
97  {
98      // brute force search is fast enough for a small array.
99      for(int i = 0; i < ARRAYSIZE(DefaultActivePcrBanks); i++)
100      {
101          if(DefaultActivePcrBanks[i] == pcrAlg)
102          {
103              return TRUE;
104          }
105      }
106      return FALSE;
107  }
108
109  // Fill a given buffer with the PCR initialization value for a particular PCR and hash
110  // combination, and return its length. If the platform doesn't have a value, then
111  // the result size is expected to be zero, and the rfunction will return TPM_RC_PCR.
112  // If a valid is not available, then the core TPM library will ignore the value and
113  // treat it as non-existent and provide a default.
114  // If the buffer is not large enough for a pcr consistent with pcrAlg, then the
115  // platform will return TPM_RC_FAILURE.
116  TPM_RC _platPcr_GetInitialValueForPcr(
117      UINT32    pcrNumber,          // IN: PCR to be initialized
118      TPM_ALG_ID pcrAlg,            // IN: Algorithm of the PCR Bank being initialized
119      BYTE       startupLocality,    // IN: locality where startup is being called from
120      BYTE*      pcrData,           // OUT: buffer to put PCR initialization value into
121      uint16_t   bufferSize,        // IN: maximum size of value buffer can hold
122      uint16_t*  pcrLength // OUT: size of initialization value returned in pcrBuffer
123  )
124  {
125      // If the reset locality contains locality 4, then this
126      // indicates a DRTM PCR where the reset value is all ones,
127      // otherwise it is all zero. Don't check with equal because
128      // resetLocality is a bitfield of multiple values and does
129      // not support extended localities.
130      uint16_t pcrSize = CryptHashGetDigestSize(pcrAlg);
131      pAssert_RC(pcrNumber < _platPcr_NumberOfPcrs());
132      pAssert_RC(bufferSize >= pcrSize) pAssert_RC(pcrLength != NULL);
133
134      PCR_Attributes pcrAttributes =
135          _platPcr_GetPcrInitializationAttributes(pcrNumber);
136      BYTE defaultValue = 0;
137      // PCRs that can be cleared from locality 4 are DRTM and initialize to all 0xFF
138      if((pcrAttributes.resetLocality & 0x10) != 0)
139      {
140          defaultValue = 0xFF;
141      }
142      MemorySet(pcrData, defaultValue, pcrSize);
143      if(pcrNumber == HCRTM_PCR)
144      {
145          pcrData[pcrSize - 1] = startupLocality;
146      }
147
148      // platform could provide a value here if the platform has initialization rules
149      // different from the original PC Client spec (the default used by the Core
150      // library).
151      *pcrLength = pcrSize;
152      return TPM_RC_SUCCESS;
153  }

```

C.1.16. /Platform/src/PowerPlat.c

```

1  /** Includes and Function Prototypes
2
3  #include "Platform.h"
4

```

```

5  /*** Functions
6
7  /***_plat_Signal_PowerOn()
8  // Signal platform power on
9  LIB_EXPORT int _plat_Signal_PowerOn(void)
10 {
11     // Reset the timer
12     _plat_TimerReset();
13
14     // Need to indicate that we lost power
15     s_powerLost = TRUE;
16
17     return 0;
18 }
19
20 /***_plat_WasPowerLost()
21 // Test whether power was lost before a _TPM_Init.
22 //
23 // This function will clear the "hardware" indication of power loss before return.
24 // This means that there can only be one spot in the TPM code where this value
25 // gets read. This method is used here as it is the most difficult to manage in the
26 // TPM code and, if the hardware actually works this way, it is hard to make it
27 // look like anything else. So, the burden is placed on the TPM code rather than the
28 // platform code
29 // Return Type: int
30 //     TRUE(1)         power was lost
31 //     FALSE(0)        power was not lost
32 LIB_EXPORT int _plat_WasPowerLost(void)
33 {
34     int retVal = s_powerLost;
35     s_powerLost = FALSE;
36     return retVal;
37 }
38
39 /***_plat_Signal_Reset()
40 // This a TPM reset without a power loss.
41 LIB_EXPORT int _plat_Signal_Reset(void)
42 {
43     // Initialize locality
44     s_locality = 0;
45
46     // Command cancel
47     s_isCanceled = FALSE;
48
49     _TPM_Init();
50
51     // if we are doing reset but did not have a power failure, then we should
52     // not need to reload NV ...
53
54     return 0;
55 }
56
57 /***_plat_Signal_PowerOff()
58 // Signal platform power off
59 LIB_EXPORT void _plat_Signal_PowerOff(void)
60 {
61     // Prepare NV memory for power off
62     _plat_NVDisable((void*)FALSE, 0);
63
64     #if ACT_SUPPORT
65         // Disable tick ACT tick processing
66         _plat_ACT_EnableTicks(FALSE);
67     #endif
68
69     return;
70 }

```

C.1.17. /Platform/src/PPPlat.c

```

1  /** Description
2
3  //   This module simulates the physical presence interface pins on the TPM.
4
5  /** Includes
6  #include "Platform.h"
7
8  /** Functions
9
10 /***_plat_PhysicalPresenceAsserted()
11 // Check if physical presence is signaled
12 // Return Type: int
13 //   TRUE(1)           if physical presence is signaled
14 //   FALSE(0)          if physical presence is not signaled
15 LIB_EXPORT int _plat_PhysicalPresenceAsserted(void)
16 {
17     // Do not know how to check physical presence without real hardware.
18     // so always return TRUE;
19     return s_physicalPresence;
20 }
21
22 /***_plat_Signal_PhysicalPresenceOn()
23 // Signal physical presence on
24 LIB_EXPORT void _plat_Signal_PhysicalPresenceOn(void)
25 {
26     s_physicalPresence = TRUE;
27     return;
28 }
29
30 /***_plat_Signal_PhysicalPresenceOff()
31 // Signal physical presence off
32 LIB_EXPORT void _plat_Signal_PhysicalPresenceOff(void)
33 {
34     s_physicalPresence = FALSE;
35     return;
36 }

```

C.1.18. /Platform/src/RunCommand.c

```

1  /**Introduction
2  // This module provides the platform specific entry and fail processing. The
3  // _plat_RunCommand() function is used to call to ExecuteCommand() in the TPM code.
4  // This function does whatever processing is necessary to set up the platform
5  // in anticipation of the call to the TPM including setup for error processing.
6  //
7  // The _plat_Fail() function is called when there is a failure in the TPM. The TPM
8  // code will have set the flag to indicate that the TPM is in failure mode.
9  // This call will then recursively call ExecuteCommand in order to build the
10 // failure mode response. When ExecuteCommand() returns to _plat_Fail(), the
11 // platform will do some platform specif operation to return to the environment in
12 // which the TPM is executing. For a simulator, setjmp/longjmp is used. For an OS,
13 // a system exit to the OS would be appropriate.
14
15 /** Includes and locals
16 #include "Platform.h"
17 #include <assert.h>
18 #include <setjmp.h>
19 #include <stdio.h>
20
21 jmp_buf s_jumpBuffer;
22
23 // The following extern globals are copied here from Global.h to avoid including all
  of Tpm.h here.

```

```

24 // TODO: Improve the interface by which these values are shared.
25 extern BOOL g_inFailureMode; // Indicates that the TPM is in failure mode
26 #if ALLOW_FORCE_FAILURE_MODE
27 extern BOOL g_forceFailureMode; // flag to force failure mode during test
28 #endif
29 #if FAIL_TRACE
30 // The name of the function that triggered failure mode.
31 extern const char* s_failFunctionName;
32 #endif // FAIL_TRACE
33 extern UINT32 s_failFunction;
34 extern UINT32 s_failLine;
35 extern UINT32 s_failCode;
36
37 /** Functions
38
39 ****_plat_RunCommand()
40 // This version of RunCommand will set up a jump_buf and call ExecuteCommand(). If
41 // the command executes without failing, it will return and RunCommand will return.
42 // If there is a failure in the command, then _plat_Fail() is called and it will
43 // longjump back to RunCommand which will call ExecuteCommand again. However, this
44 // time, the TPM will be in failure mode so ExecuteCommand will simply build
45 // a failure response and return.
46 LIB_EXPORT void _plat_RunCommand(
47     uint32_t requestSize, // IN: command buffer size
48     unsigned char* request, // IN: command buffer
49     uint32_t* responseSize, // IN/OUT: response buffer size
50     unsigned char** response // IN/OUT: response buffer
51 )
52 {
53     setjmp(s_jumpBuffer);
54     ExecuteCommand(requestSize, request, responseSize, response);
55 }
56
57 ****_plat_Fail()
58 // This is the platform depended failure exit for the TPM.
59 LIB_EXPORT NORETURN void _plat_Fail(void)
60 {
61
62 #if ALLOW_FORCE_FAILURE_MODE
63     // The simulator asserts during unexpected (i.e., un-forced) failure modes.
64     if(!g_forceFailureMode)
65     {
66         fprintf(stderr, "Unexpected failure mode (code %d) in ", s_failCode);
67 # if FAIL_TRACE
68         fprintf(stderr, "function '%s' (line %d)\n", s_failFunctionName, s_failLine);
69 # else // FAIL_TRACE
70         fprintf(stderr, "location code 0x%0x\n", s_locationCode);
71 # endif // FAIL_TRACE
72         assert(FALSE);
73     }
74
75     // Clear the forced-failure mode flag for next time.
76     g_forceFailureMode = FALSE;
77 #endif // ALLOW_FORCE_FAILURE_MODE
78
79     longjmp(&s_jumpBuffer[0], 1);
80 }

```

C.1.19. /Platform/src/Unique.c

