

Trusted Platform Module Library

Part 4: Supporting Routines

Family “2.0”

Level 00 Revision 01.64

December 24, 2020

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.

TCG PUBLIC REVIEW

Copyright © TCG 2006-2021

TCG

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	2
4.2.1 Introduction	2
4.2.2 Unmarshaling Code Prototype.....	2
4.2.2.1 Simple Types and Structures.....	2
4.2.2.2 Union Types.....	3
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	5
4.4 Function Prototypes.....	5
4.5 Portability.....	6
5 Header Files.....	7
5.1 Introduction	7
5.2 BaseType.h	7
5.3 Capabilities.h	8
5.4 CommandAttributeData.h.....	9
5.5 CommandAttributes.h	23
5.6 CommandDispatchData.h	24
5.7 Commands.h	105
5.8 CompilerDependencies.h	113
5.9 Global.h	115
5.9.1 Description	115
5.9.2 Includes.....	115
5.9.3 Defines and Types.....	115
5.9.3.1 Size Types.....	115
5.9.3.2 Other Types.....	115
5.9.4 Loaded Object Structures.....	116
5.9.4.1 Description	116
5.9.4.2 OBJECT_ATTRIBUTES.....	116
5.9.4.3 OBJECT Structure.....	117
5.9.4.4 HASH_OBJECT Structure	118
5.9.4.5 ANY_OBJECT.....	118
5.9.5 AUTH_DUP Types	118
5.9.6 Active Session Context	118
5.9.6.1 Description	118
5.9.6.2 SESSION_ATTRIBUTES	119
5.9.6.3 SESSION Structure.....	119

5.9.7	PCR	120
5.9.7.1	PCR_SAVE Structure	120
5.9.7.2	PCR_POLICY	121
5.9.7.3	PCR_AUTHVALUE	121
5.9.8	STARTUP_TYPE	121
5.9.9	NV	121
5.9.9.1	NV_INDEX	121
5.9.9.2	NV_REF	122
5.9.9.3	NV_PIN	122
5.9.10	COMMIT_INDEX_MASK	122
5.9.11	RAM Global Values	122
5.9.11.1	Description	122
5.9.11.2	Crypto Self-Test Values	123
5.9.11.3	g_rcIndex[]	123
5.9.11.4	g_exclusiveAuditSession	123
5.9.11.5	g_time	123
5.9.11.6	g_timeEpoch	123
5.9.11.7	g_phEnable	123
5.9.11.8	g_pcrReConfig	124
5.9.11.9	g_DRTMHandle	124
5.9.11.10	g_DrtmPreStartup	124
5.9.11.11	g_StartupLocality3	124
5.9.11.12	TPM_SU_NONE	124
5.9.11.13	TPM_SU_DA_USED	124
5.9.11.14	Startup Flags	124
5.9.11.15	g_daUsed	125
5.9.11.16	g_updateNV	125
5.9.11.17	g_powerWasLost	125
5.9.11.18	g_clearOrderly	125
5.9.11.19	g_prevOrderlyState	126
5.9.11.20	g_nvOk	126
5.9.11.21	g_platformUnique	126
5.9.12	Persistent Global Values	126
5.9.12.1	Description	126
5.9.12.2	PERSISTENT_DATA	126
5.9.12.3	ORDERLY_DATA	129
5.9.12.4	STATE_CLEAR_DATA	130
5.9.12.5	State Reset Data	130
5.9.13	NV Layout	132
5.9.14	Global Macro Definitions	132
5.9.15	From CryptTest.c	134
5.9.16	From Manufacture.c	134
5.9.17	Private data	134
5.9.17.1	From SessionProcess.c	134
5.9.17.2	From DA.c	135
5.9.17.3	From NV.c	135
5.9.17.4	From Object.c	136
5.9.17.5	From PCR.c	136
5.9.17.6	From Session.c	137
5.9.17.7	From IoBuffers.c	137
5.9.17.8	From TPMFail.c	137
5.9.17.9	From ACT_spt.c	138
5.9.17.10	From CommandCodeAttributes.c	138

5.10	GpMacros.h	139
5.10.1	Introduction	139
5.10.2	For Self-test	139
5.10.3	For Failures	139
5.10.4	Derived from Vendor-specific values	140
5.10.5	Compile-time Checks	140
5.11	InternalRoutines.h	145
5.12	LibSupport.h	147
5.13	MinMax.h	147
5.14	NV.h	148
5.14.1	Index Type Definitions	148
5.14.2	Attribute Macros	148
5.14.3	Orderly RAM Values	149
5.15	TPMB.h	151
5.16	Tpm.h	152
5.17	TpmBuildSwitches.h	153
5.18	TpmError.h	159
5.19	TpmTypes.h	160
5.20	VendorString.h	199
5.21	swap.h	200
5.22	ACT.h	202
6	Main	206
6.1	Introduction	206
6.2	ExecCommand.c	206
6.2.1	Introduction	206
6.2.2	Includes	206
6.2.3	ExecuteCommand()	206
6.3	CommandDispatcher.c	211
6.3.1	Introduction	211
6.3.2	Includes and Typedefs	211
6.3.3	Marshal/Unmarshal Functions	213
6.3.3.1	ParseHandleBuffer()	213
6.3.3.2	CommandDispatcher()	214
6.4	SessionProcess.c	219
6.4.1	Introduction	219
6.4.2	Includes and Data Definitions	219
6.4.3	Authorization Support Functions	219
6.4.3.1	IsDAExempted()	219
6.4.3.2	IncrementLockout()	220
6.4.3.3	IsSessionBindEntity()	221
6.4.3.4	IsPolicySessionRequired()	222
6.4.3.5	IsAuthValueAvailable()	223
6.4.3.6	IsAuthPolicyAvailable()	225
6.4.4	Session Parsing Functions	226
6.4.4.1	ClearCpRpHashes()	226
6.4.4.2	GetCpHashPointer()	227
6.4.4.3	GetRpHashPointer()	227
6.4.4.4	ComputeCpHash()	228
6.4.4.5	GetCpHash()	228
6.4.4.6	CompareTemplateHash()	229
6.4.4.7	CompareNameHash()	230

6.4.4.8	CheckPWAAuthSession()	230
6.4.4.9	ComputeCommandHMAC()	231
6.4.4.10	CheckSessionHMAC()	232
6.4.4.11	CheckPolicyAuthSession()	233
6.4.4.12	RetrieveSessionData()	235
6.4.4.13	CheckLockedOut()	238
6.4.4.14	CheckAuthSession()	239
6.4.4.15	CheckCommandAudit()	242
6.4.4.16	ParseSessionBuffer()	242
6.4.4.17	CheckAuthNoSession()	245
6.4.5	Response Session Processing	245
6.4.5.1	Introduction	245
6.4.5.2	ComputeRpHash()	245
6.4.5.3	InitAuditSession()	246
6.4.5.4	UpdateAuditDigest()	246
6.4.5.5	Audit()	247
6.4.5.6	CommandAudit()	247
6.4.5.7	UpdateAuditSessionStatus()	248
6.4.5.8	ComputeResponseHMAC()	249
6.4.5.9	UpdateInternalSession()	250
6.4.5.10	BuildSingleResponseAuth()	250
6.4.5.11	UpdateAllNonceTPM()	251
6.4.5.12	BuildResponseSession()	251
6.4.5.13	SessionRemoveAssociationToHandle()	253
7	Command Support Functions	254
7.1	Introduction	254
7.2	Attestation Command Support (Attest_spt.c)	254
7.2.1	Includes	254
7.2.2	Functions	254
7.2.2.1	FillInAttestInfo()	254
7.2.2.2	SignAttestInfo()	255
7.2.2.3	IsSigningObject()	256
7.3	Context Management Command Support (Context_spt.c)	257
7.3.1	Includes	257
7.3.2	Functions	257
7.3.2.1	ComputeContextProtectionKey()	257
7.3.2.2	ComputeContextIntegrity()	258
7.3.2.3	SequenceDataExport();	259
7.3.2.4	SequenceDataImport();	259
7.4	Policy Command Support (Policy_spt.c)	260
7.4.1	Includes	260
7.4.2	Functions	260
7.4.2.1	PolicyParameterChecks()	260
7.4.2.2	PolicyContextUpdate()	261
7.4.2.3	ComputeAuthTimeout()	262
7.4.2.4	PolicyDigestClear()	262
7.4.2.5	PolicySptCheckCondition()	263
7.5	NV Command Support (NV_spt.c)	265
7.5.1	Includes	265
7.5.2	Functions	265
7.5.2.1	NvReadAccessChecks()	265

7.5.2.2	NvWriteAccessChecks()	266
7.5.2.3	NvClearOrderly()	266
7.5.2.4	NvIsPinPassIndex()	267
7.6	Object Command Support (Object_spt.c)	268
7.6.1	Includes	268
7.6.2	Local Functions	268
7.6.2.1	GetIV2BSize()	268
7.6.2.2	ComputeProtectionKeyParms()	268
7.6.2.3	ComputeOuterIntegrity()	269
7.6.2.4	ComputeInnerIntegrity()	270
7.6.2.5	ProduceInnerIntegrity()	270
7.6.2.6	CheckInnerIntegrity()	271
7.6.3	Public Functions	272
7.6.3.1	AdjustAuthSize()	272
7.6.3.2	AreAttributesForParent()	272
7.6.3.3	CreateChecks()	272
7.6.3.4	SchemeChecks()	273
7.6.3.5	PublicAttributesValidation()	277
7.6.3.6	FillInCreationData()	278
7.6.3.7	GetSeedForKDF()	279
7.6.3.8	ProduceOuterWrap()	280
7.6.3.9	UnwrapOuter()	281
7.6.3.10	MarshalSensitive()	282
7.6.3.11	SensitiveToPrivate()	283
7.6.3.12	PrivateToSensitive()	284
7.6.3.13	SensitiveToDuplicate()	285
7.6.3.14	DuplicateToSensitive()	287
7.6.3.15	SecretToCredential()	289
7.6.3.16	CredentialToSecret()	290
7.6.3.17	MemoryRemoveTrailingZeros()	291
7.6.3.18	SetLabelAndContext()	291
7.6.3.19	UnmarshalToPublic()	292
7.6.3.20	ObjectSetExternal()	292
7.7	Encrypt Decrypt Support (EncryptDecrypt_spt.c)	293
7.8	ACT Support (ACT_spt.c)	295
7.8.1	Introduction	295
7.8.2	Includes	295
7.8.3	Functions	295
7.8.3.1	_ActResume()	295
7.8.3.2	_ActStartup()	295
7.8.3.3	_ActSaveState()	296
7.8.3.4	_ActGetSignaled()	296
7.8.3.5	ActShutdown()	297
7.8.3.6	ActIsImplemented()	297
7.8.3.7	ActCounterUpdate()	297
7.8.3.8	ActGetCapabilityData()	298
8	Subsystem	300
8.1	CommandAudit.c	300
8.1.1	Introduction	300
8.1.2	Includes	300
8.1.3	Functions	300
8.1.3.1	CommandAuditPreInstall_Init()	300

8.1.3.2	CommandAuditStartup()	300
8.1.3.3	CommandAuditSet()	301
8.1.3.4	CommandAuditClear()	301
8.1.3.5	CommandAuditIsRequired()	302
8.1.3.6	CommandAuditCapGetCCList()	302
8.1.3.7	CommandAuditGetDigest()	303
8.2	DA.c	305
8.2.1	Introduction	305
8.2.2	Includes and Data Definitions	305
8.2.3	Functions	305
8.2.3.1	DAPreInstall_Init()	305
8.2.3.2	DAStartup()	305
8.2.3.3	DARegisterFailure()	306
8.2.3.4	DASelfHeal()	307
8.3	Hierarchy.c	309
8.3.1	Introduction	309
8.3.2	Includes	309
8.3.3	Functions	309
8.3.3.1	HierarchyPreInstall()	309
8.3.3.2	HierarchyStartup()	310
8.3.3.3	HierarchyGetProof()	310
8.3.3.4	HierarchyGetPrimarySeed()	311
8.3.3.5	HierarchyIsEnabled()	311
8.4	NvDynamic.c	313
8.4.1	Introduction	313
8.4.2	Includes, Defines and Data Definitions	313
8.4.3	Local Functions	313
8.4.3.1	NvNext()	313
8.4.3.2	NvNextByType()	314
8.4.3.3	NvNextIndex()	314
8.4.3.4	NvNextEvict()	315
8.4.3.5	NvGetEnd()	315
8.4.3.6	NvGetFreeBytes()	315
8.4.3.7	NvTestSpace()	315
8.4.3.8	NvWriteNvListEnd()	316
8.4.3.9	NvAdd()	317
8.4.3.10	NvDelete()	318
8.4.4	RAM-based NV Index Data Access Functions	319
8.4.4.1	Introduction	319
8.4.4.2	NvRamNext()	319
8.4.4.3	NvRamGetEnd()	319
8.4.4.4	NvRamTestSpaceIndex()	320
8.4.4.5	NvRamGetIndex()	320
8.4.4.6	NvUpdateIndexOrderlyData()	320
8.4.4.7	NvAddRAM()	321
8.4.4.8	NvDeleteRAM()	321
8.4.4.9	NvReadIndex()	322
8.4.4.10	NvReadObject()	322
8.4.4.11	NvFindEvict()	322
8.4.4.12	NvIndexIsDefined()	323
8.4.4.13	NvConditionallyWrite()	323
8.4.4.14	NvReadNvIndexAttributes()	324
8.4.4.15	NvReadRamIndexAttributes()	324

8.4.4.16	NvWriteNvIndexAttributes()	324
8.4.4.17	NvWriteRamIndexAttributes()	324
8.4.5	Externally Accessible Functions	325
8.4.5.1	NvIsPlatformPersistentHandle()	325
8.4.5.2	NvIsOwnerPersistentHandle()	325
8.4.5.3	NvIndexIsAccessible()	325
8.4.5.4	NvGetEvictObject()	327
8.4.5.5	NvIndexCacheInit()	327
8.4.5.6	NvGetIndexData()	327
8.4.5.7	NvHashIndexData()	328
8.4.5.8	NvGetUINT64Data()	329
8.4.5.9	NvWriteIndexAttributes()	329
8.4.5.10	NvWriteIndexAuth()	329
8.4.5.11	NvGetIndexInfo()	330
8.4.5.12	NvWriteIndexData()	330
8.4.5.13	NvWriteUINT64Data()	332
8.4.5.14	NvGetIndexName()	332
8.4.5.15	NvGetNameByIndexHandle()	333
8.4.5.16	NvDefineIndex()	333
8.4.5.17	NvAddEvictObject()	334
8.4.5.18	NvDeleteIndex()	335
8.4.5.19	NvDeleteEvict()	335
8.4.5.20	NvFlushHierarchy()	336
8.4.5.21	NvSetGlobalLock()	337
8.4.5.22	InsertSort()	338
8.4.5.23	NvCapGetPersistent()	338
8.4.5.24	NvCapGetIndex()	339
8.4.5.25	NvCapGetIndexNumber()	340
8.4.5.26	NvCapGetPersistentNumber()	340
8.4.5.27	NvCapGetPersistentAvail()	341
8.4.5.28	NvCapGetCounterNumber()	341
8.4.5.29	NvSetStartupAttributes()	341
8.4.5.30	NvEntityStartup()	342
8.4.5.31	NvCapGetCounterAvail()	343
8.4.5.32	NvFindHandle()	344
8.4.6	NV Max Counter	344
8.4.6.1	Introduction	344
8.4.6.2	NvReadMaxCount()	344
8.4.6.3	NvUpdateMaxCount()	344
8.4.6.4	NvSetMaxCount()	345
8.4.6.5	NvGetMaxCount()	345
8.5	NvReserved.c	346
8.5.1	Introduction	346
8.5.2	Includes, Defines	346
8.5.3	Functions	346
8.5.3.1	NvInitStatic()	346
8.5.3.2	NvCheckState()	347
8.5.3.3	NvCommit()	347
8.5.3.4	NvPowerOn()	347
8.5.3.5	NvManufacture()	348
8.5.3.6	NvRead()	348
8.5.3.7	NvWrite()	348
8.5.3.8	NvUpdatePersistent()	349
8.5.3.9	NvClearPersistent()	349
8.5.3.10	NvReadPersistent()	349

8.6	Object.c	350
8.6.1	Introduction	350
8.6.2	Includes and Data Definitions	350
8.6.3	Functions	350
8.6.3.1	ObjectFlush()	350
8.6.3.2	ObjectSetInUse()	350
8.6.3.3	ObjectStartup()	350
8.6.3.4	ObjectCleanupEvict()	351
8.6.3.5	IsObjectPresent()	351
8.6.3.6	ObjectIsSequence()	352
8.6.3.7	HandleToObject()	352
8.6.3.8	GetQualifiedName()	352
8.6.3.9	ObjectGetHierarchy()	353
8.6.3.10	GetHierarchy()	353
8.6.3.11	FindEmptyObjectSlot()	354
8.6.3.12	ObjectAllocateSlot()	354
8.6.3.13	ObjectSetLoadedAttributes()	354
8.6.3.14	ObjectLoad()	356
8.6.3.15	AllocateSequenceSlot()	357
8.6.3.16	ObjectCreateHMACSequence()	358
8.6.3.17	ObjectCreateHashSequence()	359
8.6.3.18	ObjectCreateEventSequence()	359
8.6.3.19	ObjectTerminateEvent()	359
8.6.3.20	ObjectContextLoad()	360
8.6.3.21	FlushObject()	361
8.6.3.22	ObjectFlushHierarchy()	361
8.6.3.23	ObjectLoadEvict()	362
8.6.3.24	ObjectComputeName()	362
8.6.3.25	PublicMarshalAndComputeName()	363
8.6.3.26	ComputeQualifiedName()	363
8.6.3.27	ObjectIsStorage()	364
8.6.3.28	ObjectCapGetLoaded()	364
8.6.3.29	ObjectCapGetTransientAvail()	365
8.6.3.30	ObjectGetPublicAttributes()	366
8.7	PCR.c	367
8.7.1	Introduction	367
8.7.2	Includes, Defines, and Data Definitions	367
8.7.3	Functions	367
8.7.3.1	PCRBelongsAuthGroup()	367
8.7.3.2	PCRBelongsPolicyGroup()	368
8.7.3.3	PCRBelongsTCBGroup()	368
8.7.3.4	PCRPolicyIsAvailable()	369
8.7.3.5	PCRGetAuthValue()	369
8.7.3.6	PCRGetAuthPolicy()	370
8.7.3.7	PCRSimStart()	370
8.7.3.8	GetSavedPcrPointer()	371
8.7.3.9	PcrIsAllocated()	371
8.7.3.10	GetPcrPointer()	372
8.7.3.11	IsPcrSelected()	372
8.7.3.12	FilterPcr()	373
8.7.3.13	PcrDrtm()	373
8.7.3.14	PCR_ClearAuth()	374
8.7.3.15	PCRStartup()	374
8.7.3.16	PCRStateSave()	376
8.7.3.17	PCRIsStateSaved()	376
8.7.3.18	PCRIsResetAllowed()	377

8.7.3.19	PCRChanged()	377
8.7.3.20	PCRIsExtendAllowed()	378
8.7.3.21	PCRExtend()	378
8.7.3.22	PCRComputeCurrentDigest()	379
8.7.3.23	PCRRead()	380
8.7.3.24	PCRAllocate()	381
8.7.3.25	PCRSetValue()	383
8.7.3.26	PCRResetDynamics()	383
8.7.3.27	PCRCapGetAllocation()	384
8.7.3.28	PCRSetSelectBit()	384
8.7.3.29	PCRGetProperty()	385
8.7.3.30	PCRCapGetProperties()	386
8.7.3.31	PCRCapGetHandles()	387
8.8	PP.c	389
8.8.1	Introduction	389
8.8.2	Includes	389
8.8.3	Functions	389
8.8.3.1	PhysicalPresencePreInstall_Init()	389
8.8.3.2	PhysicalPresenceCommandSet()	389
8.8.3.3	PhysicalPresenceCommandClear()	390
8.8.3.4	PhysicalPresencelsRequired()	390
8.8.3.5	PhysicalPresenceCapGetCCList()	390
8.9	Session.c	392
8.9.1	Introduction	392
8.9.2	Includes, Defines, and Local Variables	392
8.9.3	File Scope Function -- ContextIdSetOldest()	392
8.9.4	Startup Function -- SessionStartup()	392
8.9.5	Access Functions	393
8.9.5.1	SessionIsLoaded()	393
8.9.5.2	SessionIsSaved()	394
8.9.5.3	SequenceNumberForSavedContextIsValid()	394
8.9.5.4	SessionPCRValuesCurrent()	395
8.9.5.5	SessionGet()	395
8.9.6	Utility Functions	396
8.9.6.1	ContextIdSessionCreate()	396
8.9.6.2	SessionCreate()	397
8.9.6.3	SessionContextSave()	399
8.9.6.4	SessionContextLoad()	400
8.9.6.5	SessionFlush()	401
8.9.6.6	SessionComputeBoundEntity()	402
8.9.6.7	SessionSetStartTime()	403
8.9.6.8	SessionResetPolicyData()	403
8.9.6.9	SessionCapGetLoaded()	404
8.9.6.10	SessionCapGetSaved()	405
8.9.6.11	SessionCapGetLoadedNumber()	406
8.9.6.12	SessionCapGetLoadedAvail()	406
8.9.6.13	SessionCapGetActiveNumber()	406
8.9.6.14	SessionCapGetActiveAvail()	406
8.10	Time.c	408
8.10.1	Introduction	408
8.10.2	Includes	408
8.10.3	Functions	408
8.10.3.1	TimePowerOn()	408

8.10.3.2	TimeNewEpoch()	408
8.10.3.3	TimeStartup()	408
8.10.3.4	TimeClockUpdate()	409
8.10.3.5	TimeUpdate()	409
8.10.3.6	TimeUpdateToCurrent()	410
8.10.3.7	TimeSetAdjustRate()	410
8.10.3.8	TimeGetMarshaled()	411
8.10.3.9	TimeFillInfo()	411
9	Support	413
9.1	AlgorithmCap.c	413
9.1.1	Description	413
9.1.2	Includes and Defines	413
9.1.3	AlgorithmCapGetImplemented()	415
9.1.4	AlgorithmGetImplementedVector()	416
9.2	Bits.c	417
9.2.1	Introduction	417
9.2.2	Includes	417
9.2.3	Functions	417
9.2.3.1	TestBit()	417
9.2.3.2	SetBit()	417
9.2.3.3	ClearBit()	417
9.3	CommandCodeAttributes.c	419
9.3.1	Introduction	419
9.3.2	Includes and Defines	419
9.3.3	Command Attribute Functions	419
9.3.3.1	NextImplementedIndex()	419
9.3.3.2	GetClosestCommandIndex()	420
9.3.3.3	CommandCodeToComandIndex()	422
9.3.3.4	GetNextCommandIndex()	423
9.3.3.5	GetCommandCode()	423
9.3.3.6	CommandAuthRole()	424
9.3.3.7	EncryptSize()	424
9.3.3.8	DecryptSize()	425
9.3.3.9	IsSessionAllowed()	425
9.3.3.10	IsHandleInResponse()	425
9.3.3.11	IsWriteOperation()	425
9.3.3.12	IsReadOperation()	426
9.3.3.13	CommandCapGetCCList()	427
9.3.3.14	IsVendorCommand()	427
9.4	Entity.c	429
9.4.1	Description	429
9.4.2	Includes	429
9.4.3	Functions	429
9.4.3.1	EntityGetLoadStatus()	429
9.4.3.2	EntityGetAuthValue()	431
9.4.3.3	EntityGetAuthPolicy()	433
9.4.3.4	EntityGetName()	434
9.4.3.5	EntityGetHierarchy()	435
9.5	Global.c	437
9.5.1	Description	437
9.5.2	Defines and Includes	437

9.6	Handle.c.....	438
9.6.1	Description	438
9.6.2	Includes.....	438
9.6.3	Functions	438
9.6.3.1	HandleGetType().....	438
9.6.3.2	NextPermanentHandle()	438
9.6.3.3	PermanentCapGetHandles().....	439
9.6.3.4	PermanentHandleGetPolicy()	440
9.7	IoBuffers.c.....	441
9.7.1	Includes and Data Definitions.....	441
9.7.2	Buffers and Functions	441
9.7.2.1	MemoryIoBufferAllocationReset()	441
9.7.2.2	MemoryIoBufferZero()	441
9.7.2.3	MemoryGetInBuffer().....	441
9.7.2.4	MemoryGetOutBuffer()	442
9.7.2.5	IsLabelProperlyFormatted()	442
9.8	Locality.c.....	443
9.8.1	Includes.....	443
9.8.2	LocalityGetAttributes()	443
9.9	Manufacture.c	444
9.9.1	Description	444
9.9.2	Includes and Data Definitions.....	444
9.9.3	Functions	444
9.9.3.1	TPM_Manufacture()	444
9.9.3.2	TPM_TearDown().....	445
9.9.3.3	TpmEndSimulation().....	446
9.10	Marshal.c	447
9.10.1	Introduction	447
9.10.2	Unmarshal and Marshal a Value.....	447
9.10.3	Unmarshal and Marshal a Union	448
9.10.4	Unmarshal and Marshal a Structure	450
9.10.5	Unmarshal and Marshal an Array	451
9.10.6	TPM2B Handling.....	453
9.10.7	Table Marshal Headers	453
9.10.7.1	TableMarshal.h	453
9.10.7.2	TableMarshalDefines.h.....	458
9.10.7.3	TableMarshalTypes.h	480
9.10.8	Table Marshal Source	504
9.10.8.1	TableDrivenMarshal.c	504
9.10.8.2	TableMarshalData.c	521
9.11	MathOnByteBuffers.c.....	543
9.11.1	Introduction	543
9.11.2	Functions	543
9.11.2.1	UnsignedCmpB().....	543
9.11.2.2	SignedCompareB().....	543
9.11.2.3	ModExpB().....	544
9.11.2.4	DivideB().....	545
9.11.2.5	AdjustNumberB().....	546
9.11.2.6	ShiftLeft().....	546

9.12	Memory.c	548
9.12.1	Description	548
9.12.2	Includes and Data Definitions	548
9.12.3	Functions	548
9.12.3.1	MemoryCopy()	548
9.12.3.2	MemoryEqual()	548
9.12.3.3	MemoryCopy2B()	549
9.12.3.4	MemoryConcat2B()	549
9.12.3.5	MemoryEqual2B()	549
9.12.3.6	MemorySet()	550
9.12.3.7	MemoryPad2B()	550
9.12.3.8	UInt16ToByteArray()	550
9.12.3.9	UInt32ToByteArray()	550
9.12.3.10	UInt64ToByteArray()	551
9.12.3.11	ByteArrayToUInt8()	551
9.12.3.12	ByteArrayToUInt16()	551
9.12.3.13	ByteArrayToUInt32()	551
9.12.3.14	ByteArrayToUInt64()	552
9.13	Power.c	553
9.13.1	Description	553
9.13.2	Includes and Data Definitions	553
9.13.3	Functions	553
9.13.3.1	TPMInit()	553
9.13.3.2	TPMRegisterStartup()	553
9.13.3.3	TPMIsStarted()	553
9.14	PropertyCap.c	555
9.14.1	Description	555
9.14.2	Includes	555
9.14.3	Functions	555
9.14.3.1	TPMPropertyIsDefined()	555
9.14.3.2	TPMCapGetProperties()	562
9.15	Response.c	564
9.15.1	Description	564
9.15.2	Includes and Defines	564
9.15.3	BuildResponseHeader()	564
9.16	ResponseCodeProcessing.c	565
9.16.1	Description	565
9.16.2	Includes and Defines	565
9.16.3	RcSafeAddToResult()	565
9.17	TpmFail.c	566
9.17.1	Includes, Defines, and Types	566
9.17.2	Typedefs	566
9.17.3	Local Functions	567
9.17.3.1	MarshalUInt16()	567
9.17.3.2	MarshalUInt32()	567
9.17.3.3	Unmarshal32()	567
9.17.3.4	Unmarshal16()	568
9.17.4	Public Functions	568
9.17.4.1	SetForceFailureMode()	568

9.17.4.2	TpmLogFailure()	568
9.17.4.3	TpmFail()	569
9.17.4.4	TpmFailureMode()	569
9.17.4.5	UnmarshalFail()	572
10	Cryptographic Functions	573
10.1	Headers	573
10.1.1	BnValues.h	573
10.1.1.1	Introduction	573
10.1.1.2	Defines	573
10.1.2	CryptEcc.h	579
10.1.2.1	Introduction	579
10.1.2.2	Structures	579
10.1.2.2.1	Macros	579
10.1.3	CryptHash.h	580
10.1.3.1	Introduction	580
10.1.3.2	Hash-related Structures	580
10.1.3.3	HMAC State Structures	584
10.1.4	CryptRand.h	585
10.1.4.1	Introduction	585
10.1.4.2	DRBG Structures and Defines	585
10.1.5	CryptRsa.h	588
10.1.6	CryptTest.h	589
10.1.7	HashTestData.h	590
10.1.8	KdfTestData.h	592
10.1.9	RsaTestData.h	593
10.1.10	SelfTest.h	599
10.1.10.1	Introduction	599
10.1.10.2	Defines	599
10.1.11	SupportLibraryFunctionPrototypes_fp.h	600
10.1.11.1	Introduction	600
10.1.11.2	SupportLibInit()	600
10.1.11.3	MathLibraryCompatibilityCheck()	600
10.1.11.4	BnModMult()	600
10.1.11.5	BnMult()	601
10.1.11.6	BnDiv()	601
10.1.11.7	BnMod()	601
10.1.11.8	BnGcd()	601
10.1.11.9	BnModExp()	601
10.1.11.10	BnModInverse()	601
10.1.11.11	BnEccModMult()	601
10.1.11.12	BnEccModMult2()	602
10.1.11.13	BnEccAdd()	602
10.1.11.14	BnCurveInitialize()	602
10.1.11.14.1	BnCurveFree()	602
10.1.12	SymmetricTestData.h	603
10.1.13	SymmetricTest.h	606
10.1.13.1	Introduction	606
10.1.13.2	Symmetric Test Structures	606
10.1.14	EccTestData.h	607

10.1.15 CryptSym.h.....	609
10.1.15.1 Introduction.....	609
10.1.15.2 Includes, Defines, and Typedefs.....	609
10.1.16 OIDs.h.....	611
10.1.17 PRNG_TestVectors.h.....	615
10.1.18 TpmAsn1.h.....	616
10.1.18.1 Introduction.....	616
10.1.18.2 Includes.....	616
10.1.18.3 Defined Constants.....	616
10.1.18.3.1 ASN.1 Universal Types (Class 00b).....	616
10.1.18.4 Macros.....	616
10.1.18.4.1 Unmarshaling Macros.....	616
10.1.18.4.2 Marshaling Macros.....	617
10.1.18.5 Structures.....	617
10.1.19 X509.h.....	617
10.1.19.1 Introduction.....	617
10.1.19.2 Includes.....	617
10.1.19.3 Defined Constants.....	618
10.1.19.3.1 X509 Application-specific types.....	618
10.1.19.4 Structures.....	618
10.1.19.5 Global X509 Constants.....	618
10.1.20 TpmAlgorithmDefines.h.....	619
10.2 Source.....	626
10.2.1 AlgorithmTests.c.....	626
10.2.1.1 Introduction.....	626
10.2.1.2 Includes and Defines.....	626
10.2.1.3 Hash Tests.....	626
10.2.1.3.1 Description.....	626
10.2.1.3.2 TestHash().....	627
10.2.1.4 Symmetric Test Functions.....	627
10.2.1.4.1 Makelv().....	627
10.2.1.4.2 TestSymmetricAlgorithm().....	628
10.2.1.4.3 AllSymsAreDone().....	628
10.2.1.4.4 AllModesAreDone().....	629
10.2.1.4.5 TestSymmetric().....	629
10.2.1.5 RSA Tests.....	630
10.2.1.5.1 Introduction.....	630
10.2.1.5.2 RsaKeyInitialize().....	631
10.2.1.5.3 TestRsaEncryptDecrypt().....	631
10.2.1.5.4 TestRsaSignAndVerify().....	632
10.2.1.5.5 TestRSA().....	634
10.2.1.6 ECC Tests.....	634
10.2.1.6.1 LoadEccParameter().....	635
10.2.1.6.2 LoadEccPoint().....	635
10.2.1.6.3 TestECDH().....	635
10.2.1.6.4 TestEccSignAndVerify().....	635
10.2.1.6.5 TestKDFa().....	637

10.2.1.6.6 TestEcc()	637
10.2.1.6.7 TestAlgorithm()	638
10.2.2 BnConvert.c	641
10.2.2.1 Introduction	641
10.2.2.2 Includes	641
10.2.2.3 Functions	641
10.2.2.3.1 BnFromBytes()	641
10.2.2.3.2 BnFrom2B()	642
10.2.2.3.3 BnFromHex()	642
10.2.2.3.4 BnToBytes()	643
10.2.2.3.5 BnTo2B()	644
10.2.2.3.6 BnPointFrom2B()	644
10.2.2.3.7 BnPointTo2B()	644
10.2.3 BnMath.c	646
10.2.3.1 Introduction	646
10.2.3.2 Includes	646
10.2.3.3 Functions	646
10.2.3.3.1 AddSame()	646
10.2.3.3.2 CarryProp()	647
10.2.3.3.3 BnAdd()	647
10.2.3.3.4 BnAddWord()	648
10.2.3.3.5 SubSame()	648
10.2.3.3.6 BorrowProp()	649
10.2.3.3.7 BnSub()	649
10.2.3.3.8 BnSubWord()	649
10.2.3.3.9 BnUnsignedCmp()	650
10.2.3.3.10 BnUnsignedCmpWord()	650
10.2.3.3.11 BnModWord()	651
10.2.3.3.12 Msb()	651
10.2.3.3.13 BnMsb()	651
10.2.3.3.14 BnSizeInBits()	652
10.2.3.3.15 BnSetWord()	652
10.2.3.3.16 BnSetBit()	652
10.2.3.3.17 BnTestBit()	653
10.2.3.3.18 BnMaskBits()	653
10.2.3.3.19 BnShiftRight()	654
10.2.3.3.20 BnGetRandomBits()	654
10.2.3.3.21 BnGenerateRandomInRange()	655
10.2.4 BnMemory.c	657
10.2.4.1 Introduction	657
10.2.4.2 Includes	657
10.2.4.3 Functions	657
10.2.4.3.1 BnSetTop()	657
10.2.4.3.2 BnClearTop()	657
10.2.4.3.3 BnInitializeWord()	658
10.2.4.3.4 BnInit()	658
10.2.4.3.5 BnCopy()	658
10.2.4.3.6 BnPointCopy()	659
10.2.4.3.7 BnInitializePoint()	659
10.2.5 CryptCmac.c	660
10.2.5.1 Introduction	660
10.2.5.2 Includes, Defines, and Typedefs	660
10.2.5.3 Functions	660

10.2.5.3.1 CryptCmacStart()	660
10.2.5.3.2 CryptCmacData()	660
10.2.5.3.3 CryptCmacEnd()	661
10.2.6 CryptUtil.c	663
10.2.6.1 Introduction	663
10.2.6.2 Includes	663
10.2.6.3 Hash/HMAC Functions	663
10.2.6.3.1 CryptHmacSign()	663
10.2.6.3.2 CryptHMACVerifySignature()	663
10.2.6.3.3 CryptGenerateKeyedHash()	664
10.2.6.3.4 CryptIsSchemeAnonymous()	665
10.2.6.4 Symmetric Functions	665
10.2.6.4.1 ParmDecryptSym()	665
10.2.6.4.2 ParmEncryptSym()	666
10.2.6.4.3 CryptGenerateKeySymmetric()	667
10.2.6.4.4 CryptXORObfuscation()	668
10.2.6.5 Initialization and shut down	668
10.2.6.5.1 CryptInit()	668
10.2.6.5.2 CryptStartup()	669
10.2.6.6 Algorithm-Independent Functions	670
10.2.6.6.1 Introduction	670
10.2.6.6.2 CryptIsAsymAlgorithm()	670
10.2.6.6.3 CryptSecretEncrypt()	670
10.2.6.6.4 CryptSecretDecrypt()	672
10.2.6.6.5 CryptParameterEncryption()	675
10.2.6.6.6 CryptParameterDecryption()	676
10.2.6.6.7 CryptComputeSymmetricUnique()	677
10.2.6.6.8 CryptCreateObject()	678
10.2.6.6.9 CryptGetSignHashAlg()	680
10.2.6.6.10 CryptIsSplitSign()	681
10.2.6.6.11 CryptIsAsymSignScheme()	681
10.2.6.6.12 CryptIsAsymDecryptScheme()	682
10.2.6.6.13 CryptSelectSignScheme()	683
10.2.6.6.14 CryptSign()	685
10.2.6.6.15 CryptValidateSignature()	686
10.2.6.6.16 CryptGetTestResult()	687
10.2.6.6.17 CryptValidateKeys()	687
10.2.6.6.18 CryptSelectMac()	690
10.2.6.6.19 CryptMacIsValidForKey()	691
10.2.6.6.20 CryptSmacIsValidAlg()	691
10.2.6.6.21 CryptSymModelsValid()	692
10.2.7 CryptSelfTest.c	693
10.2.7.1 Introduction	693
10.2.7.2 Functions	693
10.2.7.2.1 RunSelfTest()	693
10.2.7.2.2 CryptSelfTest()	693
10.2.7.2.3 CryptIncrementalSelfTest()	694
10.2.7.2.4 CryptInitializeToTest()	695
10.2.7.2.5 CryptTestAlgorithm()	695
10.2.8 CryptEccData.c	697
10.2.9 CryptDes.c	707

10.2.9.1	Introduction.....	707
10.2.9.2	Includes, Defines, and Typedefs.....	707
10.2.9.2.1	CryptSetOddByteParity()	707
10.2.9.2.2	CryptDesIsWeakKey()	708
10.2.9.2.3	CryptDesValidateKey()	708
10.2.9.2.4	CryptGenerateKeyDes()	709
10.2.10	CryptEccKeyExchange.c	710
10.2.10.1	Introduction.....	710
10.2.10.2	Functions	710
10.2.10.2.1	avf1()	710
10.2.10.2.2	C_2_2_MQV()	710
10.2.10.2.3	C_2_2_ECDH()	712
10.2.10.2.4	CryptEcc2PhaseKeyExchange().....	712
10.2.10.2.5	ComputeWForSM2().....	713
10.2.10.2.6	avfSm2()	714
10.2.10.2.7	SM2KeyExchange().....	714
10.2.11	CryptEccMain.c	716
10.2.11.1	Includes and Defines	716
10.2.11.2	Functions	716
10.2.11.2.1	CryptEccInit()	716
10.2.11.2.2	CryptEccStartup().....	716
10.2.11.2.3	ClearPoint2B(generic)	716
10.2.11.2.4	CryptEccGetParametersByCurveId()	717
10.2.11.2.5	CryptEccGetKeySizeForCurve()	717
10.2.11.2.6	GetCurveData().....	717
10.2.11.2.7	CryptEccGetOID()	718
10.2.11.2.8	CryptEccGetCurveByIndex()	718
10.2.11.2.9	CryptEccGetParameter()	718
10.2.11.2.10	CryptCapGetECCCurve().....	719
10.2.11.2.11	CryptGetCurveSignScheme()	720
10.2.11.2.12	CryptGenerateR()	720
10.2.11.2.13	CryptCommit()	722
10.2.11.2.14	CryptEndCommit()	722
10.2.11.2.15	CryptEccGetParameters()	722
10.2.11.2.16	BnGetCurvePrime()	723
10.2.11.2.17	BnGetCurveOrder()	723
10.2.11.2.18	BnIsOnCurve()	723
10.2.11.2.19	BnIsValidPrivateEcc()	724
10.2.11.2.20	BnPointMul()	724
10.2.11.2.21	BnEccGetPrivate().....	725
10.2.11.2.22	BnEccGenerateKeyPair().....	726
10.2.11.2.23	CryptEccNewKeyPair(**)	726
10.2.11.2.24	CryptEccPointMultiply()	727
10.2.11.2.25	CryptEccIsPointOnCurve()	728
10.2.11.2.26	CryptEccGenerateKey()	728
10.2.12	CryptEccSignature.c	730
10.2.12.1	Includes and Defines	730
10.2.12.2	Utility Functions	730
10.2.12.2.1	EcdsaDigest()	730
10.2.12.2.2	BnSchnorrSign().....	730
10.2.12.3	Signing Functions	731
10.2.12.3.1	BnSignEcdsa()	731

10.2.12.3.2	BnSignEcdaa()	732
10.2.12.3.3	SchnorrReduce()	734
10.2.12.3.4	SchnorrEcc()	734
10.2.12.3.5	BnHexEqual()	735
10.2.12.3.6	BnSignEcSm2()	736
10.2.12.3.7	CryptEccSign()	737
10.2.12.3.8	BnValidateSignatureEcdsa()	739
10.2.12.3.9	BnValidateSignatureEcSm2()	740
10.2.12.3.10	BnValidateSignatureEcSchnorr()	741
10.2.12.3.11	CryptEccValidateSignature()	742
10.2.12.3.12	CryptEccCommitCompute()	743
10.2.13	CryptHash.c	745
10.2.13.1	Description	745
10.2.13.2	Includes, Defines, and Types	745
10.2.13.3	Obligatory Initialization Functions	745
10.2.13.3.1	CryptHashInit()	745
10.2.13.3.2	CryptHashStartup()	745
10.2.13.4	Hash Information Access Functions	746
10.2.13.4.1	Introduction	746
10.2.13.4.2	CryptGetHashDef()	746
10.2.13.4.3	CryptHashIsValidAlg()	746
10.2.13.4.4	CryptHashGetAlgByIndex()	746
10.2.13.4.5	CryptHashGetDigestSize()	747
10.2.13.4.6	CryptHashGetBlockSize()	747
10.2.13.4.7	CryptHashGetOid()	747
10.2.13.4.8	CryptHashGetContextAlg()	748
10.2.13.5	State Import and Export	748
10.2.13.5.1	CryptHashCopyState()	748
10.2.13.5.2	CryptHashExportState()	748
10.2.13.5.3	CryptHashImportState()	749
10.2.13.6	State Modification Functions	750
10.2.13.6.1	HashEnd()	750
10.2.13.6.2	CryptHashStart()	750
10.2.13.6.3	CryptDigestUpdate()	751
10.2.13.6.4	CryptHashEnd()	751
10.2.13.6.5	CryptHashBlock()	752
10.2.13.6.6	CryptDigestUpdate2B()	752
10.2.13.6.7	CryptHashEnd2B()	753
10.2.13.6.8	CryptDigestUpdateInt()	753
10.2.13.7	HMAC Functions	753
10.2.13.7.1	CryptHmacStart()	753
10.2.13.7.2	CryptHmacEnd()	754
10.2.13.7.3	CryptHmacStart2B()	755
10.2.13.7.4	CryptHmacEnd2B()	756
10.2.13.8	Mask and Key Generation Functions	756
10.2.13.8.1	CryptMGF_KDF()	756
10.2.13.8.2	CryptKDFa()	757
10.2.13.8.3	CryptKDFe()	758
10.2.14	CryptPrime.c	761
10.2.14.1	Introduction	761
10.2.14.2	Functions	761

10.2.14.2.1	Root2()	761
10.2.14.2.2	IsPrimeInt()	761
10.2.14.2.3	BnIsProbablyPrime()	762
10.2.14.2.4	MillerRabinRounds()	763
10.2.14.2.5	MillerRabin()	763
10.2.14.2.6	RsaCheckPrime()	764
10.2.14.2.7	RsaAdjustPrimeCandidate()	765
10.2.14.2.8	BnGeneratePrimeForRSA()	766
10.2.15	CryptPrimeSieve.c	767
10.2.15.1	Includes and defines	767
10.2.15.2	Functions	767
10.2.15.2.1	RsaAdjustPrimeLimit()	767
10.2.15.2.2	RsaNextPrime()	767
10.2.15.2.3	BitsInArray()	769
10.2.15.2.4	FindNthSetBit()	769
10.2.15.2.5	PrimeSieve()	770
10.2.15.2.6	SetFieldSize()	772
10.2.15.2.7	PrimeSelectWithSieve()	772
10.2.15.2.8	PrintTuple()	774
10.2.15.2.9	RsaSimulationEnd()	774
10.2.15.2.10	GetSieveStats()	774
10.2.15.2.11	RsaSimulationEnd()	775
10.2.16	CryptRand.c	776
10.2.16.1	Introduction	776
10.2.16.2	Derivation Functions	776
10.2.16.2.1	Description	776
10.2.16.2.2	Derivation Function Defines and Structures	777
10.2.16.2.3	DfCompute()	777
10.2.16.2.4	DfStart()	777
10.2.16.2.5	DfUpdate()	778
10.2.16.2.6	DfEnd()	778
10.2.16.2.7	DfBuffer()	779
10.2.16.2.8	DRBG_GetEntropy()	779
10.2.16.2.9	IncrementIv()	780
10.2.16.2.10	EncryptDRBG()	780
10.2.16.2.11	DRBG_Update()	781
10.2.16.2.12	DRBG_Reseed()	782
10.2.16.2.13	DRBG_SelfTest()	783
10.2.16.3	Public Interface	784
10.2.16.3.1	Description	784
10.2.16.3.2	CryptRandomStir()	784
10.2.16.3.3	CryptRandomGenerate()	785
10.2.16.3.4	DRBG_InstantiateSeededKdf()	785
10.2.16.3.5	DRBG_AdditionalData()	786
10.2.16.3.6	DRBG_InstantiateSeeded()	786
10.2.16.3.7	CryptRandStartup()	787
10.2.16.3.8	CryptRandInit()	787
10.2.16.3.9	DRBG_Generate()	788
10.2.16.3.10	DRBG_Instantiate()	790
10.2.16.3.11	DRBG_Uninstantiate()	791
10.2.17	CryptRsa.c	792
10.2.17.1	Introduction	792
10.2.17.2	Includes	792

10.2.17.3	Obligatory Initialization Functions	792
10.2.17.3.1	CryptRsaInit()	792
10.2.17.3.2	CryptRsaStartup()	792
10.2.17.4	Internal Functions	792
10.2.17.4.1	RsaInitializeExponent()	792
10.2.17.4.2	MakePgreaterThanQ()	793
10.2.17.4.3	PackExponent()	793
10.2.17.4.4	UnpackExponent()	794
10.2.17.4.5	ComputePrivateExponent()	794
10.2.17.4.6	RsaPrivateKeyOp()	795
10.2.17.4.7	RSAEP()	795
10.2.17.4.8	RSADP()	796
10.2.17.4.9	OaepEncode()	797
10.2.17.4.10	OaepDecode()	798
10.2.17.4.11	PKCS1v1_5Encode()	799
10.2.17.4.12	RSASD_Decode()	800
10.2.17.4.13	CryptRsaPssSaltSize()	801
10.2.17.4.14	PssEncode()	801
10.2.17.4.15	PssDecode()	802
10.2.17.4.16	MakeDerTag()	804
10.2.17.4.17	RSASSA_Encode()	805
10.2.17.4.18	RSASSA_Decode()	806
10.2.17.5	Externally Accessible Functions	807
10.2.17.5.1	CryptRsaSelectScheme()	807
10.2.17.5.2	CryptRsaLoadPrivateExponent()	807
10.2.17.5.3	CryptRsaEncrypt()	808
10.2.17.5.4	CryptRsaDecrypt()	810
10.2.17.5.5	CryptRsaSign()	811
10.2.17.5.6	CryptRsaValidateSignature()	812
10.2.17.5.7	CryptRsaGenerateKey()	813
10.2.18	CryptSmac.c	816
10.2.18.1	Introduction	816
10.2.18.2	Includes, Defines, and Typedefs	816
10.2.18.2.1	CryptSmacStart()	816
10.2.18.2.2	CryptMacStart()	816
10.2.18.2.3	CryptMacEnd()	817
10.2.18.2.4	CryptMacEnd2B()	817
10.2.19	CryptSym.c	818
10.2.19.1	Introduction	818
10.2.19.2	Includes, Defines, and Typedefs	818
10.2.19.3	Initialization and Data Access Functions	818
10.2.19.3.1	CryptSymInit()	818
10.2.19.3.2	CryptSymStartup()	818
10.2.19.3.3	CryptGetSymmetricBlockSize()	818
10.2.19.4	Symmetric Encryption	819
10.2.19.4.1	CryptSymmetricDecrypt()	822
10.2.19.4.2	CryptSymKeyValidate()	824
10.2.20	PrimeData.c	826
10.2.21	RsaKeyCache.c	832
10.2.21.1	Introduction	832
10.2.21.2	Includes, Types, Locals, and Defines	832

10.2.21.2.1	RsaKeyCacheControl()	833
10.2.21.2.2	InitializeKeyCache()	833
10.2.21.2.3	KeyCacheLoaded()	834
10.2.21.2.4	GetCachedRsaKey()	835
10.2.22	Ticket.c	836
10.2.22.1	Introduction	836
10.2.22.2	Includes	836
10.2.22.3	Functions	836
10.2.22.3.1	TicketIsSafe()	836
10.2.22.3.2	TicketComputeVerified()	836
10.2.22.3.3	TicketComputeAuth()	837
10.2.22.3.4	TicketComputeHashCheck()	838
10.2.22.3.5	TicketComputeCreation()	838
10.2.23	TpmAsn1.c	840
10.2.23.1	Includes	840
10.2.23.2	Unmarshaling Functions	840
10.2.23.2.1	ASN1UnmarshalContextInitialize()	840
10.2.23.2.2	ASN1DecodeLength()	840
10.2.23.2.3	ASN1NextTag()	841
10.2.23.2.4	ASN1GetBitStringValue()	842
10.2.23.3	Marshaling Functions	843
10.2.23.3.1	Introduction	843
10.2.23.3.2	ASN1InitializeMarshalContext()	843
10.2.23.3.3	ASN1StartMarshalContext()	843
10.2.23.3.4	ASN1EndMarshalContext()	844
10.2.23.3.5	ASN1EndEncapsulation()	844
10.2.23.3.6	ASN1PushByte()	845
10.2.23.3.7	ASN1PushBytes()	845
10.2.23.3.8	ASN1PushNull()	845
10.2.23.3.9	ASN1PushLength()	846
10.2.23.3.10	ASN1PushTagAndLength()	846
10.2.23.3.11	ASN1PushTaggedOctetString()	847
10.2.23.3.12	ASN1PushUINT()	847
10.2.23.3.13	ASN1PushInteger()	847
10.2.23.3.14	ASN1PushOID()	848
10.2.24	X509_ECC.c	849
10.2.24.1	Includes	849
10.2.24.2	Functions	849
10.2.24.2.1	X509PushPoint()	849
10.2.24.2.2	X509AddSigningAlgorithmECC()	849
10.2.24.2.3	X509AddPublicECC()	850
10.2.25	X509_RSA.c	852
10.2.25.1	Includes	852
10.2.25.2	Functions	852
10.2.25.2.1	X509AddSigningAlgorithmRSA()	852
10.2.25.2.2	X509AddPublicRSA()	854
10.2.26	X509_spt.c	856
10.2.26.1	Includes	856
10.2.26.2	Unmarshaling Functions	856

10.2.26.2.1	X509FindExtensionByOID()	856
10.2.26.2.2	X509GetExtensionBits()	857
10.2.26.2.3	X509ProcessExtensions()	857
10.2.26.3	Marshaling Functions	859
10.2.26.3.1	X509AddSigningAlgorithm()	859
10.2.26.3.2	X509AddPublicKey()	859
10.2.26.3.3	X509PushAlgorithmIdentifierSequence()	860
10.2.27	AC_spt.c	861
10.2.27.1	Includes and Structures	861
10.2.27.2	Functions	861
10.2.27.2.1	AcToCapabilities()	861
10.2.27.2.2	AcIsAccessible()	861
10.2.27.2.3	AcCapabilitiesGet()	862
10.2.27.2.4	AcSendObject()	862
10.2.28	CryptEccCrypt.c	864
10.2.28.1	Includes and Defines	864
10.2.28.2	Functions	864
10.2.28.2.1	CryptEccSelectScheme()	864
10.2.28.2.2	CryptEccEncrypt()	864
10.2.28.2.3	CryptEccDecrypt()	866
Annex A (informative)	Implementation Dependent	868
A.1	Introduction	868
A.2	TpmProfile.h	868
A.3	TpmSizeChecks.c	880
A.3.1.	Includes, Defines, and Types	880
A.3.2.	TpmSizeChecks()	880
Annex B (informative)	Library-Specific	883
B.1	Introduction	883
B.2	OpenSSL-Specific Files	884
B.2.1.	Introduction	884
B.2.2.	Header Files	884
B.2.2.1.	TpmToOsslHash.h	884
B.2.2.1.1.	Introduction	884
B.2.2.1.2.	Links to the OpenSSL HASH code	884
B.2.2.2.	TpmToOsslMath.h	888
B.2.2.2.1.	Introduction	888
B.2.2.2.2.	Macros and Defines	888
B.2.2.3.	TpmToOsslSym.h	890
B.2.2.3.1.	Introduction	890
B.2.2.3.2.	Links to the OpenSSL symmetric algorithms.	890
B.2.2.3.3.	Links to the OpenSSL AES code	891
B.2.2.3.4.	Links to the OpenSSL DES code	891
B.2.2.3.5.	Links to the OpenSSL SM4 code	891
B.2.2.3.6.	Links to the OpenSSL CAMELLIA code	892
B.2.3.	Source Files	893
B.2.3.1.	TpmToOsslDesSupport.c	893
B.2.3.1.1.	Introduction	893

B.2.3.1.2. Defines and Includes	893
B.2.3.1.3. Functions	893
B.2.3.2. TpmToOsslMath.c	895
B.2.3.2.1. Introduction	895
B.2.3.2.2. Includes and Defines	895
B.2.3.2.3. Functions	895
B.2.3.2.4. BnEccAdd()	905
B.2.3.3. TpmToOsslSupport.c	906
B.2.3.3.1. Introduction	906
B.2.3.3.2. Defines and Includes	906
Annex C (informative) Simulation Environment	908
C.1 Introduction	908
C.2 Cancel.c	908
C.2.1. Description	908
C.2.2. Includes, Typedefs, Structures, and Defines	908
C.2.3. Functions	908
C.2.3.1. _plat_IsCanceled()	908
C.2.3.2. _plat_SetCancel()	908
C.2.3.3. _plat_ClearCancel()	909
C.3 Clock.c	910
C.3.1. Description	910
C.3.2. Includes and Data Definitions	910
C.3.3. Simulator Functions	910
C.3.3.1. Introduction	910
C.3.3.2. _plat_TimerReset()	910
C.3.3.3. _plat_TimerRestart()	910
C.3.4. Functions Used by TPM	911
C.3.4.1. Introduction	911
C.3.4.2. _plat_RealTime()	911
C.3.4.3. _plat_TimerRead()	911
C.3.4.4. _plat_TimerWasReset()	913
C.3.4.5. _plat_TimerWasStopped()	913
C.3.4.6. _plat_ClockAdjustRate()	913
C.4 Entropy.c	915
C.4.1. Includes and Local Values	915
C.4.2. Functions	915
C.4.2.1. rand32()	915
C.4.2.2. _plat_GetEntropy()	915
C.5 LocalityPlat.c	918
C.5.1. Includes	918
C.5.2. Functions	918
C.5.2.1. _plat_LocalityGet()	918
C.5.2.2. _plat_LocalitySet()	918
C.6 NVMem.c	919
C.6.1. Description	919
C.6.2. Includes and Local	919
C.6.3. Functions	919

C.6.3.1.	NvFileOpen()	919
C.6.3.2.	NvFileCommit()	920
C.6.3.3.	NvFileSize()	920
C.6.3.4.	_plat_NvErrors()	921
C.6.3.5.	_plat_NVEnable()	921
C.6.3.6.	_plat_NVDisable()	922
C.6.3.7.	_plat_IsNvAvailable()	922
C.6.3.8.	_plat_NvMemoryRead()	923
C.6.3.9.	_plat_NvIsDifferent()	923
C.6.3.10.	_plat_NvMemoryWrite()	924
C.6.3.11.	_plat_NvMemoryClear()	924
C.6.3.12.	_plat_NvMemoryMove()	924
C.6.3.13.	_plat_NvCommit()	925
C.6.3.14.	_plat_SetNvAvail()	925
C.6.3.15.	_plat_ClearNvAvail()	925
C.6.3.16.	_plat_NVNeedsManufacture()	925
C.7	PowerPlat.c	927
C.7.1.	Includes and Function Prototypes	927
C.7.2.	Functions	927
C.7.2.1.	_plat_Signal_PowerOn()	927
C.7.2.2.	_plat_WasPowerLost()	927
C.7.2.3.	_plat_Signal_Reset()	927
C.7.2.4.	_plat_Signal_PowerOff()	928
C.8	PlatformData.h	929
C.9	PlatformData.c	931
C.9.1.	Description	931
C.9.2.	Includes	931
C.10	PPPlat.c	932
C.10.1.	Description	932
C.10.2.	Includes	932
C.10.3.	Functions	932
C.10.3.1.	_plat_PhysicalPresenceAsserted()	932
C.10.3.2.	_plat_Signal_PhysicalPresenceOn()	932
C.10.3.3.	_plat_Signal_PhysicalPresenceOff()	932
C.11	RunCommand.c	933
C.11.1.	Introduction	933
C.11.2.	Includes and locals	933
C.11.3.	Functions	933
C.11.3.1.	_plat_RunCommand()	933
C.11.3.2.	_plat_Fail()	933
C.12	Unique.c	934
C.12.1.	Introduction	934
C.12.2.	Includes	934
C.12.3.	_plat_GetUnique()	934
C.13	DebugHelpers.c	935
C.13.1.	Description	935
C.13.2.	Includes and Local	935
C.13.2.1.	fileOpen()	935
C.13.2.2.	DebugFileInit()	935
C.13.2.3.	DebugDumpBuffer()	936

C.14 Platform.h	937
C.15 PlatformACT.h	938
C.16 PlatformACT.c	941
C.16.1. Includes	941
C.16.2. Functions	941
C.16.2.1. ActSignal()	941
C.16.2.2. ActGetDataPointer()	942
C.16.2.3. _plat_ACT_GetImplemented()	942
C.16.2.4. _plat_ACT_GetRemaining()	943
C.16.2.5. _plat_ACT_GetSignaled()	943
C.16.2.6. _plat_ACT_SetSignaled()	943
C.16.2.7. _plat_ACT_GetPending()	943
C.16.2.8. _plat_ACT_UpdateCounter()	944
C.16.2.9. _plat_ACT_EnableTicks()	944
C.16.2.10. ActDecrement()	944
C.16.2.11. _plat_ACT_Tick()	945
C.16.2.12. ActZero()	945
C.16.2.13. _plat_ACT_Initialize()	946
C.17 PlatformClock.h	947
Annex D (informative) Remote Procedure Interface	948
D.1 Introduction	948
D.2 TpmTcpProtocol.h	949
D.2.1. Introduction	949
D.2.2. Typedefs and Defines	949
D.2.3. TPM Commands	949
D.2.4. Enumerations and Structures	949
D.3 TcpServer.c	951
D.3.1. Description	951
D.3.2. Includes, Locals, Defines and Function Prototypes	951
D.3.3. Functions	952
D.3.3.1. CreateSocket()	952
D.3.3.2. PlatformServer()	952
D.3.3.3. PlatformSvcRoutine()	954
D.3.3.4. PlatformSignalService()	955
D.3.3.5. RegularCommandService()	955
D.3.3.6. SimulatorTimeServiceRoutine()	956
D.3.3.7. ActTimeService()	957
D.3.3.8. StartTcpServer()	958
D.3.3.9. ReadBytes()	958
D.3.3.10. WriteBytes()	959
D.3.3.11. WriteUINT32()	959
D.3.3.12. ReadUINT32()	960
D.3.3.13. ReadVarBytes()	960
D.3.3.14. WriteVarBytes()	960
D.3.3.15. TpmServer()	961
D.4 TPMCmdp.c	963
D.4.1. Description	963
D.4.2. Includes and Data Definitions	963
D.4.3. Functions	963
D.4.3.1. Signal_PowerOn()	963
D.4.3.2. Signal_Restart()	964
D.4.3.3. Signal_PowerOff()	964

D.4.3.4.	_rpc_ForceFailureMode()	964
D.4.3.5.	_rpc_Signal_PhysicalPresenceOn()	964
D.4.3.6.	_rpc_Signal_PhysicalPresenceOff()	965
D.4.3.7.	_rpc_Signal_Hash_Start()	965
D.4.3.8.	_rpc_Signal_Hash_Data()	965
D.4.3.9.	_rpc_Signal_HashEnd()	965
D.4.3.10.	_rpc_Send_Command()	966
D.4.3.11.	_rpc_Signal_CancelOn()	966
D.4.3.12.	_rpc_Signal_CancelOff()	966
D.4.3.13.	_rpc_Signal_NvOn()	967
D.4.3.14.	_rpc_Signal_NvOff()	967
D.4.3.15.	_rpc_RsaKeyCacheControl()	967
D.4.3.16.	_rpc_ACT_GetSignaled()	968
D.5	TPMCmds.c	969
D.5.1.	Description	969
D.5.2.	Includes, Defines, Data Definitions, and Function Prototypes	969
D.5.3.	Functions	969
D.5.3.1.	Assert()	969
D.5.3.2.	Usage()	970
D.5.3.3.	CmdLineParser_Init()	970
D.5.3.4.	CmdLineParser_More()	970
D.5.3.5.	CmdLineParser_IsOpt()	971
D.5.3.6.	CmdLineParser_IsOptPresent()	971
D.5.3.7.	CmdLineParser_Done()	972
D.5.3.8.	main()	972

Trusted Platform Module Library

Part 4: Supporting Routines

1 Scope

This part contains C code that describes the algorithms and methods used by the command code in TPM 2.0 Part 3. The code in this document augments TPM 2.0 Part 2 and TPM 2.0 Part 3 to provide a complete description of a TPM, including the supporting framework for the code that performs the command actions.

Any TPM 2.0 Part 4 code may be replaced by code that provides similar results when interfacing to the action code in TPM 2.0 Part 3. The behavior of code in this document that is not included in an annex is *normative*, as observed at the interfaces with TPM 2.0 Part 3 code. Code in an annex is provided for completeness, that is, to allow a full implementation of the specification from the provided code.

The code in parts 3 and 4 is written to define the behavior of a compliant TPM. In some cases (e.g., firmware update), it is not possible to provide a compliant implementation. In those cases, any implementation provided by the vendor that meets the general description of the function provided in TPM 2.0 Part 3 would be compliant.

The code in parts 3 and 4 is not written to meet any particular level of conformance nor does this specification require that a TPM meet any particular level of conformance.

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 `TPMI_ALG_HASH` data type is used in many places. In some cases, `TPM_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 TYPE 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.

```

1  #if CC_Create // Command must be enabled
2
3  #ifndef _Create_FP_H_
4  #define _Create_FP_H_

```

Input structure definition

```

5  typedef struct {
6      TPMI_DH_OBJECT          parentHandle;
7      TPM2B_SENSITIVE_CREATE inSensitive;
8      TPM2B_PUBLIC            inPublic;
9      TPM2B_DATA              outsideInfo;
10     TPML_PCR_SELECTION       creationPCR;
11 } Create_In;

```

Output structure definition

```

12 typedef struct {
13     TPM2B_PRIVATE outPrivate;
14     TPM2B_PUBLIC  outPublic;

```

```

15     TPM2B_CREATION_DATA      creationData;
16     TPM2B_DIGEST             creationHash;
17     TPMT_TK_CREATION          creationTicket;
18 } Create_Out;

```

Response code modifiers

```

19 #define RC_Create_parentHandle (TPM_RC_H + TPM_RC_1)
20 #define RC_Create_inSensitive (TPM_RC_P + TPM_RC_1)
21 #define RC_Create_inPublic     (TPM_RC_P + TPM_RC_2)
22 #define RC_Create_outsideInfo  (TPM_RC_P + TPM_RC_3)
23 #define RC_Create_creationPCR  (TPM_RC_P + TPM_RC_4)

```

Function prototype

```

24 TPM_RC
25 TPM2_Create(
26     Create_In          *in,
27     Create_Out          *out
28 );
29
30 #endif // _Create_FP_H_
31 #endif // CC_Create

```

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 Header Files

5.1 Introduction

The files in this clause are used to define values that are used in multiple parts of the specification and are not confined to a single module.

5.2 BaseTypes.h

```
1  #ifndef _BASE_TYPES_H_
2  #define _BASE_TYPES_H_
```

NULL definition

```
3  #ifndef NULL
4  #define NULL          (0)
5  #endif
6
7  typedef uint8_t      UINT8;
8  typedef uint8_t      BYTE;
9  typedef int8_t       INT8;
10 typedef int          BOOL;
11 typedef uint16_t     UINT16;
12 typedef int16_t      INT16;
13 typedef uint32_t     UINT32;
14 typedef int32_t      INT32;
15 typedef uint64_t     UINT64;
16 typedef int64_t      INT64;
17
18 #endif // _BASE_TYPES_H_
```

5.3 Capabilities.h

This file contains defines for the number of capability values that will fit into the largest data buffer.

These defines are used in various function in the "support" and the "subsystem" code groups. A module that supports a type that is returned by a capability will have a function that returns the capabilities of the type.

EXAMPLE PCR.c contains PCRCapGetHandles() and PCRCapGetProperties().

```

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
14 #define      MAX_AC_CAPABILITIES (MAX_CAP_DATA / sizeof(TPMS_AC_OUTPUT))
15
16 #endif

```


5.4 CommandAttributeData.h

This file should only be included by CommandCodeAttributes.c

```

1  #ifndef _COMMAND_CODE_ATTRIBUTES_
2
3  #include "CommandAttributes.h"
4
5  #if COMPRESSED_LISTS
6  #   define PAD_LIST 0
7  #else
8  #   define PAD_LIST 1
9  #endif

```

This is the command code attribute array for GetCapability(). Both this array and `s_commandAttributes` provides command code attributes, but tuned for different purpose

```

10 const TPMA_CC s_ccAttr [] = {
11 #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
12     TPMA_CC_INITIALIZER(0x011F, 0, 1, 0, 0, 2, 0, 0, 0),
13 #endif
14 #if (PAD_LIST || CC_EvictControl)
15     TPMA_CC_INITIALIZER(0x0120, 0, 1, 0, 0, 2, 0, 0, 0),
16 #endif
17 #if (PAD_LIST || CC_HierarchyControl)
18     TPMA_CC_INITIALIZER(0x0121, 0, 1, 1, 0, 1, 0, 0, 0),
19 #endif
20 #if (PAD_LIST || CC_NV_UndefineSpace)
21     TPMA_CC_INITIALIZER(0x0122, 0, 1, 0, 0, 2, 0, 0, 0),
22 #endif
23 #if (PAD_LIST )
24     TPMA_CC_INITIALIZER(0x0123, 0, 0, 0, 0, 0, 0, 0, 0),
25 #endif
26 #if (PAD_LIST || CC_ChangeEPS)
27     TPMA_CC_INITIALIZER(0x0124, 0, 1, 1, 0, 1, 0, 0, 0),
28 #endif
29 #if (PAD_LIST || CC_ChangePPS)
30     TPMA_CC_INITIALIZER(0x0125, 0, 1, 1, 0, 1, 0, 0, 0),
31 #endif
32 #if (PAD_LIST || CC_Clear)
33     TPMA_CC_INITIALIZER(0x0126, 0, 1, 1, 0, 1, 0, 0, 0),
34 #endif
35 #if (PAD_LIST || CC_ClearControl)
36     TPMA_CC_INITIALIZER(0x0127, 0, 1, 0, 0, 1, 0, 0, 0),
37 #endif
38 #if (PAD_LIST || CC_ClockSet)
39     TPMA_CC_INITIALIZER(0x0128, 0, 1, 0, 0, 1, 0, 0, 0),
40 #endif
41 #if (PAD_LIST || CC_HierarchyChangeAuth)
42     TPMA_CC_INITIALIZER(0x0129, 0, 1, 0, 0, 1, 0, 0, 0),
43 #endif
44 #if (PAD_LIST || CC_NV_DefineSpace)
45     TPMA_CC_INITIALIZER(0x012A, 0, 1, 0, 0, 1, 0, 0, 0),
46 #endif
47 #if (PAD_LIST || CC_PCR_Allocate)
48     TPMA_CC_INITIALIZER(0x012B, 0, 1, 0, 0, 1, 0, 0, 0),
49 #endif
50 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
51     TPMA_CC_INITIALIZER(0x012C, 0, 1, 0, 0, 1, 0, 0, 0),
52 #endif
53 #if (PAD_LIST || CC_PP_Commands)
54     TPMA_CC_INITIALIZER(0x012D, 0, 1, 0, 0, 1, 0, 0, 0),
55 #endif
56 #if (PAD_LIST || CC_SetPrimaryPolicy)

```

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

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

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

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

```

321     TPMA_CC_INITIALIZER(0x0186, 0, 0, 0, 0, 0, 1, 0, 0),
322 #endif
323 #if (PAD_LIST || CC_PolicyPhysicalPresence)
324     TPMA_CC_INITIALIZER(0x0187, 0, 0, 0, 0, 0, 1, 0, 0),
325 #endif
326 #if (PAD_LIST || CC_PolicyDuplicationSelect)
327     TPMA_CC_INITIALIZER(0x0188, 0, 0, 0, 0, 0, 1, 0, 0),
328 #endif
329 #if (PAD_LIST || CC_PolicyGetDigest)
330     TPMA_CC_INITIALIZER(0x0189, 0, 0, 0, 0, 0, 1, 0, 0),
331 #endif
332 #if (PAD_LIST || CC_TestParms)
333     TPMA_CC_INITIALIZER(0x018A, 0, 0, 0, 0, 0, 0, 0, 0),
334 #endif
335 #if (PAD_LIST || CC_Commit)
336     TPMA_CC_INITIALIZER(0x018B, 0, 0, 0, 0, 0, 1, 0, 0),
337 #endif
338 #if (PAD_LIST || CC_PolicyPassword)
339     TPMA_CC_INITIALIZER(0x018C, 0, 0, 0, 0, 0, 1, 0, 0),
340 #endif
341 #if (PAD_LIST || CC_ZGen_2Phase)
342     TPMA_CC_INITIALIZER(0x018D, 0, 0, 0, 0, 0, 1, 0, 0),
343 #endif
344 #if (PAD_LIST || CC_EC_Ephemeral)
345     TPMA_CC_INITIALIZER(0x018E, 0, 0, 0, 0, 0, 0, 0, 0),
346 #endif
347 #if (PAD_LIST || CC_PolicyNvWritten)
348     TPMA_CC_INITIALIZER(0x018F, 0, 0, 0, 0, 0, 1, 0, 0),
349 #endif
350 #if (PAD_LIST || CC_PolicyTemplate)
351     TPMA_CC_INITIALIZER(0x0190, 0, 0, 0, 0, 0, 1, 0, 0),
352 #endif
353 #if (PAD_LIST || CC_CreateLoaded)
354     TPMA_CC_INITIALIZER(0x0191, 0, 0, 0, 0, 0, 1, 1, 0),
355 #endif
356 #if (PAD_LIST || CC_PolicyAuthorizeNV)
357     TPMA_CC_INITIALIZER(0x0192, 0, 0, 0, 0, 0, 3, 0, 0),
358 #endif
359 #if (PAD_LIST || CC_EncryptDecrypt2)
360     TPMA_CC_INITIALIZER(0x0193, 0, 0, 0, 0, 0, 1, 0, 0),
361 #endif
362 #if (PAD_LIST || CC_AC_GetCapability)
363     TPMA_CC_INITIALIZER(0x0194, 0, 0, 0, 0, 0, 1, 0, 0),
364 #endif
365 #if (PAD_LIST || CC_AC_Send)
366     TPMA_CC_INITIALIZER(0x0195, 0, 0, 0, 0, 0, 3, 0, 0),
367 #endif
368 #if (PAD_LIST || CC_Policy_AC_SendSelect)
369     TPMA_CC_INITIALIZER(0x0196, 0, 0, 0, 0, 0, 1, 0, 0),
370 #endif
371 #if (PAD_LIST || CC_CertifyX509)
372     TPMA_CC_INITIALIZER(0x0197, 0, 0, 0, 0, 0, 2, 0, 0),
373 #endif
374 #if (PAD_LIST || CC_ACT_SetTimeout)
375     TPMA_CC_INITIALIZER(0x0198, 0, 0, 0, 0, 0, 1, 0, 0),
376 #endif
377 #if (PAD_LIST || CC_ECC_Encrypt)
378     TPMA_CC_INITIALIZER(0x0199, 0, 0, 0, 0, 0, 1, 0, 0),
379 #endif
380 #if (PAD_LIST || CC_ECC_Decrypt)
381     TPMA_CC_INITIALIZER(0x019A, 0, 0, 0, 0, 0, 1, 0, 0),
382 #endif
383 #if (PAD_LIST || CC_Vendor_TCG_Test)
384     TPMA_CC_INITIALIZER(0x0000, 0, 0, 0, 0, 0, 0, 0, 1),
385 #endif
386     TPMA_ZERO_INITIALIZER()

```



```
387 };
```

This is the command code attribute structure.

```
388 const COMMAND_ATTRIBUTES s_commandAttributes [] = {
389 #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
390     (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpaceSpecial * // 0x011F
391         (IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),
392 #endif
393 #if (PAD_LIST || CC_EvictControl)
394     (COMMAND_ATTRIBUTES)(CC_EvictControl * // 0x0120
395         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
396 #endif
397 #if (PAD_LIST || CC_HierarchyControl)
398     (COMMAND_ATTRIBUTES)(CC_HierarchyControl * // 0x0121
399         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
400 #endif
401 #if (PAD_LIST || CC_NV_UndefineSpace)
402     (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpace * // 0x0122
403         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
404 #endif
405 #if (PAD_LIST )
406     (COMMAND_ATTRIBUTES)(0), // 0x0123
407 #endif
408 #if (PAD_LIST || CC_ChangeEPS)
409     (COMMAND_ATTRIBUTES)(CC_ChangeEPS * // 0x0124
410         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
411 #endif
412 #if (PAD_LIST || CC_ChangePPS)
413     (COMMAND_ATTRIBUTES)(CC_ChangePPS * // 0x0125
414         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
415 #endif
416 #if (PAD_LIST || CC_Clear)
417     (COMMAND_ATTRIBUTES)(CC_Clear * // 0x0126
418         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
419 #endif
420 #if (PAD_LIST || CC_ClearControl)
421     (COMMAND_ATTRIBUTES)(CC_ClearControl * // 0x0127
422         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
423 #endif
424 #if (PAD_LIST || CC_ClockSet)
425     (COMMAND_ATTRIBUTES)(CC_ClockSet * // 0x0128
426         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
427 #endif
428 #if (PAD_LIST || CC_HierarchyChangeAuth)
429     (COMMAND_ATTRIBUTES)(CC_HierarchyChangeAuth * // 0x0129
430         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
431 #endif
432 #if (PAD_LIST || CC_NV_DefineSpace)
433     (COMMAND_ATTRIBUTES)(CC_NV_DefineSpace * // 0x012A
434         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
435 #endif
436 #if (PAD_LIST || CC_PCR_Allocate)
437     (COMMAND_ATTRIBUTES)(CC_PCR_Allocate * // 0x012B
438         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
439 #endif
440 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
441     (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthPolicy * // 0x012C
442         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
443 #endif
444 #if (PAD_LIST || CC_PP_Commands)
445     (COMMAND_ATTRIBUTES)(CC_PP_Commands * // 0x012D
446         (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),
447 #endif
448 #if (PAD_LIST || CC_SetPrimaryPolicy)
```

```

449         (COMMAND_ATTRIBUTES) (CC_SetPrimaryPolicy          * // 0x012E
450         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
451 #endif
452 #if (PAD_LIST || CC_FieldUpgradeStart)
453         (COMMAND_ATTRIBUTES) (CC_FieldUpgradeStart          * // 0x012F
454         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),
455 #endif
456 #if (PAD_LIST || CC_ClockRateAdjust)
457         (COMMAND_ATTRIBUTES) (CC_ClockRateAdjust            * // 0x0130
458         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
459 #endif
460 #if (PAD_LIST || CC_CreatePrimary)
461         (COMMAND_ATTRIBUTES) (CC_CreatePrimary              * // 0x0131
462         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
463 #endif
464 #if (PAD_LIST || CC_NV_GlobalWriteLock)
465         (COMMAND_ATTRIBUTES) (CC_NV_GlobalWriteLock         * // 0x0132
466         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
467 #endif
468 #if (PAD_LIST || CC_GetCommandAuditDigest)
469         (COMMAND_ATTRIBUTES) (CC_GetCommandAuditDigest      * // 0x0133
470         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
471 #endif
472 #if (PAD_LIST || CC_NV_Increment)
473         (COMMAND_ATTRIBUTES) (CC_NV_Increment               * // 0x0134
474         (IS_IMPLEMENTED+HANDLE_1_USER)),
475 #endif
476 #if (PAD_LIST || CC_NV_SetBits)
477         (COMMAND_ATTRIBUTES) (CC_NV_SetBits                 * // 0x0135
478         (IS_IMPLEMENTED+HANDLE_1_USER)),
479 #endif
480 #if (PAD_LIST || CC_NV_Extend)
481         (COMMAND_ATTRIBUTES) (CC_NV_Extend                  * // 0x0136
482         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
483 #endif
484 #if (PAD_LIST || CC_NV_Write)
485         (COMMAND_ATTRIBUTES) (CC_NV_Write                   * // 0x0137
486         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
487 #endif
488 #if (PAD_LIST || CC_NV_WriteLock)
489         (COMMAND_ATTRIBUTES) (CC_NV_WriteLock               * // 0x0138
490         (IS_IMPLEMENTED+HANDLE_1_USER)),
491 #endif
492 #if (PAD_LIST || CC_DictionaryAttackLockReset)
493         (COMMAND_ATTRIBUTES) (CC_DictionaryAttackLockReset * // 0x0139
494         (IS_IMPLEMENTED+HANDLE_1_USER)),
495 #endif
496 #if (PAD_LIST || CC_DictionaryAttackParameters)
497         (COMMAND_ATTRIBUTES) (CC_DictionaryAttackParameters * // 0x013A
498         (IS_IMPLEMENTED+HANDLE_1_USER)),
499 #endif
500 #if (PAD_LIST || CC_NV_ChangeAuth)
501         (COMMAND_ATTRIBUTES) (CC_NV_ChangeAuth              * // 0x013B
502         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),
503 #endif
504 #if (PAD_LIST || CC_PCR_Event)
505         (COMMAND_ATTRIBUTES) (CC_PCR_Event                  * // 0x013C
506         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
507 #endif
508 #if (PAD_LIST || CC_PCR_Reset)
509         (COMMAND_ATTRIBUTES) (CC_PCR_Reset                   * // 0x013D
510         (IS_IMPLEMENTED+HANDLE_1_USER)),
511 #endif
512 #if (PAD_LIST || CC_SequenceComplete)
513         (COMMAND_ATTRIBUTES) (CC_SequenceComplete           * // 0x013E
514         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),

```



```

515 #endif
516 #if (PAD_LIST || CC_SetAlgorithmSet)
517     (COMMAND_ATTRIBUTES) (CC_SetAlgorithmSet * // 0x013F
518         (IS_IMPLEMENTED+HANDLE_1_USER)),
519 #endif
520 #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
521     (COMMAND_ATTRIBUTES) (CC_SetCommandCodeAuditStatus * // 0x0140
522         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
523 #endif
524 #if (PAD_LIST || CC_FieldUpgradeData)
525     (COMMAND_ATTRIBUTES) (CC_FieldUpgradeData * // 0x0141
526         (IS_IMPLEMENTED+DECRYPT_2)),
527 #endif
528 #if (PAD_LIST || CC_IncrementalSelfTest)
529     (COMMAND_ATTRIBUTES) (CC_IncrementalSelfTest * // 0x0142
530         (IS_IMPLEMENTED)),
531 #endif
532 #if (PAD_LIST || CC_SelfTest)
533     (COMMAND_ATTRIBUTES) (CC_SelfTest * // 0x0143
534         (IS_IMPLEMENTED)),
535 #endif
536 #if (PAD_LIST || CC_Startup)
537     (COMMAND_ATTRIBUTES) (CC_Startup * // 0x0144
538         (IS_IMPLEMENTED+NO_SESSIONS)),
539 #endif
540 #if (PAD_LIST || CC_Shutdown)
541     (COMMAND_ATTRIBUTES) (CC_Shutdown * // 0x0145
542         (IS_IMPLEMENTED)),
543 #endif
544 #if (PAD_LIST || CC_StirRandom)
545     (COMMAND_ATTRIBUTES) (CC_StirRandom * // 0x0146
546         (IS_IMPLEMENTED+DECRYPT_2)),
547 #endif
548 #if (PAD_LIST || CC_ActivateCredential)
549     (COMMAND_ATTRIBUTES) (CC_ActivateCredential * // 0x0147
550         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
551 #endif
552 #if (PAD_LIST || CC_Certify)
553     (COMMAND_ATTRIBUTES) (CC_Certify * // 0x0148
554         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
555 #endif
556 #if (PAD_LIST || CC_PolicyNV)
557     (COMMAND_ATTRIBUTES) (CC_PolicyNV * // 0x0149
558         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL)),
559 #endif
560 #if (PAD_LIST || CC_CertifyCreation)
561     (COMMAND_ATTRIBUTES) (CC_CertifyCreation * // 0x014A
562         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
563 #endif
564 #if (PAD_LIST || CC_Duplicate)
565     (COMMAND_ATTRIBUTES) (CC_Duplicate * // 0x014B
566         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),
567 #endif
568 #if (PAD_LIST || CC_GetTime)
569     (COMMAND_ATTRIBUTES) (CC_GetTime * // 0x014C
570         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
571 #endif
572 #if (PAD_LIST || CC_GetSessionAuditDigest)
573     (COMMAND_ATTRIBUTES) (CC_GetSessionAuditDigest * // 0x014D
574         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
575 #endif
576 #if (PAD_LIST || CC_NV_Read)
577     (COMMAND_ATTRIBUTES) (CC_NV_Read * // 0x014E
578         (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
579 #endif
580 #if (PAD_LIST || CC_NV_ReadLock)

```

```

581         (COMMAND_ATTRIBUTES) (CC_NV_ReadLock * // 0x014F
582         (IS_IMPLEMENTED+HANDLE_1_USER)),
583 #endif
584 #if (PAD_LIST || CC_ObjectChangeAuth)
585         (COMMAND_ATTRIBUTES) (CC_ObjectChangeAuth * // 0x0150
586         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),
587 #endif
588 #if (PAD_LIST || CC_PolicySecret)
589         (COMMAND_ATTRIBUTES) (CC_PolicySecret * // 0x0151
590         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL+ENCRYPT_2)),
591 #endif
592 #if (PAD_LIST || CC_Rewrap)
593         (COMMAND_ATTRIBUTES) (CC_Rewrap * // 0x0152
594         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
595 #endif
596 #if (PAD_LIST || CC_Create)
597         (COMMAND_ATTRIBUTES) (CC_Create * // 0x0153
598         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
599 #endif
600 #if (PAD_LIST || CC_ECDH_ZGen)
601         (COMMAND_ATTRIBUTES) (CC_ECDH_ZGen * // 0x0154
602         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
603 #endif
604 #if (PAD_LIST || (CC_HMAC || CC_MAC))
605         (COMMAND_ATTRIBUTES) ((CC_HMAC || CC_MAC) * // 0x0155
606         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
607 #endif
608 #if (PAD_LIST || CC_Import)
609         (COMMAND_ATTRIBUTES) (CC_Import * // 0x0156
610         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
611 #endif
612 #if (PAD_LIST || CC_Load)
613         (COMMAND_ATTRIBUTES) (CC_Load * // 0x0157
614         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),
615 #endif
616 #if (PAD_LIST || CC_Quote)
617         (COMMAND_ATTRIBUTES) (CC_Quote * // 0x0158
618         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
619 #endif
620 #if (PAD_LIST || CC_RSA_Decrypt)
621         (COMMAND_ATTRIBUTES) (CC_RSA_Decrypt * // 0x0159
622         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
623 #endif
624 #if (PAD_LIST )
625         (COMMAND_ATTRIBUTES) (0), // 0x015A
626 #endif
627 #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
628         (COMMAND_ATTRIBUTES) ((CC_HMAC_Start || CC_MAC_Start) * // 0x015B
629         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),
630 #endif
631 #if (PAD_LIST || CC_SequenceUpdate)
632         (COMMAND_ATTRIBUTES) (CC_SequenceUpdate * // 0x015C
633         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
634 #endif
635 #if (PAD_LIST || CC_Sign)
636         (COMMAND_ATTRIBUTES) (CC_Sign * // 0x015D
637         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
638 #endif
639 #if (PAD_LIST || CC_Unseal)
640         (COMMAND_ATTRIBUTES) (CC_Unseal * // 0x015E
641         (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
642 #endif
643 #if (PAD_LIST )
644         (COMMAND_ATTRIBUTES) (0), // 0x015F
645 #endif
646 #if (PAD_LIST || CC_PolicySigned)

```

```

647         (COMMAND_ATTRIBUTES) (CC_PolicySigned * // 0x0160
648         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL+ENCRYPT_2)),
649 #endif
650 #if (PAD_LIST || CC_ContextLoad)
651     (COMMAND_ATTRIBUTES) (CC_ContextLoad * // 0x0161
652     (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
653 #endif
654 #if (PAD_LIST || CC_ContextSave)
655     (COMMAND_ATTRIBUTES) (CC_ContextSave * // 0x0162
656     (IS_IMPLEMENTED+NO_SESSIONS)),
657 #endif
658 #if (PAD_LIST || CC_ECDH_KeyGen)
659     (COMMAND_ATTRIBUTES) (CC_ECDH_KeyGen * // 0x0163
660     (IS_IMPLEMENTED+ENCRYPT_2)),
661 #endif
662 #if (PAD_LIST || CC_EncryptDecrypt)
663     (COMMAND_ATTRIBUTES) (CC_EncryptDecrypt * // 0x0164
664     (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
665 #endif
666 #if (PAD_LIST || CC_FlushContext)
667     (COMMAND_ATTRIBUTES) (CC_FlushContext * // 0x0165
668     (IS_IMPLEMENTED+NO_SESSIONS)),
669 #endif
670 #if (PAD_LIST )
671     (COMMAND_ATTRIBUTES) (0), // 0x0166
672 #endif
673 #if (PAD_LIST || CC_LoadExternal)
674     (COMMAND_ATTRIBUTES) (CC_LoadExternal * // 0x0167
675     (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
676 #endif
677 #if (PAD_LIST || CC_MakeCredential)
678     (COMMAND_ATTRIBUTES) (CC_MakeCredential * // 0x0168
679     (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
680 #endif
681 #if (PAD_LIST || CC_NV_ReadPublic)
682     (COMMAND_ATTRIBUTES) (CC_NV_ReadPublic * // 0x0169
683     (IS_IMPLEMENTED+ENCRYPT_2)),
684 #endif
685 #if (PAD_LIST || CC_PolicyAuthorize)
686     (COMMAND_ATTRIBUTES) (CC_PolicyAuthorize * // 0x016A
687     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
688 #endif
689 #if (PAD_LIST || CC_PolicyAuthValue)
690     (COMMAND_ATTRIBUTES) (CC_PolicyAuthValue * // 0x016B
691     (IS_IMPLEMENTED+ALLOW_TRIAL)),
692 #endif
693 #if (PAD_LIST || CC_PolicyCommandCode)
694     (COMMAND_ATTRIBUTES) (CC_PolicyCommandCode * // 0x016C
695     (IS_IMPLEMENTED+ALLOW_TRIAL)),
696 #endif
697 #if (PAD_LIST || CC_PolicyCounterTimer)
698     (COMMAND_ATTRIBUTES) (CC_PolicyCounterTimer * // 0x016D
699     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
700 #endif
701 #if (PAD_LIST || CC_PolicyCpHash)
702     (COMMAND_ATTRIBUTES) (CC_PolicyCpHash * // 0x016E
703     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
704 #endif
705 #if (PAD_LIST || CC_PolicyLocality)
706     (COMMAND_ATTRIBUTES) (CC_PolicyLocality * // 0x016F
707     (IS_IMPLEMENTED+ALLOW_TRIAL)),
708 #endif
709 #if (PAD_LIST || CC_PolicyNameHash)
710     (COMMAND_ATTRIBUTES) (CC_PolicyNameHash * // 0x0170
711     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
712 #endif

```

```

713 #if (PAD_LIST || CC_PolicyOR)
714     (COMMAND_ATTRIBUTES)(CC_PolicyOR
715         (IS_IMPLEMENTED+ALLOW_TRIAL)),
716 #endif
717 #if (PAD_LIST || CC_PolicyTicket)
718     (COMMAND_ATTRIBUTES)(CC_PolicyTicket
719         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
720 #endif
721 #if (PAD_LIST || CC_ReadPublic)
722     (COMMAND_ATTRIBUTES)(CC_ReadPublic
723         (IS_IMPLEMENTED+ENCRYPT_2)),
724 #endif
725 #if (PAD_LIST || CC_RSA_Encrypt)
726     (COMMAND_ATTRIBUTES)(CC_RSA_Encrypt
727         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
728 #endif
729 #if (PAD_LIST )
730     (COMMAND_ATTRIBUTES)(0),
731 #endif
732 #if (PAD_LIST || CC_StartAuthSession)
733     (COMMAND_ATTRIBUTES)(CC_StartAuthSession
734         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
735 #endif
736 #if (PAD_LIST || CC_VerifySignature)
737     (COMMAND_ATTRIBUTES)(CC_VerifySignature
738         (IS_IMPLEMENTED+DECRYPT_2)),
739 #endif
740 #if (PAD_LIST || CC_ECC_Parameters)
741     (COMMAND_ATTRIBUTES)(CC_ECC_Parameters
742         (IS_IMPLEMENTED)),
743 #endif
744 #if (PAD_LIST || CC_FirmwareRead)
745     (COMMAND_ATTRIBUTES)(CC_FirmwareRead
746         (IS_IMPLEMENTED+ENCRYPT_2)),
747 #endif
748 #if (PAD_LIST || CC_GetCapability)
749     (COMMAND_ATTRIBUTES)(CC_GetCapability
750         (IS_IMPLEMENTED)),
751 #endif
752 #if (PAD_LIST || CC_GetRandom)
753     (COMMAND_ATTRIBUTES)(CC_GetRandom
754         (IS_IMPLEMENTED+ENCRYPT_2)),
755 #endif
756 #if (PAD_LIST || CC_GetTestResult)
757     (COMMAND_ATTRIBUTES)(CC_GetTestResult
758         (IS_IMPLEMENTED+ENCRYPT_2)),
759 #endif
760 #if (PAD_LIST || CC_Hash)
761     (COMMAND_ATTRIBUTES)(CC_Hash
762         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
763 #endif
764 #if (PAD_LIST || CC_PCR_Read)
765     (COMMAND_ATTRIBUTES)(CC_PCR_Read
766         (IS_IMPLEMENTED)),
767 #endif
768 #if (PAD_LIST || CC_PolicyPCR)
769     (COMMAND_ATTRIBUTES)(CC_PolicyPCR
770         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
771 #endif
772 #if (PAD_LIST || CC_PolicyRestart)
773     (COMMAND_ATTRIBUTES)(CC_PolicyRestart
774         (IS_IMPLEMENTED+ALLOW_TRIAL)),
775 #endif
776 #if (PAD_LIST || CC_ReadClock)
777     (COMMAND_ATTRIBUTES)(CC_ReadClock
778         (IS_IMPLEMENTED)),

```

```

779 #endif
780 #if (PAD_LIST || CC_PCR_Extend)
781     (COMMAND_ATTRIBUTES) (CC_PCR_Extend * // 0x0182
782     (IS_IMPLEMENTED+HANDLE_1_USER)),
783 #endif
784 #if (PAD_LIST || CC_PCR_SetAuthValue)
785     (COMMAND_ATTRIBUTES) (CC_PCR_SetAuthValue * // 0x0183
786     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
787 #endif
788 #if (PAD_LIST || CC_NV_Certify)
789     (COMMAND_ATTRIBUTES) (CC_NV_Certify * // 0x0184
790     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
791 #endif
792 #if (PAD_LIST || CC_EventSequenceComplete)
793     (COMMAND_ATTRIBUTES) (CC_EventSequenceComplete * // 0x0185
794     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),
795 #endif
796 #if (PAD_LIST || CC_HashSequenceStart)
797     (COMMAND_ATTRIBUTES) (CC_HashSequenceStart * // 0x0186
798     (IS_IMPLEMENTED+DECRYPT_2+R_HANDLE)),
799 #endif
800 #if (PAD_LIST || CC_PolicyPhysicalPresence)
801     (COMMAND_ATTRIBUTES) (CC_PolicyPhysicalPresence * // 0x0187
802     (IS_IMPLEMENTED+ALLOW_TRIAL)),
803 #endif
804 #if (PAD_LIST || CC_PolicyDuplicationSelect)
805     (COMMAND_ATTRIBUTES) (CC_PolicyDuplicationSelect * // 0x0188
806     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
807 #endif
808 #if (PAD_LIST || CC_PolicyGetDigest)
809     (COMMAND_ATTRIBUTES) (CC_PolicyGetDigest * // 0x0189
810     (IS_IMPLEMENTED+ALLOW_TRIAL+ENCRYPT_2)),
811 #endif
812 #if (PAD_LIST || CC_TestParms)
813     (COMMAND_ATTRIBUTES) (CC_TestParms * // 0x018A
814     (IS_IMPLEMENTED)),
815 #endif
816 #if (PAD_LIST || CC_Commit)
817     (COMMAND_ATTRIBUTES) (CC_Commit * // 0x018B
818     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
819 #endif
820 #if (PAD_LIST || CC_PolicyPassword)
821     (COMMAND_ATTRIBUTES) (CC_PolicyPassword * // 0x018C
822     (IS_IMPLEMENTED+ALLOW_TRIAL)),
823 #endif
824 #if (PAD_LIST || CC_ZGen_2Phase)
825     (COMMAND_ATTRIBUTES) (CC_ZGen_2Phase * // 0x018D
826     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
827 #endif
828 #if (PAD_LIST || CC_EC_Ephemeral)
829     (COMMAND_ATTRIBUTES) (CC_EC_Ephemeral * // 0x018E
830     (IS_IMPLEMENTED+ENCRYPT_2)),
831 #endif
832 #if (PAD_LIST || CC_PolicyNvWritten)
833     (COMMAND_ATTRIBUTES) (CC_PolicyNvWritten * // 0x018F
834     (IS_IMPLEMENTED+ALLOW_TRIAL)),
835 #endif
836 #if (PAD_LIST || CC_PolicyTemplate)
837     (COMMAND_ATTRIBUTES) (CC_PolicyTemplate * // 0x0190
838     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
839 #endif
840 #if (PAD_LIST || CC_CreateLoaded)
841     (COMMAND_ATTRIBUTES) (CC_CreateLoaded * // 0x0191
842     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
843 #endif
844 #if (PAD_LIST || CC_PolicyAuthorizeNV)

```

```

845         (COMMAND_ATTRIBUTES) (CC_PolicyAuthorizeNV          * // 0x0192
846         (IS_IMPLEMENTED+HANDLE_1_USER+ALLOW_TRIAL)),
847     #endif
848     #if (PAD_LIST || CC_EncryptDecrypt2)
849         (COMMAND_ATTRIBUTES) (CC_EncryptDecrypt2            * // 0x0193
850         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
851     #endif
852     #if (PAD_LIST || CC_AC_GetCapability)
853         (COMMAND_ATTRIBUTES) (CC_AC_GetCapability            * // 0x0194
854         (IS_IMPLEMENTED)),
855     #endif
856     #if (PAD_LIST || CC_AC_Send)
857         (COMMAND_ATTRIBUTES) (CC_AC_Send                     * // 0x0195
858         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+HANDLE_2_USER)),
859     #endif
860     #if (PAD_LIST || CC_Policy_AC_SendSelect)
861         (COMMAND_ATTRIBUTES) (CC_Policy_AC_SendSelect        * // 0x0196
862         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
863     #endif
864     #if (PAD_LIST || CC_CertifyX509)
865         (COMMAND_ATTRIBUTES) (CC_CertifyX509                 * // 0x0197
866         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
867     #endif
868     #if (PAD_LIST || CC_ACT_SetTimeout)
869         (COMMAND_ATTRIBUTES) (CC_ACT_SetTimeout              * // 0x0198
870         (IS_IMPLEMENTED+HANDLE_1_USER)),
871     #endif
872     #if (PAD_LIST || CC_ECC_Encrypt)
873         (COMMAND_ATTRIBUTES) (CC_ECC_Encrypt                  * // 0x0199
874         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
875     #endif
876     #if (PAD_LIST || CC_ECC_Decrypt)
877         (COMMAND_ATTRIBUTES) (CC_ECC_Decrypt                  * // 0x019A
878         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
879     #endif
880     #if (PAD_LIST || CC_Vendor_TCG_Test)
881         (COMMAND_ATTRIBUTES) (CC_Vendor_TCG_Test              * // 0x0000
882         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
883     #endif
884     0
885 };
886
887 #endif // _COMMAND_CODE_ATTRIBUTES_