```

1 /** Introduction
2 // In some implementations of the TPM, the hardware can provide a secret
3 // value to the TPM. This secret value is statistically unique to the
4 // instance of the TPM. Typical uses of this value are to provide
5 // personalization to the random number generation and as a shared secret

```

```

6  // between the TPM and the manufacturer.
7
8  /** Includes
9  #include "Platform.h"
10
11 #if VENDOR_PERMANENT_AUTH_ENABLED == YES
12
13 const char notReallyUnique[] = "This is not really a unique value. A real "
14                               "unique value should"
15                               " be generated by the platform.";
16
17 /** _plat_GetUnique()
18 // This function is used to access the platform-specific vendor unique values.
19 // This function places the unique value in the provided buffer ('b')
20 // and returns the number of bytes transferred. The function will not
21 // copy more data than 'bSize'.
22 // NOTE: If a platform unique value has unequal distribution of uniqueness
23 // and 'bSize' is smaller than the size of the unique value, the 'bSize'
24 // portion with the most uniqueness should be returned.
25 //
26 // 'which' indicates the unique value to return:
27 // 0 = RESERVED, do not use
28 // 1 = the VENDOR_PERMANENT_AUTH_HANDLE authorization value for this device
29 LIB_EXPORT uint32_t _plat_GetUnique(uint32_t which, // which vendor value to return?
30                                    uint32_t bSize, // size of the buffer
31                                    unsigned char* b // output buffer
32 )
33 {
34     const char* from = notReallyUnique;
35     uint32_t retVal = 0;
36
37     if(which == 1)
38     {
39         const size_t uSize =
40             sizeof(notReallyUnique) <= bSize ? sizeof(notReallyUnique) : bSize;
41         MemoryCopy(b, notReallyUnique, uSize);
42     }
43     // else fall through to default 0
44
45     return retVal;
46 }
47
48 #endif

```

C.1.20. /Platform/src/VendorInfo.c

```

1  /** Introduction
2  // Provide vendor-specific version and identifiers to core TPM library for
3  // return in capabilities. These may not be compile time constants and therefore
4  // are provided by platform callbacks. These platform functions are expected to
5  // always be available, even in failure mode.
6  //
7  /** Includes
8  #include "Platform.h"
9
10 // In this sample platform, these are compile time constants, but are not required to
11 // be.
12 #define MANUFACTURER "XYZ "
13 #define VENDOR_STRING_1 "xCG "
14 #define VENDOR_STRING_2 "fTPM"
15 #define VENDOR_STRING_3 "\0\0\0\0"
16 #define VENDOR_STRING_4 "\0\0\0\0"
17 #define FIRMWARE_V1 (0x20170619)
18 #define FIRMWARE_V2 (0x00163636)
19 #define MAX_SVN 255

```



```

19
20 static uint32_t currentHash = FIRMWARE_V2;
21 static uint16_t currentSvn = 10;
22
23 // Similar to the Core Library's ByteArrayToUint32, but usable in Platform code.
24 static uint32_t StringToUint32(char s[4])
25 {
26     uint8_t* b = (uint8_t*)s; // Avoid promotion to a signed integer type
27     return (((uint32_t)b[0] << 8 | b[1]) << 8 | b[2]) << 8 | b[3];
28 }
29
30 // return the 4 character Manufacturer Capability code. This
31 // should come from the platform library since that is provided by the manufacturer
32 LIB_EXPORT uint32_t _plat__GetManufacturerCapabilityCode()
33 {
34     return StringToUint32(MANUFACTURER);
35 }
36
37 // return the 4 character VendorStrings for Capabilities.
38 // Index is ONE-BASED, and may be in the range [1,4] inclusive.
39 // Any other index returns all zeros. The return value will be interpreted
40 // as an array of 4 ASCII characters (with no null terminator)
41 LIB_EXPORT uint32_t _plat__GetVendorCapabilityCode(int index)
42 {
43     switch(index)
44     {
45         case 1:
46             return StringToUint32(VENDOR_STRING_1);
47         case 2:
48             return StringToUint32(VENDOR_STRING_2);
49         case 3:
50             return StringToUint32(VENDOR_STRING_3);
51         case 4:
52             return StringToUint32(VENDOR_STRING_4);
53     }
54     return 0;
55 }
56
57 // return the most-significant 32-bits of the TPM Firmware Version reported by
58 // getCapability.
59 LIB_EXPORT uint32_t _plat__GetTpmFirmwareVersionHigh()
60 {
61     return FIRMWARE_V1;
62 }
63
64 // return the least-significant 32-bits of the TPM Firmware Version reported by
65 // getCapability.
66 LIB_EXPORT uint32_t _plat__GetTpmFirmwareVersionLow()
67 {
68     return FIRMWARE_V2;
69 }
70
71 // return the TPM Firmware SVN reported by getCapability.
72 LIB_EXPORT uint16_t _plat__GetTpmFirmwareSvn(void)
73 {
74     return currentSvn;
75 }
76
77 // return the TPM Firmware maximum SVN reported by getCapability.
78 LIB_EXPORT uint16_t _plat__GetTpmFirmwareMaxSvn(void)
79 {
80     return MAX_SVN;
81 }
82
83 // Called by the simulator to set the TPM Firmware SVN reported by
84 // getCapability.

```

```

85  LIB_EXPORT void _plat__SetTpmFirmwareHash(uint32_t hash)
86  {
87      currentHash = hash;
88  }
89
90  // Called by the simulator to set the TPM Firmware SVN reported by
91  // getCapability.
92  LIB_EXPORT void _plat__SetTpmFirmwareSvn(uint16_t svn)
93  {
94      currentSvn = MIN(svn, MAX_SVN);
95  }
96
97  #if SVN_LIMITED_SUPPORT
98  // Dummy implementation for obtaining a Firmware SVN Secret bound
99  // to the given SVN.
100  LIB_EXPORT int _plat__GetTpmFirmwareSvnSecret(uint16_t svn,
101                                              uint16_t secret_buf_size,
102                                              uint8_t* secret_buf,
103                                              uint16_t* secret_size)
104  {
105      int i;
106
107      if(svn > currentSvn)
108      {
109          return -1;
110      }
111
112      // INSECURE dummy implementation: repeat the SVN into the secret buffer.
113      for(i = 0; i < secret_buf_size; ++i)
114      {
115          secret_buf[i] = ((uint8_t*)&svn)[i % sizeof(svn)];
116      }
117
118      *secret_size = secret_buf_size;
119
120      return 0;
121  }
122  #endif // SVN_LIMITED_SUPPORT
123
124  #if FW_LIMITED_SUPPORT
125  // Dummy implementation for obtaining a Firmware Secret bound
126  // to the current firmware image.
127  LIB_EXPORT int _plat__GetTpmFirmwareSecret(
128      uint16_t secret_buf_size, uint8_t* secret_buf, uint16_t* secret_size)
129  {
130      int i;
131
132      // INSECURE dummy implementation: repeat the firmware hash into the
133      // secret buffer.
134      for(i = 0; i < secret_buf_size; ++i)
135      {
136          secret_buf[i] = ((uint8_t*)&currentHash)[i % sizeof(currentHash)];
137      }
138
139      *secret_size = secret_buf_size;
140
141      return 0;
142  }
143  #endif // FW_LIMITED_SUPPORT
144
145  // return the TPM Type returned by TPM_PT_VENDOR_TPM_TYPE
146  LIB_EXPORT uint32_t _plat__GetTpmType()
147  {
148      return 1; // just the value the reference code has returned in the past.
149  }
150

```

Annex D (informative) Remote Procedure Interface

D.1 Introduction

These files provide an RPC interface for a TPM simulation.

The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in TPM 2.0 Part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as _TPM_HashStart.

D.1.1. /Simulator/include/TpmTcpProtocol.h