```


5.5 CommandAttributes.h

The attributes defined in this file are produced by the parser that creates the structure definitions from Part 3. The attributes are defined in that parser and should track the attributes being tested in CommandCodeAttributes.c. Generally, when an attribute is added to this list, new code will be needed in CommandCodeAttributes.c to test it.

```

1  #ifndef COMMAND_ATTRIBUTES_H
2  #define COMMAND_ATTRIBUTES_H
3
4  typedef uint16_t COMMAND_ATTRIBUTES;
5  #define NOT_IMPLEMENTED ((COMMAND_ATTRIBUTES) 0)
6  #define ENCRYPT_2 ((COMMAND_ATTRIBUTES) 1 << 0)
7  #define ENCRYPT_4 ((COMMAND_ATTRIBUTES) 1 << 1)
8  #define DECRYPT_2 ((COMMAND_ATTRIBUTES) 1 << 2)
9  #define DECRYPT_4 ((COMMAND_ATTRIBUTES) 1 << 3)
10 #define HANDLE_1_USER ((COMMAND_ATTRIBUTES) 1 << 4)
11 #define HANDLE_1_ADMIN ((COMMAND_ATTRIBUTES) 1 << 5)
12 #define HANDLE_1_DUP ((COMMAND_ATTRIBUTES) 1 << 6)
13 #define HANDLE_2_USER ((COMMAND_ATTRIBUTES) 1 << 7)
14 #define PP_COMMAND ((COMMAND_ATTRIBUTES) 1 << 8)
15 #define IS_IMPLEMENTED ((COMMAND_ATTRIBUTES) 1 << 9)
16 #define NO_SESSIONS ((COMMAND_ATTRIBUTES) 1 << 10)
17 #define NV_COMMAND ((COMMAND_ATTRIBUTES) 1 << 11)
18 #define PP_REQUIRED ((COMMAND_ATTRIBUTES) 1 << 12)
19 #define R_HANDLE ((COMMAND_ATTRIBUTES) 1 << 13)
20 #define ALLOW_TRIAL ((COMMAND_ATTRIBUTES) 1 << 14)
21
22 #endif // COMMAND_ATTRIBUTES_H

```

5.6 CommandDispatchData.h

This file should only be included by CommandCodeAttributes.c

```
1  #ifndef _COMMAND_TABLE_DISPATCH_
```

Define the stop value

```
2  #define END_OF_LIST      0xff
3  #define ADD_FLAG        0x80
```

These macros provide some variability in how the data is encoded. They also make the lines a little shorter. ;-)

```
4  #if TABLE_DRIVEN_MARSHAL
5  #   define UNMARSHAL_DISPATCH(name)      (marshalIndex_t)name##_MARSHAL_REF
6  #   define MARSHAL_DISPATCH(name)        (marshalIndex_t)name##_MARSHAL_REF
7  #   define _UNMARSHAL_T_                  marshalIndex_t
8  #   define _MARSHAL_T_                    marshalIndex_t
9  #else
10 #   define UNMARSHAL_DISPATCH(name)      (UNMARSHAL_t)name##_Unmarshal
11 #   define MARSHAL_DISPATCH(name)        (MARSHAL_t)name##_Marshal
12 #   define _UNMARSHAL_T_                  UNMARSHAL_t
13 #   define _MARSHAL_T_                    MARSHAL_t
14 #endif
```

The *unmarshalArray* contains the dispatch functions for the unmarshaling code. The defines in this array are used to make it easier to cross reference the unmarshaling values in the types array of each command

```
15 const _UNMARSHAL_T_ unmarshalArray[] = {
16 #define TPMI_DH_CONTEXT_H_UNMARSHAL      0
17     UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
18 #define TPMI_RH_AC_H_UNMARSHAL            (TPMI_DH_CONTEXT_H_UNMARSHAL + 1)
19     UNMARSHAL_DISPATCH(TPMI_RH_AC),
20 #define TPMI_RH_ACT_H_UNMARSHAL           (TPMI_RH_AC_H_UNMARSHAL + 1)
21     UNMARSHAL_DISPATCH(TPMI_RH_ACT),
22 #define TPMI_RH_CLEAR_H_UNMARSHAL         (TPMI_RH_ACT_H_UNMARSHAL + 1)
23     UNMARSHAL_DISPATCH(TPMI_RH_CLEAR),
24 #define TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL (TPMI_RH_CLEAR_H_UNMARSHAL + 1)
25     UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY_AUTH),
26 #define TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL \
27     (TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL + 1)
28     UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY_POLICY),
29 #define TPMI_RH_LOCKOUT_H_UNMARSHAL       \
30     (TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL + 1)
31     UNMARSHAL_DISPATCH(TPMI_RH_LOCKOUT),
32 #define TPMI_RH_NV_AUTH_H_UNMARSHAL        (TPMI_RH_LOCKOUT_H_UNMARSHAL + 1)
33     UNMARSHAL_DISPATCH(TPMI_RH_NV_AUTH),
34 #define TPMI_RH_NV_INDEX_H_UNMARSHAL      (TPMI_RH_NV_AUTH_H_UNMARSHAL + 1)
35     UNMARSHAL_DISPATCH(TPMI_RH_NV_INDEX),
36 #define TPMI_RH_PLATFORM_H_UNMARSHAL      (TPMI_RH_NV_INDEX_H_UNMARSHAL + 1)
37     UNMARSHAL_DISPATCH(TPMI_RH_PLATFORM),
38 #define TPMI_RH_PROVISION_H_UNMARSHAL     (TPMI_RH_PLATFORM_H_UNMARSHAL + 1)
39     UNMARSHAL_DISPATCH(TPMI_RH_PROVISION),
40 #define TPMI_SH_HMAC_H_UNMARSHAL          (TPMI_RH_PROVISION_H_UNMARSHAL + 1)
41     UNMARSHAL_DISPATCH(TPMI_SH_HMAC),
42 #define TPMI_SH_POLICY_H_UNMARSHAL        (TPMI_SH_HMAC_H_UNMARSHAL + 1)
43     UNMARSHAL_DISPATCH(TPMI_SH_POLICY),
44 // HANDLE_FIRST_FLAG_TYPE is the first handle that needs a flag when called.
45 #define HANDLE_FIRST_FLAG_TYPE           (TPMI_SH_POLICY_H_UNMARSHAL + 1)
46 #define TPMI_DH_ENTITY_H_UNMARSHAL       (TPMI_SH_POLICY_H_UNMARSHAL + 1)
47     UNMARSHAL_DISPATCH(TPMI_DH_ENTITY),
```



```

48 #define TPMI_DH_OBJECT_H_UNMARSHAL (TPMI_DH_ENTITY_H_UNMARSHAL + 1)
49     UNMARSHAL_DISPATCH(TPMI_DH_OBJECT),
50 #define TPMI_DH_PARENT_H_UNMARSHAL (TPMI_DH_OBJECT_H_UNMARSHAL + 1)
51     UNMARSHAL_DISPATCH(TPMI_DH_PARENT),
52 #define TPMI_DH_PCR_H_UNMARSHAL (TPMI_DH_PARENT_H_UNMARSHAL + 1)
53     UNMARSHAL_DISPATCH(TPMI_DH_PCR),
54 #define TPMI_RH_ENDORSEMENT_H_UNMARSHAL (TPMI_DH_PCR_H_UNMARSHAL + 1)
55     UNMARSHAL_DISPATCH(TPMI_RH_ENDORSEMENT),
56 #define TPMI_RH_HIERARCHY_H_UNMARSHAL \
57     (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)
58     UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
59 // PARAMETER_FIRST_TYPE marks the end of the handle list.
60 #define PARAMETER_FIRST_TYPE (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
61 #define TPM2B_DATA_P_UNMARSHAL (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
62     UNMARSHAL_DISPATCH(TPM2B_DATA),
63 #define TPM2B_DIGEST_P_UNMARSHAL (TPM2B_DATA_P_UNMARSHAL + 1)
64     UNMARSHAL_DISPATCH(TPM2B_DIGEST),
65 #define TPM2B_ECC_PARAMETER_P_UNMARSHAL (TPM2B_DIGEST_P_UNMARSHAL + 1)
66     UNMARSHAL_DISPATCH(TPM2B_ECC_PARAMETER),
67 #define TPM2B_ECC_POINT_P_UNMARSHAL \
68     (TPM2B_ECC_PARAMETER_P_UNMARSHAL + 1)
69     UNMARSHAL_DISPATCH(TPM2B_ECC_POINT),
70 #define TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL (TPM2B_ECC_POINT_P_UNMARSHAL + 1)
71     UNMARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
72 #define TPM2B_EVENT_P_UNMARSHAL \
73     (TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL + 1)
74     UNMARSHAL_DISPATCH(TPM2B_EVENT),
75 #define TPM2B_ID_OBJECT_P_UNMARSHAL (TPM2B_EVENT_P_UNMARSHAL + 1)
76     UNMARSHAL_DISPATCH(TPM2B_ID_OBJECT),
77 #define TPM2B_IV_P_UNMARSHAL (TPM2B_ID_OBJECT_P_UNMARSHAL + 1)
78     UNMARSHAL_DISPATCH(TPM2B_IV),
79 #define TPM2B_MAX_BUFFER_P_UNMARSHAL (TPM2B_IV_P_UNMARSHAL + 1)
80     UNMARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
81 #define TPM2B_MAX_NV_BUFFER_P_UNMARSHAL (TPM2B_MAX_BUFFER_P_UNMARSHAL + 1)
82     UNMARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
83 #define TPM2B_NAME_P_UNMARSHAL \
84     (TPM2B_MAX_NV_BUFFER_P_UNMARSHAL + 1)
85     UNMARSHAL_DISPATCH(TPM2B_NAME),
86 #define TPM2B_NV_PUBLIC_P_UNMARSHAL (TPM2B_NAME_P_UNMARSHAL + 1)
87     UNMARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
88 #define TPM2B_PRIVATE_P_UNMARSHAL (TPM2B_NV_PUBLIC_P_UNMARSHAL + 1)
89     UNMARSHAL_DISPATCH(TPM2B_PRIVATE),
90 #define TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL (TPM2B_PRIVATE_P_UNMARSHAL + 1)
91     UNMARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
92 #define TPM2B_SENSITIVE_P_UNMARSHAL \
93     (TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL + 1)
94     UNMARSHAL_DISPATCH(TPM2B_SENSITIVE),
95 #define TPM2B_SENSITIVE_CREATE_P_UNMARSHAL (TPM2B_SENSITIVE_P_UNMARSHAL + 1)
96     UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_CREATE),
97 #define TPM2B_SENSITIVE_DATA_P_UNMARSHAL \
98     (TPM2B_SENSITIVE_CREATE_P_UNMARSHAL + 1)
99     UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
100 #define TPM2B_TEMPLATE_P_UNMARSHAL \
101     (TPM2B_SENSITIVE_DATA_P_UNMARSHAL + 1)
102     UNMARSHAL_DISPATCH(TPM2B_TEMPLATE),
103 #define TPM2B_TIMEOUT_P_UNMARSHAL (TPM2B_TEMPLATE_P_UNMARSHAL + 1)
104     UNMARSHAL_DISPATCH(TPM2B_TIMEOUT),
105 #define TPMI_DH_CONTEXT_P_UNMARSHAL (TPM2B_TIMEOUT_P_UNMARSHAL + 1)
106     UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
107 #define TPMI_DH_PERSISTENT_P_UNMARSHAL (TPMI_DH_CONTEXT_P_UNMARSHAL + 1)
108     UNMARSHAL_DISPATCH(TPMI_DH_PERSISTENT),
109 #define TPMI_ECC_CURVE_P_UNMARSHAL (TPMI_DH_PERSISTENT_P_UNMARSHAL + 1)
110     UNMARSHAL_DISPATCH(TPMI_ECC_CURVE),
111 #define TPMI_YES_NO_P_UNMARSHAL (TPMI_ECC_CURVE_P_UNMARSHAL + 1)
112     UNMARSHAL_DISPATCH(TPMI_YES_NO),
113 #define TPML_ALG_P_UNMARSHAL (TPMI_YES_NO_P_UNMARSHAL + 1)

```

```

114         UNMARSHAL_DISPATCH(TPML_ALG),
115 #define TPML_CC_P_UNMARSHAL (TPML_ALG_P_UNMARSHAL + 1)
116         UNMARSHAL_DISPATCH(TPML_CC),
117 #define TPML_DIGEST_P_UNMARSHAL (TPML_CC_P_UNMARSHAL + 1)
118         UNMARSHAL_DISPATCH(TPML_DIGEST),
119 #define TPML_DIGEST_VALUES_P_UNMARSHAL (TPML_DIGEST_P_UNMARSHAL + 1)
120         UNMARSHAL_DISPATCH(TPML_DIGEST_VALUES),
121 #define TPML_PCR_SELECTION_P_UNMARSHAL (TPML_DIGEST_VALUES_P_UNMARSHAL + 1)
122         UNMARSHAL_DISPATCH(TPML_PCR_SELECTION),
123 #define TPMS_CONTEXT_P_UNMARSHAL (TPML_PCR_SELECTION_P_UNMARSHAL + 1)
124         UNMARSHAL_DISPATCH(TPMS_CONTEXT),
125 #define TPMT_PUBLIC_PARMS_P_UNMARSHAL (TPMS_CONTEXT_P_UNMARSHAL + 1)
126         UNMARSHAL_DISPATCH(TPMT_PUBLIC_PARMS),
127 #define TPMT_TK_AUTH_P_UNMARSHAL (TPMT_PUBLIC_PARMS_P_UNMARSHAL + 1)
128         UNMARSHAL_DISPATCH(TPMT_TK_AUTH),
129 #define TPMT_TK_CREATION_P_UNMARSHAL (TPMT_TK_AUTH_P_UNMARSHAL + 1)
130         UNMARSHAL_DISPATCH(TPMT_TK_CREATION),
131 #define TPMT_TK_HASHCHECK_P_UNMARSHAL (TPMT_TK_CREATION_P_UNMARSHAL + 1)
132         UNMARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
133 #define TPMT_TK_VERIFIED_P_UNMARSHAL (TPMT_TK_HASHCHECK_P_UNMARSHAL + 1)
134         UNMARSHAL_DISPATCH(TPMT_TK_VERIFIED),
135 #define TPM_AT_P_UNMARSHAL (TPMT_TK_VERIFIED_P_UNMARSHAL + 1)
136         UNMARSHAL_DISPATCH(TPM_AT),
137 #define TPM_CAP_P_UNMARSHAL (TPM_AT_P_UNMARSHAL + 1)
138         UNMARSHAL_DISPATCH(TPM_CAP),
139 #define TPM_CLOCK_ADJUST_P_UNMARSHAL (TPM_CAP_P_UNMARSHAL + 1)
140         UNMARSHAL_DISPATCH(TPM_CLOCK_ADJUST),
141 #define TPM_EO_P_UNMARSHAL (TPM_CLOCK_ADJUST_P_UNMARSHAL + 1)
142         UNMARSHAL_DISPATCH(TPM_EO),
143 #define TPM_SE_P_UNMARSHAL (TPM_EO_P_UNMARSHAL + 1)
144         UNMARSHAL_DISPATCH(TPM_SE),
145 #define TPM_SU_P_UNMARSHAL (TPM_SE_P_UNMARSHAL + 1)
146         UNMARSHAL_DISPATCH(TPM_SU),
147 #define UINT16_P_UNMARSHAL (TPM_SU_P_UNMARSHAL + 1)
148         UNMARSHAL_DISPATCH(UINT16),
149 #define UINT32_P_UNMARSHAL (UINT16_P_UNMARSHAL + 1)
150         UNMARSHAL_DISPATCH(UINT32),
151 #define UINT64_P_UNMARSHAL (UINT32_P_UNMARSHAL + 1)
152         UNMARSHAL_DISPATCH(UINT64),
153 #define UINT8_P_UNMARSHAL (UINT64_P_UNMARSHAL + 1)
154         UNMARSHAL_DISPATCH(UINT8),
155 // PARAMETER_FIRST_FLAG_TYPE is the first parameter to need a flag.
156 #define PARAMETER_FIRST_FLAG_TYPE (UINT8_P_UNMARSHAL + 1)
157 #define TPM2B_PUBLIC_P_UNMARSHAL (UINT8_P_UNMARSHAL + 1)
158         UNMARSHAL_DISPATCH(TPM2B_PUBLIC),
159 #define TPMI_ALG_CIPHER_MODE_P_UNMARSHAL (TPM2B_PUBLIC_P_UNMARSHAL + 1)
160         UNMARSHAL_DISPATCH(TPMI_ALG_CIPHER_MODE),
161 #define TPMI_ALG_HASH_P_UNMARSHAL \
162         (TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + 1)
163         UNMARSHAL_DISPATCH(TPMI_ALG_HASH),
164 #define TPMI_ALG_MAC_SCHEME_P_UNMARSHAL (TPMI_ALG_HASH_P_UNMARSHAL + 1)
165         UNMARSHAL_DISPATCH(TPMI_ALG_MAC_SCHEME),
166 #define TPMI_DH_PCR_P_UNMARSHAL \
167         (TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + 1)
168         UNMARSHAL_DISPATCH(TPMI_DH_PCR),
169 #define TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL (TPMI_DH_PCR_P_UNMARSHAL + 1)
170         UNMARSHAL_DISPATCH(TPMI_ECC_KEY_EXCHANGE),
171 #define TPMI_RH_ENABLES_P_UNMARSHAL \
172         (TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL + 1)
173         UNMARSHAL_DISPATCH(TPMI_RH_ENABLES),
174 #define TPMI_RH_HIERARCHY_P_UNMARSHAL (TPMI_RH_ENABLES_P_UNMARSHAL + 1)
175         UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
176 #define TPMT_KDF_SCHEME_P_UNMARSHAL (TPMI_RH_HIERARCHY_P_UNMARSHAL + 1)
177         UNMARSHAL_DISPATCH(TPMT_KDF_SCHEME),
178 #define TPMT_RSA_DECRYPT_P_UNMARSHAL (TPMT_KDF_SCHEME_P_UNMARSHAL + 1)
179         UNMARSHAL_DISPATCH(TPMT_RSA_DECRYPT),

```

```

180 #define TPMT_SIGNATURE_P_UNMARSHAL (TPMT_RSA_DECRYPT_P_UNMARSHAL + 1)
181     UNMARSHAL_DISPATCH(TPMT_SIGNATURE),
182 #define TPMT_SIG_SCHEME_P_UNMARSHAL (TPMT_SIGNATURE_P_UNMARSHAL + 1)
183     UNMARSHAL_DISPATCH(TPMT_SIG_SCHEME),
184 #define TPMT_SYM_DEF_P_UNMARSHAL (TPMT_SIG_SCHEME_P_UNMARSHAL + 1)
185     UNMARSHAL_DISPATCH(TPMT_SYM_DEF),
186 #define TPMT_SYM_DEF_OBJECT_P_UNMARSHAL (TPMT_SYM_DEF_P_UNMARSHAL + 1)
187     UNMARSHAL_DISPATCH(TPMT_SYM_DEF_OBJECT)
188 // PARAMETER_LAST_TYPE is the end of the command parameter list.
189 #define PARAMETER_LAST_TYPE (TPMT_SYM_DEF_OBJECT_P_UNMARSHAL)
190 };

```

The *marshalArray* contains the dispatch functions for the marshaling code. The defines in this array are used to make it easier to cross reference the marshaling values in the types array of each command

```

191 const _MARSHAL_T marshalArray[] = {
192
193 #define UINT32_H_MARSHAL 0
194     MARSHAL_DISPATCH(UINT32),
195 // RESPONSE_PARAMETER_FIRST_TYPE marks the end of the response handles.
196 #define RESPONSE_PARAMETER_FIRST_TYPE (UINT32_H_MARSHAL + 1)
197 #define TPM2B_ATTEST_P_MARSHAL (UINT32_H_MARSHAL + 1)
198     MARSHAL_DISPATCH(TPM2B_ATTEST),
199 #define TPM2B_CREATION_DATA_P_MARSHAL (TPM2B_ATTEST_P_MARSHAL + 1)
200     MARSHAL_DISPATCH(TPM2B_CREATION_DATA),
201 #define TPM2B_DATA_P_MARSHAL (TPM2B_CREATION_DATA_P_MARSHAL + 1)
202     MARSHAL_DISPATCH(TPM2B_DATA),
203 #define TPM2B_DIGEST_P_MARSHAL (TPM2B_DATA_P_MARSHAL + 1)
204     MARSHAL_DISPATCH(TPM2B_DIGEST),
205 #define TPM2B_ECC_POINT_P_MARSHAL (TPM2B_DIGEST_P_MARSHAL + 1)
206     MARSHAL_DISPATCH(TPM2B_ECC_POINT),
207 #define TPM2B_ENCRYPTED_SECRET_P_MARSHAL (TPM2B_ECC_POINT_P_MARSHAL + 1)
208     MARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
209 #define TPM2B_ID_OBJECT_P_MARSHAL \
210     (TPM2B_ENCRYPTED_SECRET_P_MARSHAL + 1)
211     MARSHAL_DISPATCH(TPM2B_ID_OBJECT),
212 #define TPM2B_IV_P_MARSHAL (TPM2B_ID_OBJECT_P_MARSHAL + 1)
213     MARSHAL_DISPATCH(TPM2B_IV),
214 #define TPM2B_MAX_BUFFER_P_MARSHAL (TPM2B_IV_P_MARSHAL + 1)
215     MARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
216 #define TPM2B_MAX_NV_BUFFER_P_MARSHAL (TPM2B_MAX_BUFFER_P_MARSHAL + 1)
217     MARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
218 #define TPM2B_NAME_P_MARSHAL (TPM2B_MAX_NV_BUFFER_P_MARSHAL + 1)
219     MARSHAL_DISPATCH(TPM2B_NAME),
220 #define TPM2B_NV_PUBLIC_P_MARSHAL (TPM2B_NAME_P_MARSHAL + 1)
221     MARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
222 #define TPM2B_PRIVATE_P_MARSHAL (TPM2B_NV_PUBLIC_P_MARSHAL + 1)
223     MARSHAL_DISPATCH(TPM2B_PRIVATE),
224 #define TPM2B_PUBLIC_P_MARSHAL (TPM2B_PRIVATE_P_MARSHAL + 1)
225     MARSHAL_DISPATCH(TPM2B_PUBLIC),
226 #define TPM2B_PUBLIC_KEY_RSA_P_MARSHAL (TPM2B_PUBLIC_P_MARSHAL + 1)
227     MARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
228 #define TPM2B_SENSITIVE_DATA_P_MARSHAL (TPM2B_PUBLIC_KEY_RSA_P_MARSHAL + 1)
229     MARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
230 #define TPM2B_TIMEOUT_P_MARSHAL (TPM2B_SENSITIVE_DATA_P_MARSHAL + 1)
231     MARSHAL_DISPATCH(TPM2B_TIMEOUT),
232 #define UINT8_P_MARSHAL (TPM2B_TIMEOUT_P_MARSHAL + 1)
233     MARSHAL_DISPATCH(UINT8),
234 #define TPML_AC_CAPABILITIES_P_MARSHAL (UINT8_P_MARSHAL + 1)
235     MARSHAL_DISPATCH(TPML_AC_CAPABILITIES),
236 #define TPML_ALG_P_MARSHAL (TPML_AC_CAPABILITIES_P_MARSHAL + 1)
237     MARSHAL_DISPATCH(TPML_ALG),
238 #define TPML_DIGEST_P_MARSHAL (TPML_ALG_P_MARSHAL + 1)
239     MARSHAL_DISPATCH(TPML_DIGEST),
240 #define TPML_DIGEST_VALUES_P_MARSHAL (TPML_DIGEST_P_MARSHAL + 1)

```

```

241     MARSHAL_DISPATCH(TPML_DIGEST_VALUES),
242 #define TPML_PCR_SELECTION_P_MARSHAL (TPML_DIGEST_VALUES_P_MARSHAL + 1)
243     MARSHAL_DISPATCH(TPML_PCR_SELECTION),
244 #define TPMS_AC_OUTPUT_P_MARSHAL (TPML_PCR_SELECTION_P_MARSHAL + 1)
245     MARSHAL_DISPATCH(TPMS_AC_OUTPUT),
246 #define TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL (TPMS_AC_OUTPUT_P_MARSHAL + 1)
247     MARSHAL_DISPATCH(TPMS_ALGORITHM_DETAIL_ECC),
248 #define TPMS_CAPABILITY_DATA_P_MARSHAL \
249     (TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL + 1)
250     MARSHAL_DISPATCH(TPMS_CAPABILITY_DATA),
251 #define TPMS_CONTEXT_P_MARSHAL (TPMS_CAPABILITY_DATA_P_MARSHAL + 1)
252     MARSHAL_DISPATCH(TPMS_CONTEXT),
253 #define TPMS_TIME_INFO_P_MARSHAL (TPMS_CONTEXT_P_MARSHAL + 1)
254     MARSHAL_DISPATCH(TPMS_TIME_INFO),
255 #define TPMT_HA_P_MARSHAL (TPMS_TIME_INFO_P_MARSHAL + 1)
256     MARSHAL_DISPATCH(TPMT_HA),
257 #define TPMT_SIGNATURE_P_MARSHAL (TPMT_HA_P_MARSHAL + 1)
258     MARSHAL_DISPATCH(TPMT_SIGNATURE),
259 #define TPMT_TK_AUTH_P_MARSHAL (TPMT_SIGNATURE_P_MARSHAL + 1)
260     MARSHAL_DISPATCH(TPMT_TK_AUTH),
261 #define TPMT_TK_CREATION_P_MARSHAL (TPMT_TK_AUTH_P_MARSHAL + 1)
262     MARSHAL_DISPATCH(TPMT_TK_CREATION),
263 #define TPMT_TK_HASHCHECK_P_MARSHAL (TPMT_TK_CREATION_P_MARSHAL + 1)
264     MARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
265 #define TPMT_TK_VERIFIED_P_MARSHAL (TPMT_TK_HASHCHECK_P_MARSHAL + 1)
266     MARSHAL_DISPATCH(TPMT_TK_VERIFIED),
267 #define UINT32_P_MARSHAL (TPMT_TK_VERIFIED_P_MARSHAL + 1)
268     MARSHAL_DISPATCH(UINT32),
269 #define UINT16_P_MARSHAL (UINT32_P_MARSHAL + 1)
270     MARSHAL_DISPATCH(UINT16)
271 // RESPONSE_PARAMETER_LAST_TYPE is the end of the response parameter list.
272 #define RESPONSE_PARAMETER_LAST_TYPE (UINT16_P_MARSHAL)
273 };

```

This list of aliases allows the types in the `_COMMAND_DESCRIPTOR_T` to match the types in the command/response templates of part 3.

```

274 #define INT32_P_UNMARSHAL      UINT32_P_UNMARSHAL
275 #define TPM2B_AUTH_P_UNMARSHAL TPM2B_DIGEST_P_UNMARSHAL
276 #define TPM2B_NONCE_P_UNMARSHAL TPM2B_DIGEST_P_UNMARSHAL
277 #define TPM2B_OPERAND_P_UNMARSHAL TPM2B_DIGEST_P_UNMARSHAL
278 #define TPMA_LOCALITY_P_UNMARSHAL UINT8_P_UNMARSHAL
279 #define TPM_CC_P_UNMARSHAL      UINT32_P_UNMARSHAL
280 #define TPMI_DH_CONTEXT_H_MARSHAL UINT32_H_MARSHAL
281 #define TPMI_DH_OBJECT_H_MARSHAL  UINT32_H_MARSHAL
282 #define TPMI_SH_AUTH_SESSION_H_MARSHAL UINT32_H_MARSHAL
283 #define TPM_HANDLE_H_MARSHAL      UINT32_H_MARSHAL
284 #define TPM2B_NONCE_P_MARSHAL    TPM2B_DIGEST_P_MARSHAL
285 #define TPMI_YES_NO_P_MARSHAL    UINT8_P_MARSHAL
286 #define TPM_RC_P_MARSHAL        UINT32_P_MARSHAL
287
288 #if CC_Startup
289
290 #include "Startup_fp.h"
291
292 typedef TPM_RC (Startup_Entry) (
293     Startup_In *in
294 );
295
296 typedef const struct {
297     Startup_Entry *entry;
298     UINT16 inSize;
299     UINT16 outSize;
300     UINT16 offsetOfTypes;
301     BYTE types[3];

```

```

302 } Startup_COMMAND_DESCRIPTOR_t;
303
304 Startup_COMMAND_DESCRIPTOR_t _StartupData = {
305     /* entry */          /* &TPM2_Startup,
306     /* inSize */         /* (UINT16) (sizeof(Startup_In)),
307     /* outSize */        /* 0,
308     /* offsetOfTypes */  /* offsetof(Startup_COMMAND_DESCRIPTOR_t, types),
309     /* offsets */        /* // No parameter offsets;
310     /* types */          /* {TPM_SU_P_UNMARSHAL,
311                           /*     END_OF_LIST,
312                           /*     END_OF_LIST}
313 };
314
315 #define _StartupDataAddress (&_StartupData)
316 #else
317 #define _StartupDataAddress 0
318 #endif // CC_Startup
319
320 #if CC_Shutdown
321
322 #include "Shutdown_fp.h"
323
324 typedef TPM_RC (Shutdown_Entry)(
325     Shutdown_In          *in
326 );
327
328 typedef const struct {
329     Shutdown_Entry      *entry;
330     UINT16               inSize;
331     UINT16               outSize;
332     UINT16               offsetOfTypes;
333     BYTE                 types[3];
334 } Shutdown_COMMAND_DESCRIPTOR_t;
335
336 Shutdown_COMMAND_DESCRIPTOR_t _ShutdownData = {
337     /* entry */          /* &TPM2_Shutdown,
338     /* inSize */         /* (UINT16) (sizeof(Shutdown_In)),
339     /* outSize */        /* 0,
340     /* offsetOfTypes */  /* offsetof(Shutdown_COMMAND_DESCRIPTOR_t, types),
341     /* offsets */        /* // No parameter offsets;
342     /* types */          /* {TPM_SU_P_UNMARSHAL,
343                           /*     END_OF_LIST,
344                           /*     END_OF_LIST}
345 };
346
347 #define _ShutdownDataAddress (&_ShutdownData)
348 #else
349 #define _ShutdownDataAddress 0
350 #endif // CC_Shutdown
351
352 #if CC_SelfTest
353
354 #include "SelfTest_fp.h"
355
356 typedef TPM_RC (SelfTest_Entry)(
357     SelfTest_In          *in
358 );
359
360 typedef const struct {
361     SelfTest_Entry      *entry;
362     UINT16               inSize;
363     UINT16               outSize;
364     UINT16               offsetOfTypes;
365     BYTE                 types[3];
366 } SelfTest_COMMAND_DESCRIPTOR_t;
367

```



```

368 SelfTest_COMMAND_DESCRIPTOR_t _SelfTestData = {
369     /* entry */ &TPM2_SelfTest,
370     /* inSize */ (UINT16) (sizeof(SelfTest_In)),
371     /* outSize */ 0,
372     /* offsetOfTypes */ offsetof(SelfTest_COMMAND_DESCRIPTOR_t, types),
373     /* offsets */ // No parameter offsets;
374     /* types */ {TPMI_YES_NO_P_UNMARSHAL,
375                 END_OF_LIST,
376                 END_OF_LIST};
377 };
378
379 #define _SelfTestDataAddress (&_SelfTestData)
380 #else
381 #define _SelfTestDataAddress 0
382 #endif // CC_SelfTest
383
384 #if CC_IncrementalSelfTest
385
386 #include "IncrementalSelfTest_fp.h"
387
388 typedef TPM_RC (IncrementalSelfTest_Entry) (
389     IncrementalSelfTest_In *in,
390     IncrementalSelfTest_Out *out
391 );
392
393 typedef const struct {
394     IncrementalSelfTest_Entry *entry;
395     UINT16 inSize;
396     UINT16 outSize;
397     UINT16 offsetOfTypes;
398     BYTE types[4];
399 } IncrementalSelfTest_COMMAND_DESCRIPTOR_t;
400
401 IncrementalSelfTest_COMMAND_DESCRIPTOR_t _IncrementalSelfTestData = {
402     /* entry */ &TPM2_IncrementalSelfTest,
403     /* inSize */ (UINT16) (sizeof(IncrementalSelfTest_In)),
404     /* outSize */ (UINT16) (sizeof(IncrementalSelfTest_Out)),
405     /* offsetOfTypes */ offsetof(IncrementalSelfTest_COMMAND_DESCRIPTOR_t,
406     types),
407     /* offsets */ // No parameter offsets;
408     /* types */ {TPML_ALG_P_UNMARSHAL,
409                 END_OF_LIST,
410                 TPML_ALG_P_MARSHAL,
411                 END_OF_LIST};
412 };
413
414 #define _IncrementalSelfTestDataAddress (&_IncrementalSelfTestData)
415 #else
416 #define _IncrementalSelfTestDataAddress 0
417 #endif // CC_IncrementalSelfTest
418
419 #if CC_GetTestResult
420
421 #include "GetTestResult_fp.h"
422
423 typedef TPM_RC (GetTestResult_Entry) (
424     GetTestResult_Out *out
425 );
426
427 typedef const struct {
428     GetTestResult_Entry *entry;
429     UINT16 inSize;
430     UINT16 outSize;
431     UINT16 offsetOfTypes;
432     paramOffsets[1];
433     BYTE types[4];

```

```

433 } GetTestResult_COMMAND_DESCRIPTOR_t;
434
435 GetTestResult_COMMAND_DESCRIPTOR_t _GetTestResultData = {
436     /* entry */ &TPM2_GetTestResult,
437     /* inSize */ 0,
438     /* outSize */ (UINT16) (sizeof(GetTestResult_Out)),
439     /* offsetOfTypes */ offsetof(GetTestResult_COMMAND_DESCRIPTOR_t, types),
440     /* offsets */ { (UINT16) (offsetof(GetTestResult_Out, testResult))},
441     /* types */ {END_OF_LIST,
442                 TPM2B_MAX_BUFFER_P_MARSHAL,
443                 TPM_RC_P_MARSHAL,
444                 END_OF_LIST};
445 };
446
447 #define _GetTestResultDataAddress (&_GetTestResultData)
448 #else
449 #define _GetTestResultDataAddress 0
450 #endif // CC_GetTestResult
451
452 #if CC_StartAuthSession
453
454 #include "StartAuthSession_fp.h"
455
456 typedef TPM_RC (StartAuthSession_Entry) (
457     StartAuthSession_In *in,
458     StartAuthSession_Out *out
459 );
460
461 typedef const struct {
462     StartAuthSession_Entry *entry;
463     UINT16 inSize;
464     UINT16 outSize;
465     UINT16 offsetOfTypes;
466     UINT16 paramOffsets[7];
467     BYTE types[11];
468 } StartAuthSession_COMMAND_DESCRIPTOR_t;
469
470 StartAuthSession_COMMAND_DESCRIPTOR_t _StartAuthSessionData = {
471     /* entry */ &TPM2_StartAuthSession,
472     /* inSize */ (UINT16) (sizeof(StartAuthSession_In)),
473     /* outSize */ (UINT16) (sizeof(StartAuthSession_Out)),
474     /* offsetOfTypes */ offsetof(StartAuthSession_COMMAND_DESCRIPTOR_t, types),
475     /* offsets */ { (UINT16) (offsetof(StartAuthSession_In, bind)),
476                   (UINT16) (offsetof(StartAuthSession_In, nonceCaller)),
477                   (UINT16) (offsetof(StartAuthSession_In, encryptedSalt)),
478                   (UINT16) (offsetof(StartAuthSession_In, sessionType)),
479                   (UINT16) (offsetof(StartAuthSession_In, symmetric)),
480                   (UINT16) (offsetof(StartAuthSession_In, authHash)),
481                   (UINT16) (offsetof(StartAuthSession_Out, nonceTPM))},
482     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
483                 TPMI_DH_ENTITY_H_UNMARSHAL + ADD_FLAG,
484                 TPM2B_NONCE_P_UNMARSHAL,
485                 TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
486                 TPM_SE_P_UNMARSHAL,
487                 TPMT_SYM_DEF_P_UNMARSHAL + ADD_FLAG,
488                 TPMI_ALG_HASH_P_UNMARSHAL,
489                 END_OF_LIST,
490                 TPMI_SH_AUTH_SESSION_H_MARSHAL,
491                 TPM2B_NONCE_P_MARSHAL,
492                 END_OF_LIST};
493 };
494
495 #define _StartAuthSessionDataAddress (&_StartAuthSessionData)
496 #else
497 #define _StartAuthSessionDataAddress 0
498 #endif // CC_StartAuthSession

```

```

499
500 #if CC_PolicyRestart
501
502 #include "PolicyRestart_fp.h"
503
504 typedef TPM_RC (PolicyRestart_Entry) (
505     PolicyRestart_In      *in
506 );
507
508 typedef const struct {
509     PolicyRestart_Entry    *entry;
510     UINT16                 inSize;
511     UINT16                 outSize;
512     UINT16                 offsetOfTypes;
513     BYTE                   types[3];
514 } PolicyRestart_COMMAND_DESCRIPTOR_t;
515
516 PolicyRestart_COMMAND_DESCRIPTOR_t _PolicyRestartData = {
517     /* entry */           &TPM2_PolicyRestart,
518     /* inSize */          (UINT16) (sizeof(PolicyRestart_In)),
519     /* outSize */         0,
520     /* offsetOfTypes */   offsetof(PolicyRestart_COMMAND_DESCRIPTOR_t, types),
521     /* offsets */          // No parameter offsets;
522     /* types */            {TPMI_SH_POLICY_H_UNMARSHAL,
523                             END_OF_LIST,
524                             END_OF_LIST}
525 };
526
527 #define _PolicyRestartDataAddress (&_PolicyRestartData)
528 #else
529 #define _PolicyRestartDataAddress 0
530 #endif // CC_PolicyRestart
531
532 #if CC_Create
533
534 #include "Create_fp.h"
535
536 typedef TPM_RC (Create_Entry) (
537     Create_In              *in,
538     Create_Out             *out
539 );
540
541 typedef const struct {
542     Create_Entry           *entry;
543     UINT16                 inSize;
544     UINT16                 outSize;
545     UINT16                 offsetOfTypes;
546     UINT16                 paramOffsets[8];
547     BYTE                   types[12];
548 } Create_COMMAND_DESCRIPTOR_t;
549
550 Create_COMMAND_DESCRIPTOR_t _CreateData = {
551     /* entry */           &TPM2_Create,
552     /* inSize */          (UINT16) (sizeof(Create_In)),
553     /* outSize */         (UINT16) (sizeof(Create_Out)),
554     /* offsetOfTypes */   offsetof(Create_COMMAND_DESCRIPTOR_t, types),
555     /* offsets */          {(UINT16) (offsetof(Create_In, inSensitive)),
556                             (UINT16) (offsetof(Create_In, inPublic)),
557                             (UINT16) (offsetof(Create_In, outsideInfo)),
558                             (UINT16) (offsetof(Create_In, creationPCR)),
559                             (UINT16) (offsetof(Create_Out, outPublic)),
560                             (UINT16) (offsetof(Create_Out, creationData)),
561                             (UINT16) (offsetof(Create_Out, creationHash)),
562                             (UINT16) (offsetof(Create_Out, creationTicket))},
563     /* types */            {TPMI_DH_OBJECT_H_UNMARSHAL,
564                             TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,

```



```

565         TPM2B_PUBLIC_P_UNMARSHAL,
566         TPM2B_DATA_P_UNMARSHAL,
567         TPML_PCR_SELECTION_P_UNMARSHAL,
568         END_OF_LIST,
569         TPM2B_PRIVATE_P_MARSHAL,
570         TPM2B_PUBLIC_P_MARSHAL,
571         TPM2B_CREATION_DATA_P_MARSHAL,
572         TPM2B_DIGEST_P_MARSHAL,
573         TPMT_TK_CREATION_P_MARSHAL,
574         END_OF_LIST}
575 };
576
577 #define _CreateDataAddress (&_CreateData)
578 #else
579 #define _CreateDataAddress 0
580 #endif // CC_Create
581
582 #if CC_Load
583
584 #include "Load_fp.h"
585
586 typedef TPM_RC (Load_Entry) (
587     Load_In          *in,
588     Load_Out         *out
589 );
590
591 typedef const struct {
592     Load_Entry        *entry;
593     UINT16            inSize;
594     UINT16            outSize;
595     UINT16            offsetOfTypes;
596     UINT16            paramOffsets[3];
597     BYTE              types[7];
598 } Load_COMMAND_DESCRIPTOR_t;
599
600 Load_COMMAND_DESCRIPTOR_t _LoadData = {
601     /* entry */          &TPM2_Load,
602     /* inSize */         (UINT16) (sizeof(Load_In)),
603     /* outSize */        (UINT16) (sizeof(Load_Out)),
604     /* offsetOfTypes */  offsetof(Load_COMMAND_DESCRIPTOR_t, types),
605     /* offsets */        {(UINT16) (offsetof(Load_In, inPrivate)),
606                          (UINT16) (offsetof(Load_In, inPublic)),
607                          (UINT16) (offsetof(Load_Out, name))},
608     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
609                          TPM2B_PRIVATE_P_UNMARSHAL,
610                          TPM2B_PUBLIC_P_UNMARSHAL,
611                          END_OF_LIST,
612                          TPM_HANDLE_H_MARSHAL,
613                          TPM2B_NAME_P_MARSHAL,
614                          END_OF_LIST}
615 };
616
617 #define _LoadDataAddress (&_LoadData)
618 #else
619 #define _LoadDataAddress 0
620 #endif // CC_Load
621
622 #if CC_LoadExternal
623
624 #include "LoadExternal_fp.h"
625
626 typedef TPM_RC (LoadExternal_Entry) (
627     LoadExternal_In   *in,
628     LoadExternal_Out   *out
629 );
630

```

```

631 typedef const struct {
632     LoadExternal_Entry    *entry;
633     UINT16                 inSize;
634     UINT16                 outSize;
635     UINT16                 offsetOfTypes;
636     UINT16                 paramOffsets[3];
637     BYTE                   types[7];
638 } LoadExternal_COMMAND_DESCRIPTOR_t;
639
640 LoadExternal_COMMAND_DESCRIPTOR_t _LoadExternalData = {
641     /* entry */           &TPM2_LoadExternal,
642     /* inSize */          (UINT16) (sizeof(LoadExternal_In)),
643     /* outSize */         (UINT16) (sizeof(LoadExternal_Out)),
644     /* offsetOfTypes */   offsetof(LoadExternal_COMMAND_DESCRIPTOR_t, types),
645     /* offsets */         { (UINT16) (offsetof(LoadExternal_In, inPublic)),
646                           (UINT16) (offsetof(LoadExternal_In, hierarchy)),
647                           (UINT16) (offsetof(LoadExternal_Out, name)) },
648     /* types */           { TPM2B_SENSITIVE_P_UNMARSHAL,
649                           TPM2B_PUBLIC_P_UNMARSHAL + ADD_FLAG,
650                           TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
651                           END_OF_LIST,
652                           TPM_HANDLE_H_MARSHAL,
653                           TPM2B_NAME_P_MARSHAL,
654                           END_OF_LIST };
655 };
656
657 #define _LoadExternalDataAddress (&_LoadExternalData)
658 #else
659 #define _LoadExternalDataAddress 0
660 #endif // CC_LoadExternal
661
662 #if CC_ReadPublic
663
664 #include "ReadPublic_fp.h"
665
666 typedef TPM_RC (ReadPublic_Entry) (
667     ReadPublic_In          *in,
668     ReadPublic_Out         *out
669 );
670
671 typedef const struct {
672     ReadPublic_Entry        *entry;
673     UINT16                 inSize;
674     UINT16                 outSize;
675     UINT16                 offsetOfTypes;
676     UINT16                 paramOffsets[2];
677     BYTE                   types[6];
678 } ReadPublic_COMMAND_DESCRIPTOR_t;
679
680 ReadPublic_COMMAND_DESCRIPTOR_t _ReadPublicData = {
681     /* entry */           &TPM2_ReadPublic,
682     /* inSize */          (UINT16) (sizeof(ReadPublic_In)),
683     /* outSize */         (UINT16) (sizeof(ReadPublic_Out)),
684     /* offsetOfTypes */   offsetof(ReadPublic_COMMAND_DESCRIPTOR_t, types),
685     /* offsets */         { (UINT16) (offsetof(ReadPublic_Out, name)),
686                           (UINT16) (offsetof(ReadPublic_Out, qualifiedName)) },
687     /* types */           { TPMI_DH_OBJECT_H_UNMARSHAL,
688                           END_OF_LIST,
689                           TPM2B_PUBLIC_P_MARSHAL,
690                           TPM2B_NAME_P_MARSHAL,
691                           TPM2B_NAME_P_MARSHAL,
692                           END_OF_LIST };
693 };
694
695 #define _ReadPublicDataAddress (&_ReadPublicData)
696 #else

```

```

697 #define ReadPublicDataAddress 0
698 #endif // CC_ReadPublic
699
700 #if CC_ActivateCredential
701
702 #include "ActivateCredential_fp.h"
703
704 typedef TPM_RC (ActivateCredential_Entry) (
705     ActivateCredential_In      *in,
706     ActivateCredential_Out     *out
707 );
708
709 typedef const struct {
710     ActivateCredential_Entry    *entry;
711     UINT16                      inSize;
712     UINT16                      outSize;
713     UINT16                      offsetOfTypes;
714     UINT16                      paramOffsets[3];
715     BYTE                        types[7];
716 } ActivateCredential_COMMAND_DESCRIPTOR_t;
717
718 ActivateCredential_COMMAND_DESCRIPTOR_t _ActivateCredentialData = {
719     /* entry */           &TPM2_ActivateCredential,
720     /* inSize */          (UINT16) (sizeof(ActivateCredential_In)),
721     /* outSize */         (UINT16) (sizeof(ActivateCredential_Out)),
722     /* offsetOfTypes */   offsetof(ActivateCredential_COMMAND_DESCRIPTOR_t,
723     types),
724     /* offsets */         { (UINT16) (offsetof(ActivateCredential_In, keyHandle)),
725                             (UINT16) (offsetof(ActivateCredential_In,
726 credentialBlob)),
727                             (UINT16) (offsetof(ActivateCredential_In, secret))},
728     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
729                             TPMI_DH_OBJECT_H_UNMARSHAL,
730                             TPM2B_ID_OBJECT_P_UNMARSHAL,
731                             TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
732                             END_OF_LIST,
733                             TPM2B_DIGEST_P_MARSHAL,
734                             END_OF_LIST};
735 };
736
737 #define _ActivateCredentialDataAddress (&_ActivateCredentialData)
738 #else
739 #define _ActivateCredentialDataAddress 0
740 #endif // CC_ActivateCredential
741
742 #if CC_MakeCredential
743
744 #include "MakeCredential_fp.h"
745
746 typedef TPM_RC (MakeCredential_Entry) (
747     MakeCredential_In      *in,
748     MakeCredential_Out     *out
749 );
750
751 typedef const struct {
752     MakeCredential_Entry    *entry;
753     UINT16                  inSize;
754     UINT16                  outSize;
755     UINT16                  offsetOfTypes;
756     UINT16                  paramOffsets[3];
757     BYTE                    types[7];
758 } MakeCredential_COMMAND_DESCRIPTOR_t;
759
760 MakeCredential_COMMAND_DESCRIPTOR_t _MakeCredentialData = {
761     /* entry */           &TPM2_MakeCredential,
762     /* inSize */          (UINT16) (sizeof(MakeCredential_In)),