```

1  /** Introduction
2
3  // TPM commands are communicated as uint8_t streams on a TCP connection. The TPM
4  // command protocol is enveloped with the interface protocol described in this
5  // file. The command is indicated by a uint32 with one of the values below. Most
6  // commands take no parameters return no TPM errors. In these cases the TPM
7  // interface protocol acknowledges that command processing is completed by returning
8  // a uint32=0. The command TPM_SIGNAL_HASH_DATA takes a uint32-prepended variable
9  // length byte array and the interface protocol acknowledges command completion
10 // with a uint32=0. Most TPM commands are enveloped using the TPM_SEND_COMMAND
11 // interface command. The parameters are as indicated below. The interface layer
12 // also appends a UIN32=0 to the TPM response for regularity.
13
14 /** Typedefs and Defines
15 #ifndef TCP_TPM_PROTOCOL_H
16 #define TCP_TPM_PROTOCOL_H
17
18 /** TPM Commands.
19 // All commands acknowledge processing by returning a uint32 == 0 except where noted
20 #define TPM_SIGNAL_POWER_ON      1
21 #define TPM_SIGNAL_POWER_OFF    2
22 #define TPM_SIGNAL_PHYS_PRE_ON   3
23 #define TPM_SIGNAL_PHYS_PRE_OFF 4
24 #define TPM_SIGNAL_HASH_START   5
25 #define TPM_SIGNAL_HASH_DATA    6
26 // {uint32_t BufferSize, uint8_t[BufferSize] Buffer}
27 #define TPM_SIGNAL_HASH_END 7
28 #define TPM_SEND_COMMAND     8
29 // {uint8_t Locality, uint32_t InBufferSize, uint8_t[InBufferSize] InBuffer} ->
30 //   {uint32_t OutBufferSize, uint8_t[OutBufferSize] OutBuffer}
31
32 #define TPM_SIGNAL_CANCEL_ON      9
33 #define TPM_SIGNAL_CANCEL_OFF    10
34 #define TPM_SIGNAL_NV_ON         11
35 #define TPM_SIGNAL_NV_OFF        12
36 #define TPM_SIGNAL_KEY_CACHE_ON  13
37 #define TPM_SIGNAL_KEY_CACHE_OFF 14
38
39 #define TPM_REMOTE_HANDSHAKE      15
40 #define TPM_SET_ALTERNATIVE_RESULT 16
41
42 #define TPM_SIGNAL_RESET          17
43 #define TPM_SIGNAL_RESTART        18
44
45 #define TPM_SESSION_END          20
46 #define TPM_STOP                  21
47

```

```

48 #define TPM_GET_COMMAND_RESPONSE_SIZES 25
49
50 #define TPM_ACT_GET_SIGNED 26
51
52 #define TPM_TEST_FAILURE_MODE 30
53
54 #define TPM_SET_FW_HASH 35
55 #define TPM_SET_FW_SVN 36
56
57 /** Enumerations and Structures
58 enum TpmEndPointInfo
59 {
60     tpmPlatformAvailable = 0x01,
61     tpmUsesTbs           = 0x02,
62     tpmInRawMode         = 0x04,
63     tpmSupportsPP        = 0x08
64 };
65
66 #ifdef _MSC_VER
67 # pragma warning(push, 3)
68 #endif
69
70 // Existing RPC interface type definitions retained so that the implementation
71 // can be re-used
72 typedef struct in_buffer
73 {
74     unsigned long BufferSize;
75     unsigned char* Buffer;
76 } _IN_BUFFER;
77
78 typedef unsigned char* _OUTPUT_BUFFER;
79
80 typedef struct out_buffer
81 {
82     uint32_t BufferSize;
83     _OUTPUT_BUFFER Buffer;
84 } _OUT_BUFFER;
85
86 #ifdef _MSC_VER
87 # pragma warning(pop)
88 #endif
89
90 #ifndef WIN32
91 typedef unsigned long DWORD;
92 typedef void* LPVOID;
93 #endif
94
95 #endif

```

D.1.2. /Simulator/include/prototypes/Simulator_fp.h

```

1  /* (Auto-generated)
2   * Created by TpmPrototypes; Version 3.0 July 18, 2017
3   * Date: Mar 4, 2020 Time: 02:36:45PM
4   */
5
6 #ifndef _SIMULATOR_FP_H_
7 #define _SIMULATOR_FP_H_
8
9 /** From TcpServer.c
10
11 #ifdef _MSC_VER
12 #elif defined(__unix__) || defined(__APPLE__)
13 #endif
14

```

```

15  /*** PlatformServer()
16  // This function processes incoming platform requests.
17  bool PlatformServer(SOCKET s);
18
19  /*** PlatformSvcRoutine()
20  // This function is called to set up the socket interfaces to listen for
21  // commands.
22  DWORD WINAPI PlatformSvcRoutine(LPVOID port);
23
24  /*** PlatformSignalService()
25  // This function starts a new thread waiting for platform signals.
26  // Platform signals are processed one at a time in the order in which they are
27  // received.
28  // If PickPorts is true, the server finds the next available port if the specified
29  // port was unavailable.
30  int PlatformSignalService(int PortNumber, bool PickPorts);
31
32  /*** RegularCommandService()
33  // This function services regular commands.
34  // If PickPorts is true, the server finds the next available port if the specified
35  // port was unavailable.
36  int RegularCommandService(int PortNumber, bool PickPorts);
37
38  /*** StartTcpServer()
39  // This is the main entry-point to the TCP server. The server listens on the port
40  // specified.
41  // If PickPorts is true, the server finds the next available port if the specified
42  // port was unavailable.
43  //
44  // Note that there is no way to specify the network interface in this implementation.
45  int StartTcpServer(int PortNumber, bool PickPorts);
46
47  /*** ReadBytes()
48  // This function reads the indicated number of bytes ('NumBytes') into buffer
49  // from the indicated socket.
50  bool ReadBytes(SOCKET s, char* buffer, int NumBytes);
51
52  /*** WriteBytes()
53  // This function will send the indicated number of bytes ('NumBytes') to the
54  // indicated socket
55  bool WriteBytes(SOCKET s, char* buffer, int NumBytes);
56
57  /*** WriteUINT32()
58  // Send 4 byte integer
59  bool WriteUINT32(SOCKET s, uint32_t val);
60
61  /*** ReadUINT32()
62  // Function to read 4 byte integer from socket.
63  bool ReadUINT32(SOCKET s, uint32_t* val);
64
65  /*** ReadVarBytes()
66  // Get a uint32-length-prepended binary array. Note that the 4-byte length is
67  // in network byte order (big-endian).
68  bool ReadVarBytes(SOCKET s, char* buffer, uint32_t* BytesReceived, int MaxLen);
69
70  /*** WriteVarBytes()
71  // Send a UINT32-length-prepended binary array. Note that the 4-byte length is
72  // in network byte order (big-endian).
73  bool WriteVarBytes(SOCKET s, char* buffer, int BytesToSend);
74
75  /*** TpmServer()
76  // Processing incoming TPM command requests using the protocol / interface
77  // defined above.
78  bool TpmServer(SOCKET s);
79
80  /*** From TPMCmdp.c

```

```

81
82 #ifdef _MSC_VER
83 #elif defined(__unix__) || defined(__APPLE__)
84 #endif
85
86 /*** Signal_PowerOn()
87 // This function processes a power-on indication. Among other things, it
88 // calls the _TPM_Init() handler.
89 void _rpc__Signal_PowerOn(bool isReset);
90
91 /*** Signal_Restart()
92 // This function processes the clock restart indication. All it does is call
93 // the platform function.
94 void _rpc__Signal_Restart(void);
95
96 /***Signal_PowerOff()
97 // This function processes the power off indication. Its primary function is
98 // to set a flag indicating that the next power on indication should cause
99 // _TPM_Init() to be called.
100 void _rpc__Signal_PowerOff(void);
101
102 /*** _rpc__ForceFailureMode()
103 // This function is used to debug the Failure Mode logic of the TPM. It will set
104 // a flag in the TPM code such that the next call to TPM2_SelfTest() will result
105 // in a failure, putting the TPM into Failure Mode.
106 void _rpc__ForceFailureMode(void);
107
108 /*** _rpc__Signal_PhysicalPresenceOn()
109 // This function is called to simulate activation of the physical presence "pin".
110 void _rpc__Signal_PhysicalPresenceOn(void);
111
112 /*** _rpc__Signal_PhysicalPresenceOff()
113 // This function is called to simulate deactivation of the physical presence "pin".
114 void _rpc__Signal_PhysicalPresenceOff(void);
115
116 /*** _rpc__Signal_Hash_Start()
117 // This function is called to simulate a _TPM_Hash_Start event. It will call
118 //
119 void _rpc__Signal_Hash_Start(void);
120
121 /*** _rpc__Signal_Hash_Data()
122 // This function is called to simulate a _TPM_Hash_Data event.
123 void _rpc__Signal_Hash_Data(_IN_BUFFER input);
124
125 /*** _rpc__Signal_HashEnd()
126 // This function is called to simulate a _TPM_Hash_End event.
127 void _rpc__Signal_HashEnd(void);
128
129 /*** _rpc__Send_Command()
130 // This is the interface to the TPM code.
131 // Return Type: void
132 void _rpc__Send_Command(
133     unsigned char locality, _IN_BUFFER request, _OUT_BUFFER* response);
134
135 /*** _rpc__Signal_CancelOn()
136 // This function is used to turn on the indication to cancel a command in process.
137 // An executing command is not interrupted. The command code may periodically check
138 // this indication to see if it should abort the current command processing and
139 // returned TPM_RC_CANCELLED.
140 void _rpc__Signal_CancelOn(void);
141
142 /*** _rpc__Signal_CancelOff()
143 // This function is used to turn off the indication to cancel a command in process.
144 void _rpc__Signal_CancelOff(void);
145
146 /*** _rpc__Signal_NvOn()

```