```

```

761     /* outSize */      (UINT16) (sizeof(MakeCredential_Out)),
762     /* offsetOfTypes */  offsetof(MakeCredential_COMMAND_DESCRIPTOR_t, types),
763     /* offsets */        { (UINT16) (offsetof(MakeCredential_In, credential)),
764                           (UINT16) (offsetof(MakeCredential_In, objectName)),
765                           (UINT16) (offsetof(MakeCredential_Out, secret))},
766     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
767                           TPM2B_DIGEST_P_UNMARSHAL,
768                           TPM2B_NAME_P_UNMARSHAL,
769                           END_OF_LIST,
770                           TPM2B_ID_OBJECT_P_MARSHAL,
771                           TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
772                           END_OF_LIST}
773 };
774
775 #define _MakeCredentialDataAddress (&_MakeCredentialData)
776 #else
777 #define _MakeCredentialDataAddress 0
778 #endif // CC_MakeCredential
779
780 #if CC_Unseal
781
782 #include "Unseal_fp.h"
783
784 typedef TPM_RC (Unseal_Entry) (
785     Unseal_In          *in,
786     Unseal_Out         *out
787 );
788
789 typedef const struct {
790     Unseal_Entry        *entry;
791     UINT16              inSize;
792     UINT16              outSize;
793     UINT16              offsetOfTypes;
794     BYTE                types[4];
795 } Unseal_COMMAND_DESCRIPTOR_t;
796
797 Unseal_COMMAND_DESCRIPTOR_t _UnsealData = {
798     /* entry */          &TPM2_Unseal,
799     /* inSize */         (UINT16) (sizeof(Unseal_In)),
800     /* outSize */        (UINT16) (sizeof(Unseal_Out)),
801     /* offsetOfTypes */  offsetof(Unseal_COMMAND_DESCRIPTOR_t, types),
802     /* offsets */        // No parameter offsets;
803     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
804                           END_OF_LIST,
805                           TPM2B_SENSITIVE_DATA_P_MARSHAL,
806                           END_OF_LIST}
807 };
808
809 #define _UnsealDataAddress (&_UnsealData)
810 #else
811 #define _UnsealDataAddress 0
812 #endif // CC_Unseal
813
814 #if CC_ObjectChangeAuth
815
816 #include "ObjectChangeAuth_fp.h"
817
818 typedef TPM_RC (ObjectChangeAuth_Entry) (
819     ObjectChangeAuth_In *in,
820     ObjectChangeAuth_Out *out
821 );
822
823 typedef const struct {
824     ObjectChangeAuth_Entry *entry;
825     UINT16                 inSize;
826     UINT16                 outSize;

```

```

827     UINT16                offsetOfTypes;
828     UINT16                paramOffsets[2];
829     BYTE                  types[6];
830 } ObjectChangeAuth_COMMAND_DESCRIPTOR_t;
831
832 ObjectChangeAuth_COMMAND_DESCRIPTOR_t _ObjectChangeAuthData = {
833     /* entry */           &TPM2_ObjectChangeAuth,
834     /* inSize */          (UINT16) (sizeof(ObjectChangeAuth_In)),
835     /* outSize */         (UINT16) (sizeof(ObjectChangeAuth_Out)),
836     /* offsetOfTypes */   offsetof(ObjectChangeAuth_COMMAND_DESCRIPTOR_t, types),
837     /* offsets */         { (UINT16) (offsetof(ObjectChangeAuth_In, parentHandle)),
838                           (UINT16) (offsetof(ObjectChangeAuth_In, newAuth)) },
839     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
840                           TPMI_DH_OBJECT_H_UNMARSHAL,
841                           TPM2B_AUTH_P_UNMARSHAL,
842                           END_OF_LIST,
843                           TPM2B_PRIVATE_P_MARSHAL,
844                           END_OF_LIST};
845 };
846
847 #define _ObjectChangeAuthDataAddress (&_ObjectChangeAuthData)
848 #else
849 #define _ObjectChangeAuthDataAddress 0
850 #endif // CC_ObjectChangeAuth
851
852 #if CC_CreateLoaded
853
854 #include "CreateLoaded_fp.h"
855
856 typedef TPM_RC (CreateLoaded_Entry) (
857     CreateLoaded_In          *in,
858     CreateLoaded_Out         *out
859 );
860
861 typedef const struct {
862     CreateLoaded_Entry        *entry;
863     UINT16                    inSize;
864     UINT16                    outSize;
865     UINT16                    offsetOfTypes;
866     UINT16                    paramOffsets[5];
867     BYTE                      types[9];
868 } CreateLoaded_COMMAND_DESCRIPTOR_t;
869
870 CreateLoaded_COMMAND_DESCRIPTOR_t _CreateLoadedData = {
871     /* entry */           &TPM2_CreateLoaded,
872     /* inSize */          (UINT16) (sizeof(CreateLoaded_In)),
873     /* outSize */         (UINT16) (sizeof(CreateLoaded_Out)),
874     /* offsetOfTypes */   offsetof(CreateLoaded_COMMAND_DESCRIPTOR_t, types),
875     /* offsets */         { (UINT16) (offsetof(CreateLoaded_In, inSensitive)),
876                           (UINT16) (offsetof(CreateLoaded_In, inPublic)),
877                           (UINT16) (offsetof(CreateLoaded_Out, outPrivate)),
878                           (UINT16) (offsetof(CreateLoaded_Out, outPublic)),
879                           (UINT16) (offsetof(CreateLoaded_Out, name)) },
880     /* types */           {TPMI_DH_PARENT_H_UNMARSHAL + ADD_FLAG,
881                           TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
882                           TPM2B_TEMPLATE_P_UNMARSHAL,
883                           END_OF_LIST,
884                           TPM_HANDLE_H_MARSHAL,
885                           TPM2B_PRIVATE_P_MARSHAL,
886                           TPM2B_PUBLIC_P_MARSHAL,
887                           TPM2B_NAME_P_MARSHAL,
888                           END_OF_LIST};
889 };
890
891 #define _CreateLoadedDataAddress (&_CreateLoadedData)
892 #else

```

```

893 #define _CreateLoadedDataAddress 0
894 #endif // CC_CreateLoaded
895
896 #if CC_Duplicate
897
898 #include "Duplicate_fp.h"
899
900 typedef TPM_RC (Duplicate_Entry) (
901     Duplicate_In          *in,
902     Duplicate_Out         *out
903 );
904
905 typedef const struct {
906     Duplicate_Entry        *entry;
907     UINT16                 inSize;
908     UINT16                 outSize;
909     UINT16                 offsetOfTypes;
910     UINT16                 paramOffsets[5];
911     BYTE                   types[9];
912 } Duplicate_COMMAND_DESCRIPTOR_t;
913
914 Duplicate_COMMAND_DESCRIPTOR_t _DuplicateData = {
915     /* entry */           &TPM2_Duplicate,
916     /* inSize */          (UINT16) (sizeof(Duplicate_In)),
917     /* outSize */         (UINT16) (sizeof(Duplicate_Out)),
918     /* offsetOfTypes */   offsetof(Duplicate_COMMAND_DESCRIPTOR_t, types),
919     /* offsets */         { (UINT16) (offsetof(Duplicate_In, newParentHandle)),
920                           (UINT16) (offsetof(Duplicate_In, encryptionKeyIn)),
921                           (UINT16) (offsetof(Duplicate_In, symmetricAlg)),
922                           (UINT16) (offsetof(Duplicate_Out, duplicate)),
923                           (UINT16) (offsetof(Duplicate_Out, outSymSeed)) },
924     /* types */           { TPMI_DH_OBJECT_H_UNMARSHAL,
925                           TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
926                           TPM2B_DATA_P_UNMARSHAL,
927                           TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
928                           END_OF_LIST,
929                           TPM2B_DATA_P_MARSHAL,
930                           TPM2B_PRIVATE_P_MARSHAL,
931                           TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
932                           END_OF_LIST }
933 };
934
935 #define _DuplicateDataAddress (&_DuplicateData)
936 #else
937 #define _DuplicateDataAddress 0
938 #endif // CC_Duplicate
939
940 #if CC_Rewrap
941
942 #include "Rewrap_fp.h"
943
944 typedef TPM_RC (Rewrap_Entry) (
945     Rewrap_In             *in,
946     Rewrap_Out            *out
947 );
948
949 typedef const struct {
950     Rewrap_Entry          *entry;
951     UINT16                 inSize;
952     UINT16                 outSize;
953     UINT16                 offsetOfTypes;
954     UINT16                 paramOffsets[5];
955     BYTE                   types[9];
956 } Rewrap_COMMAND_DESCRIPTOR_t;
957
958 Rewrap_COMMAND_DESCRIPTOR_t _RewrapData = {

```



```

959     /* entry          */ &TPM2_Rewrap,
960     /* inSize         */ (UINT16) (sizeof(Rewrap_In)),
961     /* outSize        */ (UINT16) (sizeof(Rewrap_Out)),
962     /* offsetOfTypes  */ offsetof(Rewrap_COMMAND_DESCRIPTOR_t, types),
963     /* offsets        */ { (UINT16) (offsetof(Rewrap_In, newParent)),
964                          (UINT16) (offsetof(Rewrap_In, inDuplicate)),
965                          (UINT16) (offsetof(Rewrap_In, name)),
966                          (UINT16) (offsetof(Rewrap_In, inSymSeed)),
967                          (UINT16) (offsetof(Rewrap_Out, outSymSeed)) },
968     /* types          */ {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
969                          TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
970                          TPM2B_PRIVATE_P_UNMARSHAL,
971                          TPM2B_NAME_P_UNMARSHAL,
972                          TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
973                          END_OF_LIST,
974                          TPM2B_PRIVATE_P_MARSHAL,
975                          TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
976                          END_OF_LIST};
977 };
978
979 #define _RewrapDataAddress (&_RewrapData)
980 #else
981 #define _RewrapDataAddress 0
982 #endif // CC_Rewrap
983
984 #if CC_Import
985
986 #include "Import_fp.h"
987
988 typedef TPM_RC (Import_Entry) (
989     Import_In          *in,
990     Import_Out         *out
991 );
992
993 typedef const struct {
994     Import_Entry        *entry;
995     UINT16              inSize;
996     UINT16              outSize;
997     UINT16              offsetOfTypes;
998     UINT16              paramOffsets[5];
999     BYTE                types[9];
1000 } Import_COMMAND_DESCRIPTOR_t;
1001
1002 Import_COMMAND_DESCRIPTOR_t _ImportData = {
1003     /* entry          */ &TPM2_Import,
1004     /* inSize         */ (UINT16) (sizeof(Import_In)),
1005     /* outSize        */ (UINT16) (sizeof(Import_Out)),
1006     /* offsetOfTypes  */ offsetof(Import_COMMAND_DESCRIPTOR_t, types),
1007     /* offsets        */ { (UINT16) (offsetof(Import_In, encryptionKey)),
1008                          (UINT16) (offsetof(Import_In, objectPublic)),
1009                          (UINT16) (offsetof(Import_In, duplicate)),
1010                          (UINT16) (offsetof(Import_In, inSymSeed)),
1011                          (UINT16) (offsetof(Import_In, symmetricAlg)) },
1012     /* types          */ {TPMI_DH_OBJECT_H_UNMARSHAL,
1013                          TPM2B_DATA_P_UNMARSHAL,
1014                          TPM2B_PUBLIC_P_UNMARSHAL,
1015                          TPM2B_PRIVATE_P_UNMARSHAL,
1016                          TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
1017                          TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
1018                          END_OF_LIST,
1019                          TPM2B_PRIVATE_P_MARSHAL,
1020                          END_OF_LIST};
1021 };
1022
1023 #define _ImportDataAddress (&_ImportData)
1024 #else

```

```

1025 #define ImportDataAddress 0
1026 #endif // CC_Import
1027
1028 #if CC_RSA_Encrypt
1029
1030 #include "RSA_Encrypt_fp.h"
1031
1032 typedef TPM_RC (RSA_Encrypt_Entry) (
1033     RSA_Encrypt_In      *in,
1034     RSA_Encrypt_Out     *out
1035 );
1036
1037 typedef const struct {
1038     RSA_Encrypt_Entry     *entry;
1039     UINT16                inSize;
1040     UINT16                outSize;
1041     UINT16                offsetOfTypes;
1042     UINT16                paramOffsets[3];
1043     BYTE                  types[7];
1044 } RSA_Encrypt_COMMAND_DESCRIPTOR_t;
1045
1046 RSA_Encrypt_COMMAND_DESCRIPTOR_t _RSA_EncryptData = {
1047     /* entry */           &TPM2_RSA_Encrypt,
1048     /* inSize */         (UINT16) (sizeof(RSA_Encrypt_In)),
1049     /* outSize */        (UINT16) (sizeof(RSA_Encrypt_Out)),
1050     /* offsetOfTypes */  offsetof(RSA_Encrypt_COMMAND_DESCRIPTOR_t, types),
1051     /* offsets */        {(UINT16) (offsetof(RSA_Encrypt_In, message)),
1052                          (UINT16) (offsetof(RSA_Encrypt_In, inScheme)),
1053                          (UINT16) (offsetof(RSA_Encrypt_In, label))},
1054     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1055                          TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
1056                          TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
1057                          TPM2B_DATA_P_UNMARSHAL,
1058                          END_OF_LIST,
1059                          TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
1060                          END_OF_LIST}
1061 };
1062
1063 #define _RSA_EncryptDataAddress (&_RSA_EncryptData)
1064 #else
1065 #define _RSA_EncryptDataAddress 0
1066 #endif // CC_RSA_Encrypt
1067
1068 #if CC_RSA_Decrypt
1069
1070 #include "RSA_Decrypt_fp.h"
1071
1072 typedef TPM_RC (RSA_Decrypt_Entry) (
1073     RSA_Decrypt_In      *in,
1074     RSA_Decrypt_Out     *out
1075 );
1076
1077 typedef const struct {
1078     RSA_Decrypt_Entry     *entry;
1079     UINT16                inSize;
1080     UINT16                outSize;
1081     UINT16                offsetOfTypes;
1082     UINT16                paramOffsets[3];
1083     BYTE                  types[7];
1084 } RSA_Decrypt_COMMAND_DESCRIPTOR_t;
1085
1086 RSA_Decrypt_COMMAND_DESCRIPTOR_t _RSA_DecryptData = {
1087     /* entry */           &TPM2_RSA_Decrypt,
1088     /* inSize */         (UINT16) (sizeof(RSA_Decrypt_In)),
1089     /* outSize */        (UINT16) (sizeof(RSA_Decrypt_Out)),
1090     /* offsetOfTypes */  offsetof(RSA_Decrypt_COMMAND_DESCRIPTOR_t, types),

```



```

1091     /* offsets */      {(UINT16) (offsetof(RSA_Decrypt_In, cipherText)),
1092                        (UINT16) (offsetof(RSA_Decrypt_In, inScheme)),
1093                        (UINT16) (offsetof(RSA_Decrypt_In, label))},
1094     /* types */        {TPMI_DH_OBJECT_H_UNMARSHAL,
1095                        TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
1096                        TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
1097                        TPM2B_DATA_P_UNMARSHAL,
1098                        END_OF_LIST,
1099                        TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
1100                        END_OF_LIST}
1101 };
1102
1103 #define _RSA_DecryptDataAddress (&_RSA_DecryptData)
1104 #else
1105 #define _RSA_DecryptDataAddress 0
1106 #endif // CC_RSA_Decrypt
1107
1108 #if CC_ECDH_KeyGen
1109
1110 #include "ECDH_KeyGen_fp.h"
1111
1112 typedef TPM_RC (ECDH_KeyGen_Entry) (
1113     ECDH_KeyGen_In      *in,
1114     ECDH_KeyGen_Out     *out
1115 );
1116
1117 typedef const struct {
1118     ECDH_KeyGen_Entry    *entry;
1119     UINT16               inSize;
1120     UINT16               outSize;
1121     UINT16               offsetOfTypes;
1122     UINT16               paramOffsets[1];
1123     BYTE                 types[5];
1124 } ECDH_KeyGen_COMMAND_DESCRIPTOR_t;
1125
1126 ECDH_KeyGen_COMMAND_DESCRIPTOR_t _ECDH_KeyGenData = {
1127     /* entry */          &TPM2_ECDH_KeyGen,
1128     /* inSize */         (UINT16) (sizeof(ECDH_KeyGen_In)),
1129     /* outSize */        (UINT16) (sizeof(ECDH_KeyGen_Out)),
1130     /* offsetOfTypes */  offsetof(ECDH_KeyGen_COMMAND_DESCRIPTOR_t, types),
1131     /* offsets */        {(UINT16) (offsetof(ECDH_KeyGen_Out, pubPoint))},
1132     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1133                          END_OF_LIST,
1134                          TPM2B_ECC_POINT_P_MARSHAL,
1135                          TPM2B_ECC_POINT_P_MARSHAL,
1136                          END_OF_LIST}
1137 };
1138
1139 #define _ECDH_KeyGenDataAddress (&_ECDH_KeyGenData)
1140 #else
1141 #define _ECDH_KeyGenDataAddress 0
1142 #endif // CC_ECDH_KeyGen
1143
1144 #if CC_ECDH_ZGen
1145
1146 #include "ECDH_ZGen_fp.h"
1147
1148 typedef TPM_RC (ECDH_ZGen_Entry) (
1149     ECDH_ZGen_In        *in,
1150     ECDH_ZGen_Out       *out
1151 );
1152
1153 typedef const struct {
1154     ECDH_ZGen_Entry      *entry;
1155     UINT16               inSize;
1156     UINT16               outSize;

```

```

1157     UINT16             offsetOfTypes;
1158     UINT16             paramOffsets[1];
1159     BYTE               types[5];
1160 } ECDH_ZGen_COMMAND_DESCRIPTOR_t;
1161
1162 ECDH_ZGen_COMMAND_DESCRIPTOR_t _ECDH_ZGenData = {
1163     /* entry */          &TPM2_ECDH_ZGen,
1164     /* inSize */         (UINT16) (sizeof(ECDH_ZGen_In)),
1165     /* outSize */        (UINT16) (sizeof(ECDH_ZGen_Out)),
1166     /* offsetOfTypes */  offsetof(ECDH_ZGen_COMMAND_DESCRIPTOR_t, types),
1167     /* offsets */         {(UINT16) (offsetof(ECDH_ZGen_In, inPoint))},
1168     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1169                          TPM2B_ECC_POINT_P_UNMARSHAL,
1170                          END_OF_LIST,
1171                          TPM2B_ECC_POINT_P_MARSHAL,
1172                          END_OF_LIST}
1173 };
1174
1175 #define _ECDH_ZGenDataAddress (&_ECDH_ZGenData)
1176 #else
1177 #define _ECDH_ZGenDataAddress 0
1178 #endif // CC_ECDH_ZGen
1179
1180 #if CC_ECC_Parameters
1181
1182 #include "ECC_Parameters_fp.h"
1183
1184 typedef TPM_RC (ECC_Parameters_Entry) (
1185     ECC_Parameters_In      *in,
1186     ECC_Parameters_Out     *out
1187 );
1188
1189 typedef const struct {
1190     ECC_Parameters_Entry    *entry;
1191     UINT16                  inSize;
1192     UINT16                  outSize;
1193     UINT16                  offsetOfTypes;
1194     BYTE                    types[4];
1195 } ECC_Parameters_COMMAND_DESCRIPTOR_t;
1196
1197 ECC_Parameters_COMMAND_DESCRIPTOR_t _ECC_ParametersData = {
1198     /* entry */          &TPM2_ECC_Parameters,
1199     /* inSize */         (UINT16) (sizeof(ECC_Parameters_In)),
1200     /* outSize */        (UINT16) (sizeof(ECC_Parameters_Out)),
1201     /* offsetOfTypes */  offsetof(ECC_Parameters_COMMAND_DESCRIPTOR_t, types),
1202     /* offsets */         // No parameter offsets;
1203     /* types */          {TPMI_ECC_CURVE_P_UNMARSHAL,
1204                          END_OF_LIST,
1205                          TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL,
1206                          END_OF_LIST}
1207 };
1208
1209 #define _ECC_ParametersDataAddress (&_ECC_ParametersData)
1210 #else
1211 #define _ECC_ParametersDataAddress 0
1212 #endif // CC_ECC_Parameters
1213
1214 #if CC_ZGen_2Phase
1215
1216 #include "ZGen_2Phase_fp.h"
1217
1218 typedef TPM_RC (ZGen_2Phase_Entry) (
1219     ZGen_2Phase_In         *in,
1220     ZGen_2Phase_Out        *out
1221 );
1222

```

```

1223 typedef const struct {
1224     ZGen_2Phase_Entry      *entry;
1225     UINT16                 inSize;
1226     UINT16                 outSize;
1227     UINT16                 offsetOfTypes;
1228     UINT16                 paramOffsets[5];
1229     BYTE                   types[9];
1230 } ZGen_2Phase_COMMAND_DESCRIPTOR_t;
1231
1232 ZGen_2Phase_COMMAND_DESCRIPTOR_t_ZGen_2PhaseData = {
1233     /* entry */           &TPM2_ZGen_2Phase,
1234     /* inSize */          (UINT16) (sizeof(ZGen_2Phase_In)),
1235     /* outSize */         (UINT16) (sizeof(ZGen_2Phase_Out)),
1236     /* offsetOfTypes */   offsetof(ZGen_2Phase_COMMAND_DESCRIPTOR_t, types),
1237     /* offsets */         { (UINT16) (offsetof(ZGen_2Phase_In, inQsB)),
1238                           (UINT16) (offsetof(ZGen_2Phase_In, inQeB)),
1239                           (UINT16) (offsetof(ZGen_2Phase_In, inScheme)),
1240                           (UINT16) (offsetof(ZGen_2Phase_In, counter)),
1241                           (UINT16) (offsetof(ZGen_2Phase_Out, outZ2)) },
1242     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1243                           TPM2B_ECC_POINT_P_UNMARSHAL,
1244                           TPM2B_ECC_POINT_P_UNMARSHAL,
1245                           TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL,
1246                           UINT16_P_UNMARSHAL,
1247                           END_OF_LIST,
1248                           TPM2B_ECC_POINT_P_MARSHAL,
1249                           TPM2B_ECC_POINT_P_MARSHAL,
1250                           END_OF_LIST};
1251 };
1252
1253 #define _ZGen_2PhaseDataAddress (&_ZGen_2PhaseData)
1254 #else
1255 #define _ZGen_2PhaseDataAddress 0
1256 #endif // CC_ZGen_2Phase
1257
1258 #if CC_ECC_Encrypt
1259
1260 #include "ECC_Encrypt_fp.h"
1261
1262 typedef TPM_RC (ECC_Encrypt_Entry) (
1263     ECC_Encrypt_In      *in,
1264     ECC_Encrypt_Out     *out
1265 );
1266
1267 typedef const struct {
1268     ECC_Encrypt_Entry      *entry;
1269     UINT16                 inSize;
1270     UINT16                 outSize;
1271     UINT16                 offsetOfTypes;
1272     UINT16                 paramOffsets[4];
1273     BYTE                   types[8];
1274 } ECC_Encrypt_COMMAND_DESCRIPTOR_t;
1275
1276 ECC_Encrypt_COMMAND_DESCRIPTOR_t_ECC_EncryptData = {
1277     /* entry */           &TPM2_ECC_Encrypt,
1278     /* inSize */          (UINT16) (sizeof(ECC_Encrypt_In)),
1279     /* outSize */         (UINT16) (sizeof(ECC_Encrypt_Out)),
1280     /* offsetOfTypes */   offsetof(ECC_Encrypt_COMMAND_DESCRIPTOR_t, types),
1281     /* offsets */         { (UINT16) (offsetof(ECC_Encrypt_In, plainText)),
1282                           (UINT16) (offsetof(ECC_Encrypt_In, inScheme)),
1283                           (UINT16) (offsetof(ECC_Encrypt_Out, C2)),
1284                           (UINT16) (offsetof(ECC_Encrypt_Out, C3)) },
1285     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1286                           TPM2B_MAX_BUFFER_P_UNMARSHAL,
1287                           TPMT_KDF_SCHEME_P_UNMARSHAL + ADD_FLAG,
1288                           END_OF_LIST,

```

```

1289             TPM2B_ECC_POINT_P_MARSHAL,
1290             TPM2B_MAX_BUFFER_P_MARSHAL,
1291             TPM2B_DIGEST_P_MARSHAL,
1292             END_OF_LIST}
1293 };
1294
1295 #define _ECC_EncryptDataAddress (&_ECC_EncryptData)
1296 #else
1297 #define _ECC_EncryptDataAddress 0
1298 #endif // CC_ECC_Encrypt
1299
1300 #if CC_ECC_Decrypt
1301
1302 #include "ECC_Decrypt_fp.h"
1303
1304 typedef TPM_RC (ECC_Decrypt_Entry) (
1305     ECC_Decrypt_In          *in,
1306     ECC_Decrypt_Out         *out
1307 );
1308
1309 typedef const struct {
1310     ECC_Decrypt_Entry      *entry;
1311     UINT16                 inSize;
1312     UINT16                 outSize;
1313     UINT16                 offsetOfTypes;
1314     UINT16                 paramOffsets[4];
1315     BYTE                   types[8];
1316 } ECC_Decrypt_COMMAND_DESCRIPTOR_t;
1317
1318 ECC_Decrypt_COMMAND_DESCRIPTOR_t _ECC_DecryptData = {
1319     /* entry */           &TPM2_ECC_Decrypt,
1320     /* inSize */          (UINT16) (sizeof(ECC_Decrypt_In)),
1321     /* outSize */         (UINT16) (sizeof(ECC_Decrypt_Out)),
1322     /* offsetOfTypes */   offsetof(ECC_Decrypt_COMMAND_DESCRIPTOR_t, types),
1323     /* offsets */         { (UINT16) (offsetof(ECC_Decrypt_In, C1)),
1324                           (UINT16) (offsetof(ECC_Decrypt_In, C2)),
1325                           (UINT16) (offsetof(ECC_Decrypt_In, C3)),
1326                           (UINT16) (offsetof(ECC_Decrypt_In, inScheme)) },
1327     /* types */           { TPMI_DH_OBJECT_H_UNMARSHAL,
1328                           TPM2B_ECC_POINT_P_UNMARSHAL,
1329                           TPM2B_MAX_BUFFER_P_UNMARSHAL,
1330                           TPM2B_DIGEST_P_UNMARSHAL,
1331                           TPMT_KDF_SCHEME_P_UNMARSHAL + ADD_FLAG,
1332                           END_OF_LIST,
1333                           TPM2B_MAX_BUFFER_P_MARSHAL,
1334                           END_OF_LIST }
1335 };
1336
1337 #define _ECC_DecryptDataAddress (&_ECC_DecryptData)
1338 #else
1339 #define _ECC_DecryptDataAddress 0
1340 #endif // CC_ECC_Decrypt
1341
1342 #if CC_EncryptDecrypt
1343
1344 #include "EncryptDecrypt_fp.h"
1345
1346 typedef TPM_RC (EncryptDecrypt_Entry) (
1347     EncryptDecrypt_In      *in,
1348     EncryptDecrypt_Out     *out
1349 );
1350
1351 typedef const struct {
1352     EncryptDecrypt_Entry   *entry;
1353     UINT16                 inSize;
1354     UINT16                 outSize;

```

```

1355     UINT16             offsetOfTypes;
1356     UINT16             paramOffsets[5];
1357     BYTE               types[9];
1358 } EncryptDecrypt_COMMAND_DESCRIPTOR_t;
1359
1360 EncryptDecrypt_COMMAND_DESCRIPTOR_t _EncryptDecryptData = {
1361     /* entry */          &TPM2_EncryptDecrypt,
1362     /* inSize */         (UINT16) (sizeof(EncryptDecrypt_In)),
1363     /* outSize */        (UINT16) (sizeof(EncryptDecrypt_Out)),
1364     /* offsetOfTypes */  offsetof(EncryptDecrypt_COMMAND_DESCRIPTOR_t, types),
1365     /* offsets */        { (UINT16) (offsetof(EncryptDecrypt_In, decrypt)),
1366                          (UINT16) (offsetof(EncryptDecrypt_In, mode)),
1367                          (UINT16) (offsetof(EncryptDecrypt_In, ivIn)),
1368                          (UINT16) (offsetof(EncryptDecrypt_In, inData)),
1369                          (UINT16) (offsetof(EncryptDecrypt_Out, ivOut))},
1370     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1371                          TPMI_YES_NO_P_UNMARSHAL,
1372                          TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1373                          TPM2B_IV_P_UNMARSHAL,
1374                          TPM2B_MAX_BUFFER_P_UNMARSHAL,
1375                          END_OF_LIST,
1376                          TPM2B_MAX_BUFFER_P_MARSHAL,
1377                          TPM2B_IV_P_MARSHAL,
1378                          END_OF_LIST};
1379 };
1380
1381 #define _EncryptDecryptDataAddress (&_EncryptDecryptData)
1382 #else
1383 #define _EncryptDecryptDataAddress 0
1384 #endif // CC_EncryptDecrypt
1385
1386 #if CC_EncryptDecrypt2
1387
1388 #include "EncryptDecrypt2_fp.h"
1389
1390 typedef TPM_RC (EncryptDecrypt2_Entry) (
1391     EncryptDecrypt2_In      *in,
1392     EncryptDecrypt2_Out     *out
1393 );
1394
1395 typedef const struct {
1396     EncryptDecrypt2_Entry  *entry;
1397     UINT16                 inSize;
1398     UINT16                 outSize;
1399     UINT16                 offsetOfTypes;
1400     UINT16                 paramOffsets[5];
1401     BYTE                   types[9];
1402 } EncryptDecrypt2_COMMAND_DESCRIPTOR_t;
1403
1404 EncryptDecrypt2_COMMAND_DESCRIPTOR_t _EncryptDecrypt2Data = {
1405     /* entry */          &TPM2_EncryptDecrypt2,
1406     /* inSize */         (UINT16) (sizeof(EncryptDecrypt2_In)),
1407     /* outSize */        (UINT16) (sizeof(EncryptDecrypt2_Out)),
1408     /* offsetOfTypes */  offsetof(EncryptDecrypt2_COMMAND_DESCRIPTOR_t, types),
1409     /* offsets */        { (UINT16) (offsetof(EncryptDecrypt2_In, inData)),
1410                          (UINT16) (offsetof(EncryptDecrypt2_In, decrypt)),
1411                          (UINT16) (offsetof(EncryptDecrypt2_In, mode)),
1412                          (UINT16) (offsetof(EncryptDecrypt2_In, ivIn)),
1413                          (UINT16) (offsetof(EncryptDecrypt2_Out, ivOut))},
1414     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1415                          TPM2B_MAX_BUFFER_P_UNMARSHAL,
1416                          TPMI_YES_NO_P_UNMARSHAL,
1417                          TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1418                          TPM2B_IV_P_UNMARSHAL,
1419                          END_OF_LIST,
1420                          TPM2B_MAX_BUFFER_P_MARSHAL,

```

```

1421                                     TPM2B_IV_P_MARSHAL,
1422                                     END_OF_LIST}
1423 };
1424
1425 #define _EncryptDecrypt2DataAddress (&_EncryptDecrypt2Data)
1426 #else
1427 #define _EncryptDecrypt2DataAddress 0
1428 #endif // CC_EncryptDecrypt2
1429
1430 #if CC_Hash
1431
1432 #include "Hash_fp.h"
1433
1434 typedef TPM_RC (Hash_Entry) (
1435     Hash_In                *in,
1436     Hash_Out               *out
1437 );
1438
1439 typedef const struct {
1440     Hash_Entry              *entry;
1441     UINT16                  inSize;
1442     UINT16                  outSize;
1443     UINT16                  offsetOfTypes;
1444     UINT16                  paramOffsets[3];
1445     BYTE                    types[7];
1446 } Hash_COMMAND_DESCRIPTOR_t;
1447
1448 Hash_COMMAND_DESCRIPTOR_t _HashData = {
1449     /* entry */              &TPM2_Hash,
1450     /* inSize */             (UINT16) (sizeof(Hash_In)),
1451     /* outSize */            (UINT16) (sizeof(Hash_Out)),
1452     /* offsetOfTypes */      offsetof(Hash_COMMAND_DESCRIPTOR_t, types),
1453     /* offsets */            { (UINT16) (offsetof(Hash_In, hashAlg)),
1454                               (UINT16) (offsetof(Hash_In, hierarchy)),
1455                               (UINT16) (offsetof(Hash_Out, validation)) },
1456     /* types */              { TPM2B_MAX_BUFFER_P_UNMARSHAL,
1457                               TPMI_ALG_HASH_P_UNMARSHAL,
1458                               TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1459                               END_OF_LIST,
1460                               TPM2B_DIGEST_P_MARSHAL,
1461                               TPMT_TK_HASHCHECK_P_MARSHAL,
1462                               END_OF_LIST }
1463 };
1464
1465 #define _HashDataAddress (&_HashData)
1466 #else
1467 #define _HashDataAddress 0
1468 #endif // CC_Hash
1469
1470 #if CC_HMAC
1471
1472 #include "HMAC_fp.h"
1473
1474 typedef TPM_RC (HMAC_Entry) (
1475     HMAC_In                 *in,
1476     HMAC_Out                *out
1477 );
1478
1479 typedef const struct {
1480     HMAC_Entry              *entry;
1481     UINT16                  inSize;
1482     UINT16                  outSize;
1483     UINT16                  offsetOfTypes;
1484     UINT16                  paramOffsets[2];
1485     BYTE                    types[6];
1486 } HMAC_COMMAND_DESCRIPTOR_t;

```



```

1487
1488 HMAC_COMMAND_DESCRIPTOR_t _HMACData = {
1489     /* entry */          &TPM2_HMAC,
1490     /* inSize */         (UINT16) (sizeof(HMAC_In)),
1491     /* outSize */        (UINT16) (sizeof(HMAC_Out)),
1492     /* offsetOfTypes */  offsetof(HMAC_COMMAND_DESCRIPTOR_t, types),
1493     /* offsets */        {(UINT16) (offsetof(HMAC_In, buffer)),
1494                          (UINT16) (offsetof(HMAC_In, hashAlg))},
1495     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1496                          TPM2B_MAX_BUFFER_P_UNMARSHAL,
1497                          TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1498                          END_OF_LIST,
1499                          TPM2B_DIGEST_P_MARSHAL,
1500                          END_OF_LIST};
1501 };
1502
1503 #define _HMACDataAddress (&_HMACData)
1504 #else
1505 #define _HMACDataAddress 0
1506 #endif // CC_HMAC
1507
1508 #if CC_MAC
1509
1510 #include "MAC_fp.h"
1511
1512 typedef TPM_RC (MAC_Entry) (
1513     MAC_In          *in,
1514     MAC_Out         *out
1515 );
1516
1517 typedef const struct {
1518     MAC_Entry        *entry;
1519     UINT16            inSize;
1520     UINT16            outSize;
1521     UINT16            offsetOfTypes;
1522     UINT16            paramOffsets[2];
1523     BYTE              types[6];
1524 } MAC_COMMAND_DESCRIPTOR_t;
1525
1526 MAC_COMMAND_DESCRIPTOR_t _MACData = {
1527     /* entry */          &TPM2_MAC,
1528     /* inSize */         (UINT16) (sizeof(MAC_In)),
1529     /* outSize */        (UINT16) (sizeof(MAC_Out)),
1530     /* offsetOfTypes */  offsetof(MAC_COMMAND_DESCRIPTOR_t, types),
1531     /* offsets */        {(UINT16) (offsetof(MAC_In, buffer)),
1532                          (UINT16) (offsetof(MAC_In, inScheme))},
1533     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1534                          TPM2B_MAX_BUFFER_P_UNMARSHAL,
1535                          TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1536                          END_OF_LIST,
1537                          TPM2B_DIGEST_P_MARSHAL,
1538                          END_OF_LIST};
1539 };
1540
1541 #define _MACDataAddress (&_MACData)
1542 #else
1543 #define _MACDataAddress 0
1544 #endif // CC_MAC
1545
1546 #if CC_GetRandom
1547
1548 #include "GetRandom_fp.h"
1549
1550 typedef TPM_RC (GetRandom_Entry) (
1551     GetRandom_In     *in,
1552     GetRandom_Out    *out

```

```

1553 );
1554
1555 typedef const struct {
1556     GetRandom_Entry      *entry;
1557     UINT16               inSize;
1558     UINT16               outSize;
1559     UINT16               offsetOfTypes;
1560     BYTE                 types[4];
1561 } GetRandom_COMMAND_DESCRIPTOR_t;
1562
1563 GetRandom_COMMAND_DESCRIPTOR_t _GetRandomData = {
1564     /* entry      */      &TPM2_GetRandom,
1565     /* inSize     */      (UINT16) (sizeof(GetRandom_In)),
1566     /* outSize    */      (UINT16) (sizeof(GetRandom_Out)),
1567     /* offsetOfTypes */    offsetof(GetRandom_COMMAND_DESCRIPTOR_t, types),
1568     /* offsets    */      // No parameter offsets;
1569     /* types      */      {UINT16_P_UNMARSHAL,
1570                           END_OF_LIST,
1571                           TPM2B_DIGEST_P_MARSHAL,
1572                           END_OF_LIST};
1573 };
1574
1575 #define _GetRandomDataAddress (&_GetRandomData)
1576 #else
1577 #define _GetRandomDataAddress 0
1578 #endif // CC_GetRandom
1579
1580 #if CC_StirRandom
1581
1582 #include "StirRandom_fp.h"
1583
1584 typedef TPM_RC (StirRandom_Entry) (
1585     StirRandom_In      *in
1586 );
1587
1588 typedef const struct {
1589     StirRandom_Entry      *entry;
1590     UINT16               inSize;
1591     UINT16               outSize;
1592     UINT16               offsetOfTypes;
1593     BYTE                 types[3];
1594 } StirRandom_COMMAND_DESCRIPTOR_t;
1595
1596 StirRandom_COMMAND_DESCRIPTOR_t _StirRandomData = {
1597     /* entry      */      &TPM2_StirRandom,
1598     /* inSize     */      (UINT16) (sizeof(StirRandom_In)),
1599     /* outSize    */      0,
1600     /* offsetOfTypes */    offsetof(StirRandom_COMMAND_DESCRIPTOR_t, types),
1601     /* offsets    */      // No parameter offsets;
1602     /* types      */      {TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1603                           END_OF_LIST,
1604                           END_OF_LIST};
1605 };
1606
1607 #define _StirRandomDataAddress (&_StirRandomData)
1608 #else
1609 #define _StirRandomDataAddress 0
1610 #endif // CC_StirRandom
1611
1612 #if CC_HMAC_Start
1613
1614 #include "HMAC_Start_fp.h"
1615
1616 typedef TPM_RC (HMAC_Start_Entry) (
1617     HMAC_Start_In      *in,
1618     HMAC_Start_Out     *out

```



```

1619 );
1620
1621 typedef const struct {
1622     HMAC_Start_Entry      *entry;
1623     UINT16                inSize;
1624     UINT16                outSize;
1625     UINT16                offsetOfTypes;
1626     UINT16                paramOffsets[2];
1627     BYTE                  types[6];
1628 } HMAC_Start_COMMAND_DESCRIPTOR_t;
1629
1630 HMAC_Start_COMMAND_DESCRIPTOR_t _HMAC_StartData = {
1631     /* entry */          &TPM2_HMAC_Start,
1632     /* inSize */         (UINT16) (sizeof(HMAC_Start_In)),
1633     /* outSize */        (UINT16) (sizeof(HMAC_Start_Out)),
1634     /* offsetOfTypes */  offsetof(HMAC_Start_COMMAND_DESCRIPTOR_t, types),
1635     /* offsets */        {(UINT16) (offsetof(HMAC_Start_In, auth)),
1636                          (UINT16) (offsetof(HMAC_Start_In, hashAlg))},
1637     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1638                          TPM2B_AUTH_P_UNMARSHAL,
1639                          TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1640                          END_OF_LIST,
1641                          TPMI_DH_OBJECT_H_MARSHAL,
1642                          END_OF_LIST}
1643 };
1644
1645 #define _HMAC_StartDataAddress (&_HMAC_StartData)
1646 #else
1647 #define _HMAC_StartDataAddress 0
1648 #endif // CC_HMAC_Start
1649
1650 #if CC_MAC_Start
1651
1652 #include "MAC_Start_fp.h"
1653
1654 typedef TPM_RC (MAC_Start_Entry) (
1655     MAC_Start_In      *in,
1656     MAC_Start_Out     *out
1657 );
1658
1659 typedef const struct {
1660     MAC_Start_Entry      *entry;
1661     UINT16                inSize;
1662     UINT16                outSize;
1663     UINT16                offsetOfTypes;
1664     UINT16                paramOffsets[2];
1665     BYTE                  types[6];
1666 } MAC_Start_COMMAND_DESCRIPTOR_t;
1667
1668 MAC_Start_COMMAND_DESCRIPTOR_t _MAC_StartData = {
1669     /* entry */          &TPM2_MAC_Start,
1670     /* inSize */         (UINT16) (sizeof(MAC_Start_In)),
1671     /* outSize */        (UINT16) (sizeof(MAC_Start_Out)),
1672     /* offsetOfTypes */  offsetof(MAC_Start_COMMAND_DESCRIPTOR_t, types),
1673     /* offsets */        {(UINT16) (offsetof(MAC_Start_In, auth)),
1674                          (UINT16) (offsetof(MAC_Start_In, inScheme))},
1675     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1676                          TPM2B_AUTH_P_UNMARSHAL,
1677                          TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1678                          END_OF_LIST,
1679                          TPMI_DH_OBJECT_H_MARSHAL,
1680                          END_OF_LIST}
1681 };
1682
1683 #define _MAC_StartDataAddress (&_MAC_StartData)
1684 #else

```

```

1685 #define _MAC_StartDataAddress 0
1686 #endif // CC_MAC_Start
1687
1688 #if CC_HashSequenceStart
1689
1690 #include "HashSequenceStart_fp.h"
1691
1692 typedef TPM_RC (HashSequenceStart_Entry) (
1693     HashSequenceStart_In      *in,
1694     HashSequenceStart_Out     *out
1695 );
1696
1697 typedef const struct {
1698     HashSequenceStart_Entry    *entry;
1699     UINT16                     inSize;
1700     UINT16                     outSize;
1701     UINT16                     offsetOfTypes;
1702     UINT16                     paramOffsets[1];
1703     BYTE                       types[5];
1704 } HashSequenceStart_COMMAND_DESCRIPTOR_t;
1705
1706 HashSequenceStart_COMMAND_DESCRIPTOR_t _HashSequenceStartData = {
1707     /* entry */           &TPM2_HashSequenceStart,
1708     /* inSize */          (UINT16) (sizeof (HashSequenceStart_In)),
1709     /* outSize */         (UINT16) (sizeof (HashSequenceStart_Out)),
1710     /* offsetOfTypes */   offsetof (HashSequenceStart_COMMAND_DESCRIPTOR_t,
1711     types),
1712     /* offsets */         { (UINT16) (offsetof (HashSequenceStart_In, hashAlg)) },
1713     /* types */           { TPM2B_AUTH_P_UNMARSHAL,
1714                             TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1715                             END_OF_LIST,
1716                             TPMI_DH_OBJECT_H_MARSHAL,
1717                             END_OF_LIST };
1718 };
1719
1720 #define _HashSequenceStartDataAddress (&_HashSequenceStartData)
1721 #else
1722 #define _HashSequenceStartDataAddress 0
1723 #endif // CC_HashSequenceStart
1724
1725 #if CC_SequenceUpdate
1726
1727 #include "SequenceUpdate_fp.h"
1728
1729 typedef TPM_RC (SequenceUpdate_Entry) (
1730     SequenceUpdate_In      *in
1731 );
1732
1733 typedef const struct {
1734     SequenceUpdate_Entry    *entry;
1735     UINT16                     inSize;
1736     UINT16                     outSize;
1737     UINT16                     offsetOfTypes;
1738     UINT16                     paramOffsets[1];
1739     BYTE                       types[4];
1740 } SequenceUpdate_COMMAND_DESCRIPTOR_t;
1741
1742 SequenceUpdate_COMMAND_DESCRIPTOR_t _SequenceUpdateData = {
1743     /* entry */           &TPM2_SequenceUpdate,
1744     /* inSize */          (UINT16) (sizeof (SequenceUpdate_In)),
1745     /* outSize */         0,
1746     /* offsetOfTypes */   offsetof (SequenceUpdate_COMMAND_DESCRIPTOR_t, types),
1747     /* offsets */         { (UINT16) (offsetof (SequenceUpdate_In, buffer)) },
1748     /* types */           { TPMI_DH_OBJECT_H_UNMARSHAL,
1749                             TPM2B_MAX_BUFFER_P_UNMARSHAL,
1750                             END_OF_LIST,

```

```

1750                                     END_OF_LIST}
1751 };
1752
1753 #define _SequenceUpdateDataAddress (&_SequenceUpdateData)
1754 #else
1755 #define _SequenceUpdateDataAddress 0
1756 #endif // CC_SequenceUpdate
1757
1758 #if CC_SequenceComplete
1759
1760 #include "SequenceComplete_fp.h"
1761
1762 typedef TPM_RC (SequenceComplete_Entry) (
1763     SequenceComplete_In      *in,
1764     SequenceComplete_Out     *out
1765 );
1766
1767 typedef const struct {
1768     SequenceComplete_Entry  *entry;
1769     UINT16                  inSize;
1770     UINT16                  outSize;
1771     UINT16                  offsetOfTypes;
1772     UINT16                  paramOffsets[3];
1773     BYTE                    types[7];
1774 } SequenceComplete_COMMAND_DESCRIPTOR_t;
1775
1776 SequenceComplete_COMMAND_DESCRIPTOR_t _SequenceCompleteData = {
1777     /* entry */                &TPM2_SequenceComplete,
1778     /* inSize */              (UINT16) (sizeof(SequenceComplete_In)),
1779     /* outSize */             (UINT16) (sizeof(SequenceComplete_Out)),
1780     /* offsetOfTypes */       offsetof(SequenceComplete_COMMAND_DESCRIPTOR_t, types),
1781     /* offsets */              { (UINT16) (offsetof(SequenceComplete_In, buffer)),
1782                               (UINT16) (offsetof(SequenceComplete_In, hierarchy)),
1783                               (UINT16) (offsetof(SequenceComplete_Out, validation)) },
1784     /* types */                { TPMI_DH_OBJECT_H_UNMARSHAL,
1785                               TPM2B_MAX_BUFFER_P_UNMARSHAL,
1786                               TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1787                               END_OF_LIST,
1788                               TPM2B_DIGEST_P_MARSHAL,
1789                               TPMT_TK_HASHCHECK_P_MARSHAL,
1790                               END_OF_LIST }
1791 };
1792
1793 #define _SequenceCompleteDataAddress (&_SequenceCompleteData)
1794 #else
1795 #define _SequenceCompleteDataAddress 0
1796 #endif // CC_SequenceComplete
1797
1798 #if CC_EventSequenceComplete
1799
1800 #include "EventSequenceComplete_fp.h"
1801
1802 typedef TPM_RC (EventSequenceComplete_Entry) (
1803     EventSequenceComplete_In  *in,
1804     EventSequenceComplete_Out *out
1805 );
1806
1807 typedef const struct {
1808     EventSequenceComplete_Entry *entry;
1809     UINT16                      inSize;
1810     UINT16                      outSize;
1811     UINT16                      offsetOfTypes;
1812     UINT16                      paramOffsets[2];
1813     BYTE                        types[6];
1814 } EventSequenceComplete_COMMAND_DESCRIPTOR_t;
1815