```

147 // In a system where the NV memory used by the TPM is not within the TPM, the
148 // NV may not always be available. This function turns on the indicator that
149 // indicates that NV is available.
150 void _rpc__Signal_NvOn(void);
151
152 /***_ _rpc__Signal_NvOff()
153 // This function is used to set the indication that NV memory is no
154 // longer available.
155 void _rpc__Signal_NvOff(void);
156
157 /***_ _rpc__RsaKeyCacheControl()
158 // This function is used to enable/disable the use of the RSA key cache during
159 // simulation.
160 void _rpc__RsaKeyCacheControl(int state);
161
162 /***_ _rpc__ACT_GetSignaled()
163 // This function is used to count the ACT second tick.
164 bool _rpc__ACT_GetSignaled(uint32_t actHandle);
165
166 /***_ _rpc__SetTpmFirmwareHash()
167 // This function is used to modify the firmware's hash during simulation.
168 void _rpc__SetTpmFirmwareHash(uint32_t hash);
169
170 /***_ _rpc__SetTpmFirmwareSvn()
171 // This function is used to modify the firmware's SVN during simulation.
172 void _rpc__SetTpmFirmwareSvn(uint16_t svn);
173
174 /*** From TPMCmds.c
175
176 /***_ main()
177 // This is the main entry point for the simulator.
178 // It registers the interface and starts listening for clients
179 int main(int argc, char* argv[]);
180
181 #endif // _SIMULATOR_FP_H_

```

D.1.3. /Simulator/src/simulatorPrivate.h

```

1 // common headers for simulator implementation files
2
3 #ifndef SIMULATOR_PRIVATE_H
4 #define SIMULATOR_PRIVATE_H
5
6 /*** Includes, Locals, Defines and Function Prototypes
7 #include <public/tpm_public.h>
8
9 #include "simulator_sysheaders.h"
10
11 // TODO_RENAME_INC_FOLDER:prototypes refers to the platform library
12 #include <prototypes/platform_public_interface.h>
13 // TODO_RENAME_INC_FOLDER:platform_interface refers to the TPM_CoreLib platform
14 // interface
15 #include <platform_interface/tpm_to_platform_interface.h>
16 #include <platform_interface/platform_to_tpm_interface.h>
17
18 #include "TpmTcpProtocol.h"
19 #include "Simulator_fp.h"
20 #endif // SIMULATOR_PRIVATE_H

```

D.1.4. /Simulator/src/simulator_sysheaders.h

```

1 // system headers for the simulator, both Windows and Linux
2

```

```

3  #ifndef _SIMULATOR_SYSHEADERS_H_
4  #define _SIMULATOR_SYSHEADERS_H_
5  // include the system headers silencing warnings that occur with /Wall
6  #include <stdio.h>
7  #include <stdbool.h>
8  #include <stdint.h>
9  #include <stdlib.h>
10 #include <ctype.h>
11 #include <string.h>
12
13 #ifdef _MSC_VER
14 # pragma warning(push, 3)
15 // C4668 is supposed to be level 4, but this is still necessary to suppress the
16 // error. We don't want to suppress it globally because the same error can
17 // happen in the TPM code and it shouldn't be ignored in those cases because it
18 // generally means a configuration header is missing.
19 //
20 // X is not defined as a preprocessor macro, assuming 0 for #if
21 # pragma warning(disable : 4668)
22 # include <windows.h>
23 # include <winsock.h>
24 # pragma warning(pop)
25 typedef int socklen_t;
26 #elif defined(__unix__) || defined(__APPLE__)
27 # include <unistd.h>
28 # include <errno.h>
29 # include <netinet/in.h>
30 # include <sys/socket.h>
31 # include <pthread.h>
32 // simulate certain windows APIs
33 # define ZeroMemory(ptr, sz) (memset((ptr), 0, (sz)))
34 # define closesocket(x) close(x)
35 # define INVALID_SOCKET (-1)
36 # define SOCKET_ERROR (-1)
37 # define WSAGetLastError() (errno)
38 # define WSAEADDRINUSE EADDRINUSE
39 # define INT_PTR intptr_t
40 typedef int SOCKET;
41 # define _strncmpi strcasecmp
42 #else
43 # error "Unsupported platform."
44 #endif // _MSC_VER
45 #endif // _SIMULATOR_SYSHEADERS_H_

```

D.1.5. /Simulator/src/TcpServer.c

```

1  /** Description
2  //
3  // This file contains the socket interface to a TPM simulator.
4  //
5  /** Includes, Locals, Defines and Function Prototypes
6  #include "simulatorPrivate.h"
7
8  // To access key cache control in TPM
9  void RsaKeyCacheControl(int state);
10
11 #ifndef __IGNORE_STATE__
12
13 static uint32_t ServerVersion = 1;
14
15 # define MAX_BUFFER 1048576
16 char InputBuffer[MAX_BUFFER]; //The input data buffer for the simulator.
17 char OutputBuffer[MAX_BUFFER]; //The output data buffer for the simulator.
18
19 struct

```

```

20 {
21     uint32_t largestCommandSize;
22     uint32_t largestCommand;
23     uint32_t largestResponseSize;
24     uint32_t largestResponse;
25 } CommandResponseSizes = {0};
26
27 #endif // __IGNORE_STATE__
28
29 /** Functions
30
31 **** CreateSocket()
32 // This function creates a socket listening on 'PortNumber'.
33 // If PickPorts is true, the server finds the next available port if the specified
34 // port was unavailable.
35 static int CreateSocket(
36     int PortNumber, bool PickPorts, SOCKET* ListenSocket, int* ActualPort)
37 {
38     struct sockaddr_in MyAddress;
39     int res;
40     //
41     // Initialize Winsock
42     #ifdef _MSC_VER
43     WSADATA wsaData;
44     res = WSAStartup(MAKEWORD(2, 2), &wsaData);
45     if(res != 0)
46     {
47         printf("WSAStartup failed with error: %d\n", res);
48         return -1;
49     }
50 #endif
51     // create listening socket
52     *ListenSocket = socket(PF_INET, SOCK_STREAM, 0);
53     if(INVALID_SOCKET == *ListenSocket)
54     {
55         printf("Cannot create server listen socket. Error is 0x%x\n",
56             WSAGetLastError());
57         return -1;
58     }
59     // bind the listening socket to the specified port
60     ZeroMemory(&MyAddress, sizeof(MyAddress));
61     MyAddress.sin_port = htons((unsigned short)PortNumber);
62     MyAddress.sin_family = AF_INET;
63
64     res = bind(*ListenSocket, (struct sockaddr*)&MyAddress, sizeof(MyAddress));
65     if(PickPorts)
66     {
67         while(res == SOCKET_ERROR && MyAddress.sin_port < UINT16_MAX)
68         {
69             // keep trying as long as the underlying error is that the port is already
70             in use
71             if(WSAGetLastError() != WSAEADDRINUSE)
72             {
73                 break;
74             }
75             MyAddress.sin_port++;
76             res =
77                 bind(*ListenSocket, (struct sockaddr*)&MyAddress, sizeof(MyAddress));
78         }
79     }
80     if(res == SOCKET_ERROR)
81     {
82         printf("Bind error. Error is 0x%x\n", WSAGetLastError());
83         return -1;
84     }

```

```

85     // listen/wait for server connections
86     res = listen(*ListenSocket, 3);
87     if(res == SOCKET_ERROR)
88     {
89         printf("Listen error.  Error is 0x%x\n", WSAGetLastError());
90         return -1;
91     }
92
93     *ActualPort = ntohs(MyAddress.sin_port);
94     return 0;
95 }
96
97 /** PlatformServer()
98 // This function processes incoming platform requests.
99 bool PlatformServer(SOCKET s)
100 {
101     bool    OK = true;
102     uint32_t Command;
103     //
104     for(;;)
105     {
106         OK = ReadBytes(s, (char*)&Command, 4);
107         // client disconnected (or other error).  We stop processing this client
108         // and return to our caller who can stop the server or listen for another
109         // connection.
110         if(!OK)
111             return true;
112         Command = ntohl(Command);
113         switch(Command)
114         {
115             case TPM_SIGNAL_POWER_ON:
116                 _rpc_Signal_PowerOn(false);
117                 break;
118             case TPM_SIGNAL_POWER_OFF:
119                 _rpc_Signal_PowerOff();
120                 break;
121             case TPM_SIGNAL_RESET:
122                 _rpc_Signal_PowerOn(true);
123                 break;
124             case TPM_SIGNAL_RESTART:
125                 _rpc_Signal_Restart();
126                 break;
127             case TPM_SIGNAL_PHYS_PRESENCE_ON:
128                 _rpc_Signal_PhysicalPresenceOn();
129                 break;
130             case TPM_SIGNAL_PHYS_PRESENCE_OFF:
131                 _rpc_Signal_PhysicalPresenceOff();
132                 break;
133             case TPM_SIGNAL_CANCEL_ON:
134                 _rpc_Signal_CancelOn();
135                 break;
136             case TPM_SIGNAL_CANCEL_OFF:
137                 _rpc_Signal_CancelOff();
138                 break;
139             case TPM_SIGNAL_NV_ON:
140                 _rpc_Signal_NvOn();
141                 break;
142             case TPM_SIGNAL_NV_OFF:
143                 _rpc_Signal_NvOff();
144                 break;
145             case TPM_SIGNAL_KEY_CACHE_ON:
146                 _rpc_RsaKeyCacheControl(true);
147                 break;
148             case TPM_SIGNAL_KEY_CACHE_OFF:
149                 _rpc_RsaKeyCacheControl(false);
150                 break;

```

```

151         case TPM_SESSION_END:
152             // Client signaled end-of-session
153             TpmEndSimulation();
154             return true;
155         case TPM_STOP:
156             // Client requested the simulator to exit
157             return false;
158         case TPM_TEST_FAILURE_MODE:
159             _rpc_ForceFailureMode();
160             break;
161         case TPM_GET_COMMAND_RESPONSE_SIZES:
162             OK = WriteVarBytes(
163                 s, (char*)&CommandResponseSizes, sizeof(CommandResponseSizes));
164             memset(&CommandResponseSizes, 0, sizeof(CommandResponseSizes));
165             if(!OK)
166                 return true;
167             break;
168         case TPM_ACT_GET_SIGNED:
169             {
170                 uint32_t actHandle;
171                 OK = ReadUINT32(s, &actHandle);
172                 WriteUINT32(s, _rpc_ACT_GetSigned(actHandle));
173                 break;
174             }
175         case TPM_SET_FW_HASH:
176             {
177                 uint32_t hash;
178                 OK = ReadUINT32(s, &hash);
179                 _rpc_SetTpmFirmwareHash(hash);
180                 break;
181             }
182         case TPM_SET_FW_SVN:
183             {
184                 uint32_t svn;
185                 OK = ReadUINT32(s, &svn);
186                 _rpc_SetTpmFirmwareSvn((uint16_t)svn);
187                 break;
188             }
189         default:
190             printf("Unrecognized platform interface command %d\n", (int)Command);
191             WriteUINT32(s, 1);
192             return true;
193     }
194     WriteUINT32(s, 0);
195 }
196 }
197
198 /*** WritePortToFile()
199 // This function writes the given port out to a file.
200 bool WritePortToFile(const char* filename, int port)
201 {
202     FILE* f;
203
204     #ifdef _MSC_VER
205     # pragma warning(push)
206     # pragma warning(disable : 4996)
207     #endif // _MSC_VER
208     f = fopen(filename, "w");
209     #ifdef _MSC_VER
210     # pragma warning(pop)
211     #endif // _MSC_VER
212     if(f == NULL)
213     {
214         return false;
215     }
216

```

```

217     fprintf(f, "%d\n", port);
218     return fclose(f) == 0;
219 }
220
221 /*** DeletePortFile()
222 // This function deletes the port file.
223 bool DeletePortFile(const char* filename)
224 {
225     return remove(filename) == 0;
226 }
227
228 struct platformParameters
229 {
230     int port;
231     bool pickPorts;
232 };
233
234 /*** PlatformSvcRoutine()
235 // This function is called to set up the socket interfaces to listen for
236 // commands.
237 DWORD WINAPI PlatformSvcRoutine(LPVOID parms)
238 {
239     struct platformParameters* platformParms = (struct platformParameters*)parms;
240     int PortNumber = platformParms->port;
241     bool PickPorts = platformParms->pickPorts;
242     SOCKET listenSocket, serverSocket;
243     struct sockaddr_in HerAddress;
244     int res;
245     socklen_t length;
246     bool continueServing;
247     const char* portFile = "platform.port";
248
249     res = CreateSocket(PortNumber, PickPorts, &listenSocket, &PortNumber);
250     if(res != 0)
251     {
252         printf("Could not create platform service socket\n");
253         return res;
254     }
255     if(!WritePortToFile(portFile, PortNumber))
256     {
257         printf("Could not write port to %s\n", portFile);
258         return (DWORD)-1;
259     }
260     // Loop accepting connections one-by-one until we are killed or asked to stop
261     // Note the platform service is single-threaded so we don't listen for a new
262     // connection until the prior connection drops.
263     do
264     {
265         printf("Platform server listening on port %d\n", PortNumber);
266
267         // blocking accept
268         length = sizeof(HerAddress);
269         serverSocket = accept(listenSocket, (struct sockaddr*)&HerAddress, &length);
270         if(serverSocket == INVALID_SOCKET)
271         {
272             printf("Accept error. Error is 0x%x\n", WSAGetLastError());
273             return (DWORD)-1;
274         }
275         printf("Client accepted\n");
276
277         // normal behavior on client disconnection is to wait for a new client
278         // to connect
279         continueServing = PlatformServer(serverSocket);
280         closesocket(serverSocket);
281     } while(continueServing);
282     if(!DeletePortFile(portFile))

```



```

283     {
284         printf("Could not delete %s", portFile);
285         return (DWORD)-1;
286     }
287     free(parms);
288     return 0;
289 }
290
291 /** PlatformSignalService()
292 // This function starts a new thread waiting for platform signals.
293 // Platform signals are processed one at a time in the order in which they are
294 // received.
295 // If PickPorts is true, the server finds the next available port if the specified
296 // port was unavailable.
297 int PlatformSignalService(int PortNumber, bool PickPorts)
298 {
299     struct platformParameters* parms;
300
301     parms = (struct platformParameters*)malloc(sizeof(struct platformParameters));
302     parms->port = PortNumber;
303     parms->pickPorts = PickPorts;
304 #if defined(_MSC_VER)
305     HANDLE hPlatformSvc;
306     int ThreadId;
307
308     hPlatformSvc = CreateThread(NULL,
309                                0,
310                                (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
311                                (LPVOID)parms,
312                                0,
313                                (LPDWORD)&ThreadId);
314
315     if(hPlatformSvc == NULL)
316     {
317         printf("Could not create platform thread\n");
318         return -1;
319     }
320 #else
321     pthread_t thread_id;
322     int ret;
323
324     ret = pthread_create(&thread_id, NULL, (void*)PlatformSvcRoutine, (LPVOID)parms);
325     if(ret == -1)
326     {
327         printf("Could not create platform thread: %s\n", strerror(ret));
328     }
329     return ret;
330 #endif // _MSC_VER
331 }
332
333 /** RegularCommandService()
334 // This function services regular commands.
335 // If PickPorts is true, the server finds the next available port if the specified
336 // port was unavailable.
337 int RegularCommandService(int PortNumber, bool PickPorts)
338 {
339     SOCKET listenSocket;
340     SOCKET serverSocket;
341     struct sockaddr_in HerAddress;
342     int res;
343     socklen_t length;
344     bool continueServing;
345     const char* portFile = "command.port";
346
347     res = CreateSocket(PortNumber, PickPorts, &listenSocket, &PortNumber);
348     if(res != 0)

```

```

349     {
350         printf("Could not create command service socket\n");
351         return res;
352     }
353     if(!WritePortToFile(portFile, PortNumber))
354     {
355         printf("Could not write port to %s\n", portFile);
356         return -1;
357     }
358     // Loop accepting connections one-by-one until we are killed or asked to stop
359     // Note the TPM command service is single-threaded so we don't listen for
360     // a new connection until the prior connection drops.
361     do
362     {
363         printf("TPM command server listening on port %d\n", PortNumber);
364
365         // blocking accept
366         length = sizeof(HeaderAddress);
367         serverSocket = accept(listenSocket, (struct sockaddr*)&HeaderAddress, &length);
368         if(serverSocket == INVALID_SOCKET)
369         {
370             printf("Accept error. Error is 0x%x\n", WSAGetLastError());
371             return -1;
372         }
373         printf("Client accepted\n");
374
375         // normal behavior on client disconnection is to wait for a new client
376         // to connect
377         continueServing = TpmServer(serverSocket);
378         closesocket(serverSocket);
379     } while(continueServing);
380
381     if(!DeletePortFile(portFile))
382     {
383         printf("Could not delete %s", portFile);
384         return -1;
385     }
386     return 0;
387 }
388
389 #if RH_ACT_0
390
391 /*** SimulatorTimeServiceRoutine()
392 // This function is called to service the time 'ticks'.
393 static unsigned long WINAPI SimulatorTimeServiceRoutine(LPVOID notUsed)
394 {
395     // All time is in ms
396     const int64_t tick = 1000;
397     uint64_t prevTime = _plat__RealTime();
398     int64_t timeout = tick;
399
400     (void)notUsed;
401
402     while(true)
403     {
404         uint64_t curTime;
405
406         # if defined( _MSC_VER)
407             Sleep((DWORD)timeout);
408         # else
409             struct timespec req = {timeout / 1000, (timeout % 1000) * 1000};
410             struct timespec rem;
411             nanosleep(&req, &rem);
412         # endif // _MSC_VER
413         curTime = _plat__RealTime();
414

```