```

```

1816 EventSequenceComplete_COMMAND_DESCRIPTOR_t_EventSequenceCompleteData = {
1817     /* entry */ &TPM2_EventSequenceComplete,
1818     /* inSize */ (UINT16) (sizeof(EventSequenceComplete_In)),
1819     /* outSize */ (UINT16) (sizeof(EventSequenceComplete_Out)),
1820     /* offsetOfTypes */
offsetof(EventSequenceComplete_COMMAND_DESCRIPTOR_t, types),
1821     /* offsets */ { (UINT16) (offsetof(EventSequenceComplete_In,
sequenceHandle)) ,
1822
(UINT16) (offsetof(EventSequenceComplete_In,
buffer)) },
1823     /* types */ { TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1824
TPMI_DH_OBJECT_H_UNMARSHAL,
1825
TPM2B_MAX_BUFFER_P_UNMARSHAL,
1826
END_OF_LIST,
1827
TPML_DIGEST_VALUES_P_MARSHAL,
1828
END_OF_LIST }
1829 };
1830
1831 #define _EventSequenceCompleteDataAddress (&EventSequenceCompleteData)
1832 #else
1833 #define _EventSequenceCompleteDataAddress 0
1834 #endif // CC_EventSequenceComplete
1835
1836 #if CC_Certify
1837
1838 #include "Certify_fp.h"
1839
1840 typedef TPM_RC (Certify_Entry) (
1841     Certify_In *in,
1842     Certify_Out *out
1843 );
1844
1845 typedef const struct {
1846     Certify_Entry *entry;
1847     UINT16 inSize;
1848     UINT16 outSize;
1849     UINT16 offsetOfTypes;
1850     UINT16 paramOffsets[4];
1851     BYTE types[8];
1852 } Certify_COMMAND_DESCRIPTOR_t;
1853
1854 Certify_COMMAND_DESCRIPTOR_t_CertifyData = {
1855     /* entry */ &TPM2_Certify,
1856     /* inSize */ (UINT16) (sizeof(Certify_In)),
1857     /* outSize */ (UINT16) (sizeof(Certify_Out)),
1858     /* offsetOfTypes */ offsetof(Certify_COMMAND_DESCRIPTOR_t, types),
1859     /* offsets */ { (UINT16) (offsetof(Certify_In, signHandle)),
1860
(UINT16) (offsetof(Certify_In, qualifyingData)),
1861
(UINT16) (offsetof(Certify_In, inScheme)),
1862
(UINT16) (offsetof(Certify_Out, signature)) },
1863     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
1864
TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1865
TPM2B_DATA_P_UNMARSHAL,
1866
TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1867
END_OF_LIST,
1868
TPM2B_ATTEST_P_MARSHAL,
1869
TPMT_SIGNATURE_P_MARSHAL,
1870
END_OF_LIST }
1871 };
1872
1873 #define _CertifyDataAddress (&CertifyData)
1874 #else
1875 #define _CertifyDataAddress 0
1876 #endif // CC_Certify
1877
1878 #if CC_CertifyCreation

```

```

1879
1880 #include "CertifyCreation_fp.h"
1881
1882 typedef TPM_RC (CertifyCreation_Entry) (
1883     CertifyCreation_In      *in,
1884     CertifyCreation_Out     *out
1885 );
1886
1887 typedef const struct {
1888     CertifyCreation_Entry  *entry;
1889     UINT16                 inSize;
1890     UINT16                 outSize;
1891     UINT16                 offsetOfTypes;
1892     UINT16                 paramOffsets[6];
1893     BYTE                   types[10];
1894 } CertifyCreation_COMMAND_DESCRIPTOR_t;
1895
1896 CertifyCreation_COMMAND_DESCRIPTOR_t _CertifyCreationData = {
1897     /* entry          */ &TPM2_CertifyCreation,
1898     /* inSize         */ (UINT16) (sizeof(CertifyCreation_In)),
1899     /* outSize        */ (UINT16) (sizeof(CertifyCreation_Out)),
1900     /* offsetOfTypes */ offsetof(CertifyCreation_COMMAND_DESCRIPTOR_t, types),
1901     /* offsets         */ { (UINT16) (offsetof(CertifyCreation_In, objectHandle)),
1902                           (UINT16) (offsetof(CertifyCreation_In, qualifyingData)),
1903                           (UINT16) (offsetof(CertifyCreation_In, creationHash)),
1904                           (UINT16) (offsetof(CertifyCreation_In, inScheme)),
1905                           (UINT16) (offsetof(CertifyCreation_In, creationTicket)),
1906                           (UINT16) (offsetof(CertifyCreation_Out, signature))},
1907     /* types          */ {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1908                           TPMI_DH_OBJECT_H_UNMARSHAL,
1909                           TPM2B_DATA_P_UNMARSHAL,
1910                           TPM2B_DIGEST_P_UNMARSHAL,
1911                           TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1912                           TPMT_TK_CREATION_P_UNMARSHAL,
1913                           END_OF_LIST,
1914                           TPM2B_ATTEST_P_MARSHAL,
1915                           TPMT_SIGNATURE_P_MARSHAL,
1916                           END_OF_LIST}
1917 };
1918
1919 #define _CertifyCreationDataAddress (&_CertifyCreationData)
1920 #else
1921 #define _CertifyCreationDataAddress 0
1922 #endif // CC_CertifyCreation
1923
1924 #if CC_Quote
1925
1926 #include "Quote_fp.h"
1927
1928 typedef TPM_RC (Quote_Entry) (
1929     Quote_In      *in,
1930     Quote_Out     *out
1931 );
1932
1933 typedef const struct {
1934     Quote_Entry  *entry;
1935     UINT16       inSize;
1936     UINT16       outSize;
1937     UINT16       offsetOfTypes;
1938     UINT16       paramOffsets[4];
1939     BYTE         types[8];
1940 } Quote_COMMAND_DESCRIPTOR_t;
1941
1942 Quote_COMMAND_DESCRIPTOR_t _QuoteData = {
1943     /* entry          */ &TPM2_Quote,
1944     /* inSize         */ (UINT16) (sizeof(Quote_In)),

```

```

1945     /* outSize */ (UINT16) (sizeof(Quote_Out)),
1946     /* offsetOfTypes */ offsetof(Quote_COMMAND_DESCRIPTOR_t, types),
1947     /* offsets */ { (UINT16) (offsetof(Quote_In, qualifyingData)),
1948                   (UINT16) (offsetof(Quote_In, inScheme)),
1949                   (UINT16) (offsetof(Quote_In, PCRselect)),
1950                   (UINT16) (offsetof(Quote_Out, signature))},
1951     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1952                 TPM2B_DATA_P_UNMARSHAL,
1953                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1954                 TPML_PCR_SELECTION_P_UNMARSHAL,
1955                 END_OF_LIST,
1956                 TPM2B_ATTEST_P_MARSHAL,
1957                 TPMT_SIGNATURE_P_MARSHAL,
1958                 END_OF_LIST};
1959 };
1960
1961 #define _QuoteDataAddress (&_QuoteData)
1962 #else
1963 #define _QuoteDataAddress 0
1964 #endif // CC_Quote
1965
1966 #if CC_GetSessionAuditDigest
1967
1968 #include "GetSessionAuditDigest_fp.h"
1969
1970 typedef TPM_RC (GetSessionAuditDigest_Entry) (
1971     GetSessionAuditDigest_In *in,
1972     GetSessionAuditDigest_Out *out
1973 );
1974
1975 typedef const struct {
1976     GetSessionAuditDigest_Entry *entry;
1977     UINT16 inSize;
1978     UINT16 outSize;
1979     UINT16 offsetOfTypes;
1980     UINT16 paramOffsets[5];
1981     BYTE types[9];
1982 } GetSessionAuditDigest_COMMAND_DESCRIPTOR_t;
1983
1984 GetSessionAuditDigest_COMMAND_DESCRIPTOR_t _GetSessionAuditDigestData = {
1985     /* entry */ &TPM2_GetSessionAuditDigest,
1986     /* inSize */ (UINT16) (sizeof(GetSessionAuditDigest_In)),
1987     /* outSize */ (UINT16) (sizeof(GetSessionAuditDigest_Out)),
1988     /* offsetOfTypes */
1989     /* offsets */ { (UINT16) (offsetof(GetSessionAuditDigest_In,
1990     signHandle)),
1991                   (UINT16) (offsetof(GetSessionAuditDigest_In,
1992     sessionHandle)),
1993                   (UINT16) (offsetof(GetSessionAuditDigest_In,
1994     qualifyingData)),
1995                   (UINT16) (offsetof(GetSessionAuditDigest_In,
1996     inScheme)),
1997                   (UINT16) (offsetof(GetSessionAuditDigest_Out,
1998     signature))},
1999     /* types */ {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
2000                 TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
2001                 TPMI_SH_HMAC_H_UNMARSHAL,
2002                 TPM2B_DATA_P_UNMARSHAL,
2003                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2004                 END_OF_LIST,
2005                 TPM2B_ATTEST_P_MARSHAL,
2006                 TPMT_SIGNATURE_P_MARSHAL,
2007                 END_OF_LIST};
2008 };
2009

```



```

2005 #define _GetSessionAuditDigestDataAddress (&_GetSessionAuditDigestData)
2006 #else
2007 #define _GetSessionAuditDigestDataAddress 0
2008 #endif // CC_GetSessionAuditDigest
2009
2010 #if CC_GetCommandAuditDigest
2011
2012 #include "GetCommandAuditDigest_fp.h"
2013
2014 typedef TPM_RC (GetCommandAuditDigest_Entry) (
2015     GetCommandAuditDigest_In      *in,
2016     GetCommandAuditDigest_Out     *out
2017 );
2018
2019 typedef const struct {
2020     GetCommandAuditDigest_Entry    *entry;
2021     UINT16                         inSize;
2022     UINT16                         outSize;
2023     UINT16                         offsetOfTypes;
2024     UINT16                         paramOffsets[4];
2025     BYTE                           types[8];
2026 } GetCommandAuditDigest_COMMAND_DESCRIPTOR_t;
2027
2028 GetCommandAuditDigest_COMMAND_DESCRIPTOR_t _GetCommandAuditDigestData = {
2029     /* entry */                &TPM2_GetCommandAuditDigest,
2030     /* inSize */               (UINT16) (sizeof(GetCommandAuditDigest_In)),
2031     /* outSize */              (UINT16) (sizeof(GetCommandAuditDigest_Out)),
2032     /* offsetOfTypes */
offsetof(GetCommandAuditDigest_COMMAND_DESCRIPTOR_t, types),
2033     /* offsets */              { (UINT16) (offsetof(GetCommandAuditDigest_In,
signHandle)),
                                (UINT16) (offsetof(GetCommandAuditDigest_In,
qualifyingData)),
                                (UINT16) (offsetof(GetCommandAuditDigest_In,
inScheme)),
                                (UINT16) (offsetof(GetCommandAuditDigest_Out,
signature)) },
2034     /* types */                { TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
                                TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
                                TPM2B_DATA_P_UNMARSHAL,
                                TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
                                END_OF_LIST,
                                TPM2B_ATTEST_P_MARSHAL,
                                TPMT_SIGNATURE_P_MARSHAL,
                                END_OF_LIST }
2035 };
2036
2037 #define _GetCommandAuditDigestDataAddress (&_GetCommandAuditDigestData)
2038 #else
2039 #define _GetCommandAuditDigestDataAddress 0
2040 #endif // CC_GetCommandAuditDigest
2041
2042 #if CC_GetTime
2043
2044 #include "GetTime_fp.h"
2045
2046
2047 typedef TPM_RC (GetTime_Entry) (
2048     GetTime_In      *in,
2049     GetTime_Out     *out
2050 );
2051
2052 typedef const struct {
2053     GetTime_Entry    *entry;
2054     UINT16           inSize;
2055     UINT16           outSize;
2056     UINT16           offsetOfTypes;

```

```

2066     UINT16                paramOffsets[4];
2067     BYTE                  types[8];
2068 } GetTime_COMMAND_DESCRIPTOR_t;
2069
2070 GetTime_COMMAND_DESCRIPTOR_t _GetTimeData = {
2071     /* entry */           &TPM2_GetTime,
2072     /* inSize */          (UINT16) (sizeof(GetTime_In)),
2073     /* outSize */         (UINT16) (sizeof(GetTime_Out)),
2074     /* offsetOfTypes */   offsetof(GetTime_COMMAND_DESCRIPTOR_t, types),
2075     /* offsets */         {(UINT16) (offsetof(GetTime_In, signHandle)),
2076                          (UINT16) (offsetof(GetTime_In, qualifyingData)),
2077                          (UINT16) (offsetof(GetTime_In, inScheme)),
2078                          (UINT16) (offsetof(GetTime_Out, signature))},
2079     /* types */           {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
2080                          TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
2081                          TPM2B_DATA_P_UNMARSHAL,
2082                          TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2083                          END_OF_LIST,
2084                          TPM2B_ATTEST_P_MARSHAL,
2085                          TPMT_SIGNATURE_P_MARSHAL,
2086                          END_OF_LIST};
2087 };
2088
2089 #define _GetTimeDataAddress (&_GetTimeData)
2090 #else
2091 #define _GetTimeDataAddress 0
2092 #endif // CC_GetTime
2093
2094 #if CC_CertifyX509
2095
2096 #include "CertifyX509_fp.h"
2097
2098 typedef TPM_RC (CertifyX509_Entry) (
2099     CertifyX509_In      *in,
2100     CertifyX509_Out     *out
2101 );
2102
2103 typedef const struct {
2104     CertifyX509_Entry    *entry;
2105     UINT16               inSize;
2106     UINT16               outSize;
2107     UINT16               offsetOfTypes;
2108     UINT16               paramOffsets[6];
2109     BYTE                 types[10];
2110 } CertifyX509_COMMAND_DESCRIPTOR_t;
2111
2112 CertifyX509_COMMAND_DESCRIPTOR_t _CertifyX509Data = {
2113     /* entry */           &TPM2_CertifyX509,
2114     /* inSize */          (UINT16) (sizeof(CertifyX509_In)),
2115     /* outSize */         (UINT16) (sizeof(CertifyX509_Out)),
2116     /* offsetOfTypes */   offsetof(CertifyX509_COMMAND_DESCRIPTOR_t, types),
2117     /* offsets */         {(UINT16) (offsetof(CertifyX509_In, signHandle)),
2118                          (UINT16) (offsetof(CertifyX509_In, reserved)),
2119                          (UINT16) (offsetof(CertifyX509_In, inScheme)),
2120                          (UINT16) (offsetof(CertifyX509_In, partialCertificate)),
2121                          (UINT16) (offsetof(CertifyX509_Out, tbsDigest)),
2122                          (UINT16) (offsetof(CertifyX509_Out, signature))},
2123     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
2124                          TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
2125                          TPM2B_DATA_P_UNMARSHAL,
2126                          TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2127                          TPM2B_MAX_BUFFER_P_UNMARSHAL,
2128                          END_OF_LIST,
2129                          TPM2B_MAX_BUFFER_P_MARSHAL,
2130                          TPM2B_DIGEST_P_MARSHAL,
2131                          TPMT_SIGNATURE_P_MARSHAL,

```



```

2132                                     END_OF_LIST}
2133 };
2134
2135 #define _CertifyX509DataAddress (&_CertifyX509Data)
2136 #else
2137 #define _CertifyX509DataAddress 0
2138 #endif // CC_CertifyX509
2139
2140 #if CC_Commit
2141
2142 #include "Commit_fp.h"
2143
2144 typedef TPM_RC (Commit_Entry)(
2145     Commit_In          *in,
2146     Commit_Out         *out
2147 );
2148
2149 typedef const struct {
2150     Commit_Entry        *entry;
2151     UINT16              inSize;
2152     UINT16              outSize;
2153     UINT16              offsetOfTypes;
2154     UINT16              paramOffsets[6];
2155     BYTE                types[10];
2156 } Commit_COMMAND_DESCRIPTOR_t;
2157
2158 Commit_COMMAND_DESCRIPTOR_t _CommitData = {
2159     /* entry */          &TPM2_Commit,
2160     /* inSize */         (UINT16)(sizeof(Commit_In)),
2161     /* outSize */        (UINT16)(sizeof(Commit_Out)),
2162     /* offsetOfTypes */  offsetof(Commit_COMMAND_DESCRIPTOR_t, types),
2163     /* offsets */        {(UINT16)(offsetof(Commit_In, P1)),
2164                          (UINT16)(offsetof(Commit_In, s2)),
2165                          (UINT16)(offsetof(Commit_In, y2)),
2166                          (UINT16)(offsetof(Commit_Out, L)),
2167                          (UINT16)(offsetof(Commit_Out, E)),
2168                          (UINT16)(offsetof(Commit_Out, counter)))},
2169     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
2170                          TPM2B_ECC_POINT_P_UNMARSHAL,
2171                          TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
2172                          TPM2B_ECC_PARAMETER_P_UNMARSHAL,
2173                          END_OF_LIST,
2174                          TPM2B_ECC_POINT_P_MARSHAL,
2175                          TPM2B_ECC_POINT_P_MARSHAL,
2176                          TPM2B_ECC_POINT_P_MARSHAL,
2177                          UINT16_P_MARSHAL,
2178                          END_OF_LIST}
2179 };
2180
2181 #define _CommitDataAddress (&_CommitData)
2182 #else
2183 #define _CommitDataAddress 0
2184 #endif // CC_Commit
2185
2186 #if CC_EC_Ephemeral
2187
2188 #include "EC_Ephemeral_fp.h"
2189
2190 typedef TPM_RC (EC_Ephemeral_Entry)(
2191     EC_Ephemeral_In    *in,
2192     EC_Ephemeral_Out    *out
2193 );
2194
2195 typedef const struct {
2196     EC_Ephemeral_Entry  *entry;
2197     UINT16              inSize;

```

```

2198     UINT16          outSize;
2199     UINT16          offsetOfTypes;
2200     UINT16          paramOffsets[1];
2201     BYTE            types[5];
2202 } EC_Ephemeral_COMMAND_DESCRIPTOR_t;
2203
2204 EC_Ephemeral_COMMAND_DESCRIPTOR_t _EC_EphemeralData = {
2205     /* entry          */ &TPM2_EC_Ephemeral,
2206     /* inSize         */ (UINT16) (sizeof(EC_Ephemeral_In)),
2207     /* outSize        */ (UINT16) (sizeof(EC_Ephemeral_Out)),
2208     /* offsetOfTypes  */ offsetof(EC_Ephemeral_COMMAND_DESCRIPTOR_t, types),
2209     /* offsets        */ { (UINT16) (offsetof(EC_Ephemeral_Out, counter))},
2210     /* types          */ {TPMI_ECC_CURVE_P_UNMARSHAL,
2211                          END_OF_LIST,
2212                          TPM2B_ECC_POINT_P_MARSHAL,
2213                          UINT16_P_MARSHAL,
2214                          END_OF_LIST}
2215 };
2216
2217 #define _EC_EphemeralDataAddress (&_EC_EphemeralData)
2218 #else
2219 #define _EC_EphemeralDataAddress 0
2220 #endif // CC_EC_Ephemeral
2221
2222 #if CC_VerifySignature
2223
2224 #include "VerifySignature_fp.h"
2225
2226 typedef TPM_RC (VerifySignature_Entry) (
2227     VerifySignature_In      *in,
2228     VerifySignature_Out     *out
2229 );
2230
2231 typedef const struct {
2232     VerifySignature_Entry *entry;
2233     UINT16                inSize;
2234     UINT16                outSize;
2235     UINT16                offsetOfTypes;
2236     UINT16                paramOffsets[2];
2237     BYTE                  types[6];
2238 } VerifySignature_COMMAND_DESCRIPTOR_t;
2239
2240 VerifySignature_COMMAND_DESCRIPTOR_t _VerifySignatureData = {
2241     /* entry          */ &TPM2_VerifySignature,
2242     /* inSize         */ (UINT16) (sizeof(VerifySignature_In)),
2243     /* outSize        */ (UINT16) (sizeof(VerifySignature_Out)),
2244     /* offsetOfTypes  */ offsetof(VerifySignature_COMMAND_DESCRIPTOR_t, types),
2245     /* offsets        */ { (UINT16) (offsetof(VerifySignature_In, digest)),
2246                          (UINT16) (offsetof(VerifySignature_In, signature))},
2247     /* types          */ {TPMI_DH_OBJECT_H_UNMARSHAL,
2248                          TPM2B_DIGEST_P_UNMARSHAL,
2249                          TPMT_SIGNATURE_P_UNMARSHAL,
2250                          END_OF_LIST,
2251                          TPMT_TK_VERIFIED_P_MARSHAL,
2252                          END_OF_LIST}
2253 };
2254
2255 #define _VerifySignatureDataAddress (&_VerifySignatureData)
2256 #else
2257 #define _VerifySignatureDataAddress 0
2258 #endif // CC_VerifySignature
2259
2260 #if CC_Sign
2261
2262 #include "Sign_fp.h"
2263

```

```

2264 typedef TPM_RC (Sign_Entry) (
2265     Sign_In          *in,
2266     Sign_Out         *out
2267 );
2268
2269 typedef const struct {
2270     Sign_Entry        *entry;
2271     UINT16            inSize;
2272     UINT16            outSize;
2273     UINT16            offsetOfTypes;
2274     UINT16            paramOffsets[3];
2275     BYTE              types[7];
2276 } Sign_COMMAND_DESCRIPTOR_t;
2277
2278 Sign_COMMAND_DESCRIPTOR_t _SignData = {
2279     /* entry */          &TPM2_Sign,
2280     /* inSize */         (UINT16) (sizeof(Sign_In)),
2281     /* outSize */        (UINT16) (sizeof(Sign_Out)),
2282     /* offsetOfTypes */  offsetof(Sign_COMMAND_DESCRIPTOR_t, types),
2283     /* offsets */        { (UINT16) (offsetof(Sign_In, digest)),
2284                          (UINT16) (offsetof(Sign_In, inScheme)),
2285                          (UINT16) (offsetof(Sign_In, validation)) },
2286     /* types */          { TPMI_DH_OBJECT_H_UNMARSHAL,
2287                          TPM2B_DIGEST_P_UNMARSHAL,
2288                          TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
2289                          TPMT_TK_HASHCHECK_P_UNMARSHAL,
2290                          END_OF_LIST,
2291                          TPMT_SIGNATURE_P_MARSHAL,
2292                          END_OF_LIST }
2293 };
2294
2295 #define _SignDataAddress (&_SignData)
2296 #else
2297 #define _SignDataAddress 0
2298 #endif // CC_Sign
2299
2300 #if CC_SetCommandCodeAuditStatus
2301 #include "SetCommandCodeAuditStatus_fp.h"
2302
2303 typedef TPM_RC (SetCommandCodeAuditStatus_Entry) (
2304     SetCommandCodeAuditStatus_In *in
2305 );
2306
2307 typedef const struct {
2308     SetCommandCodeAuditStatus_Entry *entry;
2309     UINT16 inSize;
2310     UINT16 outSize;
2311     UINT16 offsetOfTypes;
2312     UINT16 paramOffsets[3];
2313     BYTE types[6];
2314 } SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t;
2315
2316 SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t _SetCommandCodeAuditStatusData = {
2317     /* entry */          &TPM2_SetCommandCodeAuditStatus,
2318     /* inSize */         (UINT16) (sizeof(SetCommandCodeAuditStatus_In)),
2319     /* outSize */        0,
2320     /* offsetOfTypes */  offsetof(SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t, types),
2321     /* offsets */        { (UINT16) (offsetof(SetCommandCodeAuditStatus_In, auditAlg)),
2322                          (UINT16) (offsetof(SetCommandCodeAuditStatus_In, setList)),
2323                          (UINT16) (offsetof(SetCommandCodeAuditStatus_In, clearList)) },

```

```

2325     /* types */
2326
2327     {TPMI_RH_PROVISION_H_UNMARSHAL,
2328     TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2329     TPML_CC_P_UNMARSHAL,
2330     TPML_CC_P_UNMARSHAL,
2331     END_OF_LIST,
2332     END_OF_LIST}
2333 };
2334
2335 #define _SetCommandCodeAuditStatusDataAddress (&_SetCommandCodeAuditStatusData)
2336 #else
2337 #define _SetCommandCodeAuditStatusDataAddress 0
2338 #endif // CC_SetCommandCodeAuditStatus
2339
2340 #if CC_PCR_Extend
2341
2342 #include "PCR_Extend_fp.h"
2343
2344 typedef TPM_RC (PCR_Extend_Entry) (
2345     PCR_Extend_In *in
2346 );
2347
2348 typedef const struct {
2349     PCR_Extend_Entry *entry;
2350     UINT16 inSize;
2351     UINT16 outSize;
2352     UINT16 offsetOfTypes;
2353     UINT16 paramOffsets[1];
2354     BYTE types[4];
2355 } PCR_Extend_COMMAND_DESCRIPTOR_t;
2356
2357 PCR_Extend_COMMAND_DESCRIPTOR_t _PCR_ExtendData = {
2358     /* entry */ &TPM2_PCR_Extend,
2359     /* inSize */ (UINT16) (sizeof(PCR_Extend_In)),
2360     /* outSize */ 0,
2361     /* offsetOfTypes */ offsetof(PCR_Extend_COMMAND_DESCRIPTOR_t, types),
2362     /* offsets */ {(UINT16) (offsetof(PCR_Extend_In, digests))},
2363     /* types */ {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
2364     TPMI_DIGEST_VALUES_P_UNMARSHAL,
2365     END_OF_LIST,
2366     END_OF_LIST}
2367 };
2368
2369 #define _PCR_ExtendDataAddress (&_PCR_ExtendData)
2370 #else
2371 #define _PCR_ExtendDataAddress 0
2372 #endif // CC_PCR_Extend
2373
2374 #if CC_PCR_Event
2375
2376 #include "PCR_Event_fp.h"
2377
2378 typedef TPM_RC (PCR_Event_Entry) (
2379     PCR_Event_In *in,
2380     PCR_Event_Out *out
2381 );
2382
2383 typedef const struct {
2384     PCR_Event_Entry *entry;
2385     UINT16 inSize;
2386     UINT16 outSize;
2387     UINT16 offsetOfTypes;
2388     UINT16 paramOffsets[1];
2389     BYTE types[5];
2390 } PCR_Event_COMMAND_DESCRIPTOR_t;
2391
2392 PCR_Event_COMMAND_DESCRIPTOR_t _PCR_EventData = {

```

```

2391     /* entry          */      &TPM2_PCR_Event,
2392     /* inSize        */      (UINT16) (sizeof(PCR_Event_In)),
2393     /* outSize       */      (UINT16) (sizeof(PCR_Event_Out)),
2394     /* offsetOfTypes */      offsetof(PCR_Event_COMMAND_DESCRIPTOR_t, types),
2395     /* offsets       */      {(UINT16) (offsetof(PCR_Event_In, eventData))},
2396     /* types         */      {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
2397                               TPM2B_EVENT_P_UNMARSHAL,
2398                               END_OF_LIST,
2399                               TPML_DIGEST_VALUES_P_MARSHAL,
2400                               END_OF_LIST}
2401 };
2402
2403 #define _PCR_EventDataAddress (&_PCR_EventData)
2404 #else
2405 #define _PCR_EventDataAddress 0
2406 #endif // CC_PCR_Event
2407
2408 #if CC_PCR_Read
2409
2410 #include "PCR_Read_fp.h"
2411
2412 typedef TPM_RC (PCR_Read_Entry) (
2413     PCR_Read_In          *in,
2414     PCR_Read_Out         *out
2415 );
2416
2417 typedef const struct {
2418     PCR_Read_Entry      *entry;
2419     UINT16              inSize;
2420     UINT16              outSize;
2421     UINT16              offsetOfTypes;
2422     UINT16              paramOffsets[2];
2423     BYTE                types[6];
2424 } PCR_Read_COMMAND_DESCRIPTOR_t;
2425
2426 PCR_Read_COMMAND_DESCRIPTOR_t _PCR_ReadData = {
2427     /* entry          */      &TPM2_PCR_Read,
2428     /* inSize        */      (UINT16) (sizeof(PCR_Read_In)),
2429     /* outSize       */      (UINT16) (sizeof(PCR_Read_Out)),
2430     /* offsetOfTypes */      offsetof(PCR_Read_COMMAND_DESCRIPTOR_t, types),
2431     /* offsets       */      {(UINT16) (offsetof(PCR_Read_Out, pcrSelectionOut)),
2432                               (UINT16) (offsetof(PCR_Read_Out, pcrValues))},
2433     /* types         */      {TPML_PCR_SELECTION_P_UNMARSHAL,
2434                               END_OF_LIST,
2435                               UINT32_P_MARSHAL,
2436                               TPML_PCR_SELECTION_P_MARSHAL,
2437                               TPML_DIGEST_P_MARSHAL,
2438                               END_OF_LIST}
2439 };
2440
2441 #define _PCR_ReadDataAddress (&_PCR_ReadData)
2442 #else
2443 #define _PCR_ReadDataAddress 0
2444 #endif // CC_PCR_Read
2445
2446 #if CC_PCR_Allocate
2447
2448 #include "PCR_Allocate_fp.h"
2449
2450 typedef TPM_RC (PCR_Allocate_Entry) (
2451     PCR_Allocate_In      *in,
2452     PCR_Allocate_Out     *out
2453 );
2454
2455 typedef const struct {
2456     PCR_Allocate_Entry   *entry;

```

```

2457     UINT16             inSize;
2458     UINT16             outSize;
2459     UINT16             offsetOfTypes;
2460     UINT16             paramOffsets[4];
2461     BYTE               types[8];
2462 } PCR_Allocate_COMMAND_DESCRIPTOR_t;
2463
2464 PCR_Allocate_COMMAND_DESCRIPTOR_t _PCR_AllocateData = {
2465     /* entry */          &TPM2_PCR_Allocate,
2466     /* inSize */         (UINT16) (sizeof(PCR_Allocate_In)),
2467     /* outSize */        (UINT16) (sizeof(PCR_Allocate_Out)),
2468     /* offsetOfTypes */  offsetof(PCR_Allocate_COMMAND_DESCRIPTOR_t, types),
2469     /* offsets */        { (UINT16) (offsetof(PCR_Allocate_In, pcrAllocation)),
2470                          (UINT16) (offsetof(PCR_Allocate_Out, maxPCR)),
2471                          (UINT16) (offsetof(PCR_Allocate_Out, sizeNeeded)),
2472                          (UINT16) (offsetof(PCR_Allocate_Out, sizeAvailable)) },
2473     /* types */          {TPMI_RH_PLATFORM_H_UNMARSHAL,
2474                          TPML_PCR_SELECTION_P_UNMARSHAL,
2475                          END_OF_LIST,
2476                          TPMI_YES_NO_P_MARSHAL,
2477                          UINT32_P_MARSHAL,
2478                          UINT32_P_MARSHAL,
2479                          UINT32_P_MARSHAL,
2480                          END_OF_LIST}
2481 };
2482
2483 #define _PCR_AllocateDataAddress (&_PCR_AllocateData)
2484 #else
2485 #define _PCR_AllocateDataAddress 0
2486 #endif // CC_PCR_Allocate
2487
2488 #if CC_PCR_SetAuthPolicy
2489
2490 #include "PCR_SetAuthPolicy_fp.h"
2491
2492 typedef TPM_RC (PCR_SetAuthPolicy_Entry) (
2493     PCR_SetAuthPolicy_In *in
2494 );
2495
2496 typedef const struct {
2497     PCR_SetAuthPolicy_Entry *entry;
2498     UINT16 inSize;
2499     UINT16 outSize;
2500     UINT16 offsetOfTypes;
2501     UINT16 paramOffsets[3];
2502     BYTE types[6];
2503 } PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t;
2504
2505 PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t _PCR_SetAuthPolicyData = {
2506     /* entry */          &TPM2_PCR_SetAuthPolicy,
2507     /* inSize */         (UINT16) (sizeof(PCR_SetAuthPolicy_In)),
2508     /* outSize */        0,
2509     /* offsetOfTypes */  offsetof(PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t,
2510     types),
2511     /* offsets */        { (UINT16) (offsetof(PCR_SetAuthPolicy_In, authPolicy)),
2512                          (UINT16) (offsetof(PCR_SetAuthPolicy_In, hashAlg)),
2513                          (UINT16) (offsetof(PCR_SetAuthPolicy_In, pcrNum)) },
2514     /* types */          {TPMI_RH_PLATFORM_H_UNMARSHAL,
2515                          TPM2B_DIGEST_P_UNMARSHAL,
2516                          TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2517                          TPMI_DH_PCR_P_UNMARSHAL,
2518                          END_OF_LIST,
2519                          END_OF_LIST}
2519 };
2520
2521 #define _PCR_SetAuthPolicyDataAddress (&_PCR_SetAuthPolicyData)

```



```

2522 #else
2523 #define _PCR_SetAuthPolicyDataAddress 0
2524 #endif // CC_PCR_SetAuthPolicy
2525
2526 #if CC_PCR_SetAuthValue
2527
2528 #include "PCR_SetAuthValue_fp.h"
2529
2530 typedef TPM_RC (PCR_SetAuthValue_Entry) (
2531     PCR_SetAuthValue_In      *in
2532 );
2533
2534 typedef const struct {
2535     PCR_SetAuthValue_Entry  *entry;
2536     UINT16                  inSize;
2537     UINT16                  outSize;
2538     UINT16                  offsetOfTypes;
2539     UINT16                  paramOffsets[1];
2540     BYTE                    types[4];
2541 } PCR_SetAuthValue_COMMAND_DESCRIPTOR_t;
2542
2543 PCR_SetAuthValue_COMMAND_DESCRIPTOR_t _PCR_SetAuthValueData = {
2544     /* entry          */      &TPM2_PCR_SetAuthValue,
2545     /* inSize         */      (UINT16) (sizeof(PCR_SetAuthValue_In)),
2546     /* outSize        */      0,
2547     /* offsetOfTypes  */      offsetof(PCR_SetAuthValue_COMMAND_DESCRIPTOR_t, types),
2548     /* offsets        */      {(UINT16) (offsetof(PCR_SetAuthValue_In, auth))},
2549     /* types          */      {TPMI_DH_PCR_H_UNMARSHAL,
2550                             TPM2B_DIGEST_P_UNMARSHAL,
2551                             END_OF_LIST,
2552                             END_OF_LIST}
2553 };
2554
2555 #define _PCR_SetAuthValueDataAddress (&_PCR_SetAuthValueData)
2556 #else
2557 #define _PCR_SetAuthValueDataAddress 0
2558 #endif // CC_PCR_SetAuthValue
2559
2560 #if CC_PCR_Reset
2561
2562 #include "PCR_Reset_fp.h"
2563
2564 typedef TPM_RC (PCR_Reset_Entry) (
2565     PCR_Reset_In            *in
2566 );
2567
2568 typedef const struct {
2569     PCR_Reset_Entry         *entry;
2570     UINT16                  inSize;
2571     UINT16                  outSize;
2572     UINT16                  offsetOfTypes;
2573     BYTE                    types[3];
2574 } PCR_Reset_COMMAND_DESCRIPTOR_t;
2575
2576 PCR_Reset_COMMAND_DESCRIPTOR_t _PCR_ResetData = {
2577     /* entry          */      &TPM2_PCR_Reset,
2578     /* inSize         */      (UINT16) (sizeof(PCR_Reset_In)),
2579     /* outSize        */      0,
2580     /* offsetOfTypes  */      offsetof(PCR_Reset_COMMAND_DESCRIPTOR_t, types),
2581     /* offsets        */      // No parameter offsets;
2582     /* types          */      {TPMI_DH_PCR_H_UNMARSHAL,
2583                             END_OF_LIST,
2584                             END_OF_LIST}
2585 };
2586
2587 #define _PCR_ResetDataAddress (&_PCR_ResetData)

```



```

2588 #else
2589 #define _PCR_ResetDataAddress 0
2590 #endif // CC_PCR_Reset
2591
2592 #if CC_PolicySigned
2593
2594 #include "PolicySigned_fp.h"
2595
2596 typedef TPM_RC (PolicySigned_Entry) (
2597     PolicySigned_In      *in,
2598     PolicySigned_Out     *out
2599 );
2600
2601 typedef const struct {
2602     PolicySigned_Entry    *entry;
2603     UINT16                inSize;
2604     UINT16                outSize;
2605     UINT16                offsetOfTypes;
2606     UINT16                paramOffsets[7];
2607     BYTE                  types[11];
2608 } PolicySigned_COMMAND_DESCRIPTOR_t;
2609
2610 PolicySigned_COMMAND_DESCRIPTOR_t _PolicySignedData = {
2611     /* entry */          &TPM2_PolicySigned,
2612     /* inSize */         (UINT16) (sizeof(PolicySigned_In)),
2613     /* outSize */        (UINT16) (sizeof(PolicySigned_Out)),
2614     /* offsetOfTypes */  offsetof(PolicySigned_COMMAND_DESCRIPTOR_t, types),
2615     /* offsets */        { (UINT16) (offsetof(PolicySigned_In, policySession)),
2616                          (UINT16) (offsetof(PolicySigned_In, nonceTPM)),
2617                          (UINT16) (offsetof(PolicySigned_In, cpHashA)),
2618                          (UINT16) (offsetof(PolicySigned_In, policyRef)),
2619                          (UINT16) (offsetof(PolicySigned_In, expiration)),
2620                          (UINT16) (offsetof(PolicySigned_In, auth)),
2621                          (UINT16) (offsetof(PolicySigned_Out, policyTicket)) },
2622     /* types */          { TPMI_DH_OBJECT_H_UNMARSHAL,
2623                          TPMI_SH_POLICY_H_UNMARSHAL,
2624                          TPM2B_NONCE_P_UNMARSHAL,
2625                          TPM2B_DIGEST_P_UNMARSHAL,
2626                          TPM2B_NONCE_P_UNMARSHAL,
2627                          INT32_P_UNMARSHAL,
2628                          TPMT_SIGNATURE_P_UNMARSHAL,
2629                          END_OF_LIST,
2630                          TPM2B_TIMEOUT_P_MARSHAL,
2631                          TPMT_TK_AUTH_P_MARSHAL,
2632                          END_OF_LIST }
2633 };
2634
2635 #define _PolicySignedDataAddress (&_PolicySignedData)
2636 #else
2637 #define _PolicySignedDataAddress 0
2638 #endif // CC_PolicySigned
2639
2640 #if CC_PolicySecret
2641
2642 #include "PolicySecret_fp.h"
2643
2644 typedef TPM_RC (PolicySecret_Entry) (
2645     PolicySecret_In      *in,
2646     PolicySecret_Out     *out
2647 );
2648
2649 typedef const struct {
2650     PolicySecret_Entry    *entry;
2651     UINT16                inSize;
2652     UINT16                outSize;
2653     UINT16                offsetOfTypes;

```

```

2654     UINT16                paramOffsets[6];
2655     BYTE                  types[10];
2656 } PolicySecret_COMMAND_DESCRIPTOR_t;
2657
2658 PolicySecret_COMMAND_DESCRIPTOR_t _PolicySecretData = {
2659     /* entry */           &TPM2_PolicySecret,
2660     /* inSize */          (UINT16) (sizeof(PolicySecret_In)),
2661     /* outSize */         (UINT16) (sizeof(PolicySecret_Out)),
2662     /* offsetOfTypes */   offsetof(PolicySecret_COMMAND_DESCRIPTOR_t, types),
2663     /* offsets */         { (UINT16) (offsetof(PolicySecret_In, policySession)),
2664                           (UINT16) (offsetof(PolicySecret_In, nonceTPM)),
2665                           (UINT16) (offsetof(PolicySecret_In, cpHashA)),
2666                           (UINT16) (offsetof(PolicySecret_In, policyRef)),
2667                           (UINT16) (offsetof(PolicySecret_In, expiration)),
2668                           (UINT16) (offsetof(PolicySecret_Out, policyTicket))},
2669     /* types */           {TPMI_DH_ENTITY_H_UNMARSHAL,
2670                           TPMI_SH_POLICY_H_UNMARSHAL,
2671                           TPM2B_NONCE_P_UNMARSHAL,
2672                           TPM2B_DIGEST_P_UNMARSHAL,
2673                           TPM2B_NONCE_P_UNMARSHAL,
2674                           INT32_P_UNMARSHAL,
2675                           END_OF_LIST,
2676                           TPM2B_TIMEOUT_P_MARSHAL,
2677                           TPMT_TK_AUTH_P_MARSHAL,
2678                           END_OF_LIST};
2679 };
2680
2681 #define _PolicySecretDataAddress (&_PolicySecretData)
2682 #else
2683 #define _PolicySecretDataAddress 0
2684 #endif // CC_PolicySecret
2685
2686 #if CC_PolicyTicket
2687
2688 #include "PolicyTicket_fp.h"
2689
2690 typedef TPM_RC (PolicyTicket_Entry) (
2691     PolicyTicket_In *in
2692 );
2693
2694 typedef const struct {
2695     PolicyTicket_Entry *entry;
2696     UINT16 inSize;
2697     UINT16 outSize;
2698     UINT16 offsetOfTypes;
2699     UINT16 paramOffsets[5];
2700     BYTE types[8];
2701 } PolicyTicket_COMMAND_DESCRIPTOR_t;
2702
2703 PolicyTicket_COMMAND_DESCRIPTOR_t _PolicyTicketData = {
2704     /* entry */           &TPM2_PolicyTicket,
2705     /* inSize */          (UINT16) (sizeof(PolicyTicket_In)),
2706     /* outSize */         0,
2707     /* offsetOfTypes */   offsetof(PolicyTicket_COMMAND_DESCRIPTOR_t, types),
2708     /* offsets */         { (UINT16) (offsetof(PolicyTicket_In, timeout)),
2709                           (UINT16) (offsetof(PolicyTicket_In, cpHashA)),
2710                           (UINT16) (offsetof(PolicyTicket_In, policyRef)),
2711                           (UINT16) (offsetof(PolicyTicket_In, authName)),
2712                           (UINT16) (offsetof(PolicyTicket_In, ticket))},
2713     /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2714                           TPM2B_TIMEOUT_P_UNMARSHAL,
2715                           TPM2B_DIGEST_P_UNMARSHAL,
2716                           TPM2B_NONCE_P_UNMARSHAL,
2717                           TPM2B_NAME_P_UNMARSHAL,
2718                           TPMT_TK_AUTH_P_UNMARSHAL,
2719                           END_OF_LIST,

```

```

2720                                     END_OF_LIST}
2721 };
2722
2723 #define _PolicyTicketDataAddress (&_PolicyTicketData)
2724 #else
2725 #define _PolicyTicketDataAddress 0
2726 #endif // CC_PolicyTicket
2727
2728 #if CC_PolicyOR
2729
2730 #include "PolicyOR_fp.h"
2731
2732 typedef TPM_RC (PolicyOR_Entry) (
2733     PolicyOR_In          *in
2734 );
2735
2736 typedef const struct {
2737     PolicyOR_Entry        *entry;
2738     UINT16                inSize;
2739     UINT16                outSize;
2740     UINT16                offsetOfTypes;
2741     UINT16                paramOffsets[1];
2742     BYTE                  types[4];
2743 } PolicyOR_COMMAND_DESCRIPTOR_t;
2744
2745 PolicyOR_COMMAND_DESCRIPTOR_t _PolicyORData = {
2746     /* entry          */    &TPM2_PolicyOR,
2747     /* inSize         */    (UINT16) (sizeof(PolicyOR_In)),
2748     /* outSize        */    0,
2749     /* offsetOfTypes  */    offsetof(PolicyOR_COMMAND_DESCRIPTOR_t, types),
2750     /* offsets        */    {(UINT16) (offsetof(PolicyOR_In, pHashList))},
2751     /* types          */    {TPMI_SH_POLICY_H_UNMARSHAL,
2752                             TPML_DIGEST_P_UNMARSHAL,
2753                             END_OF_LIST,
2754                             END_OF_LIST}
2755 };
2756
2757 #define _PolicyORDataAddress (&_PolicyORData)
2758 #else
2759 #define _PolicyORDataAddress 0
2760 #endif // CC_PolicyOR
2761
2762 #if CC_PolicyPCR
2763
2764 #include "PolicyPCR_fp.h"
2765
2766 typedef TPM_RC (PolicyPCR_Entry) (
2767     PolicyPCR_In          *in
2768 );
2769
2770 typedef const struct {
2771     PolicyPCR_Entry        *entry;
2772     UINT16                inSize;
2773     UINT16                outSize;
2774     UINT16                offsetOfTypes;
2775     UINT16                paramOffsets[2];
2776     BYTE                  types[5];
2777 } PolicyPCR_COMMAND_DESCRIPTOR_t;
2778
2779 PolicyPCR_COMMAND_DESCRIPTOR_t _PolicyPCRData = {
2780     /* entry          */    &TPM2_PolicyPCR,
2781     /* inSize         */    (UINT16) (sizeof(PolicyPCR_In)),
2782     /* outSize        */    0,
2783     /* offsetOfTypes  */    offsetof(PolicyPCR_COMMAND_DESCRIPTOR_t, types),
2784     /* offsets        */    {(UINT16) (offsetof(PolicyPCR_In, pcrDigest)),
2785                             (UINT16) (offsetof(PolicyPCR_In, pcrs))},
2786 
```

```

2786     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2787                  TPM2B_DIGEST_P_UNMARSHAL,
2788                  TPML_PCR_SELECTION_P_UNMARSHAL,
2789                  END_OF_LIST,
2790                  END_OF_LIST}
2791 };
2792
2793 #define _PolicyPCRDataAddress (&_PolicyPCRData)
2794 #else
2795 #define _PolicyPCRDataAddress 0
2796 #endif // CC_PolicyPCR
2797
2798 #if CC_PolicyLocality
2799
2800 #include "PolicyLocality_fp.h"
2801
2802 typedef TPM_RC (PolicyLocality_Entry) (
2803     PolicyLocality_In *in
2804 );
2805
2806 typedef const struct {
2807     PolicyLocality_Entry *entry;
2808     UINT16 inSize;
2809     UINT16 outSize;
2810     UINT16 offsetOfTypes;
2811     UINT16 paramOffsets[1];
2812     BYTE types[4];
2813 } PolicyLocality_COMMAND_DESCRIPTOR_t;
2814
2815 PolicyLocality_COMMAND_DESCRIPTOR_t _PolicyLocalityData = {
2816     /* entry */ &TPM2_PolicyLocality,
2817     /* inSize */ (UINT16) (sizeof(PolicyLocality_In)),
2818     /* outSize */ 0,
2819     /* offsetOfTypes */ offsetof(PolicyLocality_COMMAND_DESCRIPTOR_t, types),
2820     /* offsets */ { (UINT16) (offsetof(PolicyLocality_In, locality)) },
2821     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2822                  TPMA_LOCALITY_P_UNMARSHAL,
2823                  END_OF_LIST,
2824                  END_OF_LIST}
2825 };
2826
2827 #define _PolicyLocalityDataAddress (&_PolicyLocalityData)
2828 #else
2829 #define _PolicyLocalityDataAddress 0
2830 #endif // CC_PolicyLocality
2831
2832 #if CC_PolicyNV
2833
2834 #include "PolicyNV_fp.h"
2835
2836 typedef TPM_RC (PolicyNV_Entry) (
2837     PolicyNV_In *in
2838 );
2839
2840 typedef const struct {
2841     PolicyNV_Entry *entry;
2842     UINT16 inSize;
2843     UINT16 outSize;
2844     UINT16 offsetOfTypes;
2845     UINT16 paramOffsets[5];
2846     BYTE types[8];
2847 } PolicyNV_COMMAND_DESCRIPTOR_t;
2848
2849 PolicyNV_COMMAND_DESCRIPTOR_t _PolicyNVData = {
2850     /* entry */ &TPM2_PolicyNV,
2851     /* inSize */ (UINT16) (sizeof(PolicyNV_In)),

```

```

2852     /* outSize */ 0,
2853     /* offsetOfTypes */ offsetof(PolicyNV_COMMAND_DESCRIPTOR_t, types),
2854     /* offsets */ { (UINT16) (offsetof(PolicyNV_In, nvIndex)),
2855                   (UINT16) (offsetof(PolicyNV_In, policySession)),
2856                   (UINT16) (offsetof(PolicyNV_In, operandB)),
2857                   (UINT16) (offsetof(PolicyNV_In, offset)),
2858                   (UINT16) (offsetof(PolicyNV_In, operation)) },
2859     /* types */ { TPMI_RH_NV_AUTH_H_UNMARSHAL,
2860                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
2861                 TPMI_SH_POLICY_H_UNMARSHAL,
2862                 TPM2B_OPERAND_P_UNMARSHAL,
2863                 UINT16_P_UNMARSHAL,
2864                 TPM_EO_P_UNMARSHAL,
2865                 END_OF_LIST,
2866                 END_OF_LIST }
2867 };
2868
2869 #define _PolicyNVDataAddress (&_PolicyNVData)
2870 #else
2871 #define _PolicyNVDataAddress 0
2872 #endif // CC_PolicyNV
2873
2874 #if CC_PolicyCounterTimer
2875
2876 #include "PolicyCounterTimer_fp.h"
2877
2878 typedef TPM_RC (PolicyCounterTimer_Entry) (
2879     PolicyCounterTimer_In *in
2880 );
2881
2882 typedef const struct {
2883     PolicyCounterTimer_Entry *entry;
2884     UINT16 inSize;
2885     UINT16 outSize;
2886     UINT16 offsetOfTypes;
2887     UINT16 paramOffsets[3];
2888     BYTE types[6];
2889 } PolicyCounterTimer_COMMAND_DESCRIPTOR_t;
2890
2891 PolicyCounterTimer_COMMAND_DESCRIPTOR_t _PolicyCounterTimerData = {
2892     /* entry */ &TPM2_PolicyCounterTimer,
2893     /* inSize */ (UINT16) (sizeof(PolicyCounterTimer_In)),
2894     /* outSize */ 0,
2895     /* offsetOfTypes */ offsetof(PolicyCounterTimer_COMMAND_DESCRIPTOR_t,
2896     types),
2897     /* offsets */ { (UINT16) (offsetof(PolicyCounterTimer_In, operandB)),
2898                   (UINT16) (offsetof(PolicyCounterTimer_In, offset)),
2899                   (UINT16) (offsetof(PolicyCounterTimer_In,
2900     operation)) },
2901     /* types */ { TPMI_SH_POLICY_H_UNMARSHAL,
2902                 TPM2B_OPERAND_P_UNMARSHAL,
2903                 UINT16_P_UNMARSHAL,
2904                 TPM_EO_P_UNMARSHAL,
2905                 END_OF_LIST,
2906                 END_OF_LIST }
2907 };
2908
2909 #define _PolicyCounterTimerDataAddress (&_PolicyCounterTimerData)
2910 #else
2911 #define _PolicyCounterTimerDataAddress 0
2912 #endif // CC_PolicyCounterTimer
2913
2914 #if CC_PolicyCommandCode
2915
2916 #include "PolicyCommandCode_fp.h"
2917

```

```

2916 typedef TPM_RC (PolicyCommandCode_Entry) (
2917     PolicyCommandCode_In          *in
2918 );
2919
2920 typedef const struct {
2921     PolicyCommandCode_Entry      *entry;
2922     UINT16                       inSize;
2923     UINT16                       outSize;
2924     UINT16                       offsetOfTypes;
2925     UINT16                       paramOffsets[1];
2926     BYTE                         types[4];
2927 } PolicyCommandCode_COMMAND_DESCRIPTOR_t;
2928
2929 PolicyCommandCode_COMMAND_DESCRIPTOR_t _PolicyCommandCodeData = {
2930     /* entry          */      &TPM2_PolicyCommandCode,
2931     /* inSize         */      (UINT16) (sizeof(PolicyCommandCode_In)),
2932     /* outSize        */      0,
2933     /* offsetOfTypes */      offsetof(PolicyCommandCode_COMMAND_DESCRIPTOR_t,
2934     types),
2935     /* offsets        */      { (UINT16) (offsetof(PolicyCommandCode_In, code)) },
2936     /* types          */      { TPMI_SH_POLICY_H_UNMARSHAL,
2937                               TPM_CC_P_UNMARSHAL,
2938                               END_OF_LIST,
2939                               END_OF_LIST }
2940 };
2941
2942 #define _PolicyCommandCodeDataAddress (&_PolicyCommandCodeData)
2943 #else
2944 #define _PolicyCommandCodeDataAddress 0
2945 #endif // CC_PolicyCommandCode
2946
2947 #if CC_PolicyPhysicalPresence
2948 #include "PolicyPhysicalPresence_fp.h"
2949
2950 typedef TPM_RC (PolicyPhysicalPresence_Entry) (
2951     PolicyPhysicalPresence_In      *in
2952 );
2953
2954 typedef const struct {
2955     PolicyPhysicalPresence_Entry    *entry;
2956     UINT16                         inSize;
2957     UINT16                         outSize;
2958     UINT16                         offsetOfTypes;
2959     BYTE                           types[3];
2960 } PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t;
2961
2962 PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t _PolicyPhysicalPresenceData = {
2963     /* entry          */      &TPM2_PolicyPhysicalPresence,
2964     /* inSize         */      (UINT16) (sizeof(PolicyPhysicalPresence_In)),
2965     /* outSize        */      0,
2966     /* offsetOfTypes */      offsetof(PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t, types),
2967     /* offsets        */      // No parameter offsets;
2968     /* types          */      { TPMI_SH_POLICY_H_UNMARSHAL,
2969                               END_OF_LIST,
2970                               END_OF_LIST }
2971 };
2972
2973 #define _PolicyPhysicalPresenceDataAddress (&_PolicyPhysicalPresenceData)
2974 #else
2975 #define _PolicyPhysicalPresenceDataAddress 0
2976 #endif // CC_PolicyPhysicalPresence
2977
2978 #if CC_PolicyCpHash
2979

```



```

2980 #include "PolicyCpHash_fp.h"
2981
2982 typedef TPM_RC (PolicyCpHash_Entry) (
2983     PolicyCpHash_In      *in
2984 );
2985
2986 typedef const struct {
2987     PolicyCpHash_Entry    *entry;
2988     UINT16                inSize;
2989     UINT16                outSize;
2990     UINT16                offsetOfTypes;
2991     UINT16                paramOffsets[1];
2992     BYTE                  types[4];
2993 } PolicyCpHash_COMMAND_DESCRIPTOR_t;
2994
2995 PolicyCpHash_COMMAND_DESCRIPTOR_t _PolicyCpHashData = {
2996     /* entry */          &TPM2_PolicyCpHash,
2997     /* inSize */         (UINT16) (sizeof(PolicyCpHash_In)),
2998     /* outSize */        0,
2999     /* offsetOfTypes */  offsetof(PolicyCpHash_COMMAND_DESCRIPTOR_t, types),
3000     /* offsets */        {(UINT16) (offsetof(PolicyCpHash_In, cpHashA))},
3001     /* types */          {TPMI_SH_POLICY_H_UNMARSHAL,
3002                          TPM2B_DIGEST_P_UNMARSHAL,
3003                          END_OF_LIST,
3004                          END_OF_LIST}
3005 };
3006
3007 #define _PolicyCpHashDataAddress (&_PolicyCpHashData)
3008 #else
3009 #define _PolicyCpHashDataAddress 0
3010 #endif // CC_PolicyCpHash
3011
3012 #if CC_PolicyNameHash
3013
3014 #include "PolicyNameHash_fp.h"
3015
3016 typedef TPM_RC (PolicyNameHash_Entry) (
3017     PolicyNameHash_In     *in
3018 );
3019
3020 typedef const struct {
3021     PolicyNameHash_Entry  *entry;
3022     UINT16                inSize;
3023     UINT16                outSize;
3024     UINT16                offsetOfTypes;
3025     UINT16                paramOffsets[1];
3026     BYTE                  types[4];
3027 } PolicyNameHash_COMMAND_DESCRIPTOR_t;
3028
3029 PolicyNameHash_COMMAND_DESCRIPTOR_t _PolicyNameHashData = {
3030     /* entry */          &TPM2_PolicyNameHash,
3031     /* inSize */         (UINT16) (sizeof(PolicyNameHash_In)),
3032     /* outSize */        0,
3033     /* offsetOfTypes */  offsetof(PolicyNameHash_COMMAND_DESCRIPTOR_t, types),
3034     /* offsets */        {(UINT16) (offsetof(PolicyNameHash_In, nameHash))},
3035     /* types */          {TPMI_SH_POLICY_H_UNMARSHAL,
3036                          TPM2B_DIGEST_P_UNMARSHAL,
3037                          END_OF_LIST,
3038                          END_OF_LIST}
3039 };
3040
3041 #define _PolicyNameHashDataAddress (&_PolicyNameHashData)
3042 #else
3043 #define _PolicyNameHashDataAddress 0
3044 #endif // CC_PolicyNameHash
3045

```



```

3046 #if CC_PolicyDuplicationSelect
3047
3048 #include "PolicyDuplicationSelect_fp.h"
3049
3050 typedef TPM_RC (PolicyDuplicationSelect_Entry) (
3051     PolicyDuplicationSelect_In      *in
3052 );
3053
3054 typedef const struct {
3055     PolicyDuplicationSelect_Entry  *entry;
3056     UINT16                          inSize;
3057     UINT16                          outSize;
3058     UINT16                          offsetOfTypes;
3059     UINT16                          paramOffsets[3];
3060     BYTE                            types[6];
3061 } PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t;
3062
3063 PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t _PolicyDuplicationSelectData = {
3064     /* entry */                &TPM2_PolicyDuplicationSelect,
3065     /* inSize */               (UINT16) (sizeof(PolicyDuplicationSelect_In)),
3066     /* outSize */              0,
3067     /* offsetOfTypes */
offsetof(PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t, types),
3068     /* offsets */              {(UINT16) (offsetof(PolicyDuplicationSelect_In,
objectName))},
3069                               {(UINT16) (offsetof(PolicyDuplicationSelect_In,
newParentName))},
3070                               {(UINT16) (offsetof(PolicyDuplicationSelect_In,
includeObject))},
3071     /* types */                {TPMI_SH_POLICY_H_UNMARSHAL,
3072                               TPM2B_NAME_P_UNMARSHAL,
3073                               TPM2B_NAME_P_UNMARSHAL,
3074                               TPMI_YES_NO_P_UNMARSHAL,
3075                               END_OF_LIST,
3076                               END_OF_LIST}
3077 };
3078
3079 #define _PolicyDuplicationSelectDataAddress (&_PolicyDuplicationSelectData)
3080 #else
3081 #define _PolicyDuplicationSelectDataAddress 0
3082 #endif // CC_PolicyDuplicationSelect
3083
3084 #if CC_PolicyAuthorize
3085
3086 #include "PolicyAuthorize_fp.h"
3087
3088 typedef TPM_RC (PolicyAuthorize_Entry) (
3089     PolicyAuthorize_In      *in
3090 );
3091
3092 typedef const struct {
3093     PolicyAuthorize_Entry  *entry;
3094     UINT16                  inSize;
3095     UINT16                  outSize;
3096     UINT16                  offsetOfTypes;
3097     UINT16                  paramOffsets[4];
3098     BYTE                    types[7];
3099 } PolicyAuthorize_COMMAND_DESCRIPTOR_t;
3100
3101 PolicyAuthorize_COMMAND_DESCRIPTOR_t _PolicyAuthorizeData = {
3102     /* entry */                &TPM2_PolicyAuthorize,
3103     /* inSize */               (UINT16) (sizeof(PolicyAuthorize_In)),
3104     /* outSize */              0,
3105     /* offsetOfTypes */        offsetof(PolicyAuthorize_COMMAND_DESCRIPTOR_t, types),
3106     /* offsets */              {(UINT16) (offsetof(PolicyAuthorize_In, approvedPolicy)),
3107                               (UINT16) (offsetof(PolicyAuthorize_In, policyRef))},

```

```

3108             (UINT16) (offsetof(PolicyAuthorize_In, keySign)),
3109             (UINT16) (offsetof(PolicyAuthorize_In, checkTicket))),
3110     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
3111                 TPM2B_DIGEST_P_UNMARSHAL,
3112                 TPM2B_NONCE_P_UNMARSHAL,
3113                 TPM2B_NAME_P_UNMARSHAL,
3114                 TPMT_TK_VERIFIED_P_UNMARSHAL,
3115                 END_OF_LIST,
3116                 END_OF_LIST}
3117 };
3118
3119 #define _PolicyAuthorizeDataAddress (&_PolicyAuthorizeData)
3120 #else
3121 #define _PolicyAuthorizeDataAddress 0
3122 #endif // CC_PolicyAuthorize
3123
3124 #if CC_PolicyAuthValue
3125
3126 #include "PolicyAuthValue_fp.h"
3127
3128 typedef TPM_RC (PolicyAuthValue_Entry) (
3129     PolicyAuthValue_In      *in
3130 );
3131
3132 typedef const struct {
3133     PolicyAuthValue_Entry    *entry;
3134     UINT16                   inSize;
3135     UINT16                   outSize;
3136     UINT16                   offsetOfTypes;
3137     BYTE                     types[3];
3138 } PolicyAuthValue_COMMAND_DESCRIPTOR_t;
3139
3140 PolicyAuthValue_COMMAND_DESCRIPTOR_t _PolicyAuthValueData = {
3141     /* entry */ &TPM2_PolicyAuthValue,
3142     /* inSize */ (UINT16) (sizeof(PolicyAuthValue_In)),
3143     /* outSize */ 0,
3144     /* offsetOfTypes */ offsetof(PolicyAuthValue_COMMAND_DESCRIPTOR_t, types),
3145     /* offsets */ // No parameter offsets;
3146     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
3147                 END_OF_LIST,
3148                 END_OF_LIST}
3149 };
3150
3151 #define _PolicyAuthValueDataAddress (&_PolicyAuthValueData)
3152 #else
3153 #define _PolicyAuthValueDataAddress 0
3154 #endif // CC_PolicyAuthValue
3155
3156 #if CC_PolicyPassword
3157
3158 #include "PolicyPassword_fp.h"
3159
3160 typedef TPM_RC (PolicyPassword_Entry) (
3161     PolicyPassword_In      *in
3162 );
3163
3164 typedef const struct {
3165     PolicyPassword_Entry    *entry;
3166     UINT16                   inSize;
3167     UINT16                   outSize;
3168     UINT16                   offsetOfTypes;
3169     BYTE                     types[3];
3170 } PolicyPassword_COMMAND_DESCRIPTOR_t;
3171
3172 PolicyPassword_COMMAND_DESCRIPTOR_t _PolicyPasswordData = {
3173     /* entry */ &TPM2_PolicyPassword,

```

```

3174     /* inSize      */ (UINT16) (sizeof(PolicyPassword_In)),
3175     /* outSize     */ 0,
3176     /* offsetOfTypes */ offsetof(PolicyPassword_COMMAND_DESCRIPTOR_t, types),
3177     /* offsets      */ // No parameter offsets;
3178     /* types        */ {TPMI_SH_POLICY_H_UNMARSHAL,
3179                        END_OF_LIST,
3180                        END_OF_LIST};
3181 };
3182
3183 #define _PolicyPasswordDataAddress (&_PolicyPasswordData)
3184 #else
3185 #define _PolicyPasswordDataAddress 0
3186 #endif // CC_PolicyPassword
3187
3188 #if CC_PolicyGetDigest
3189
3190 #include "PolicyGetDigest_fp.h"
3191
3192 typedef TPM_RC (PolicyGetDigest_Entry) (
3193     PolicyGetDigest_In      *in,
3194     PolicyGetDigest_Out     *out
3195 );
3196
3197 typedef const struct {
3198     PolicyGetDigest_Entry *entry;
3199     UINT16                inSize;
3200     UINT16                outSize;
3201     UINT16                offsetOfTypes;
3202     BYTE                  types[4];
3203 } PolicyGetDigest_COMMAND_DESCRIPTOR_t;
3204
3205 PolicyGetDigest_COMMAND_DESCRIPTOR_t _PolicyGetDigestData = {
3206     /* entry      */ &TPM2_PolicyGetDigest,
3207     /* inSize     */ (UINT16) (sizeof(PolicyGetDigest_In)),
3208     /* outSize    */ (UINT16) (sizeof(PolicyGetDigest_Out)),
3209     /* offsetOfTypes */ offsetof(PolicyGetDigest_COMMAND_DESCRIPTOR_t, types),
3210     /* offsets     */ // No parameter offsets;
3211     /* types      */ {TPMI_SH_POLICY_H_UNMARSHAL,
3212                      END_OF_LIST,
3213                      TPM2B_DIGEST_P_MARSHAL,
3214                      END_OF_LIST};
3215 };
3216
3217 #define _PolicyGetDigestDataAddress (&_PolicyGetDigestData)
3218 #else
3219 #define _PolicyGetDigestDataAddress 0
3220 #endif // CC_PolicyGetDigest
3221
3222 #if CC_PolicyNvWritten
3223
3224 #include "PolicyNvWritten_fp.h"
3225
3226 typedef TPM_RC (PolicyNvWritten_Entry) (
3227     PolicyNvWritten_In      *in
3228 );
3229
3230 typedef const struct {
3231     PolicyNvWritten_Entry *entry;
3232     UINT16                inSize;
3233     UINT16                outSize;
3234     UINT16                offsetOfTypes;
3235     UINT16                paramOffsets[1];
3236     BYTE                  types[4];
3237 } PolicyNvWritten_COMMAND_DESCRIPTOR_t;
3238
3239 PolicyNvWritten_COMMAND_DESCRIPTOR_t _PolicyNvWrittenData = {

```

```

3240     /* entry          */ &TPM2_PolicyNvWritten,
3241     /* inSize         */ (UINT16) (sizeof(PolicyNvWritten_In)),
3242     /* outSize        */ 0,
3243     /* offsetOfTypes  */ offsetof(PolicyNvWritten_COMMAND_DESCRIPTOR_t, types),
3244     /* offsets         */ { (UINT16) (offsetof(PolicyNvWritten_In, writtenSet)) },
3245     /* types          */ { TPMI_SH_POLICY_H_UNMARSHAL,
3246                          TPMI_YES_NO_P_UNMARSHAL,
3247                          END_OF_LIST,
3248                          END_OF_LIST }
3249 };
3250
3251 #define _PolicyNvWrittenDataAddress (&_PolicyNvWrittenData)
3252 #else
3253 #define _PolicyNvWrittenDataAddress 0
3254 #endif // CC_PolicyNvWritten
3255
3256 #if CC_PolicyTemplate
3257 #include "PolicyTemplate_fp.h"
3258
3259 typedef TPM_RC (PolicyTemplate_Entry) (
3260     PolicyTemplate_In *in
3261 );
3262
3263 typedef const struct {
3264     PolicyTemplate_Entry *entry;
3265     UINT16 inSize;
3266     UINT16 outSize;
3267     UINT16 offsetOfTypes;
3268     UINT16 paramOffsets[1];
3269     BYTE types[4];
3270 } PolicyTemplate_COMMAND_DESCRIPTOR_t;
3271
3272 PolicyTemplate_COMMAND_DESCRIPTOR_t _PolicyTemplateData = {
3273     /* entry          */ &TPM2_PolicyTemplate,
3274     /* inSize         */ (UINT16) (sizeof(PolicyTemplate_In)),
3275     /* outSize        */ 0,
3276     /* offsetOfTypes  */ offsetof(PolicyTemplate_COMMAND_DESCRIPTOR_t, types),
3277     /* offsets         */ { (UINT16) (offsetof(PolicyTemplate_In, templateHash)) },
3278     /* types          */ { TPMI_SH_POLICY_H_UNMARSHAL,
3279                          TPM2B_DIGEST_P_UNMARSHAL,
3280                          END_OF_LIST,
3281                          END_OF_LIST }
3282 };
3283
3284 #define _PolicyTemplateDataAddress (&_PolicyTemplateData)
3285 #else
3286 #define _PolicyTemplateDataAddress 0
3287 #endif // CC_PolicyTemplate
3288
3289 #if CC_PolicyAuthorizeNV
3290 #include "PolicyAuthorizeNV_fp.h"
3291
3292 typedef TPM_RC (PolicyAuthorizeNV_Entry) (
3293     PolicyAuthorizeNV_In *in
3294 );
3295
3296 typedef const struct {
3297     PolicyAuthorizeNV_Entry *entry;
3298     UINT16 inSize;
3299     UINT16 outSize;
3300     UINT16 offsetOfTypes;
3301     UINT16 paramOffsets[2];
3302     BYTE types[5];
3303 } PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t;

```

```

3306
3307 PolicyAuthorizeNV_COMMAND_DESCRIPTOR t_PolicyAuthorizeNVData = {
3308     /* entry */ &TPM2_PolicyAuthorizeNV,
3309     /* inSize */ (UINT16) (sizeof(PolicyAuthorizeNV_In)),
3310     /* outSize */ 0,
3311     /* offsetOfTypes */ offsetof(PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t,
types),
3312     /* offsets */ { (UINT16) (offsetof(PolicyAuthorizeNV_In, nvIndex)),
3313                   (UINT16) (offsetof(PolicyAuthorizeNV_In,
policySession)) },
3314     /* types */ { TPMI_RH_NV_AUTH_H_UNMARSHAL,
3315                  TPMI_RH_NV_INDEX_H_UNMARSHAL,
3316                  TPMI_SH_POLICY_H_UNMARSHAL,
3317                  END_OF_LIST,
3318                  END_OF_LIST }
3319 };
3320
3321 #define _PolicyAuthorizeNVDataAddress (&_PolicyAuthorizeNVData)
3322 #else
3323 #define _PolicyAuthorizeNVDataAddress 0
3324 #endif // CC_PolicyAuthorizeNV
3325
3326 #if CC_CreatePrimary
3327
3328 #include "CreatePrimary_fp.h"
3329
3330 typedef TPM_RC (CreatePrimary_Entry) (
3331     CreatePrimary_In *in,
3332     CreatePrimary_Out *out
3333 );
3334
3335 typedef const struct {
3336     CreatePrimary_Entry *entry;
3337     UINT16 inSize;
3338     UINT16 outSize;
3339     UINT16 offsetOfTypes;
3340     UINT16 paramOffsets[9];
3341     BYTE types[13];
3342 } CreatePrimary_COMMAND_DESCRIPTOR_t;
3343
3344 CreatePrimary_COMMAND_DESCRIPTOR t_CreatePrimaryData = {
3345     /* entry */ &TPM2_CreatePrimary,
3346     /* inSize */ (UINT16) (sizeof(CreatePrimary_In)),
3347     /* outSize */ (UINT16) (sizeof(CreatePrimary_Out)),
3348     /* offsetOfTypes */ offsetof(CreatePrimary_COMMAND_DESCRIPTOR_t, types),
3349     /* offsets */ { (UINT16) (offsetof(CreatePrimary_In, inSensitive)),
3350                   (UINT16) (offsetof(CreatePrimary_In, inPublic)),
3351                   (UINT16) (offsetof(CreatePrimary_In, outsideInfo)),
3352                   (UINT16) (offsetof(CreatePrimary_In, creationPCR)),
3353                   (UINT16) (offsetof(CreatePrimary_Out, outPublic)),
3354                   (UINT16) (offsetof(CreatePrimary_Out, creationData)),
3355                   (UINT16) (offsetof(CreatePrimary_Out, creationHash)),
3356                   (UINT16) (offsetof(CreatePrimary_Out, creationTicket)),
3357                   (UINT16) (offsetof(CreatePrimary_Out, name)) },
3358     /* types */ { TPMI_RH_HIERARCHY_H_UNMARSHAL + ADD_FLAG,
3359                  TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
3360                  TPM2B_PUBLIC_P_UNMARSHAL,
3361                  TPM2B_DATA_P_UNMARSHAL,
3362                  TPML_PCR_SELECTION_P_UNMARSHAL,
3363                  END_OF_LIST,
3364                  TPM_HANDLE_H_MARSHAL,
3365                  TPM2B_PUBLIC_P_MARSHAL,
3366                  TPM2B_CREATION_DATA_P_MARSHAL,
3367                  TPM2B_DIGEST_P_MARSHAL,
3368                  TPMT_TK_CREATION_P_MARSHAL,
3369                  TPM2B_NAME_P_MARSHAL,

```

```

3370                                     END_OF_LIST}
3371 };
3372
3373 #define _CreatePrimaryDataAddress (&_CreatePrimaryData)
3374 #else
3375 #define _CreatePrimaryDataAddress 0
3376 #endif // CC_CreatePrimary
3377
3378 #if CC_HierarchyControl
3379
3380 #include "HierarchyControl_fp.h"
3381
3382 typedef TPM_RC (HierarchyControl_Entry) (
3383     HierarchyControl_In      *in
3384 );
3385
3386 typedef const struct {
3387     HierarchyControl_Entry  *entry;
3388     UINT16                   inSize;
3389     UINT16                   outSize;
3390     UINT16                   offsetOfTypes;
3391     UINT16                   paramOffsets[2];
3392     BYTE                     types[5];
3393 } HierarchyControl_COMMAND_DESCRIPTOR_t;
3394
3395 HierarchyControl_COMMAND_DESCRIPTOR_t _HierarchyControlData = {
3396     /* entry          */ &TPM2_HierarchyControl,
3397     /* inSize         */ (UINT16) (sizeof(HierarchyControl_In)),
3398     /* outSize        */ 0,
3399     /* offsetOfTypes  */ offsetof(HierarchyControl_COMMAND_DESCRIPTOR_t, types),
3400     /* offsets        */ {(UINT16) (offsetof(HierarchyControl_In, enable)),
3401                          (UINT16) (offsetof(HierarchyControl_In, state))},
3402     /* types          */ {TPMI_RH_HIERARCHY_H_UNMARSHAL,
3403                          TPMI_RH_ENABLES_P_UNMARSHAL,
3404                          TPMI_YES_NO_P_UNMARSHAL,
3405                          END_OF_LIST,
3406                          END_OF_LIST}
3407 };
3408
3409 #define _HierarchyControlDataAddress (&_HierarchyControlData)
3410 #else
3411 #define _HierarchyControlDataAddress 0
3412 #endif // CC_HierarchyControl
3413
3414 #if CC_SetPrimaryPolicy
3415
3416 #include "SetPrimaryPolicy_fp.h"
3417
3418 typedef TPM_RC (SetPrimaryPolicy_Entry) (
3419     SetPrimaryPolicy_In      *in
3420 );
3421
3422 typedef const struct {
3423     SetPrimaryPolicy_Entry  *entry;
3424     UINT16                   inSize;
3425     UINT16                   outSize;
3426     UINT16                   offsetOfTypes;
3427     UINT16                   paramOffsets[2];
3428     BYTE                     types[5];
3429 } SetPrimaryPolicy_COMMAND_DESCRIPTOR_t;
3430
3431 SetPrimaryPolicy_COMMAND_DESCRIPTOR_t _SetPrimaryPolicyData = {
3432     /* entry          */ &TPM2_SetPrimaryPolicy,
3433     /* inSize         */ (UINT16) (sizeof(SetPrimaryPolicy_In)),
3434     /* outSize        */ 0,
3435     /* offsetOfTypes  */ offsetof(SetPrimaryPolicy_COMMAND_DESCRIPTOR_t, types),

```



```

3436     /* offsets */      {(UINT16) (offsetof(SetPrimaryPolicy_In, authPolicy)),
3437                        (UINT16) (offsetof(SetPrimaryPolicy_In, hashAlg))},
3438     /* types */        {TPMI_RH_HIERARCHY_POLICY_H_UNMARSHAL,
3439                        TPM2B_DIGEST_P_UNMARSHAL,
3440                        TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
3441                        END_OF_LIST,
3442                        END_OF_LIST}
3443 };
3444
3445 #define _SetPrimaryPolicyDataAddress (&_SetPrimaryPolicyData)
3446 #else
3447 #define _SetPrimaryPolicyDataAddress 0
3448 #endif // CC_SetPrimaryPolicy
3449
3450 #if CC_ChangePPS
3451
3452 #include "ChangePPS_fp.h"
3453
3454 typedef TPM_RC (ChangePPS_Entry) (
3455     ChangePPS_In          *in
3456 );
3457
3458 typedef const struct {
3459     ChangePPS_Entry      *entry;
3460     UINT16               inSize;
3461     UINT16               outSize;
3462     UINT16               offsetOfTypes;
3463     BYTE                 types[3];
3464 } ChangePPS_COMMAND_DESCRIPTOR_t;
3465
3466 ChangePPS_COMMAND_DESCRIPTOR_t _ChangePPSData = {
3467     /* entry */          &TPM2_ChangePPS,
3468     /* inSize */         (UINT16) (sizeof(ChangePPS_In)),
3469     /* outSize */        0,
3470     /* offsetOfTypes */  offsetof(ChangePPS_COMMAND_DESCRIPTOR_t, types),
3471     /* offsets */        // No parameter offsets;
3472     /* types */          {TPMI_RH_PLATFORM_H_UNMARSHAL,
3473                          END_OF_LIST,
3474                          END_OF_LIST}
3475 };
3476
3477 #define _ChangePPSDataAddress (&_ChangePPSData)
3478 #else
3479 #define _ChangePPSDataAddress 0
3480 #endif // CC_ChangePPS
3481
3482 #if CC_ChangeEPS
3483
3484 #include "ChangeEPS_fp.h"
3485
3486 typedef TPM_RC (ChangeEPS_Entry) (
3487     ChangeEPS_In          *in
3488 );
3489
3490 typedef const struct {
3491     ChangeEPS_Entry      *entry;
3492     UINT16               inSize;
3493     UINT16               outSize;
3494     UINT16               offsetOfTypes;
3495     BYTE                 types[3];
3496 } ChangeEPS_COMMAND_DESCRIPTOR_t;
3497
3498 ChangeEPS_COMMAND_DESCRIPTOR_t _ChangeEPSData = {
3499     /* entry */          &TPM2_ChangeEPS,
3500     /* inSize */         (UINT16) (sizeof(ChangeEPS_In)),
3501     /* outSize */        0,

```



```

3502     /* offsetofTypes */    offsetof(ChangeEPS_COMMAND_DESCRIPTOR_t, types),
3503     /* offsets */          // No parameter offsets;
3504     /* types */            {TPMI_RH_PLATFORM_H_UNMARSHAL,
3505                           END_OF_LIST,
3506                           END_OF_LIST}
3507 };
3508
3509 #define _ChangeEPSDataAddress (&_ChangeEPSData)
3510 #else
3511 #define _ChangeEPSDataAddress 0
3512 #endif // CC_ChangeEPS
3513
3514 #if CC_Clear
3515
3516 #include "Clear_fp.h"
3517
3518 typedef TPM_RC (Clear_Entry)(
3519     Clear_In          *in
3520 );
3521
3522 typedef const struct {
3523     Clear_Entry        *entry;
3524     UINT16             inSize;
3525     UINT16             outSize;
3526     UINT16             offsetofTypes;
3527     BYTE               types[3];
3528 } Clear_COMMAND_DESCRIPTOR_t;
3529
3530 Clear_COMMAND_DESCRIPTOR_t _ClearData = {
3531     /* entry */          &TPM2_Clear,
3532     /* inSize */         (UINT16)(sizeof(Clear_In)),
3533     /* outSize */        0,
3534     /* offsetofTypes */  offsetof(Clear_COMMAND_DESCRIPTOR_t, types),
3535     /* offsets */        // No parameter offsets;
3536     /* types */          {TPMI_RH_CLEAR_H_UNMARSHAL,
3537                           END_OF_LIST,
3538                           END_OF_LIST}
3539 };
3540
3541 #define _ClearDataAddress (&_ClearData)
3542 #else
3543 #define _ClearDataAddress 0
3544 #endif // CC_Clear
3545
3546 #if CC_ClearControl
3547
3548 #include "ClearControl_fp.h"
3549
3550 typedef TPM_RC (ClearControl_Entry)(
3551     ClearControl_In    *in
3552 );
3553
3554 typedef const struct {
3555     ClearControl_Entry *entry;
3556     UINT16             inSize;
3557     UINT16             outSize;
3558     UINT16             offsetofTypes;
3559     UINT16             paramOffsets[1];
3560     BYTE               types[4];
3561 } ClearControl_COMMAND_DESCRIPTOR_t;
3562
3563 ClearControl_COMMAND_DESCRIPTOR_t _ClearControlData = {
3564     /* entry */          &TPM2_ClearControl,
3565     /* inSize */         (UINT16)(sizeof(ClearControl_In)),
3566     /* outSize */        0,
3567     /* offsetofTypes */  offsetof(ClearControl_COMMAND_DESCRIPTOR_t, types),

```

```

3568     /* offsets */      {(UINT16) (offsetof(ClearControl_In, disable))},
3569     /* types */        {TPMI_RH_CLEAR_H_UNMARSHAL,
3570                        TPMI_YES_NO_P_UNMARSHAL,
3571                        END_OF_LIST,
3572                        END_OF_LIST}
3573 };
3574
3575 #define _ClearControlDataAddress (&_ClearControlData)
3576 #else
3577 #define _ClearControlDataAddress 0
3578 #endif // CC_ClearControl
3579
3580 #if CC_HierarchyChangeAuth
3581
3582 #include "HierarchyChangeAuth_fp.h"
3583
3584 typedef TPM_RC (HierarchyChangeAuth_Entry) (
3585     HierarchyChangeAuth_In      *in
3586 );
3587
3588 typedef const struct {
3589     HierarchyChangeAuth_Entry  *entry;
3590     UINT16                      inSize;
3591     UINT16                      outSize;
3592     UINT16                      offsetOfTypes;
3593     UINT16                      paramOffsets[1];
3594     BYTE                        types[4];
3595 } HierarchyChangeAuth_COMMAND_DESCRIPTOR_t;
3596
3597 HierarchyChangeAuth_COMMAND_DESCRIPTOR_t _HierarchyChangeAuthData = {
3598     /* entry */          &TPM2_HierarchyChangeAuth,
3599     /* inSize */         (UINT16) (sizeof(HierarchyChangeAuth_In)),
3600     /* outSize */        0,
3601     /* offsetOfTypes */  offsetof(HierarchyChangeAuth_COMMAND_DESCRIPTOR_t,
types),
3602     /* offsets */        {(UINT16) (offsetof(HierarchyChangeAuth_In, newAuth))},
3603     /* types */          {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
3604                        TPM2B_AUTH_P_UNMARSHAL,
3605                        END_OF_LIST,
3606                        END_OF_LIST}
3607 };
3608
3609 #define _HierarchyChangeAuthDataAddress (&_HierarchyChangeAuthData)
3610 #else
3611 #define _HierarchyChangeAuthDataAddress 0
3612 #endif // CC_HierarchyChangeAuth
3613
3614 #if CC_DictionaryAttackLockReset
3615
3616 #include "DictionaryAttackLockReset_fp.h"
3617
3618 typedef TPM_RC (DictionaryAttackLockReset_Entry) (
3619     DictionaryAttackLockReset_In      *in
3620 );
3621
3622 typedef const struct {
3623     DictionaryAttackLockReset_Entry  *entry;
3624     UINT16                      inSize;
3625     UINT16                      outSize;
3626     UINT16                      offsetOfTypes;
3627     BYTE                        types[3];
3628 } DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t;
3629
3630 DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t _DictionaryAttackLockResetData = {
3631     /* entry */          &TPM2_DictionaryAttackLockReset,

```

```

3632     /* inSize */
3633     (UINT16) (sizeof(DictionaryAttackLockReset_In)),
3634     /* outSize */
3635     /* offsetOfTypes */
offsetof(DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t, types),
3636     /* offsets */
3637     /* types */
3638     {TPMI_RH_LOCKOUT_H_UNMARSHAL,
3639     END_OF_LIST,
3640     END_OF_LIST}
3641 };
3642
3643 #define DictionaryAttackLockResetDataAddress (&DictionaryAttackLockResetData)
3644 #else
3645 #define DictionaryAttackLockResetDataAddress 0
3646 #endif // CC_DictionaryAttackLockReset
3647
3648 #if CC_DictionaryAttackParameters
3649 #include "DictionaryAttackParameters_fp.h"
3650
3651 typedef TPM_RC (DictionaryAttackParameters_Entry) (
3652     DictionaryAttackParameters_In *in
3653 );
3654
3655 typedef const struct {
3656     DictionaryAttackParameters_Entry *entry;
3657     UINT16 inSize;
3658     UINT16 outSize;
3659     UINT16 offsetOfTypes;
3660     UINT16 paramOffsets[3];
3661     BYTE types[6];
3662 } DictionaryAttackParameters_COMMAND_DESCRIPTOR_t;
3663
3664 DictionaryAttackParameters_COMMAND_DESCRIPTOR_t DictionaryAttackParametersData = {
3665     /* entry */
3666     /* inSize */
3667     (UINT16) (sizeof(DictionaryAttackParameters_In)),
3668     /* outSize */
3669     /* offsetOfTypes */
offsetof(DictionaryAttackParameters_COMMAND_DESCRIPTOR_t, types),
3670     /* offsets */
3671     { (UINT16) (offsetof(DictionaryAttackParameters_In, newMaxTries)),
3672     (UINT16) (offsetof(DictionaryAttackParameters_In, newRecoveryTime)),
3673     (UINT16) (offsetof(DictionaryAttackParameters_In, lockoutRecovery)) },
3674     /* types */
3675     {TPMI_RH_LOCKOUT_H_UNMARSHAL,
3676     UINT32_P_UNMARSHAL,
3677     UINT32_P_UNMARSHAL,
3678     UINT32_P_UNMARSHAL,
3679     END_OF_LIST,
3680     END_OF_LIST}
3681 };
3682
3683 #define DictionaryAttackParametersDataAddress (&DictionaryAttackParametersData)
3684 #else
3685 #define DictionaryAttackParametersDataAddress 0
3686 #endif // CC_DictionaryAttackParameters
3687
3688 #if CC_PP_Commands
3689 #include "PP_Commands_fp.h"
3690
3691 typedef TPM_RC (PP_Commands_Entry) (
3692     PP_Commands_In *in
3693 );

```

```

3691
3692 typedef const struct {
3693     PP_Commands_Entry      *entry;
3694     UINT16                 inSize;
3695     UINT16                 outSize;
3696     UINT16                 offsetOfTypes;
3697     UINT16                 paramOffsets[2];
3698     BYTE                   types[5];
3699 } PP_Commands_COMMAND_DESCRIPTOR_t;
3700
3701 PP_Commands_COMMAND_DESCRIPTOR_t _PP_CommandsData = {
3702     /* entry */           &TPM2_PP_Commands,
3703     /* inSize */          (UINT16) (sizeof(PP_Commands_In)),
3704     /* outSize */         0,
3705     /* offsetOfTypes */   offsetof(PP_Commands_COMMAND_DESCRIPTOR_t, types),
3706     /* offsets */         {(UINT16) (offsetof(PP_Commands_In, setList)),
3707                          (UINT16) (offsetof(PP_Commands_In, clearList))},
3708     /* types */           {TPMI_RH_PLATFORM_H_UNMARSHAL,
3709                          TPML_CC_P_UNMARSHAL,
3710                          TPML_CC_P_UNMARSHAL,
3711                          END_OF_LIST,
3712                          END_OF_LIST}
3713 };
3714
3715 #define _PP_CommandsDataAddress (&_PP_CommandsData)
3716 #else
3717 #define _PP_CommandsDataAddress 0
3718 #endif // CC_PP_Commands
3719
3720 #if CC_SetAlgorithmSet
3721
3722 #include "SetAlgorithmSet_fp.h"
3723
3724 typedef TPM_RC (SetAlgorithmSet_Entry) (
3725     SetAlgorithmSet_In      *in
3726 );
3727
3728 typedef const struct {
3729     SetAlgorithmSet_Entry   *entry;
3730     UINT16                 inSize;
3731     UINT16                 outSize;
3732     UINT16                 offsetOfTypes;
3733     UINT16                 paramOffsets[1];
3734     BYTE                   types[4];
3735 } SetAlgorithmSet_COMMAND_DESCRIPTOR_t;
3736
3737 SetAlgorithmSet_COMMAND_DESCRIPTOR_t _SetAlgorithmSetData = {
3738     /* entry */           &TPM2_SetAlgorithmSet,
3739     /* inSize */          (UINT16) (sizeof(SetAlgorithmSet_In)),
3740     /* outSize */         0,
3741     /* offsetOfTypes */   offsetof(SetAlgorithmSet_COMMAND_DESCRIPTOR_t, types),
3742     /* offsets */         {(UINT16) (offsetof(SetAlgorithmSet_In, algorithmSet))},
3743     /* types */           {TPMI_RH_PLATFORM_H_UNMARSHAL,
3744                          UINT32_P_UNMARSHAL,
3745                          END_OF_LIST,
3746                          END_OF_LIST}
3747 };
3748
3749 #define _SetAlgorithmSetDataAddress (&_SetAlgorithmSetData)
3750 #else
3751 #define _SetAlgorithmSetDataAddress 0
3752 #endif // CC_SetAlgorithmSet
3753
3754 #if CC_FieldUpgradeStart
3755
3756 #include "FieldUpgradeStart_fp.h"

```

```

3757
3758 typedef TPM_RC (FieldUpgradeStart_Entry) (
3759     FieldUpgradeStart_In      *in
3760 );
3761
3762 typedef const struct {
3763     FieldUpgradeStart_Entry    *entry;
3764     UINT16                     inSize;
3765     UINT16                     outSize;
3766     UINT16                     offsetOfTypes;
3767     UINT16                     paramOffsets[3];
3768     BYTE                       types[6];
3769 } FieldUpgradeStart_COMMAND_DESCRIPTOR_t;
3770
3771 FieldUpgradeStart_COMMAND_DESCRIPTOR_t _FieldUpgradeStartData = {
3772     /* entry */           /*&TPM2_FieldUpgradeStart,
3773     /* inSize */          /*(UINT16) (sizeof(FieldUpgradeStart_In)),
3774     /* outSize */         /*0,
3775     /* offsetOfTypes */   /*offsetof(FieldUpgradeStart_COMMAND_DESCRIPTOR_t,
types),
3776     /* offsets */         /*{(UINT16) (offsetof(FieldUpgradeStart_In, keyHandle)),
3777                          /*(UINT16) (offsetof(FieldUpgradeStart_In, fuDigest)),
3778                          /*(UINT16) (offsetof(FieldUpgradeStart_In,
manifestSignature))},
3779     /* types */           /*{TPMI_RH_PLATFORM_H_UNMARSHAL,
3780                          /*TPMI_DH_OBJECT_H_UNMARSHAL,
3781                          /*TPM2B_DIGEST_P_UNMARSHAL,
3782                          /*TPMT_SIGNATURE_P_UNMARSHAL,
3783                          /*END_OF_LIST,
3784                          /*END_OF_LIST}
3785 };
3786
3787 #define _FieldUpgradeStartDataAddress (&_FieldUpgradeStartData)
3788 #else
3789 #define _FieldUpgradeStartDataAddress 0
3790 #endif // CC_FieldUpgradeStart
3791
3792 #if CC_FieldUpgradeData
3793
3794 #include "FieldUpgradeData_fp.h"
3795
3796 typedef TPM_RC (FieldUpgradeData_Entry) (
3797     FieldUpgradeData_In      *in,
3798     FieldUpgradeData_Out     *out
3799 );
3800
3801 typedef const struct {
3802     FieldUpgradeData_Entry    *entry;
3803     UINT16                     inSize;
3804     UINT16                     outSize;
3805     UINT16                     offsetOfTypes;
3806     UINT16                     paramOffsets[1];
3807     BYTE                       types[5];
3808 } FieldUpgradeData_COMMAND_DESCRIPTOR_t;
3809
3810 FieldUpgradeData_COMMAND_DESCRIPTOR_t _FieldUpgradeDataData = {
3811     /* entry */           /*&TPM2_FieldUpgradeData,
3812     /* inSize */          /*(UINT16) (sizeof(FieldUpgradeData_In)),
3813     /* outSize */         /*(UINT16) (sizeof(FieldUpgradeData_Out)),
3814     /* offsetOfTypes */   /*offsetof(FieldUpgradeData_COMMAND_DESCRIPTOR_t, types),
3815     /* offsets */         /*{(UINT16) (offsetof(FieldUpgradeData_Out, firstDigest))},
3816     /* types */           /*{TPM2B_MAX_BUFFER_P_UNMARSHAL,
3817                          /*END_OF_LIST,
3818                          /*TPMT_HA_P_MARSHAL,
3819                          /*TPMT_HA_P_MARSHAL,
3820                          /*END_OF_LIST}

```

```

3821 };
3822
3823 #define _FieldUpgradeDataDataAddress (&_FieldUpgradeDataData)
3824 #else
3825 #define _FieldUpgradeDataDataAddress 0
3826 #endif // CC_FieldUpgradeData
3827
3828 #if CC_FirmwareRead
3829
3830 #include "FirmwareRead_fp.h"
3831
3832 typedef TPM_RC (FirmwareRead_Entry) (
3833     FirmwareRead_In      *in,
3834     FirmwareRead_Out     *out
3835 );
3836
3837 typedef const struct {
3838     FirmwareRead_Entry    *entry;
3839     UINT16                 inSize;
3840     UINT16                 outSize;
3841     UINT16                 offsetOfTypes;
3842     BYTE                   types[4];
3843 } FirmwareRead_COMMAND_DESCRIPTOR_t;
3844
3845 FirmwareRead_COMMAND_DESCRIPTOR_t _FirmwareReadData = {
3846     /* entry */           &TPM2_FirmwareRead,
3847     /* inSize */          (UINT16) (sizeof(FirmwareRead_In)),
3848     /* outSize */         (UINT16) (sizeof(FirmwareRead_Out)),
3849     /* offsetOfTypes */   offsetof(FirmwareRead_COMMAND_DESCRIPTOR_t, types),
3850     /* offsets */         // No parameter offsets;
3851     /* types */           {UINT32_P_UNMARSHAL,
3852                           END_OF_LIST,
3853                           TPM2B_MAX_BUFFER_P_MARSHAL,
3854                           END_OF_LIST};
3855 };
3856
3857 #define _FirmwareReadDataAddress (&_FirmwareReadData)
3858 #else
3859 #define _FirmwareReadDataAddress 0
3860 #endif // CC_FirmwareRead
3861
3862 #if CC_ContextSave
3863
3864 #include "ContextSave_fp.h"
3865
3866 typedef TPM_RC (ContextSave_Entry) (
3867     ContextSave_In      *in,
3868     ContextSave_Out     *out
3869 );
3870
3871 typedef const struct {
3872     ContextSave_Entry    *entry;
3873     UINT16                 inSize;
3874     UINT16                 outSize;
3875     UINT16                 offsetOfTypes;
3876     BYTE                   types[4];
3877 } ContextSave_COMMAND_DESCRIPTOR_t;
3878
3879 ContextSave_COMMAND_DESCRIPTOR_t _ContextSaveData = {
3880     /* entry */           &TPM2_ContextSave,
3881     /* inSize */          (UINT16) (sizeof(ContextSave_In)),
3882     /* outSize */         (UINT16) (sizeof(ContextSave_Out)),
3883     /* offsetOfTypes */   offsetof(ContextSave_COMMAND_DESCRIPTOR_t, types),
3884     /* offsets */         // No parameter offsets;
3885     /* types */           {TPMI_DH_CONTEXT_H_UNMARSHAL,
3886                           END_OF_LIST,

```



```

3887             TPMS_CONTEXT_P_MARSHAL,
3888             END_OF_LIST}
3889 };
3890
3891 #define _ContextSaveDataAddress (&_ContextSaveData)
3892 #else
3893 #define _ContextSaveDataAddress 0
3894 #endif // CC_ContextSave
3895
3896 #if CC_ContextLoad
3897
3898 #include "ContextLoad_fp.h"
3899
3900 typedef TPM_RC (ContextLoad_Entry) (
3901     ContextLoad_In          *in,
3902     ContextLoad_Out         *out
3903 );
3904
3905 typedef const struct {
3906     ContextLoad_Entry        *entry;
3907     UINT16                   inSize;
3908     UINT16                   outSize;
3909     UINT16                   offsetOfTypes;
3910     BYTE                     types[4];
3911 } ContextLoad_COMMAND_DESCRIPTOR_t;
3912
3913 ContextLoad_COMMAND_DESCRIPTOR_t _ContextLoadData = {
3914     /* entry          */ &TPM2_ContextLoad,
3915     /* inSize         */ (UINT16) (sizeof(ContextLoad_In)),
3916     /* outSize        */ (UINT16) (sizeof(ContextLoad_Out)),
3917     /* offsetOfTypes */ offsetof(ContextLoad_COMMAND_DESCRIPTOR_t, types),
3918     /* offsets        */ // No parameter offsets;
3919     /* types          */ {TPMS_CONTEXT_P_UNMARSHAL,
3920                          END_OF_LIST,
3921                          TPMI_DH_CONTEXT_H_MARSHAL,
3922                          END_OF_LIST}
3923 };
3924
3925 #define _ContextLoadDataAddress (&_ContextLoadData)
3926 #else
3927 #define _ContextLoadDataAddress 0
3928 #endif // CC_ContextLoad
3929
3930 #if CC_FlushContext
3931
3932 #include "FlushContext_fp.h"
3933
3934 typedef TPM_RC (FlushContext_Entry) (
3935     FlushContext_In          *in
3936 );
3937
3938 typedef const struct {
3939     FlushContext_Entry        *entry;
3940     UINT16                   inSize;
3941     UINT16                   outSize;
3942     UINT16                   offsetOfTypes;
3943     BYTE                     types[3];
3944 } FlushContext_COMMAND_DESCRIPTOR_t;
3945
3946 FlushContext_COMMAND_DESCRIPTOR_t _FlushContextData = {
3947     /* entry          */ &TPM2_FlushContext,
3948     /* inSize         */ (UINT16) (sizeof(FlushContext_In)),
3949     /* outSize        */ 0,
3950     /* offsetOfTypes */ offsetof(FlushContext_COMMAND_DESCRIPTOR_t, types),
3951     /* offsets        */ // No parameter offsets;
3952     /* types          */ {TPMI_DH_CONTEXT_P_UNMARSHAL,