```

415         // May need to issue several ticks if the Sleep() took longer than asked,
416         // or no ticks at all, it Sleep() was interrupted prematurely.
417         while(prevTime < curTime - tick / 2)
418         {
419             //printf("%05lld | %05lld\n",
420             //      prevTime % 100000, (curTime - tick / 2) % 100000);
421             _plat__ACT_Tick();
422             prevTime += (uint64_t)tick;
423         }
424         // Adjust the next timeout to keep the average interval of one second
425         timeout = tick + (prevTime - curTime);
426         //prevTime = curTime;
427         //printf("%04lld | c:%05lld | p:%05llu\n",
428         //      timeout, curTime % 100000, prevTime);
429     }
430     return 0;
431 }
432
433 /*** ActTimeService()
434 // This function starts a new thread waiting to wait for time ticks.
435 // Return Type: int
436 // ==0          success
437 // !=0          failure
438 static int ActTimeService(void)
439 {
440     static bool running = false;
441     int         ret      = 0;
442     if(!running)
443     {
444 #   if defined(_MSC_VER)
445         HANDLE hThr;
446         int     ThreadId;
447         //
448         printf("Starting ACT thread...\n");
449         // Don't allow ticks to be processed before TPM is manufactured.
450         _plat__ACT_EnableTicks(false);
451
452         // Create service thread for ACT internal timer
453         hThr = CreateThread(NULL,
454                             0,
455                             (LPTHREAD_START_ROUTINE)SimulatorTimeServiceRoutine,
456                             (LPVOID)NULL,
457                             0,
458                             (LPDWORD)&ThreadId);
459         if(hThr != NULL)
460             CloseHandle(hThr);
461         else
462             ret = -1;
463 #   else
464         pthread_t thread_id;
465         //
466         ret = pthread_create(
467             &thread_id, NULL, (void*)SimulatorTimeServiceRoutine, (LPVOID)NULL);
468 #   endif // _MSC_VER
469
470         if(ret != 0)
471             printf("ACT thread Creation failed\n");
472         else
473             running = true;
474     }
475     return ret;
476 }
477
478 #endif // RH_ACT_0
479
480 /*** StartTcpServer()

```

```

481 // This is the main entry-point to the TCP server. The server listens on the port
482 // specified.
483 // If PickPorts is true, the server finds the next available port if the specified
484 // port was unavailable.
485 //
486 // Note that there is no way to specify the network interface in this implementation.
487 int StartTcpServer(int PortNumber, bool PickPorts)
488 {
489     int res;
490
491 #if ACT_SUPPORT
492 # if !RH_ACT_0
493 #   error "Compliance tests currently require ACT_0 if ACT_SUPPORT"
494 # endif
495     // Start the Time Service routine
496     res = ActTimeService();
497     if(res != 0)
498     {
499         printf("TimeService failed\n");
500         return res;
501     }
502 #endif // ACT_SUPPORT
503
504     // Start Platform Signal Processing Service
505     res = PlatformSignalService(PortNumber + 1, PickPorts);
506     if(res != 0)
507     {
508         printf("PlatformSignalService failed\n");
509         return res;
510     }
511     // Start Regular/DRTM TPM command service
512     res = RegularCommandService(PortNumber, PickPorts);
513     if(res != 0)
514     {
515         printf("RegularCommandService failed\n");
516         return res;
517     }
518     return 0;
519 }
520
521 /*** ReadBytes()
522 // This function reads the indicated number of bytes ('NumBytes') into buffer
523 // from the indicated socket.
524 bool ReadBytes(SOCKET s, char* buffer, int NumBytes)
525 {
526     int res;
527     int numGot = 0;
528     //
529     while(numGot < NumBytes)
530     {
531         res = recv(s, buffer + numGot, NumBytes - numGot, 0);
532         if(res == -1)
533         {
534             printf("Receive error. Error is 0x%x\n", WSAGetLastError());
535             return false;
536         }
537         if(res == 0)
538         {
539             return false;
540         }
541         numGot += res;
542     }
543     return true;
544 }
545
546 /*** WriteBytes()

```

```

547 // This function will send the indicated number of bytes ('NumBytes') to the
548 // indicated socket
549 bool WriteBytes(SOCKET s, char* buffer, int NumBytes)
550 {
551     int res;
552     int numSent = 0;
553     //
554     while(numSent < NumBytes)
555     {
556         res = send(s, buffer + numSent, NumBytes - numSent, 0);
557         if(res == -1)
558         {
559             if(WSAGetLastError() == 0x2745)
560             {
561                 printf("Client disconnected\n");
562             }
563             else
564             {
565                 printf("Send error. Error is 0x%x\n", WSAGetLastError());
566             }
567             return false;
568         }
569         numSent += res;
570     }
571     return true;
572 }
573
574 /*** WriteUINT32()
575 // Send 4 byte integer
576 bool WriteUINT32(SOCKET s, uint32_t val)
577 {
578     uint32_t netVal = htonl(val);
579     //
580     return WriteBytes(s, (char*)&netVal, 4);
581 }
582
583 /*** ReadUINT32()
584 // Function to read 4 byte integer from socket.
585 bool ReadUINT32(SOCKET s, uint32_t* val)
586 {
587     uint32_t netVal;
588     //
589     if(!ReadBytes(s, (char*)&netVal, 4))
590         return false;
591     *val = ntohl(netVal);
592     return true;
593 }
594
595 /*** ReadVarBytes()
596 // Get a uint32-length-prepended binary array. Note that the 4-byte length is
597 // in network byte order (big-endian).
598 bool ReadVarBytes(SOCKET s, char* buffer, uint32_t* BytesReceived, int MaxLen)
599 {
600     int length;
601     bool res;
602     //
603     res = ReadBytes(s, (char*)&length, 4);
604     if(!res)
605         return res;
606     length = ntohl(length);
607     *BytesReceived = length;
608     if(length > MaxLen)
609     {
610         printf("Buffer too big. Client says %d\n", length);
611         return false;
612     }

```

```

613     if(length == 0)
614         return true;
615     res = ReadBytes(s, buffer, length);
616     if(!res)
617         return res;
618     return true;
619 }
620
621 /*** WriteVarBytes()
622 // Send a uint32-length-prepended binary array. Note that the 4-byte length is
623 // in network byte order (big-endian).
624 bool WriteVarBytes(SOCKET s, char* buffer, int BytesToSend)
625 {
626     uint32_t netLength = htonl(BytesToSend);
627     bool res;
628     //
629     res = WriteBytes(s, (char*)&netLength, 4);
630     if(!res)
631         return res;
632     res = WriteBytes(s, buffer, BytesToSend);
633     if(!res)
634         return res;
635     return true;
636 }
637
638 /*** TpmServer()
639 // Processing incoming TPM command requests using the protocol / interface
640 // defined above.
641 bool TpmServer(SOCKET s)
642 {
643     uint32_t length;
644     uint32_t Command;
645     uint8_t locality;
646     bool OK;
647     int result;
648     int clientVersion;
649     _IN_BUFFER InBuffer;
650     _OUT_BUFFER OutBuffer;
651     //
652     for(;;)
653     {
654         OK = ReadBytes(s, (char*)&Command, 4);
655         // client disconnected (or other error). We stop processing this client
656         // and return to our caller who can stop the server or listen for another
657         // connection.
658         if(!OK)
659             return true;
660         Command = ntohl(Command);
661         switch(Command)
662         {
663             case TPM_SIGNAL_HASH_START:
664                 _rpc_Signal_Hash_Start();
665                 break;
666             case TPM_SIGNAL_HASH_END:
667                 _rpc_Signal_HashEnd();
668                 break;
669             case TPM_SIGNAL_HASH_DATA:
670                 OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
671                 if(!OK)
672                     return true;
673                 InBuffer.Buffer = (uint8_t*)InputBuffer;
674                 InBuffer.BufferSize = length;
675                 _rpc_Signal_Hash_Data(InBuffer);
676                 break;
677             case TPM_SEND_COMMAND:
678                 OK = ReadBytes(s, (char*)&locality, 1);

```