```



```

3953                                     END_OF_LIST,
3954                                     END_OF_LIST}
3955 };
3956
3957 #define _FlushContextDataAddress (&_FlushContextData)
3958 #else
3959 #define _FlushContextDataAddress 0
3960 #endif // CC_FlushContext
3961
3962 #if CC_EvictControl
3963
3964 #include "EvictControl_fp.h"
3965
3966 typedef TPM_RC (EvictControl_Entry)(
3967     EvictControl_In          *in
3968 );
3969
3970 typedef const struct {
3971     EvictControl_Entry      *entry;
3972     UINT16                  inSize;
3973     UINT16                  outSize;
3974     UINT16                  offsetOfTypes;
3975     UINT16                  paramOffsets[2];
3976     BYTE                    types[5];
3977 } EvictControl_COMMAND_DESCRIPTOR_t;
3978
3979 EvictControl_COMMAND_DESCRIPTOR_t _EvictControlData = {
3980     /* entry */           &TPM2_EvictControl,
3981     /* inSize */          (UINT16)(sizeof(EvictControl_In)),
3982     /* outSize */         0,
3983     /* offsetOfTypes */   offsetof(EvictControl_COMMAND_DESCRIPTOR_t, types),
3984     /* offsets */          {(UINT16)(offsetof(EvictControl_In, objectHandle)),
3985                             (UINT16)(offsetof(EvictControl_In, persistentHandle))},
3986     /* types */            {TPMI_RH_PROVISION_H_UNMARSHAL,
3987                             TPMI_DH_OBJECT_H_UNMARSHAL,
3988                             TPMI_DH_PERSISTENT_P_UNMARSHAL,
3989                             END_OF_LIST,
3990                             END_OF_LIST}
3991 };
3992
3993 #define _EvictControlDataAddress (&_EvictControlData)
3994 #else
3995 #define _EvictControlDataAddress 0
3996 #endif // CC_EvictControl
3997
3998 #if CC_ReadClock
3999
4000 #include "ReadClock_fp.h"
4001
4002 typedef TPM_RC (ReadClock_Entry)(
4003     ReadClock_Out          *out
4004 );
4005
4006 typedef const struct {
4007     ReadClock_Entry        *entry;
4008     UINT16                  inSize;
4009     UINT16                  outSize;
4010     UINT16                  offsetOfTypes;
4011     BYTE                    types[3];
4012 } ReadClock_COMMAND_DESCRIPTOR_t;
4013
4014 ReadClock_COMMAND_DESCRIPTOR_t _ReadClockData = {
4015     /* entry */           &TPM2_ReadClock,
4016     /* inSize */          0,
4017     /* outSize */         (UINT16)(sizeof(ReadClock_Out)),
4018     /* offsetOfTypes */   offsetof(ReadClock_COMMAND_DESCRIPTOR_t, types),

```

```

4019     /* offsets      */ // No parameter offsets;
4020     /* types        */ {END_OF_LIST,
4021                        TPMS_TIME_INFO_P_MARSHAL,
4022                        END_OF_LIST}
4023 };
4024
4025 #define _ReadClockDataAddress (&_ReadClockData)
4026 #else
4027 #define _ReadClockDataAddress 0
4028 #endif // CC_ReadClock
4029
4030 #if CC_ClockSet
4031
4032 #include "ClockSet_fp.h"
4033
4034 typedef TPM_RC (ClockSet_Entry) (
4035     ClockSet_In          *in
4036 );
4037
4038 typedef const struct {
4039     ClockSet_Entry      *entry;
4040     UINT16              inSize;
4041     UINT16              outSize;
4042     UINT16              offsetOfTypes;
4043     UINT16              paramOffsets[1];
4044     BYTE                types[4];
4045 } ClockSet_COMMAND_DESCRIPTOR_t;
4046
4047 ClockSet_COMMAND_DESCRIPTOR_t _ClockSetData = {
4048     /* entry      */ &TPM2_ClockSet,
4049     /* inSize     */ (UINT16) (sizeof(ClockSet_In)),
4050     /* outSize    */ 0,
4051     /* offsetOfTypes */ offsetof(ClockSet_COMMAND_DESCRIPTOR_t, types),
4052     /* offsets     */ {(UINT16) (offsetof(ClockSet_In, newTime))},
4053     /* types      */ {TPMI_RH_PROVISION_H_UNMARSHAL,
4054                     UINT64_P_UNMARSHAL,
4055                     END_OF_LIST,
4056                     END_OF_LIST}
4057 };
4058
4059 #define _ClockSetDataAddress (&_ClockSetData)
4060 #else
4061 #define _ClockSetDataAddress 0
4062 #endif // CC_ClockSet
4063
4064 #if CC_ClockRateAdjust
4065
4066 #include "ClockRateAdjust_fp.h"
4067
4068 typedef TPM_RC (ClockRateAdjust_Entry) (
4069     ClockRateAdjust_In    *in
4070 );
4071
4072 typedef const struct {
4073     ClockRateAdjust_Entry *entry;
4074     UINT16                inSize;
4075     UINT16                outSize;
4076     UINT16                offsetOfTypes;
4077     UINT16                paramOffsets[1];
4078     BYTE                  types[4];
4079 } ClockRateAdjust_COMMAND_DESCRIPTOR_t;
4080
4081 ClockRateAdjust_COMMAND_DESCRIPTOR_t _ClockRateAdjustData = {
4082     /* entry      */ &TPM2_ClockRateAdjust,
4083     /* inSize     */ (UINT16) (sizeof(ClockRateAdjust_In)),
4084     /* outSize    */ 0,

```

```

4085     /* offsetOfTypes */      offsetof(ClockRateAdjust_COMMAND_DESCRIPTOR_t, types),
4086     /* offsets */           { (UINT16) (offsetof(ClockRateAdjust_In, rateAdjust)) },
4087     /* types */             { TPMI_RH_PROVISION_H_UNMARSHAL,
4088                             TPM_CLOCK_ADJUST_P_UNMARSHAL,
4089                             END_OF_LIST,
4090                             END_OF_LIST };
4091 };
4092
4093 #define _ClockRateAdjustDataAddress (&_ClockRateAdjustData)
4094 #else
4095 #define _ClockRateAdjustDataAddress 0
4096 #endif // CC_ClockRateAdjust
4097
4098 #if CC_GetCapability
4099
4100 #include "GetCapability_fp.h"
4101
4102 typedef TPM_RC (GetCapability_Entry) (
4103     GetCapability_In          *in,
4104     GetCapability_Out         *out
4105 );
4106
4107 typedef const struct {
4108     GetCapability_Entry      *entry;
4109     UINT16                   inSize;
4110     UINT16                   outSize;
4111     UINT16                   offsetOfTypes;
4112     UINT16                   paramOffsets[3];
4113     BYTE                     types[7];
4114 } GetCapability_COMMAND_DESCRIPTOR_t;
4115
4116 GetCapability_COMMAND_DESCRIPTOR_t _GetCapabilityData = {
4117     /* entry */              &TPM2_GetCapability,
4118     /* inSize */             (UINT16) (sizeof(GetCapability_In)),
4119     /* outSize */            (UINT16) (sizeof(GetCapability_Out)),
4120     /* offsetOfTypes */      offsetof(GetCapability_COMMAND_DESCRIPTOR_t, types),
4121     /* offsets */            { (UINT16) (offsetof(GetCapability_In, property)),
4122                             (UINT16) (offsetof(GetCapability_In, propertyCount)),
4123                             (UINT16) (offsetof(GetCapability_Out, capabilityData)) },
4124     /* types */              { TPM_CAP_P_UNMARSHAL,
4125                             UINT32_P_UNMARSHAL,
4126                             UINT32_P_UNMARSHAL,
4127                             END_OF_LIST,
4128                             TPMI_YES_NO_P_MARSHAL,
4129                             TPMS_CAPABILITY_DATA_P_MARSHAL,
4130                             END_OF_LIST };
4131 };
4132
4133 #define _GetCapabilityDataAddress (&_GetCapabilityData)
4134 #else
4135 #define _GetCapabilityDataAddress 0
4136 #endif // CC_GetCapability
4137
4138 #if CC_TestParms
4139
4140 #include "TestParms_fp.h"
4141
4142 typedef TPM_RC (TestParms_Entry) (
4143     TestParms_In             *in
4144 );
4145
4146 typedef const struct {
4147     TestParms_Entry          *entry;
4148     UINT16                   inSize;
4149     UINT16                   outSize;
4150     UINT16                   offsetOfTypes;

```

```

4151     BYTE                types[3];
4152 } TestParms_COMMAND_DESCRIPTOR_t;
4153
4154 TestParms_COMMAND_DESCRIPTOR_t _TestParmsData = {
4155     /* entry            */    &TPM2_TestParms,
4156     /* inSize           */    (UINT16) (sizeof(TestParms_In)),
4157     /* outSize          */    0,
4158     /* offsetOfTypes    */    offsetof(TestParms_COMMAND_DESCRIPTOR_t, types),
4159     /* offsets          */    // No parameter offsets;
4160     /* types            */    {TPMT_PUBLIC_PARMS_P_UNMARSHAL,
4161                             END_OF_LIST,
4162                             END_OF_LIST};
4163 };
4164
4165 #define _TestParmsDataAddress (&_TestParmsData)
4166 #else
4167 #define _TestParmsDataAddress 0
4168 #endif // CC_TestParms
4169
4170 #if CC_NV_DefineSpace
4171
4172 #include "NV_DefineSpace_fp.h"
4173
4174 typedef TPM_RC (NV_DefineSpace_Entry) (
4175     NV_DefineSpace_In          *in
4176 );
4177
4178 typedef const struct {
4179     NV_DefineSpace_Entry      *entry;
4180     UINT16                    inSize;
4181     UINT16                    outSize;
4182     UINT16                    offsetOfTypes;
4183     UINT16                    paramOffsets[2];
4184     BYTE                      types[5];
4185 } NV_DefineSpace_COMMAND_DESCRIPTOR_t;
4186
4187 NV_DefineSpace_COMMAND_DESCRIPTOR_t _NV_DefineSpaceData = {
4188     /* entry            */    &TPM2_NV_DefineSpace,
4189     /* inSize           */    (UINT16) (sizeof(NV_DefineSpace_In)),
4190     /* outSize          */    0,
4191     /* offsetOfTypes    */    offsetof(NV_DefineSpace_COMMAND_DESCRIPTOR_t, types),
4192     /* offsets          */    {(UINT16) (offsetof(NV_DefineSpace_In, auth)),
4193                             (UINT16) (offsetof(NV_DefineSpace_In, publicInfo))},
4194     /* types            */    {TPMI_RH_PROVISION_H_UNMARSHAL,
4195                             TPM2B_AUTH_P_UNMARSHAL,
4196                             TPM2B_NV_PUBLIC_P_UNMARSHAL,
4197                             END_OF_LIST,
4198                             END_OF_LIST};
4199 };
4200
4201 #define _NV_DefineSpaceDataAddress (&_NV_DefineSpaceData)
4202 #else
4203 #define _NV_DefineSpaceDataAddress 0
4204 #endif // CC_NV_DefineSpace
4205
4206 #if CC_NV_UndefineSpace
4207
4208 #include "NV_UndefineSpace_fp.h"
4209
4210 typedef TPM_RC (NV_UndefineSpace_Entry) (
4211     NV_UndefineSpace_In        *in
4212 );
4213
4214 typedef const struct {
4215     NV_UndefineSpace_Entry      *entry;
4216     UINT16                    inSize;

```

```

4217     UINT16                outSize;
4218     UINT16                offsetOfTypes;
4219     UINT16                paramOffsets[1];
4220     BYTE                  types[4];
4221 } NV_UndefineSpace_COMMAND_DESCRIPTOR_t;
4222
4223 NV_UndefineSpace_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceData = {
4224     /* entry                */      &TPM2_NV_UndefineSpace,
4225     /* inSize              */      (UINT16) (sizeof(NV_UndefineSpace_In)),
4226     /* outSize             */      0,
4227     /* offsetOfTypes       */      offsetof(NV_UndefineSpace_COMMAND_DESCRIPTOR_t, types),
4228     /* offsets             */      {(UINT16) (offsetof(NV_UndefineSpace_In, nvIndex))},
4229     /* types               */      {TPMI_RH_PROVISION_H_UNMARSHAL,
4230                                     TPMI_RH_NV_INDEX_H_UNMARSHAL,
4231                                     END_OF_LIST,
4232                                     END_OF_LIST}
4233 };
4234
4235 #define _NV_UndefineSpaceDataAddress (&_NV_UndefineSpaceData)
4236 #else
4237 #define _NV_UndefineSpaceDataAddress 0
4238 #endif // CC_NV_UndefineSpace
4239
4240 #if CC_NV_UndefineSpaceSpecial
4241
4242 #include "NV_UndefineSpaceSpecial_fp.h"
4243
4244 typedef TPM_RC (NV_UndefineSpaceSpecial_Entry) (
4245     NV_UndefineSpaceSpecial_In      *in
4246 );
4247
4248 typedef const struct {
4249     NV_UndefineSpaceSpecial_Entry  *entry;
4250     UINT16                        inSize;
4251     UINT16                        outSize;
4252     UINT16                        offsetOfTypes;
4253     UINT16                        paramOffsets[1];
4254     BYTE                          types[4];
4255 } NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t;
4256
4257 NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t _NV_UndefineSpaceSpecialData = {
4258     /* entry                */      &TPM2_NV_UndefineSpaceSpecial,
4259     /* inSize              */      (UINT16) (sizeof(NV_UndefineSpaceSpecial_In)),
4260     /* outSize             */      0,
4261     /* offsetOfTypes       */      offsetof(NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t, types),
4262     /* offsets             */      {(UINT16) (offsetof(NV_UndefineSpaceSpecial_In,
4263 platform))},
4264     /* types               */      {TPMI_RH_NV_INDEX_H_UNMARSHAL,
4265                                     TPMI_RH_PLATFORM_H_UNMARSHAL,
4266                                     END_OF_LIST,
4267                                     END_OF_LIST}
4268 };
4269
4270 #define _NV_UndefineSpaceSpecialDataAddress (&_NV_UndefineSpaceSpecialData)
4271 #else
4272 #define _NV_UndefineSpaceSpecialDataAddress 0
4273 #endif // CC_NV_UndefineSpaceSpecial
4274
4275 #if CC_NV_ReadPublic
4276
4277 #include "NV_ReadPublic_fp.h"
4278
4279 typedef TPM_RC (NV_ReadPublic_Entry) (
4280     NV_ReadPublic_In              *in,
4281     NV_ReadPublic_Out             *out

```

```

4281 );
4282
4283 typedef const struct {
4284     NV_ReadPublic_Entry      *entry;
4285     UINT16                   inSize;
4286     UINT16                   outSize;
4287     UINT16                   offsetOfTypes;
4288     UINT16                   paramOffsets[1];
4289     BYTE                     types[5];
4290 } NV_ReadPublic_COMMAND_DESCRIPTOR_t;
4291
4292 NV_ReadPublic_COMMAND_DESCRIPTOR_t _NV_ReadPublicData = {
4293     /* entry */          &TPM2_NV_ReadPublic,
4294     /* inSize */         (UINT16)(sizeof(NV_ReadPublic_In)),
4295     /* outSize */        (UINT16)(sizeof(NV_ReadPublic_Out)),
4296     /* offsetOfTypes */  offsetof(NV_ReadPublic_COMMAND_DESCRIPTOR_t, types),
4297     /* offsets */        {(UINT16)(offsetof(NV_ReadPublic_Out, nvName))},
4298     /* types */          {TPMI_RH_NV_INDEX_H_UNMARSHAL,
4299                          END_OF_LIST,
4300                          TPM2B_NV_PUBLIC_P_MARSHAL,
4301                          TPM2B_NAME_P_MARSHAL,
4302                          END_OF_LIST};
4303 };
4304
4305 #define _NV_ReadPublicDataAddress (&_NV_ReadPublicData)
4306 #else
4307 #define _NV_ReadPublicDataAddress 0
4308 #endif // CC_NV_ReadPublic
4309
4310 #if CC_NV_Write
4311
4312 #include "NV_Write_fp.h"
4313
4314 typedef TPM_RC (NV_Write_Entry)(
4315     NV_Write_In      *in
4316 );
4317
4318 typedef const struct {
4319     NV_Write_Entry      *entry;
4320     UINT16               inSize;
4321     UINT16               outSize;
4322     UINT16               offsetOfTypes;
4323     UINT16               paramOffsets[3];
4324     BYTE                 types[6];
4325 } NV_Write_COMMAND_DESCRIPTOR_t;
4326
4327 NV_Write_COMMAND_DESCRIPTOR_t _NV_WriteData = {
4328     /* entry */          &TPM2_NV_Write,
4329     /* inSize */         (UINT16)(sizeof(NV_Write_In)),
4330     /* outSize */        0,
4331     /* offsetOfTypes */  offsetof(NV_Write_COMMAND_DESCRIPTOR_t, types),
4332     /* offsets */        {(UINT16)(offsetof(NV_Write_In, nvIndex)),
4333                          (UINT16)(offsetof(NV_Write_In, data)),
4334                          (UINT16)(offsetof(NV_Write_In, offset))},
4335     /* types */          {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4336                          TPMI_RH_NV_INDEX_H_UNMARSHAL,
4337                          TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
4338                          UINT16_P_UNMARSHAL,
4339                          END_OF_LIST,
4340                          END_OF_LIST};
4341 };
4342
4343 #define _NV_WriteDataAddress (&_NV_WriteData)
4344 #else
4345 #define _NV_WriteDataAddress 0
4346 #endif // CC_NV_Write

```



```

4347
4348 #if CC_NV_Increment
4349
4350 #include "NV_Increment_fp.h"
4351
4352 typedef TPM_RC (NV_Increment_Entry) (
4353     NV_Increment_In      *in
4354 );
4355
4356 typedef const struct {
4357     NV_Increment_Entry      *entry;
4358     UINT16                  inSize;
4359     UINT16                  outSize;
4360     UINT16                  offsetOfTypes;
4361     UINT16                  paramOffsets[1];
4362     BYTE                    types[4];
4363 } NV_Increment_COMMAND_DESCRIPTOR_t;
4364
4365 NV_Increment_COMMAND_DESCRIPTOR_t _NV_IncrementData = {
4366     /* entry */          &TPM2_NV_Increment,
4367     /* inSize */         (UINT16) (sizeof(NV_Increment_In)),
4368     /* outSize */        0,
4369     /* offsetOfTypes */  offsetof(NV_Increment_COMMAND_DESCRIPTOR_t, types),
4370     /* offsets */        {(UINT16) (offsetof(NV_Increment_In, nvIndex))},
4371     /* types */          {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4372                          TPMI_RH_NV_INDEX_H_UNMARSHAL,
4373                          END_OF_LIST,
4374                          END_OF_LIST};
4375 };
4376
4377 #define _NV_IncrementDataAddress (&_NV_IncrementData)
4378 #else
4379 #define _NV_IncrementDataAddress 0
4380 #endif // CC_NV_Increment
4381
4382 #if CC_NV_Extend
4383
4384 #include "NV_Extend_fp.h"
4385
4386 typedef TPM_RC (NV_Extend_Entry) (
4387     NV_Extend_In      *in
4388 );
4389
4390 typedef const struct {
4391     NV_Extend_Entry      *entry;
4392     UINT16                  inSize;
4393     UINT16                  outSize;
4394     UINT16                  offsetOfTypes;
4395     UINT16                  paramOffsets[2];
4396     BYTE                    types[5];
4397 } NV_Extend_COMMAND_DESCRIPTOR_t;
4398
4399 NV_Extend_COMMAND_DESCRIPTOR_t _NV_ExtendData = {
4400     /* entry */          &TPM2_NV_Extend,
4401     /* inSize */         (UINT16) (sizeof(NV_Extend_In)),
4402     /* outSize */        0,
4403     /* offsetOfTypes */  offsetof(NV_Extend_COMMAND_DESCRIPTOR_t, types),
4404     /* offsets */        {(UINT16) (offsetof(NV_Extend_In, nvIndex)),
4405                          (UINT16) (offsetof(NV_Extend_In, data))},
4406     /* types */          {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4407                          TPMI_RH_NV_INDEX_H_UNMARSHAL,
4408                          TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
4409                          END_OF_LIST,
4410                          END_OF_LIST};
4411 };
4412

```



```

4413 #define _NV_ExtendDataAddress (&_NV_ExtendData)
4414 #else
4415 #define _NV_ExtendDataAddress 0
4416 #endif // CC_NV_Extend
4417
4418 #if CC_NV_SetBits
4419
4420 #include "NV_SetBits_fp.h"
4421
4422 typedef TPM_RC (NV_SetBits_Entry) (
4423     NV_SetBits_In          *in
4424 );
4425
4426 typedef const struct {
4427     NV_SetBits_Entry      *entry;
4428     UINT16                inSize;
4429     UINT16                outSize;
4430     UINT16                offsetOfTypes;
4431     UINT16                paramOffsets[2];
4432     BYTE                  types[5];
4433 } NV_SetBits_COMMAND_DESCRIPTOR_t;
4434
4435 NV_SetBits_COMMAND_DESCRIPTOR_t _NV_SetBitsData = {
4436     /* entry */          &TPM2_NV_SetBits,
4437     /* inSize */         (UINT16) (sizeof(NV_SetBits_In)),
4438     /* outSize */        0,
4439     /* offsetOfTypes */  offsetof(NV_SetBits_COMMAND_DESCRIPTOR_t, types),
4440     /* offsets */        {(UINT16) (offsetof(NV_SetBits_In, nvIndex)),
4441                          (UINT16) (offsetof(NV_SetBits_In, bits))},
4442     /* types */          {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4443                          TPMI_RH_NV_INDEX_H_UNMARSHAL,
4444                          UINT64_P_UNMARSHAL,
4445                          END_OF_LIST,
4446                          END_OF_LIST}
4447 };
4448
4449 #define _NV_SetBitsDataAddress (&_NV_SetBitsData)
4450 #else
4451 #define _NV_SetBitsDataAddress 0
4452 #endif // CC_NV_SetBits
4453
4454 #if CC_NV_WriteLock
4455
4456 #include "NV_WriteLock_fp.h"
4457
4458 typedef TPM_RC (NV_WriteLock_Entry) (
4459     NV_WriteLock_In        *in
4460 );
4461
4462 typedef const struct {
4463     NV_WriteLock_Entry      *entry;
4464     UINT16                inSize;
4465     UINT16                outSize;
4466     UINT16                offsetOfTypes;
4467     UINT16                paramOffsets[1];
4468     BYTE                  types[4];
4469 } NV_WriteLock_COMMAND_DESCRIPTOR_t;
4470
4471 NV_WriteLock_COMMAND_DESCRIPTOR_t _NV_WriteLockData = {
4472     /* entry */          &TPM2_NV_WriteLock,
4473     /* inSize */         (UINT16) (sizeof(NV_WriteLock_In)),
4474     /* outSize */        0,
4475     /* offsetOfTypes */  offsetof(NV_WriteLock_COMMAND_DESCRIPTOR_t, types),
4476     /* offsets */        {(UINT16) (offsetof(NV_WriteLock_In, nvIndex))},
4477     /* types */          {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4478                          TPMI_RH_NV_INDEX_H_UNMARSHAL,

```

```

4479             END_OF_LIST,
4480             END_OF_LIST}
4481 };
4482
4483 #define _NV_WriteLockDataAddress (&_NV_WriteLockData)
4484 #else
4485 #define _NV_WriteLockDataAddress 0
4486 #endif // CC_NV_WriteLock
4487
4488 #if CC_NV_GlobalWriteLock
4489
4490 #include "NV_GlobalWriteLock_fp.h"
4491
4492 typedef TPM_RC (NV_GlobalWriteLock_Entry)(
4493     NV_GlobalWriteLock_In      *in
4494 );
4495
4496 typedef const struct {
4497     NV_GlobalWriteLock_Entry    *entry;
4498     UINT16                      inSize;
4499     UINT16                      outSize;
4500     UINT16                      offsetOfTypes;
4501     BYTE                        types[3];
4502 } NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t;
4503
4504 NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t _NV_GlobalWriteLockData = {
4505     /* entry */                &TPM2_NV_GlobalWriteLock,
4506     /* inSize */               (UINT16)(sizeof(NV_GlobalWriteLock_In)),
4507     /* outSize */              0,
4508     /* offsetOfTypes */        offsetof(NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t,
4509     types),
4510     /* offsets */              // No parameter offsets;
4511     /* types */                {TPMI_RH_PROVISION_H_UNMARSHAL,
4512                                END_OF_LIST,
4513                                END_OF_LIST}
4514 };
4515
4516 #define _NV_GlobalWriteLockDataAddress (&_NV_GlobalWriteLockData)
4517 #else
4518 #define _NV_GlobalWriteLockDataAddress 0
4519 #endif // CC_NV_GlobalWriteLock
4520
4521 #if CC_NV_Read
4522
4523 #include "NV_Read_fp.h"
4524
4525 typedef TPM_RC (NV_Read_Entry)(
4526     NV_Read_In      *in,
4527     NV_Read_Out     *out
4528 );
4529
4530 typedef const struct {
4531     NV_Read_Entry    *entry;
4532     UINT16            inSize;
4533     UINT16            outSize;
4534     UINT16            offsetOfTypes;
4535     UINT16            paramOffsets[3];
4536     BYTE              types[7];
4537 } NV_Read_COMMAND_DESCRIPTOR_t;
4538
4539 NV_Read_COMMAND_DESCRIPTOR_t _NV_ReadData = {
4540     /* entry */                &TPM2_NV_Read,
4541     /* inSize */               (UINT16)(sizeof(NV_Read_In)),
4542     /* outSize */              (UINT16)(sizeof(NV_Read_Out)),
4543     /* offsetOfTypes */        offsetof(NV_Read_COMMAND_DESCRIPTOR_t, types),
4544     /* offsets */              {(UINT16)(offsetof(NV_Read_In, nvIndex)),

```

```

4544             (UINT16) (offsetof(NV_Read_In, size)),
4545             (UINT16) (offsetof(NV_Read_In, offset))),
4546     /* types */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4547                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
4548                 UINT16_P_UNMARSHAL,
4549                 UINT16_P_UNMARSHAL,
4550                 END_OF_LIST,
4551                 TPM2B_MAX_NV_BUFFER_P_MARSHAL,
4552                 END_OF_LIST}
4553 };
4554
4555 #define _NV_ReadDataAddress (&_NV_ReadData)
4556 #else
4557 #define _NV_ReadDataAddress 0
4558 #endif // CC_NV_Read
4559
4560 #if CC_NV_ReadLock
4561 #include "NV_ReadLock_fp.h"
4562
4563 typedef TPM_RC (NV_ReadLock_Entry) (
4564     NV_ReadLock_In *in
4565 );
4566
4567 typedef const struct {
4568     NV_ReadLock_Entry *entry;
4569     UINT16 inSize;
4570     UINT16 outSize;
4571     UINT16 offsetOfTypes;
4572     UINT16 paramOffsets[1];
4573     BYTE types[4];
4574 } NV_ReadLock_COMMAND_DESCRIPTOR_t;
4575
4576 NV_ReadLock_COMMAND_DESCRIPTOR_t NV_ReadLockData = {
4577     /* entry */ &TPM2_NV_ReadLock,
4578     /* inSize */ (UINT16) (sizeof(NV_ReadLock_In)),
4579     /* outSize */ 0,
4580     /* offsetOfTypes */ offsetof(NV_ReadLock_COMMAND_DESCRIPTOR_t, types),
4581     /* offsets */ {(UINT16) (offsetof(NV_ReadLock_In, nvIndex))},
4582     /* types */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,
4583                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
4584                 END_OF_LIST,
4585                 END_OF_LIST}
4586 };
4587
4588 #define _NV_ReadLockDataAddress (&_NV_ReadLockData)
4589 #else
4590 #define _NV_ReadLockDataAddress 0
4591 #endif // CC_NV_ReadLock
4592
4593 #if CC_NV_ChangeAuth
4594 #include "NV_ChangeAuth_fp.h"
4595
4596 typedef TPM_RC (NV_ChangeAuth_Entry) (
4597     NV_ChangeAuth_In *in
4598 );
4599
4600 typedef const struct {
4601     NV_ChangeAuth_Entry *entry;
4602     UINT16 inSize;
4603     UINT16 outSize;
4604     UINT16 offsetOfTypes;
4605     UINT16 paramOffsets[1];
4606     BYTE types[4];
4607 } NV_ChangeAuth_COMMAND_DESCRIPTOR_t;

```

```

4610
4611 NV_ChangeAuth_COMMAND_DESCRIPTOR_t _NV_ChangeAuthData = {
4612     /* entry */ &TPM2_NV_ChangeAuth,
4613     /* inSize */ (UINT16) (sizeof(NV_ChangeAuth_In)),
4614     /* outSize */ 0,
4615     /* offsetOfTypes */ offsetof(NV_ChangeAuth_COMMAND_DESCRIPTOR_t, types),
4616     /* offsets */ {(UINT16) (offsetof(NV_ChangeAuth_In, newAuth))},
4617     /* types */ {TPMI_RH_NV_INDEX_H_UNMARSHAL,
4618                 TPM2B_AUTH_P_UNMARSHAL,
4619                 END_OF_LIST,
4620                 END_OF_LIST};
4621 };
4622
4623 #define _NV_ChangeAuthDataAddress (&_NV_ChangeAuthData)
4624 #else
4625 #define _NV_ChangeAuthDataAddress 0
4626 #endif // CC_NV_ChangeAuth
4627
4628 #if CC_NV_Certify
4629
4630 #include "NV_Certify_fp.h"
4631
4632 typedef TPM_RC (NV_Certify_Entry)(
4633     NV_Certify_In *in,
4634     NV_Certify_Out *out
4635 );
4636
4637 typedef const struct {
4638     NV_Certify_Entry *entry;
4639     UINT16 inSize;
4640     UINT16 outSize;
4641     UINT16 offsetOfTypes;
4642     UINT16 paramOffsets[7];
4643     BYTE types[11];
4644 } NV_Certify_COMMAND_DESCRIPTOR_t;
4645
4646 NV_Certify_COMMAND_DESCRIPTOR_t _NV_CertifyData = {
4647     /* entry */ &TPM2_NV_Certify,
4648     /* inSize */ (UINT16) (sizeof(NV_Certify_In)),
4649     /* outSize */ (UINT16) (sizeof(NV_Certify_Out)),
4650     /* offsetOfTypes */ offsetof(NV_Certify_COMMAND_DESCRIPTOR_t, types),
4651     /* offsets */ {(UINT16) (offsetof(NV_Certify_In, authHandle)),
4652                 (UINT16) (offsetof(NV_Certify_In, nvIndex)),
4653                 (UINT16) (offsetof(NV_Certify_In, qualifyingData)),
4654                 (UINT16) (offsetof(NV_Certify_In, inScheme)),
4655                 (UINT16) (offsetof(NV_Certify_In, size)),
4656                 (UINT16) (offsetof(NV_Certify_In, offset)),
4657                 (UINT16) (offsetof(NV_Certify_Out, signature))},
4658     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
4659                 TPMI_RH_NV_AUTH_H_UNMARSHAL,
4660                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
4661                 TPM2B_DATA_P_UNMARSHAL,
4662                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
4663                 UINT16_P_UNMARSHAL,
4664                 UINT16_P_UNMARSHAL,
4665                 END_OF_LIST,
4666                 TPM2B_ATTEST_P_MARSHAL,
4667                 TPMT_SIGNATURE_P_MARSHAL,
4668                 END_OF_LIST};
4669 };
4670
4671 #define _NV_CertifyDataAddress (&_NV_CertifyData)
4672 #else
4673 #define _NV_CertifyDataAddress 0
4674 #endif // CC_NV_Certify
4675

```

```

4676 #if CC_AC_GetCapability
4677
4678 #include "AC_GetCapability_fp.h"
4679
4680 typedef TPM_RC (AC_GetCapability_Entry) (
4681     AC_GetCapability_In      *in,
4682     AC_GetCapability_Out     *out
4683 );
4684
4685 typedef const struct {
4686     AC_GetCapability_Entry  *entry;
4687     UINT16                   inSize;
4688     UINT16                   outSize;
4689     UINT16                   offsetOfTypes;
4690     UINT16                   paramOffsets[3];
4691     BYTE                     types[7];
4692 } AC_GetCapability_COMMAND_DESCRIPTOR_t;
4693
4694 AC_GetCapability_COMMAND_DESCRIPTOR_t AC_GetCapabilityData = {
4695     /* entry */           &TPM2_AC_GetCapability,
4696     /* inSize */          (UINT16) (sizeof(AC_GetCapability_In)),
4697     /* outSize */         (UINT16) (sizeof(AC_GetCapability_Out)),
4698     /* offsetOfTypes */   offsetof(AC_GetCapability_COMMAND_DESCRIPTOR_t, types),
4699     /* offsets */         {(UINT16) (offsetof(AC_GetCapability_In, capability)),
4700                          (UINT16) (offsetof(AC_GetCapability_In, count)),
4701                          (UINT16) (offsetof(AC_GetCapability_Out,
4702 capabilitiesData)))},
4703     /* types */          {TPMI_RH_AC_H_UNMARSHAL,
4704                          TPM_AT_P_UNMARSHAL,
4705                          UINT32_P_UNMARSHAL,
4706                          END_OF_LIST,
4707                          TPMI_YES_NO_P_MARSHAL,
4708                          TPML_AC_CAPABILITIES_P_MARSHAL,
4709                          END_OF_LIST};
4710 };
4711 #define _AC_GetCapabilityDataAddress (&AC_GetCapabilityData)
4712 #else
4713 #define _AC_GetCapabilityDataAddress 0
4714 #endif // CC_AC_GetCapability
4715
4716 #if CC_AC_Send
4717
4718 #include "AC_Send_fp.h"
4719
4720 typedef TPM_RC (AC_Send_Entry) (
4721     AC_Send_In      *in,
4722     AC_Send_Out     *out
4723 );
4724
4725 typedef const struct {
4726     AC_Send_Entry  *entry;
4727     UINT16          inSize;
4728     UINT16          outSize;
4729     UINT16          offsetOfTypes;
4730     UINT16          paramOffsets[3];
4731     BYTE            types[7];
4732 } AC_Send_COMMAND_DESCRIPTOR_t;
4733
4734 AC_Send_COMMAND_DESCRIPTOR_t AC_SendData = {
4735     /* entry */           &TPM2_AC_Send,
4736     /* inSize */          (UINT16) (sizeof(AC_Send_In)),
4737     /* outSize */         (UINT16) (sizeof(AC_Send_Out)),
4738     /* offsetOfTypes */   offsetof(AC_Send_COMMAND_DESCRIPTOR_t, types),
4739     /* offsets */         {(UINT16) (offsetof(AC_Send_In, authHandle)),
4740                          (UINT16) (offsetof(AC_Send_In, ac))},

```

```

4741                                     (UINT16) (offsetof(AC_Send_In, acDataIn))),
4742     /* types */                      {TPMI_DH_OBJECT_H_UNMARSHAL,
4743                                     TPMI_RH_NV_AUTH_H_UNMARSHAL,
4744                                     TPMI_RH_AC_H_UNMARSHAL,
4745                                     TPM2B_MAX_BUFFER_P_UNMARSHAL,
4746                                     END_OF_LIST,
4747                                     TPMS_AC_OUTPUT_P_MARSHAL,
4748                                     END_OF_LIST}
4749 };
4750
4751 #define _AC_SendDataAddress (&_AC_SendData)
4752 #else
4753 #define _AC_SendDataAddress 0
4754 #endif // CC_AC_Send
4755
4756 #if CC_Policy_AC_SendSelect
4757
4758 #include "Policy_AC_SendSelect_fp.h"
4759
4760 typedef TPM_RC (Policy_AC_SendSelect_Entry) (
4761     Policy_AC_SendSelect_In *in
4762 );
4763
4764 typedef const struct {
4765     Policy_AC_SendSelect_Entry *entry;
4766     UINT16 inSize;
4767     UINT16 outSize;
4768     UINT16 offsetOfTypes;
4769     UINT16 paramOffsets[4];
4770     BYTE types[7];
4771 } Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t;
4772
4773 Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t _Policy_AC_SendSelectData = {
4774     /* entry */ &TPM2_Policy_AC_SendSelect,
4775     /* inSize */ (UINT16) (sizeof(Policy_AC_SendSelect_In)),
4776     /* outSize */ 0,
4777     /* offsetOfTypes */ offsetof(Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t,
types),
4778     /* offsets */ { (UINT16) (offsetof(Policy_AC_SendSelect_In,
objectName)),
4779
(UINT16) (offsetof(Policy_AC_SendSelect_In,
authHandleName)),
4780
(UINT16) (offsetof(Policy_AC_SendSelect_In, acName)),
4781
(UINT16) (offsetof(Policy_AC_SendSelect_In,
includeObject))},
4782     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
4783                 TPM2B_NAME_P_UNMARSHAL,
4784                 TPM2B_NAME_P_UNMARSHAL,
4785                 TPM2B_NAME_P_UNMARSHAL,
4786                 TPMI_YES_NO_P_UNMARSHAL,
4787                 END_OF_LIST,
4788                 END_OF_LIST}
4789 };
4790
4791 #define _Policy_AC_SendSelectDataAddress (&_Policy_AC_SendSelectData)
4792 #else
4793 #define _Policy_AC_SendSelectDataAddress 0
4794 #endif // CC_Policy_AC_SendSelect
4795
4796 #if CC_ACT_SetTimeout
4797
4798 #include "ACT_SetTimeout_fp.h"
4799
4800 typedef TPM_RC (ACT_SetTimeout_Entry) (
4801     ACT_SetTimeout_In *in
4802 );

```



```

4803
4804 typedef const struct {
4805     ACT_SetTimeout_Entry    *entry;
4806     UINT16                  inSize;
4807     UINT16                  outSize;
4808     UINT16                  offsetOfTypes;
4809     UINT16                  paramOffsets[1];
4810     BYTE                    types[4];
4811 } ACT_SetTimeout_COMMAND_DESCRIPTOR_t;
4812
4813 ACT_SetTimeout_COMMAND_DESCRIPTOR_t _ACT_SetTimeoutData = {
4814     /* entry */          &TPM2_ACT_SetTimeout,
4815     /* inSize */         (UINT16) (sizeof(ACT_SetTimeout_In)),
4816     /* outSize */        0,
4817     /* offsetOfTypes */  offsetof(ACT_SetTimeout_COMMAND_DESCRIPTOR_t, types),
4818     /* offsets */        {(UINT16) (offsetof(ACT_SetTimeout_In, startTimeout))},
4819     /* types */          {TPMI_RH_ACT_H_UNMARSHAL,
4820                          UINT32_P_UNMARSHAL,
4821                          END_OF_LIST,
4822                          END_OF_LIST};
4823 };
4824
4825 #define _ACT_SetTimeoutDataAddress (&_ACT_SetTimeoutData)
4826 #else
4827 #define _ACT_SetTimeoutDataAddress 0
4828 #endif // CC_ACT_SetTimeout
4829
4830 #if CC_Vendor_TCG_Test
4831
4832 #include "Vendor_TCG_Test_fp.h"
4833
4834 typedef TPM_RC (Vendor_TCG_Test_Entry) (
4835     Vendor_TCG_Test_In    *in,
4836     Vendor_TCG_Test_Out   *out
4837 );
4838
4839 typedef const struct {
4840     Vendor_TCG_Test_Entry    *entry;
4841     UINT16                  inSize;
4842     UINT16                  outSize;
4843     UINT16                  offsetOfTypes;
4844     BYTE                    types[4];
4845 } Vendor_TCG_Test_COMMAND_DESCRIPTOR_t;
4846
4847 Vendor_TCG_Test_COMMAND_DESCRIPTOR_t _Vendor_TCG_TestData = {
4848     /* entry */          &TPM2_Vendor_TCG_Test,
4849     /* inSize */         (UINT16) (sizeof(Vendor_TCG_Test_In)),
4850     /* outSize */        (UINT16) (sizeof(Vendor_TCG_Test_Out)),
4851     /* offsetOfTypes */  offsetof(Vendor_TCG_Test_COMMAND_DESCRIPTOR_t, types),
4852     /* offsets */        // No parameter offsets;
4853     /* types */          {TPM2B_DATA_P_UNMARSHAL,
4854                          END_OF_LIST,
4855                          TPM2B_DATA_P_MARSHAL,
4856                          END_OF_LIST};
4857 };
4858
4859 #define _Vendor_TCG_TestDataAddress (&_Vendor_TCG_TestData)
4860 #else
4861 #define _Vendor_TCG_TestDataAddress 0
4862 #endif // CC_Vendor_TCG_Test
4863
4864 COMMAND_DESCRIPTOR_t *s_CommandDataArray[] = {
4865     #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
4866         (COMMAND_DESCRIPTOR_t *) _NV_UndefineSpaceSpecialDataAddress,
4867     #endif // CC_NV_UndefineSpaceSpecial
4868     #if (PAD_LIST || CC_EvictControl)

```



```

4869         (COMMAND_DESCRIPTOR_t *)_EvictControlDataAddress,
4870 #endif // CC_EvictControl
4871 #if (PAD_LIST || CC_HierarchyControl)
4872         (COMMAND_DESCRIPTOR_t *)_HierarchyControlDataAddress,
4873 #endif // CC_HierarchyControl
4874 #if (PAD_LIST || CC_NV_UndefineSpace)
4875         (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceDataAddress,
4876 #endif // CC_NV_UndefineSpace
4877 #if (PAD_LIST)
4878         (COMMAND_DESCRIPTOR_t *)0,
4879 #endif //
4880 #if (PAD_LIST || CC_ChangeEPS)
4881         (COMMAND_DESCRIPTOR_t *)_ChangeEPSDataAddress,
4882 #endif // CC_ChangeEPS
4883 #if (PAD_LIST || CC_ChangePPS)
4884         (COMMAND_DESCRIPTOR_t *)_ChangePPSDataAddress,
4885 #endif // CC_ChangePPS
4886 #if (PAD_LIST || CC_Clear)
4887         (COMMAND_DESCRIPTOR_t *)_ClearDataAddress,
4888 #endif // CC_Clear
4889 #if (PAD_LIST || CC_ClearControl)
4890         (COMMAND_DESCRIPTOR_t *)_ClearControlDataAddress,
4891 #endif // CC_ClearControl
4892 #if (PAD_LIST || CC_ClockSet)
4893         (COMMAND_DESCRIPTOR_t *)_ClockSetDataAddress,
4894 #endif // CC_ClockSet
4895 #if (PAD_LIST || CC_HierarchyChangeAuth)
4896         (COMMAND_DESCRIPTOR_t *)_HierarchyChangeAuthDataAddress,
4897 #endif // CC_HierarchyChangeAuth
4898 #if (PAD_LIST || CC_NV_DefineSpace)
4899         (COMMAND_DESCRIPTOR_t *)_NV_DefineSpaceDataAddress,
4900 #endif // CC_NV_DefineSpace
4901 #if (PAD_LIST || CC_PCR_Allocate)
4902         (COMMAND_DESCRIPTOR_t *)_PCR_AllocateDataAddress,
4903 #endif // CC_PCR_Allocate
4904 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
4905         (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthPolicyDataAddress,
4906 #endif // CC_PCR_SetAuthPolicy
4907 #if (PAD_LIST || CC_PP_Commands)
4908         (COMMAND_DESCRIPTOR_t *)_PP_CommandsDataAddress,
4909 #endif // CC_PP_Commands
4910 #if (PAD_LIST || CC_SetPrimaryPolicy)
4911         (COMMAND_DESCRIPTOR_t *)_SetPrimaryPolicyDataAddress,
4912 #endif // CC_SetPrimaryPolicy
4913 #if (PAD_LIST || CC_FieldUpgradeStart)
4914         (COMMAND_DESCRIPTOR_t *)_FieldUpgradeStartDataAddress,
4915 #endif // CC_FieldUpgradeStart
4916 #if (PAD_LIST || CC_ClockRateAdjust)
4917         (COMMAND_DESCRIPTOR_t *)_ClockRateAdjustDataAddress,
4918 #endif // CC_ClockRateAdjust
4919 #if (PAD_LIST || CC_CreatePrimary)
4920         (COMMAND_DESCRIPTOR_t *)_CreatePrimaryDataAddress,
4921 #endif // CC_CreatePrimary
4922 #if (PAD_LIST || CC_NV_GlobalWriteLock)
4923         (COMMAND_DESCRIPTOR_t *)_NV_GlobalWriteLockDataAddress,
4924 #endif // CC_NV_GlobalWriteLock
4925 #if (PAD_LIST || CC_GetCommandAuditDigest)
4926         (COMMAND_DESCRIPTOR_t *)_GetCommandAuditDigestDataAddress,
4927 #endif // CC_GetCommandAuditDigest
4928 #if (PAD_LIST || CC_NV_Increment)
4929         (COMMAND_DESCRIPTOR_t *)_NV_IncrementDataAddress,
4930 #endif // CC_NV_Increment
4931 #if (PAD_LIST || CC_NV_SetBits)
4932         (COMMAND_DESCRIPTOR_t *)_NV_SetBitsDataAddress,
4933 #endif // CC_NV_SetBits
4934 #if (PAD_LIST || CC_NV_Extend)

```

```

4935         (COMMAND_DESCRIPTOR_t *)_NV_ExtendDataAddress,
4936 #endif // CC_NV_Extend
4937 #if (PAD_LIST || CC_NV_Write)
4938         (COMMAND_DESCRIPTOR_t *)_NV_WriteDataAddress,
4939 #endif // CC_NV_Write
4940 #if (PAD_LIST || CC_NV_WriteLock)
4941         (COMMAND_DESCRIPTOR_t *)_NV_WriteLockDataAddress,
4942 #endif // CC_NV_WriteLock
4943 #if (PAD_LIST || CC_DictionaryAttackLockReset)
4944         (COMMAND_DESCRIPTOR_t *)_DictionaryAttackLockResetDataAddress,
4945 #endif // CC_DictionaryAttackLockReset
4946 #if (PAD_LIST || CC_DictionaryAttackParameters)
4947         (COMMAND_DESCRIPTOR_t *)_DictionaryAttackParametersDataAddress,
4948 #endif // CC_DictionaryAttackParameters
4949 #if (PAD_LIST || CC_NV_ChangeAuth)
4950         (COMMAND_DESCRIPTOR_t *)_NV_ChangeAuthDataAddress,
4951 #endif // CC_NV_ChangeAuth
4952 #if (PAD_LIST || CC_PCR_Event)
4953         (COMMAND_DESCRIPTOR_t *)_PCR_EventDataAddress,
4954 #endif // CC_PCR_Event
4955 #if (PAD_LIST || CC_PCR_Reset)
4956         (COMMAND_DESCRIPTOR_t *)_PCR_ResetDataAddress,
4957 #endif // CC_PCR_Reset
4958 #if (PAD_LIST || CC_SequenceComplete)
4959         (COMMAND_DESCRIPTOR_t *)_SequenceCompleteDataAddress,
4960 #endif // CC_SequenceComplete
4961 #if (PAD_LIST || CC_SetAlgorithmSet)
4962         (COMMAND_DESCRIPTOR_t *)_SetAlgorithmSetDataAddress,
4963 #endif // CC_SetAlgorithmSet
4964 #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
4965         (COMMAND_DESCRIPTOR_t *)_SetCommandCodeAuditStatusDataAddress,
4966 #endif // CC_SetCommandCodeAuditStatus
4967 #if (PAD_LIST || CC_FieldUpgradeData)
4968         (COMMAND_DESCRIPTOR_t *)_FieldUpgradeDataDataAddress,
4969 #endif // CC_FieldUpgradeData
4970 #if (PAD_LIST || CC_IncrementalSelfTest)
4971         (COMMAND_DESCRIPTOR_t *)_IncrementalSelfTestDataAddress,
4972 #endif // CC_IncrementalSelfTest
4973 #if (PAD_LIST || CC_SelfTest)
4974         (COMMAND_DESCRIPTOR_t *)_SelfTestDataAddress,
4975 #endif // CC_SelfTest
4976 #if (PAD_LIST || CC_Startup)
4977         (COMMAND_DESCRIPTOR_t *)_StartupDataAddress,
4978 #endif // CC_Startup
4979 #if (PAD_LIST || CC_Shutdown)
4980         (COMMAND_DESCRIPTOR_t *)_ShutdownDataAddress,
4981 #endif // CC_Shutdown
4982 #if (PAD_LIST || CC_StirRandom)
4983         (COMMAND_DESCRIPTOR_t *)_StirRandomDataAddress,
4984 #endif // CC_StirRandom
4985 #if (PAD_LIST || CC_ActivateCredential)
4986         (COMMAND_DESCRIPTOR_t *)_ActivateCredentialDataAddress,
4987 #endif // CC_ActivateCredential
4988 #if (PAD_LIST || CC_Certify)
4989         (COMMAND_DESCRIPTOR_t *)_CertifyDataAddress,
4990 #endif // CC_Certify
4991 #if (PAD_LIST || CC_PolicyNV)
4992         (COMMAND_DESCRIPTOR_t *)_PolicyNVDataAddress,
4993 #endif // CC_PolicyNV
4994 #if (PAD_LIST || CC_CertifyCreation)
4995         (COMMAND_DESCRIPTOR_t *)_CertifyCreationDataAddress,
4996 #endif // CC_CertifyCreation
4997 #if (PAD_LIST || CC_Duplicate)
4998         (COMMAND_DESCRIPTOR_t *)_DuplicateDataAddress,
4999 #endif // CC_Duplicate
5000 #if (PAD_LIST || CC_GetTime)