```

679         if(!OK)
680             return true;
681         OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
682         if(!OK)
683             return true;
684         InBuffer.Buffer      = (uint8_t*)InputBuffer;
685         InBuffer.BufferSize  = length;
686         OutBuffer.BufferSize = MAX_BUFFER;
687         OutBuffer.Buffer     = (_OUTPUT_BUFFER)OutputBuffer;
688         // record the number of bytes in the command if it is the largest
689         // we have seen so far.
690         if(InBuffer.BufferSize > CommandResponseSizes.largestCommandSize)
691         {
692             CommandResponseSizes.largestCommandSize = InBuffer.BufferSize;
693             memcpy(&CommandResponseSizes.largestCommand,
694                 &InputBuffer[6],
695                 sizeof(uint32_t));
696         }
697         _rpc_Send_Command(locality, InBuffer, &OutBuffer);
698         // record the number of bytes in the response if it is the largest
699         // we have seen so far.
700         if(OutBuffer.BufferSize > CommandResponseSizes.largestResponseSize)
701         {
702             CommandResponseSizes.largestResponseSize = OutBuffer.BufferSize;
703             memcpy(&CommandResponseSizes.largestResponse,
704                 &OutputBuffer[6],
705                 sizeof(uint32_t));
706         }
707         OK = WriteVarBytes(s, (char*)OutBuffer.Buffer, OutBuffer.BufferSize);
708         if(!OK)
709             return true;
710         break;
711     case TPM_REMOTE_HANDSHAKE:
712         OK = ReadBytes(s, (char*)&clientVersion, 4);
713         if(!OK)
714             return true;
715         if(clientVersion == 0)
716         {
717             printf("Unsupported client version (0).\n");
718             return true;
719         }
720         OK &= WriteUINT32(s, ServerVersion);
721         OK &= WriteUINT32(
722             s, tpmInRawMode | tpmPlatformAvailable | tpmSupportsPP);
723         break;
724     case TPM_SET_ALTERNATIVE_RESULT:
725         OK = ReadBytes(s, (char*)&result, 4);
726         if(!OK)
727             return true;
728         // Alternative result is not applicable to the simulator.
729         break;
730     case TPM_SESSION_END:
731         // Client signaled end-of-session
732         return true;
733     case TPM_STOP:
734         // Client requested the simulator to exit
735         return false;
736     default:
737         printf("Unrecognized TPM interface command %d\n", (int)Command);
738         return true;
739     }
740     OK = WriteUINT32(s, 0);
741     if(!OK)
742         return true;
743 }
744 }

```

D.1.6. /Simulator/src/

```

1  /** Description
2  // This file contains the functions that process the commands received on the
3  // control port or the command port of the simulator. The control port is used
4  // to allow simulation of hardware events (such as, _TPM_Hash_Start) to test
5  // the simulated TPM's reaction to those events. This improves code coverage
6  // of the testing.
7
8  /** Includes and Data Definitions
9  #include "simulatorPrivate.h"
10
11 static bool s_isPowerOn = false;
12
13 /** Functions
14
15 /*** Signal_PowerOn()
16 // This function processes a power-on indication. Among other things, it
17 // calls the _TPM_Init() handler.
18 void _rpc__Signal_PowerOn(bool isReset)
19 {
20     // if power is on and this is not a call to do TPM reset then return
21     if(s_isPowerOn && !isReset)
22         return;
23     // If this is a reset but power is not on, then return
24     if(isReset && !s_isPowerOn)
25         return;
26     // Unless this is just a reset, pass power on signal to platform
27     if(!isReset)
28         _plat__Signal_PowerOn();
29     // Power on and reset both lead to _TPM_Init()
30     _plat__Signal_Reset();
31
32     // Set state as power on
33     s_isPowerOn = true;
34 }
35
36 /*** Signal_Restart()
37 // This function processes the clock restart indication. All it does is call
38 // the platform function.
39 void _rpc__Signal_Restart(void)
40 {
41     _plat__TimerRestart();
42 }
43
44 /***Signal_PowerOff()
45 // This function processes the power off indication. Its primary function is
46 // to set a flag indicating that the next power on indication should cause
47 // _TPM_Init() to be called.
48 void _rpc__Signal_PowerOff(void)
49 {
50     if(s_isPowerOn)
51         // Pass power off signal to platform
52         _plat__Signal_PowerOff();
53     // This could be redundant, but...
54     s_isPowerOn = false;
55
56     return;
57 }
58
59 /*** _rpc__ForceFailureMode()
60 // This function is used to debug the Failure Mode logic of the TPM. It will set
61 // a flag in the TPM code such that the next call to TPM2_SelfTest() will result
62 // in a failure, putting the TPM into Failure Mode.
63 void _rpc__ForceFailureMode(void)
64 {

```

```

65  #if SIMULATION
66      SetForceFailureMode();
67  #endif
68      return;
69  }
70
71  /***_rpc_Signal_PhysicalPresenceOn()
72  // This function is called to simulate activation of the physical presence "pin".
73  void _rpc_Signal_PhysicalPresenceOn(void)
74  {
75      // If TPM power is on...
76      if(s_isPowerOn)
77          // ... pass physical presence on to platform
78          _plat_Signal_PhysicalPresenceOn();
79      return;
80  }
81
82  /***_rpc_Signal_PhysicalPresenceOff()
83  // This function is called to simulate deactivation of the physical presence "pin".
84  void _rpc_Signal_PhysicalPresenceOff(void)
85  {
86      // If TPM is power on...
87      if(s_isPowerOn)
88          // ... pass physical presence off to platform
89          _plat_Signal_PhysicalPresenceOff();
90      return;
91  }
92
93  /***_rpc_Signal_Hash_Start()
94  // This function is called to simulate a _TPM_Hash_Start event. It will call
95  //
96  void _rpc_Signal_Hash_Start(void)
97  {
98      // If TPM power is on...
99      if(s_isPowerOn)
100          // ... pass _TPM_Hash_Start signal to TPM
101          _TPM_Hash_Start();
102      return;
103  }
104
105  /***_rpc_Signal_Hash_Data()
106  // This function is called to simulate a _TPM_Hash_Data event.
107  void _rpc_Signal_Hash_Data(_IN_BUFFER input)
108  {
109      // If TPM power is on...
110      if(s_isPowerOn)
111          // ... pass _TPM_Hash_Data signal to TPM
112          _TPM_Hash_Data(input.BufferSize, input.Buffer);
113      return;
114  }
115
116  /***_rpc_Signal_HashEnd()
117  // This function is called to simulate a _TPM_Hash_End event.
118  void _rpc_Signal_HashEnd(void)
119  {
120      // If TPM power is on...
121      if(s_isPowerOn)
122          // ... pass _TPM_HashEnd signal to TPM
123          _TPM_Hash_End();
124      return;
125  }
126
127  /***_rpc_Send_Command()
128  // This is the interface to the TPM code.
129  // Return Type: void
130  void _rpc_Send_Command(

```

```

131     unsigned char locality, _IN_BUFFER request, _OUT_BUFFER* response)
132 {
133     // If TPM is power off, reject any commands.
134     if(!s_isPowerOn)
135     {
136         response->BufferSize = 0;
137         return;
138     }
139     // Set the locality of the command so that it doesn't change during the command
140     _plat__LocalitySet(locality);
141     // Do implementation-specific command dispatch
142     _plat__RunCommand(
143         request.BufferSize, request.Buffer, &response->BufferSize, &response->Buffer);
144     return;
145 }
146
147 /***_rpc_Signal_CancelOn()
148 // This function is used to turn on the indication to cancel a command in process.
149 // An executing command is not interrupted. The command code may periodically check
150 // this indication to see if it should abort the current command processing and
151 // returned TPM_RC_CANCELLED.
152 void _rpc__Signal_CancelOn(void)
153 {
154     // If TPM power is on...
155     if(s_isPowerOn)
156         // ... set the platform canceling flag.
157         _plat__SetCancel();
158     return;
159 }
160
161 /***_rpc_Signal_CancelOff()
162 // This function is used to turn off the indication to cancel a command in process.
163 void _rpc__Signal_CancelOff(void)
164 {
165     // If TPM power is on...
166     if(s_isPowerOn)
167         // ... set the platform canceling flag.
168         _plat__ClearCancel();
169     return;
170 }
171
172 /***_rpc_Signal_NvOn()
173 // In a system where the NV memory used by the TPM is not within the TPM, the
174 // NV may not always be available. This function turns on the indicator that
175 // indicates that NV is available.
176 void _rpc__Signal_NvOn(void)
177 {
178     // If TPM power is on...
179     if(s_isPowerOn)
180         // ... make the NV available
181         _plat__SetNvAvail();
182     return;
183 }
184
185 /***_rpc_Signal_NvOff()
186 // This function is used to set the indication that NV memory is no
187 // longer available.
188 void _rpc__Signal_NvOff(void)
189 {
190     // If TPM power is on...
191     if(s_isPowerOn)
192         // ... make NV not available
193         _plat__ClearNvAvail();
194     return;
195 }
196

```

```

197 void RsaKeyCacheControl(int state);
198
199 /***_rpc_RsaKeyCacheControl()
200 // This function is used to enable/disable the use of the RSA key cache during
201 // simulation.
202 void _rpc_RsaKeyCacheControl(int state)
203 {
204 #if USE_RSA_KEY_CACHE
205     RsaKeyCacheControl(state);
206 #else
207     NOT_REFERENCED(state);
208 #endif
209     return;
210 }
211
212 /***_rpc_ACT_GetSignaled()
213 // This function is used to count the ACT second tick.
214 bool _rpc_ACT_GetSignaled(uint32_t actHandle)
215 {
216 #if ACT_SUPPORT
217     // If TPM power is on...
218     if(s_isPowerOn)
219         // ... query the platform
220         return _plat_ACT_GetSignaled(actHandle - TPM_RH_ACT_0);
221 #else // ACT_SUPPORT
222     NOT_REFERENCED(actHandle);
223 #endif // ACT_SUPPORT
224     return false;
225 }
226
227 /***_rpc_SetTpmFirmwareHash()
228 // This function is used to modify the firmware's hash during simulation.
229 void _rpc_SetTpmFirmwareHash(uint32_t hash)
230 {
231     _plat_SetTpmFirmwareHash(hash);
232 }
233
234 /***_rpc_SetTpmFirmwareSvn()
235 // This function is used to modify the firmware's SVN during simulation.
236 void _rpc_SetTpmFirmwareSvn(uint16_t svn)
237 {
238     _plat_SetTpmFirmwareSvn(svn);
239 }

```

D.1.7. /Simulator/src/

```

1 /***_Description
2 // This file contains the entry point for the simulator.
3
4 /***_Includes, Defines, Data Definitions, and Function Prototypes
5 #include "simulatorPrivate.h"
6
7 #define PURPOSE \
8     "TPM 2.0 Reference Simulator.\n" \
9     "Copyright (c) Microsoft Corporation. All rights reserved."
10
11 #define DEFAULT_TPM_PORT 2321
12
13 // Information about command line arguments (does not include program name)
14 static uint32_t s_ArgsMask = 0; // Bit mask of unmatched command line args
15 static int s_Argc = 0;
16 static const char** s_Argv = NULL;
17
18 /***_Functions
19

```

```

20  #if DEBUG
21  /*** Assert()
22  // This function implements a run-time assertion.
23  // Computation of its parameters must not result in any side effects, as these
24  // computations will be stripped from the release builds.
25  static void Assert(bool cond, const char* msg)
26  {
27      if(cond)
28          return;
29      fputs(msg, stderr);
30      exit(2);
31  }
32  #else
33  # define Assert(cond, msg)
34  #endif
35
36  /*** Usage()
37  // This function prints the proper calling sequence for the simulator.
38  static void Usage(const char* programName)
39  {
40      fprintf(stderr, "%s\n\n", PURPOSE);
41      fprintf(stderr,
42          "Usage:  %s [PortNum] [opts]\n\n"
43          "Starts the TPM server listening on TCP port PortNum (by default "
44          "%d).\n\n"
45          "An option can be in the short form (one letter preceded with '-' or "
46          "'/')\n"
47          "or in the full form (preceded with '--' or no option marker at all).\n"
48          "Possible options are:\n"
49          "  -h (--help) or ? - print this message\n"
50          "  -m (--manufacture) - forces NV state of the TPM simulator to be "
51          "(re)manufactured\n"
52          "  -p (--pick_ports) - choose the next available TCP ports "
53          "automatically "
54          "if PortNum is not available\n",
55          programName,
56          DEFAULT_TPM_PORT);
57      exit(1);
58  }
59
60  /*** CmdLineParser_Init()
61  // This function initializes command line option parser.
62  static bool CmdLineParser_Init(int argc, char* argv[], int maxOpts)
63  {
64      if(argc == 1)
65          return false;
66
67      if(maxOpts && (argc - 1) > maxOpts)
68      {
69          fprintf(stderr, "No more than %d options can be specified\n\n", maxOpts);
70          Usage(argv[0]);
71      }
72
73      s_Argc      = argc - 1;
74      s_Argv      = (const char**) (argv + 1);
75      s_ArgsMask  = (1 << s_Argc) - 1;
76      return true;
77  }
78
79  /*** CmdLineParser_More()
80  // Returns true if there are unparsed options still.
81  static bool CmdLineParser_More(void)
82  {
83      return s_ArgsMask != 0;
84  }
85

```



```

86  /*** CmdLineParser_IsOpt()
87  // This function determines if the given command line parameter represents a valid
88  // option.
89  static bool CmdLineParser_IsOpt(
90      const char* opt,          // Command line parameter to check
91      const char* optFull,     // Expected full name
92      const char* optShort,    // Expected short (single letter) name
93      bool        dashed      // The parameter is preceded by a single dash
94  )
95  {
96      return 0 == strcmp(opt, optFull)
97          || (optShort && opt[0] == optShort[0] && opt[1] == 0)
98          || (dashed && opt[0] == '-' && 0 == strcmp(opt + 1, optFull));
99  }
100
101  /*** CmdLineParser_IsOptPresent()
102  // This function determines if the given command line parameter represents a valid
103  // option.
104  static bool CmdLineParser_IsOptPresent(const char* optFull, const char* optShort)
105  {
106      int i;
107      int curArgBit;
108      Assert(s_Argv != NULL, "InitCmdLineOptParser(argc, argv) has not been invoked\n");
109      Assert(optFull && optFull[0],
110          "Full form of a command line option must be present.\n"
111          "If only a short (single letter) form is supported, it must be"
112          "specified as the full one.\n");
113      Assert(!optShort || (optShort[0] && !optShort[1]),
114          "If a short form of an option is specified, it must consist "
115          "of a single letter only.\n");
116
117      if(!CmdLineParser_More())
118          return false;
119
120      for(i = 0, curArgBit = 1; i < s_Argc; ++i, curArgBit <= 1)
121      {
122          const char* opt = s_Argv[i];
123          if((s_ArgsMask & curArgBit) && opt
124              && (0 == strcmp(opt, optFull)
125                  || ((opt[0] == '/' || opt[0] == '-')
126                      && CmdLineParser_IsOpt(
127                          opt + 1, optFull, optShort, opt[0] == '-'))))
128          {
129              s_ArgsMask ^= curArgBit;
130              return true;
131          }
132      }
133      return false;
134  }
135
136  /*** CmdLineParser_Done()
137  // This function notifies the parser that no more options are needed.
138  static void CmdLineParser_Done(const char* programName)
139  {
140      char delim = ':';
141      int i;
142      int curArgBit;
143
144      if(!CmdLineParser_More())
145          return;
146
147      fprintf(stderr,
148          "Command line contains unknown option%s",
149          s_ArgsMask & (s_ArgsMask - 1) ? "s" : "");
150      for(i = 0, curArgBit = 1; i < s_Argc; ++i, curArgBit <= 1)
151      {

```

```

152         if(s_ArgsMask & curArgBit)
153         {
154             fprintf(stderr, "%c %s", delim, s_Argv[i]);
155             delim = ',';
156         }
157     }
158     fprintf(stderr, "\n\n");
159     Usage(programName);
160 }
161
162 /*** main()
163 // This is the main entry point for the simulator.
164 // It registers the interface and starts listening for clients
165 int main(int argc, char* argv[])
166 {
167     bool manufacture = false;
168     bool pick_ports = false;
169     int PortNum      = DEFAULT_TPM_PORT;
170
171     // Parse command line options
172
173     if(CmdLineParser_Init(argc, argv, 2))
174     {
175         if(CmdLineParser_IsOptPresent("?", "?")
176             || CmdLineParser_IsOptPresent("help", "h"))
177         {
178             Usage(argv[0]);
179         }
180         if(CmdLineParser_IsOptPresent("manufacture", "m"))
181         {
182             manufacture = true;
183         }
184         if(CmdLineParser_IsOptPresent("pick_ports", "p"))
185         {
186             pick_ports = true;
187         }
188         if(CmdLineParser_More())
189         {
190             int i;
191             for(i = 0; i < s_Argc; ++i)
192             {
193                 char* nptr = NULL;
194                 int portNum = (int)strtol(s_Argv[i], &nptr, 0);
195                 if(s_Argv[i] != nptr)
196                 {
197                     // A numeric option is found
198                     if(!nptr && portNum > 0 && portNum < 65535)
199                     {
200                         PortNum = portNum;
201                         s_ArgsMask ^= 1 << i;
202                         break;
203                     }
204                     fprintf(stderr, "Invalid numeric option %s\n\n", s_Argv[i]);
205                     Usage(argv[0]);
206                 }
207             }
208         }
209         CmdLineParser_Done(argv[0]);
210     }
211     printf("LIBRARY_COMPATIBILITY_CHECK is %s\n",
212           (LIBRARY_COMPATIBILITY_CHECK ? "ON" : "OFF"));
213     // Enable NV memory
214     _plat__NVEnable(NULL, 0);
215
216     if(manufacture || _plat__NVNeedsManufacture())
217     {

```

```
218     printf("Manufacturing NV state...\n");
219     if (TPM_Manufacture(MANUF_FIRST_TIME) != MANUF_OK)
220     {
221         // if the manufacture didn't work, then make sure that the NV file doesn't
222         // survive. This prevents manufacturing failures from being ignored the
223         // next time the code is run.
224         _plat__NVDisable((void*)TRUE, 0);
225         exit(1);
226     }
227     // Coverage test - repeated manufacturing attempt
228     if (TPM_Manufacture(MANUF_REMANUFACTURE) != MANUF_ALREADY_DONE)
229     {
230         exit(2);
231     }
232     // Coverage test - re-manufacturing
233     TPM_TearDown();
234     if (TPM_Manufacture(MANUF_FIRST_TIME) != MANUF_OK)
235     {
236         exit(3);
237     }
238 }
239 // Disable NV memory
240 _plat__NVDisable((void*)FALSE, 0);
241
242 StartTcpServer(PortNum, pick_ports);
243 return EXIT_SUCCESS;
244 }
245
```