```

```

5001         (COMMAND_DESCRIPTOR_t *)_GetTimeDataAddress,
5002 #endif // CC_GetTime
5003 #if (PAD_LIST || CC_GetSessionAuditDigest)
5004         (COMMAND_DESCRIPTOR_t *)_GetSessionAuditDigestDataAddress,
5005 #endif // CC_GetSessionAuditDigest
5006 #if (PAD_LIST || CC_NV_Read)
5007         (COMMAND_DESCRIPTOR_t *)_NV_ReadDataAddress,
5008 #endif // CC_NV_Read
5009 #if (PAD_LIST || CC_NV_ReadLock)
5010         (COMMAND_DESCRIPTOR_t *)_NV_ReadLockDataAddress,
5011 #endif // CC_NV_ReadLock
5012 #if (PAD_LIST || CC_ObjectChangeAuth)
5013         (COMMAND_DESCRIPTOR_t *)_ObjectChangeAuthDataAddress,
5014 #endif // CC_ObjectChangeAuth
5015 #if (PAD_LIST || CC_PolicySecret)
5016         (COMMAND_DESCRIPTOR_t *)_PolicySecretDataAddress,
5017 #endif // CC_PolicySecret
5018 #if (PAD_LIST || CC_Rewrap)
5019         (COMMAND_DESCRIPTOR_t *)_RewrapDataAddress,
5020 #endif // CC_Rewrap
5021 #if (PAD_LIST || CC_Create)
5022         (COMMAND_DESCRIPTOR_t *)_CreateDataAddress,
5023 #endif // CC_Create
5024 #if (PAD_LIST || CC_ECDH_ZGen)
5025         (COMMAND_DESCRIPTOR_t *)_ECDH_ZGenDataAddress,
5026 #endif // CC_ECDH_ZGen
5027 #if (PAD_LIST || (CC_HMAC || CC_MAC))
5028 #   if CC_HMAC
5029         (COMMAND_DESCRIPTOR_t *)_HMACDataAddress,
5030 #   endif
5031 #   if CC_MAC
5032         (COMMAND_DESCRIPTOR_t *)_MACDataAddress,
5033 #   endif
5034 #   if (CC_HMAC || CC_MAC) > 1
5035 #       error "More than one aliased command defined"
5036 #   endif
5037 #endif // CC_HMAC CC_MAC
5038 #if (PAD_LIST || CC_Import)
5039         (COMMAND_DESCRIPTOR_t *)_ImportDataAddress,
5040 #endif // CC_Import
5041 #if (PAD_LIST || CC_Load)
5042         (COMMAND_DESCRIPTOR_t *)_LoadDataAddress,
5043 #endif // CC_Load
5044 #if (PAD_LIST || CC_Quote)
5045         (COMMAND_DESCRIPTOR_t *)_QuoteDataAddress,
5046 #endif // CC_Quote
5047 #if (PAD_LIST || CC_RSA_Decrypt)
5048         (COMMAND_DESCRIPTOR_t *)_RSA_DecryptDataAddress,
5049 #endif // CC_RSA_Decrypt
5050 #if (PAD_LIST)
5051         (COMMAND_DESCRIPTOR_t *)0,
5052 #endif //
5053 #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
5054 #   if CC_HMAC_Start
5055         (COMMAND_DESCRIPTOR_t *)_HMAC_StartDataAddress,
5056 #   endif
5057 #   if CC_MAC_Start
5058         (COMMAND_DESCRIPTOR_t *)_MAC_StartDataAddress,
5059 #   endif
5060 #   if (CC_HMAC_Start || CC_MAC_Start) > 1
5061 #       error "More than one aliased command defined"
5062 #   endif
5063 #endif // CC_HMAC_Start CC_MAC_Start
5064 #if (PAD_LIST || CC_SequenceUpdate)
5065         (COMMAND_DESCRIPTOR_t *)_SequenceUpdateDataAddress,
5066 #endif // CC_SequenceUpdate

```

```
5067 #if (PAD_LIST || CC_Sign)
5068     (COMMAND_DESCRIPTOR_t *)_SignDataAddress,
5069 #endif // CC_Sign
5070 #if (PAD_LIST || CC_Unseal)
5071     (COMMAND_DESCRIPTOR_t *)_UnsealDataAddress,
5072 #endif // CC_Unseal
5073 #if (PAD_LIST)
5074     (COMMAND_DESCRIPTOR_t *)0,
5075 #endif //
5076 #if (PAD_LIST || CC_PolicySigned)
5077     (COMMAND_DESCRIPTOR_t *)_PolicySignedDataAddress,
5078 #endif // CC_PolicySigned
5079 #if (PAD_LIST || CC_ContextLoad)
5080     (COMMAND_DESCRIPTOR_t *)_ContextLoadDataAddress,
5081 #endif // CC_ContextLoad
5082 #if (PAD_LIST || CC_ContextSave)
5083     (COMMAND_DESCRIPTOR_t *)_ContextSaveDataAddress,
5084 #endif // CC_ContextSave
5085 #if (PAD_LIST || CC_ECDH_KeyGen)
5086     (COMMAND_DESCRIPTOR_t *)_ECDH_KeyGenDataAddress,
5087 #endif // CC_ECDH_KeyGen
5088 #if (PAD_LIST || CC_EncryptDecrypt)
5089     (COMMAND_DESCRIPTOR_t *)_EncryptDecryptDataAddress,
5090 #endif // CC_EncryptDecrypt
5091 #if (PAD_LIST || CC_FlushContext)
5092     (COMMAND_DESCRIPTOR_t *)_FlushContextDataAddress,
5093 #endif // CC_FlushContext
5094 #if (PAD_LIST)
5095     (COMMAND_DESCRIPTOR_t *)0,
5096 #endif //
5097 #if (PAD_LIST || CC_LoadExternal)
5098     (COMMAND_DESCRIPTOR_t *)_LoadExternalDataAddress,
5099 #endif // CC_LoadExternal
5100 #if (PAD_LIST || CC_MakeCredential)
5101     (COMMAND_DESCRIPTOR_t *)_MakeCredentialDataAddress,
5102 #endif // CC_MakeCredential
5103 #if (PAD_LIST || CC_NV_ReadPublic)
5104     (COMMAND_DESCRIPTOR_t *)_NV_ReadPublicDataAddress,
5105 #endif // CC_NV_ReadPublic
5106 #if (PAD_LIST || CC_PolicyAuthorize)
5107     (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeDataAddress,
5108 #endif // CC_PolicyAuthorize
5109 #if (PAD_LIST || CC_PolicyAuthValue)
5110     (COMMAND_DESCRIPTOR_t *)_PolicyAuthValueDataAddress,
5111 #endif // CC_PolicyAuthValue
5112 #if (PAD_LIST || CC_PolicyCommandCode)
5113     (COMMAND_DESCRIPTOR_t *)_PolicyCommandCodeDataAddress,
5114 #endif // CC_PolicyCommandCode
5115 #if (PAD_LIST || CC_PolicyCounterTimer)
5116     (COMMAND_DESCRIPTOR_t *)_PolicyCounterTimerDataAddress,
5117 #endif // CC_PolicyCounterTimer
5118 #if (PAD_LIST || CC_PolicyCpHash)
5119     (COMMAND_DESCRIPTOR_t *)_PolicyCpHashDataAddress,
5120 #endif // CC_PolicyCpHash
5121 #if (PAD_LIST || CC_PolicyLocality)
5122     (COMMAND_DESCRIPTOR_t *)_PolicyLocalityDataAddress,
5123 #endif // CC_PolicyLocality
5124 #if (PAD_LIST || CC_PolicyNameHash)
5125     (COMMAND_DESCRIPTOR_t *)_PolicyNameHashDataAddress,
5126 #endif // CC_PolicyNameHash
5127 #if (PAD_LIST || CC_PolicyOR)
5128     (COMMAND_DESCRIPTOR_t *)_PolicyORDataAddress,
5129 #endif // CC_PolicyOR
5130 #if (PAD_LIST || CC_PolicyTicket)
5131     (COMMAND_DESCRIPTOR_t *)_PolicyTicketDataAddress,
5132 #endif // CC_PolicyTicket
```

```

5133 #if (PAD_LIST || CC_ReadPublic)
5134     (COMMAND_DESCRIPTOR_t *)_ReadPublicDataAddress,
5135 #endif // CC_ReadPublic
5136 #if (PAD_LIST || CC_RSA_Encrypt)
5137     (COMMAND_DESCRIPTOR_t *)_RSA_EncryptDataAddress,
5138 #endif // CC_RSA_Encrypt
5139 #if (PAD_LIST)
5140     (COMMAND_DESCRIPTOR_t *)0,
5141 #endif //
5142 #if (PAD_LIST || CC_StartAuthSession)
5143     (COMMAND_DESCRIPTOR_t *)_StartAuthSessionDataAddress,
5144 #endif // CC_StartAuthSession
5145 #if (PAD_LIST || CC_VerifySignature)
5146     (COMMAND_DESCRIPTOR_t *)_VerifySignatureDataAddress,
5147 #endif // CC_VerifySignature
5148 #if (PAD_LIST || CC_ECC_Parameters)
5149     (COMMAND_DESCRIPTOR_t *)_ECC_ParametersDataAddress,
5150 #endif // CC_ECC_Parameters
5151 #if (PAD_LIST || CC_FirmwareRead)
5152     (COMMAND_DESCRIPTOR_t *)_FirmwareReadDataAddress,
5153 #endif // CC_FirmwareRead
5154 #if (PAD_LIST || CC_GetCapability)
5155     (COMMAND_DESCRIPTOR_t *)_GetCapabilityDataAddress,
5156 #endif // CC_GetCapability
5157 #if (PAD_LIST || CC_GetRandom)
5158     (COMMAND_DESCRIPTOR_t *)_GetRandomDataAddress,
5159 #endif // CC_GetRandom
5160 #if (PAD_LIST || CC_GetTestResult)
5161     (COMMAND_DESCRIPTOR_t *)_GetTestResultDataAddress,
5162 #endif // CC_GetTestResult
5163 #if (PAD_LIST || CC_Hash)
5164     (COMMAND_DESCRIPTOR_t *)_HashDataAddress,
5165 #endif // CC_Hash
5166 #if (PAD_LIST || CC_PCR_Read)
5167     (COMMAND_DESCRIPTOR_t *)_PCR_ReadDataAddress,
5168 #endif // CC_PCR_Read
5169 #if (PAD_LIST || CC_PolicyPCR)
5170     (COMMAND_DESCRIPTOR_t *)_PolicyPCRDataAddress,
5171 #endif // CC_PolicyPCR
5172 #if (PAD_LIST || CC_PolicyRestart)
5173     (COMMAND_DESCRIPTOR_t *)_PolicyRestartDataAddress,
5174 #endif // CC_PolicyRestart
5175 #if (PAD_LIST || CC_ReadClock)
5176     (COMMAND_DESCRIPTOR_t *)_ReadClockDataAddress,
5177 #endif // CC_ReadClock
5178 #if (PAD_LIST || CC_PCR_Extend)
5179     (COMMAND_DESCRIPTOR_t *)_PCR_ExtendDataAddress,
5180 #endif // CC_PCR_Extend
5181 #if (PAD_LIST || CC_PCR_SetAuthValue)
5182     (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthValueDataAddress,
5183 #endif // CC_PCR_SetAuthValue
5184 #if (PAD_LIST || CC_NV_Certify)
5185     (COMMAND_DESCRIPTOR_t *)_NV_CertifyDataAddress,
5186 #endif // CC_NV_Certify
5187 #if (PAD_LIST || CC_EventSequenceComplete)
5188     (COMMAND_DESCRIPTOR_t *)_EventSequenceCompleteDataAddress,
5189 #endif // CC_EventSequenceComplete
5190 #if (PAD_LIST || CC_HashSequenceStart)
5191     (COMMAND_DESCRIPTOR_t *)_HashSequenceStartDataAddress,
5192 #endif // CC_HashSequenceStart
5193 #if (PAD_LIST || CC_PolicyPhysicalPresence)
5194     (COMMAND_DESCRIPTOR_t *)_PolicyPhysicalPresenceDataAddress,
5195 #endif // CC_PolicyPhysicalPresence
5196 #if (PAD_LIST || CC_PolicyDuplicationSelect)
5197     (COMMAND_DESCRIPTOR_t *)_PolicyDuplicationSelectDataAddress,
5198 #endif // CC_PolicyDuplicationSelect

```



```

5199 #if (PAD_LIST || CC_PolicyGetDigest)
5200     (COMMAND_DESCRIPTOR_t *)_PolicyGetDigestDataAddress,
5201 #endif // CC_PolicyGetDigest
5202 #if (PAD_LIST || CC_TestParms)
5203     (COMMAND_DESCRIPTOR_t *)_TestParmsDataAddress,
5204 #endif // CC_TestParms
5205 #if (PAD_LIST || CC_Commit)
5206     (COMMAND_DESCRIPTOR_t *)_CommitDataAddress,
5207 #endif // CC_Commit
5208 #if (PAD_LIST || CC_PolicyPassword)
5209     (COMMAND_DESCRIPTOR_t *)_PolicyPasswordDataAddress,
5210 #endif // CC_PolicyPassword
5211 #if (PAD_LIST || CC_ZGen_2Phase)
5212     (COMMAND_DESCRIPTOR_t *)_ZGen_2PhaseDataAddress,
5213 #endif // CC_ZGen_2Phase
5214 #if (PAD_LIST || CC_EC_Ephemeral)
5215     (COMMAND_DESCRIPTOR_t *)_EC_EphemeralDataAddress,
5216 #endif // CC_EC_Ephemeral
5217 #if (PAD_LIST || CC_PolicyNvWritten)
5218     (COMMAND_DESCRIPTOR_t *)_PolicyNvWrittenDataAddress,
5219 #endif // CC_PolicyNvWritten
5220 #if (PAD_LIST || CC_PolicyTemplate)
5221     (COMMAND_DESCRIPTOR_t *)_PolicyTemplateDataAddress,
5222 #endif // CC_PolicyTemplate
5223 #if (PAD_LIST || CC_CreateLoaded)
5224     (COMMAND_DESCRIPTOR_t *)_CreateLoadedDataAddress,
5225 #endif // CC_CreateLoaded
5226 #if (PAD_LIST || CC_PolicyAuthorizeNV)
5227     (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeNVDataAddress,
5228 #endif // CC_PolicyAuthorizeNV
5229 #if (PAD_LIST || CC_EncryptDecrypt2)
5230     (COMMAND_DESCRIPTOR_t *)_EncryptDecrypt2DataAddress,
5231 #endif // CC_EncryptDecrypt2
5232 #if (PAD_LIST || CC_AC_GetCapability)
5233     (COMMAND_DESCRIPTOR_t *)_AC_GetCapabilityDataAddress,
5234 #endif // CC_AC_GetCapability
5235 #if (PAD_LIST || CC_AC_Send)
5236     (COMMAND_DESCRIPTOR_t *)_AC_SendDataAddress,
5237 #endif // CC_AC_Send
5238 #if (PAD_LIST || CC_Policy_AC_SendSelect)
5239     (COMMAND_DESCRIPTOR_t *)_Policy_AC_SendSelectDataAddress,
5240 #endif // CC_Policy_AC_SendSelect
5241 #if (PAD_LIST || CC_CertifyX509)
5242     (COMMAND_DESCRIPTOR_t *)_CertifyX509DataAddress,
5243 #endif // CC_CertifyX509
5244 #if (PAD_LIST || CC_ACT_SetTimeout)
5245     (COMMAND_DESCRIPTOR_t *)_ACT_SetTimeoutDataAddress,
5246 #endif // CC_ACT_SetTimeout
5247 #if (PAD_LIST || CC_ECC_Encrypt)
5248     (COMMAND_DESCRIPTOR_t *)_ECC_EncryptDataAddress,
5249 #endif // CC_ECC_Encrypt
5250 #if (PAD_LIST || CC_ECC_Decrypt)
5251     (COMMAND_DESCRIPTOR_t *)_ECC_DecryptDataAddress,
5252 #endif // CC_ECC_Decrypt
5253 #if (PAD_LIST || CC_Vendor_TCG_Test)
5254     (COMMAND_DESCRIPTOR_t *)_Vendor_TCG_TestDataAddress,
5255 #endif // CC_Vendor_TCG_Test
5256     0
5257 };
5258
5259 #endif // _COMMAND_TABLE_DISPATCH_

```

5.7 Commands.h

```
1  #ifndef _COMMANDS_H_
2  #define _COMMANDS_H_
```

Start-up

```
3  #if CC_Startup
4  #include "Startup_fp.h"
5  #endif
6  #if CC_Shutdown
7  #include "Shutdown_fp.h"
8  #endif
```

Testing

```
9  #if CC_SelfTest
10 #include "SelfTest_fp.h"
11 #endif
12 #if CC_IncrementalSelfTest
13 #include "IncrementalSelfTest_fp.h"
14 #endif
15 #if CC_GetTestResult
16 #include "GetTestResult_fp.h"
17 #endif
```

Session Commands

```
18 #if CC_StartAuthSession
19 #include "StartAuthSession_fp.h"
20 #endif
21 #if CC_PolicyRestart
22 #include "PolicyRestart_fp.h"
23 #endif
```

Object Commands

```
24 #if CC_Create
25 #include "Create_fp.h"
26 #endif
27 #if CC_Load
28 #include "Load_fp.h"
29 #endif
30 #if CC_LoadExternal
31 #include "LoadExternal_fp.h"
32 #endif
33 #if CC_ReadPublic
34 #include "ReadPublic_fp.h"
35 #endif
36 #if CC_ActivateCredential
37 #include "ActivateCredential_fp.h"
38 #endif
39 #if CC_MakeCredential
40 #include "MakeCredential_fp.h"
41 #endif
42 #if CC_Unseal
43 #include "Unseal_fp.h"
44 #endif
45 #if CC_ObjectChangeAuth
46 #include "ObjectChangeAuth_fp.h"
47 #endif
48 #if CC_CreateLoaded
49 #include "CreateLoaded_fp.h"
```



```
50 #endif
```

Duplication Commands

```
51 #if CC_Duplicate
52 #include "Duplicate_fp.h"
53 #endif
54 #if CC_Rewrap
55 #include "Rewrap_fp.h"
56 #endif
57 #if CC_Import
58 #include "Import_fp.h"
59 #endif
```

Asymmetric Primitives

```
60 #if CC_RSA_Encrypt
61 #include "RSA_Encrypt_fp.h"
62 #endif
63 #if CC_RSA_Decrypt
64 #include "RSA_Decrypt_fp.h"
65 #endif
66 #if CC_ECDH_KeyGen
67 #include "ECDH_KeyGen_fp.h"
68 #endif
69 #if CC_ECDH_ZGen
70 #include "ECDH_ZGen_fp.h"
71 #endif
72 #if CC_ECC_Parameters
73 #include "ECC_Parameters_fp.h"
74 #endif
75 #if CC_ZGen_2Phase
76 #include "ZGen_2Phase_fp.h"
77 #endif
78 #if CC_ECC_Encrypt
79 #include "ECC_Encrypt_fp.h"
80 #endif
81 #if CC_ECC_Decrypt
82 #include "ECC_Decrypt_fp.h"
83 #endif
```

Symmetric Primitives

```
84 #if CC_EncryptDecrypt
85 #include "EncryptDecrypt_fp.h"
86 #endif
87 #if CC_EncryptDecrypt2
88 #include "EncryptDecrypt2_fp.h"
89 #endif
90 #if CC_Hash
91 #include "Hash_fp.h"
92 #endif
93 #if CC_HMAC
94 #include "HMAC_fp.h"
95 #endif
96 #if CC_MAC
97 #include "MAC_fp.h"
98 #endif
```

Random Number Generator

```
99 #if CC_GetRandom
100 #include "GetRandom_fp.h"
101 #endif
```

```
102  #if CC_StirRandom
103  #include "StirRandom_fp.h"
104  #endif
```

Hash/HMAC/Event Sequences

```
105  #if CC_HMAC_Start
106  #include "HMAC_Start_fp.h"
107  #endif
108  #if CC_MAC_Start
109  #include "MAC_Start_fp.h"
110  #endif
111  #if CC_HashSequenceStart
112  #include "HashSequenceStart_fp.h"
113  #endif
114  #if CC_SequenceUpdate
115  #include "SequenceUpdate_fp.h"
116  #endif
117  #if CC_SequenceComplete
118  #include "SequenceComplete_fp.h"
119  #endif
120  #if CC_EventSequenceComplete
121  #include "EventSequenceComplete_fp.h"
122  #endif
```

Attestation Commands

```
123  #if CC_Certify
124  #include "Certify_fp.h"
125  #endif
126  #if CC_CertifyCreation
127  #include "CertifyCreation_fp.h"
128  #endif
129  #if CC_Quote
130  #include "Quote_fp.h"
131  #endif
132  #if CC_GetSessionAuditDigest
133  #include "GetSessionAuditDigest_fp.h"
134  #endif
135  #if CC_GetCommandAuditDigest
136  #include "GetCommandAuditDigest_fp.h"
137  #endif
138  #if CC_GetTime
139  #include "GetTime_fp.h"
140  #endif
141  #if CC_CertifyX509
142  #include "CertifyX509_fp.h"
143  #endif
```

Ephemeral EC Keys

```
144  #if CC_Commit
145  #include "Commit_fp.h"
146  #endif
147  #if CC_EC_Ephemeral
148  #include "EC_Ephemeral_fp.h"
149  #endif
```

Signing and Signature Verification

```
150  #if CC_VerifySignature
151  #include "VerifySignature_fp.h"
152  #endif
153  #if CC_Sign
```

```
154 #include "Sign_fp.h"
155 #endif
```

Command Audit

```
156 #if CC_SetCommandCodeAuditStatus
157 #include "SetCommandCodeAuditStatus_fp.h"
158 #endif
```

Integrity Collection (PCR)

```
159 #if CC_PCR_Extend
160 #include "PCR_Extend_fp.h"
161 #endif
162 #if CC_PCR_Event
163 #include "PCR_Event_fp.h"
164 #endif
165 #if CC_PCR_Read
166 #include "PCR_Read_fp.h"
167 #endif
168 #if CC_PCR_Allocate
169 #include "PCR_Allocate_fp.h"
170 #endif
171 #if CC_PCR_SetAuthPolicy
172 #include "PCR_SetAuthPolicy_fp.h"
173 #endif
174 #if CC_PCR_SetAuthValue
175 #include "PCR_SetAuthValue_fp.h"
176 #endif
177 #if CC_PCR_Reset
178 #include "PCR_Reset_fp.h"
179 #endif
```

Enhanced Authorization (EA) Commands

```
180 #if CC_PolicySigned
181 #include "PolicySigned_fp.h"
182 #endif
183 #if CC_PolicySecret
184 #include "PolicySecret_fp.h"
185 #endif
186 #if CC_PolicyTicket
187 #include "PolicyTicket_fp.h"
188 #endif
189 #if CC_PolicyOR
190 #include "PolicyOR_fp.h"
191 #endif
192 #if CC_PolicyPCR
193 #include "PolicyPCR_fp.h"
194 #endif
195 #if CC_PolicyLocality
196 #include "PolicyLocality_fp.h"
197 #endif
198 #if CC_PolicyNV
199 #include "PolicyNV_fp.h"
200 #endif
201 #if CC_PolicyCounterTimer
202 #include "PolicyCounterTimer_fp.h"
203 #endif
204 #if CC_PolicyCommandCode
205 #include "PolicyCommandCode_fp.h"
206 #endif
207 #if CC_PolicyPhysicalPresence
208 #include "PolicyPhysicalPresence_fp.h"
```

```
209 #endif
210 #if CC_PolicyCpHash
211 #include "PolicyCpHash_fp.h"
212 #endif
213 #if CC_PolicyNameHash
214 #include "PolicyNameHash_fp.h"
215 #endif
216 #if CC_PolicyDuplicationSelect
217 #include "PolicyDuplicationSelect_fp.h"
218 #endif
219 #if CC_PolicyAuthorize
220 #include "PolicyAuthorize_fp.h"
221 #endif
222 #if CC_PolicyAuthValue
223 #include "PolicyAuthValue_fp.h"
224 #endif
225 #if CC_PolicyPassword
226 #include "PolicyPassword_fp.h"
227 #endif
228 #if CC_PolicyGetDigest
229 #include "PolicyGetDigest_fp.h"
230 #endif
231 #if CC_PolicyNvWritten
232 #include "PolicyNvWritten_fp.h"
233 #endif
234 #if CC_PolicyTemplate
235 #include "PolicyTemplate_fp.h"
236 #endif
237 #if CC_PolicyAuthorizeNV
238 #include "PolicyAuthorizeNV_fp.h"
239 #endif
```

Hierarchy Commands

```
240 #if CC_CreatePrimary
241 #include "CreatePrimary_fp.h"
242 #endif
243 #if CC_HierarchyControl
244 #include "HierarchyControl_fp.h"
245 #endif
246 #if CC_SetPrimaryPolicy
247 #include "SetPrimaryPolicy_fp.h"
248 #endif
249 #if CC_ChangePPS
250 #include "ChangePPS_fp.h"
251 #endif
252 #if CC_ChangeEPS
253 #include "ChangeEPS_fp.h"
254 #endif
255 #if CC_Clear
256 #include "Clear_fp.h"
257 #endif
258 #if CC_ClearControl
259 #include "ClearControl_fp.h"
260 #endif
261 #if CC_HierarchyChangeAuth
262 #include "HierarchyChangeAuth_fp.h"
263 #endif
```

Dictionary Attack Functions

```
264 #if CC_DictionaryAttackLockReset
265 #include "DictionaryAttackLockReset_fp.h"
266 #endif
267 #if CC_DictionaryAttackParameters
```

```
268 #include "DictionaryAttackParameters_fp.h"
269 #endif
```

Miscellaneous Management Functions

```
270 #if CC_PP_Commands
271 #include "PP_Commands_fp.h"
272 #endif
273 #if CC_SetAlgorithmSet
274 #include "SetAlgorithmSet_fp.h"
275 #endif
```

Field Upgrade

```
276 #if CC_FieldUpgradeStart
277 #include "FieldUpgradeStart_fp.h"
278 #endif
279 #if CC_FieldUpgradeData
280 #include "FieldUpgradeData_fp.h"
281 #endif
282 #if CC_FirmwareRead
283 #include "FirmwareRead_fp.h"
284 #endif
```

Context Management

```
285 #if CC_ContextSave
286 #include "ContextSave_fp.h"
287 #endif
288 #if CC_ContextLoad
289 #include "ContextLoad_fp.h"
290 #endif
291 #if CC_FlushContext
292 #include "FlushContext_fp.h"
293 #endif
294 #if CC_EvictControl
295 #include "EvictControl_fp.h"
296 #endif
```

Clocks and Timers

```
297 #if CC_ReadClock
298 #include "ReadClock_fp.h"
299 #endif
300 #if CC_ClockSet
301 #include "ClockSet_fp.h"
302 #endif
303 #if CC_ClockRateAdjust
304 #include "ClockRateAdjust_fp.h"
305 #endif
```

Capability Commands

```
306 #if CC_GetCapability
307 #include "GetCapability_fp.h"
308 #endif
309 #if CC_TestParms
310 #include "TestParms_fp.h"
311 #endif
```

Non-volatile Storage

```
312 #if CC_NV_DefineSpace
```

```
313 #include "NV_DefineSpace_fp.h"
314 #endif
315 #if CC_NV_UndefineSpace
316 #include "NV_UndefineSpace_fp.h"
317 #endif
318 #if CC_NV_UndefineSpaceSpecial
319 #include "NV_UndefineSpaceSpecial_fp.h"
320 #endif
321 #if CC_NV_ReadPublic
322 #include "NV_ReadPublic_fp.h"
323 #endif
324 #if CC_NV_Write
325 #include "NV_Write_fp.h"
326 #endif
327 #if CC_NV_Increment
328 #include "NV_Increment_fp.h"
329 #endif
330 #if CC_NV_Extend
331 #include "NV_Extend_fp.h"
332 #endif
333 #if CC_NV_SetBits
334 #include "NV_SetBits_fp.h"
335 #endif
336 #if CC_NV_WriteLock
337 #include "NV_WriteLock_fp.h"
338 #endif
339 #if CC_NV_GlobalWriteLock
340 #include "NV_GlobalWriteLock_fp.h"
341 #endif
342 #if CC_NV_Read
343 #include "NV_Read_fp.h"
344 #endif
345 #if CC_NV_ReadLock
346 #include "NV_ReadLock_fp.h"
347 #endif
348 #if CC_NV_ChangeAuth
349 #include "NV_ChangeAuth_fp.h"
350 #endif
351 #if CC_NV_Certify
352 #include "NV_Certify_fp.h"
353 #endif
```

Attached Components

```
354 #if CC_AC_GetCapability
355 #include "AC_GetCapability_fp.h"
356 #endif
357 #if CC_AC_Send
358 #include "AC_Send_fp.h"
359 #endif
360 #if CC_Policy_AC_SendSelect
361 #include "Policy_AC_SendSelect_fp.h"
362 #endif
```

Authenticated Countdown Timer

```
363 #if CC_ACT_SetTimeout
364 #include "ACT_SetTimeout_fp.h"
365 #endif
```

Vendor Specific

```
366 #if CC_Vendor_TCG_Test
367 #include "Vendor_TCG_Test_fp.h"
```

```
368 #endif  
369  
370 #endif
```

DRAFT

5.8 CompilerDependencies.h

This file contains the build switches. This contains switches for multiple versions of the crypto-library so some may not apply to your environment.

```

1  #ifndef COMPILER_DEPENDENCIES_H
2  #define COMPILER_DEPENDENCIES_H
3
4  #ifdef GCC
5  #   undef MSC_VER
6  #   undef WIN32
7  #endif
8
9  #ifdef MSC_VER

```

These definitions are for the Microsoft compiler

Endian conversion for aligned structures

```

10 #   define REVERSE_ENDIAN_16(_Number) _byteswap_ushort(_Number)
11 #   define REVERSE_ENDIAN_32(_Number) _byteswap_ulong(_Number)
12 #   define REVERSE_ENDIAN_64(_Number) _byteswap_uint64(_Number)

```

Avoid compiler warning for in line of stdio (or not)

```

13 // #define _NO_CRT_STDIO_INLINE
14

```

This macro is used to handle LIB_EXPORT of function and variable names in lieu of a .def file. Visual Studio requires that functions be explicitly exported and imported.

```

15 #   define LIB_EXPORT __declspec(dllexport) // VS compatible version
16 #   define LIB_IMPORT __declspec(dllimport)

```

This is defined to indicate a function that does not return. Microsoft compilers do not support the _Noreturn function parameter.

```

17 #   define NORETURN __declspec(noreturn)
18 #   if MSC_VER >= 1400 // SAL processing when needed
19 #       include <sal.h>
20 #   endif
21
22 #   ifdef WIN64
23 #       define _INTPTR 2
24 #   else
25 #       define _INTPTR 1
26 #   endif
27
28 #define NOT_REFERENCED(x) (x)

```

Lower the compiler error warning for system include files. They tend not to be that clean and there is no reason to sort through all the spurious errors that they generate when the normal error level is set to /Wall

```

29 #   define _REDUCE_WARNING_LEVEL_(n) \
30     __pragma(warning(push, n))

```

Restore the compiler warning level

```

31 #   define _NORMAL_WARNING_LEVEL_ \
32     __pragma(warning(pop))
33 #   include <stdint.h>

```

```

34 #endif
35
36 #ifndef _MSC_VER
37 #ifndef WINAPI
38 #   define WINAPI
39 #endif
40 #   define __pragma(x)
41 #   define REVERSE_ENDIAN_16(_Number) __builtin_bswap16(_Number)
42 #   define REVERSE_ENDIAN_32(_Number) __builtin_bswap32(_Number)
43 #   define REVERSE_ENDIAN_64(_Number) __builtin_bswap64(_Number)
44 #endif
45
46 #if defined(__GNUC__)
47 #   define NORETURN __attribute__((noreturn))
48 #   include <stdint.h>
49 #endif

```

Things that are not defined should be defined as NULL

```

50 #ifndef NORETURN
51 #   define NORETURN
52 #endif
53 #ifndef LIB_EXPORT
54 #   define LIB_EXPORT
55 #endif
56 #ifndef LIB_IMPORT
57 #   define LIB_IMPORT
58 #endif
59 #ifndef _REDUCE_WARNING_LEVEL_
60 #   define _REDUCE_WARNING_LEVEL_(n)
61 #endif
62 #ifndef _NORMAL_WARNING_LEVEL_
63 #   define _NORMAL_WARNING_LEVEL_
64 #endif
65 #ifndef NOT_REFERENCED
66 #   define NOT_REFERENCED(x) (x = x)
67 #endif
68
69 #ifdef _POSIX_
70 typedef int SOCKET;
71 #endif
72
73 #endif // _COMPILER_DEPENDENCIES_H_

```

5.9 Global.h

5.9.1 Description

This file contains internal global type definitions and data declarations that are need between subsystems. The instantiation of global data is in Global.c. The initialization of global data is in the subsystem that is the primary owner of the data.

The first part of this file has the *typedefs* for structures and other defines used in many portions of the code. After the *typedef* section, is a section that defines global values that are only present in RAM. The next three sections define the structures for the NV data areas: persistent, orderly, and state save. Additional sections define the data that is used in specific modules. That data is private to the module but is collected here to simplify the management of the instance data.

All the data is instanced in Global.c.

```
1  #if !defined _TPM_H_
2  #error "Should only be instanced in TPM.h"
3  #endif
```

5.9.2 Includes

```
4  #ifndef          GLOBAL_H
5  #define          GLOBAL_H
6
7  _REDUCE_WARNING_LEVEL_(2)
8  #include <string.h>
9  #include <stddef.h>
10 _NORMAL_WARNING_LEVEL_
11
12 #include "Capabilities.h"
13 #include "TpmTypes.h"
14 #include "CommandAttributes.h"
15 #include "CryptTest.h"
16 #include "BnValues.h"
17 #include "CryptHash.h"
18 #include "CryptSym.h"
19 #include "CryptRand.h"
20 #include "CryptEcc.h"
21 #include "CryptRsa.h"
22 #include "CryptTest.h"
23 #include "TpmError.h"
24 #include "NV.h"
25 #include "ACT.h"
```

5.9.3 Defines and Types

5.9.3.1 Size Types

These types are used to differentiate the two different size values used.

NUMBYTES is used when a size is a number of bytes (usually a TPM2B)

```
26 typedef UINT16  NUMBYTES;
```

5.9.3.2 Other Types

An AUTH_VALUE is a BYTE array containing a digest (TPMU_HA)

```
27  typedef BYTE    AUTH_VALUE[sizeof(TPMU_HA)];
```

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

```
28  typedef BYTE    TIME_INFO[sizeof(TPMS_TIME_INFO)];
```

A NAME is a BYTE array that can contain a TPMU_NAME

```
29  typedef BYTE    NAME[sizeof(TPMU_NAME)];
```

Definition for a PROOF value

```
30  TPM2B_TYPE(PROOF, PROOF_SIZE);
```

Definition for a Primary Seed value

```
31  TPM2B_TYPE(SEED, PRIMARY_SEED_SIZE);
```

A CLOCK_NONCE is used to tag the time value in the authorization session and in the ticket computation so that the ticket expires when there is a time discontinuity. When the clock stops during normal operation, the nonce is 64-bit value kept in RAM but it is a 32-bit counter when the clock only stops during power events.

```
32  #if CLOCK_STOPS
33  typedef UINT64    CLOCK_NONCE;
34  #else
35  typedef UINT32    CLOCK_NONCE;
36  #endif
```

5.9.4 Loaded Object Structures

5.9.4.1 Description

The structures in this section define the object layout as it exists in TPM memory.

Two types of objects are defined: an ordinary object such as a key, and a sequence object that may be a hash, HMAC, or event.

5.9.4.2 OBJECT_ATTRIBUTES

An OBJECT_ATTRIBUTES structure contains the variable attributes of an object. These properties are not part of the public properties but are used by the TPM in managing the object. An OBJECT_ATTRIBUTES is used in the definition of the OBJECT data type.

```
37  typedef struct
38  {
39      unsigned    publicOnly : 1;    //0) SET if only the public portion of
40                                     // an object is loaded
41      unsigned    epsHierarchy : 1;  //1) SET if the object belongs to EPS
42                                     // Hierarchy
43      unsigned    ppsHierarchy : 1;  //2) SET if the object belongs to PPS
44                                     // Hierarchy
45      unsigned    spsHierarchy : 1;  //3) SET if the object belongs to SPS
46                                     // Hierarchy
47      unsigned    evict : 1;         //4) SET if the object is a platform or
48                                     // owner evict object. Platform-
49                                     // evict object belongs to PPS
50                                     // hierarchy, owner-evict object
51                                     // belongs to SPS or EPS hierarchy.
```

```

52                                     // This bit is also used to mark a
53                                     // completed sequence object so it
54                                     // will be flush when the
55                                     // SequenceComplete command succeeds.
56     unsigned           primary : 1;    //5) SET for a primary object
57     unsigned           temporary : 1;  //6) SET for a temporary object
58     unsigned           stClear : 1;    //7) SET for an stClear object
59     unsigned           hmacSeq : 1;    //8) SET for an HMAC or MAC sequence
60                                     // object
61     unsigned           hashSeq : 1;    //9) SET for a hash sequence object
62     unsigned           eventSeq : 1;   //10) SET for an event sequence object
63     unsigned           ticketSafe : 1; //11) SET if a ticket is safe to create
64                                     // for hash sequence object
65     unsigned           firstBlock : 1; //12) SET if the first block of hash
66                                     // data has been received. It
67                                     // works with ticketSafe bit
68     unsigned           isParent : 1;   //13) SET if the key has the proper
69                                     // attributes to be a parent key
70 //     unsigned         privateExp : 1; //14) SET when the private exponent
71 //                                     // of an RSA key has been validated.
72     unsigned           not_used_14 : 1;
73     unsigned           occupied : 1;   //15) SET when the slot is occupied.
74     unsigned           derivation : 1; //16) SET when the key is a derivation
75                                     // parent
76     unsigned           external : 1;   //17) SET when the object is loaded with
77                                     // TPM2_LoadExternal();
78 } OBJECT_ATTRIBUTES;
79
80 #if ALG_RSA

```

There is an overload of the `sensitive.rsa.t.size` field of a `TPMT_SENSITIVE` when an RSA key is loaded. When the *sensitive*→*sensitive* contains an RSA key with all of the CRT values, then the MSB of the size field will be set to indicate that the buffer contains all 5 of the CRT private key values.

```

81 #define      RSA_prime_flag      0x8000
82 #endif

```

5.9.4.3 OBJECT Structure

An `OBJECT` structure holds the object public, sensitive, and meta-data associated. This structure is implementation dependent. For this implementation, the structure is not optimized for space but rather for clarity of the reference implementation. Other implementations may choose to overlap portions of the structure that are not used simultaneously. These changes would necessitate changes to the source code but those changes would be compatible with the reference implementation.

```

83 typedef struct OBJECT
84 {
85     // The attributes field is required to be first followed by the publicArea.
86     // This allows the overlay of the object structure and a sequence structure
87     OBJECT_ATTRIBUTES  attributes;    // object attributes
88     TPMT_PUBLIC        publicArea;    // public area of an object
89     TPMT_SENSITIVE     sensitive;     // sensitive area of an object
90     TPM2B_NAME         qualifiedName;  // object qualified name
91     TPMI_DH_OBJECT     evictHandle;   // if the object is an evict object,
92                                     // the original handle is kept here.
93                                     // The 'working' handle will be the
94                                     // handle of an object slot.
95     TPM2B_NAME         name;          // Name of the object name. Kept here
96                                     // to avoid repeatedly computing it.
97 } OBJECT;

```

5.9.4.4 HASH_OBJECT Structure

This structure holds a hash sequence object or an event sequence object.

The first four components of this structure are manually set to be the same as the first four components of the object structure. This prevents the object from being inadvertently misused as sequence objects occupy the same memory as a regular object. A debug check is present to make sure that the offsets are what they are supposed to be.

NOTE In a future version, this will probably be renamed as SEQUENCE_OBJECT

```

98  typedef struct HASH_OBJECT
99  {
100      OBJECT_ATTRIBUTES    attributes;           // The attributes of the HASH object
101      TPMI_ALG_PUBLIC      type;                 // algorithm
102      TPMI_ALG_HASH        nameAlg;              // name algorithm
103      TPMA_OBJECT          objectAttributes;     // object attributes
104
105      // The data below is unique to a sequence object
106      TPM2B_AUTH           auth;                 // authorization for use of sequence
107      union
108      {
109          HASH_STATE       hashState[HASH_COUNT];
110          HMAC_STATE       hmacState;
111          state;
112      } HASH_OBJECT;
113
114  typedef BYTE    HASH_OBJECT_BUFFER[sizeof(HASH_OBJECT)];

```

5.9.4.5 ANY_OBJECT

This is the union for holding either a sequence object or a regular object for ContextSave() and ContextLoad().

```

115  typedef union ANY_OBJECT
116  {
117      OBJECT          entity;
118      HASH_OBJECT     hash;
119  } ANY_OBJECT;
120
121  typedef BYTE    ANY_OBJECT_BUFFER[sizeof(ANY_OBJECT)];

```

5.9.5 AUTH_DUP Types

These values are used in the authorization processing.

```

122  typedef UINT32    AUTH_ROLE;
123  #define AUTH_NONE    ((AUTH_ROLE) (0))
124  #define AUTH_USER     ((AUTH_ROLE) (1))
125  #define AUTH_ADMIN    ((AUTH_ROLE) (2))
126  #define AUTH_DUP      ((AUTH_ROLE) (3))

```

5.9.6 Active Session Context

5.9.6.1 Description

The structures in this section define the internal structure of a session context.

5.9.6.2 SESSION_ATTRIBUTES

The attributes in the SESSION_ATTRIBUTES structure track the various properties of the session. It maintains most of the tracking state information for the policy session. It is used within the SESSION structure.

```

127 typedef struct SESSION_ATTRIBUTES
128 {
129     unsigned    isPolicy : 1;           //(1) SET if the session may only be used
130                                           // for policy
131     unsigned    isAudit : 1;           //(2) SET if the session is used for audit
132     unsigned    isBound : 1;          //(3) SET if the session is bound to with an
133                                           // entity. This attribute will be CLEAR
134                                           // if either isPolicy or isAudit is SET.
135     unsigned    isCpHashDefined : 1;  //(4) SET if the cpHash has been defined
136                                           // This attribute is not SET unless
137                                           // 'isPolicy' is SET.
138     unsigned    isAuthValueNeeded : 1; //(5) SET if the authValue is required for
139                                           // computing the session HMAC. This
140                                           // attribute is not SET unless 'isPolicy'
141                                           // is SET.
142     unsigned    isPasswordNeeded : 1; //(6) SET if a password authValue is required
143                                           // for authorization This attribute is not
144                                           // SET unless 'isPolicy' is SET.
145     unsigned    isPPRequired : 1;      //(7) SET if physical presence is required to
146                                           // be asserted when the authorization is
147                                           // checked. This attribute is not SET
148                                           // unless 'isPolicy' is SET.
149     unsigned    isTrialPolicy : 1;     //(8) SET if the policy session is created
150                                           // for trial of the policy's policyHash
151                                           // generation. This attribute is not SET
152                                           // unless 'isPolicy' is SET.
153     unsigned    isDaBound : 1;         //(9) SET if the bind entity had noDA CLEAR.
154                                           // If this is SET, then an authorization
155                                           // failure using this session will count
156                                           // against lockout even if the object
157                                           // being authorized is exempt from DA.
158     unsigned    isLockoutBound : 1;    //(10) SET if the session is bound to
159                                           // lockoutAuth.
160     unsigned    includeAuth : 1;       //(11) This attribute is SET when the
161                                           // authValue of an object is to be
162                                           // included in the computation of the
163                                           // HMAC key for the command and response
164                                           // computations. (was 'requestWasBound')
165     unsigned    checkNvWritten : 1;    //(12) SET if the TPMA_NV_WRITTEN attribute
166                                           // needs to be checked when the policy is
167                                           // used for authorization for NV access.
168                                           // If this is SET for any other type, the
169                                           // policy will fail.
170     unsigned    nvWrittenState : 1;    //(13) SET if TPMA_NV_WRITTEN is required to
171                                           // be SET. Used when 'checkNvWritten' is
172                                           // SET
173     unsigned    isTemplateSet : 1;     //(14) SET if the templateHash needs to be
174                                           // checked for Create, CreatePrimary, or
175                                           // CreateLoaded.
176 } SESSION_ATTRIBUTES;

```

5.9.6.3 SESSION Structure

The SESSION structure contains all the context of a session except for the associated *contextID*.

NOTE The *contextID* of a session is only relevant when the session context is stored off the TPM.

```

177 typedef struct SESSION
178 {
179     SESSION_ATTRIBUTES  attributes;           // session attributes
180     UINT32              pcrCounter;           // PCR counter value when PCR is
181                                           // included (policy session)
182                                           // If no PCR is included, this
183                                           // value is 0.
184     UINT64              startTime;            // The value in g_time when the session
185                                           // was started (policy session)
186     UINT64              timeout;              // The timeout relative to g_time
187                                           // There is no timeout if this value
188                                           // is 0.
189     CLOCK_NONCE         epoch;                // The g_clockEpoch value when the
190                                           // session was started. If g_clockEpoch
191                                           // does not match this value when the
192                                           // timeout is used, then
193                                           // then the command will fail.
194     TPM_CC              commandCode;           // command code (policy session)
195     TPM_ALG_ID          authHashAlg;          // session hash algorithm
196     TPMA_LOCALITY        commandLocality;      // command locality (policy session)
197     TPMT_SYM_DEF         symmetric;            // session symmetric algorithm (if any)
198     TPM2B_AUTH           sessionKey;           // session secret value used for
199                                           // this session
200     TPM2B_NONCE          nonceTPM;            // last TPM-generated nonce for
201                                           // generating HMAC and encryption keys
202     union
203     {
204         TPM2B_NAME       boundEntity;         // value used to track the entity to
205                                           // which the session is bound
206
207         TPM2B_DIGEST      cpHash;             // the required cpHash value for the
208                                           // command being authorized
209         TPM2B_DIGEST      nameHash;           // the required nameHash
210         TPM2B_DIGEST      templateHash;       // the required template for creation
211     } u1;
212
213     union
214     {
215         TPM2B_DIGEST      auditDigest;        // audit session digest
216         TPM2B_DIGEST      policyDigest;       // policyHash
217     } u2;                                     // audit log and policyHash may
218                                           // share space to save memory
219 } SESSION;
220
221 #define EXPIRES_ON_RESET    INT32_MIN
222 #define TIMEOUT_ON_RESET    UINT64_MAX
223 #define EXPIRES_ON_RESTART  (INT32_MIN + 1)
224 #define TIMEOUT_ON_RESTART  (UINT64_MAX - 1)
225
226 typedef BYTE              SESSION_BUF[sizeof(SESSION)];

```

5.9.7 PCR

5.9.7.1 PCR_SAVE Structure

The PCR_SAVE structure type contains the PCR data that are saved across power cycles. Only the static PCR are required to be saved across power cycles. The DRTM and resettable PCR are not saved. The number of static and resettable PCR is determined by the platform-specific specification to which the TPM is built.

```

227 #define PCR_SAVE_SPACE(HASH, Hash)  BYTE Hash[NUM_STATIC_PCR][HASH##_DIGEST_SIZE];

```

```

228
229 typedef struct PCR_SAVE
230 {
231     FOR_EACH_HASH(PCR_SAVE_SPACE)
232
233     // This counter increments whenever the PCR are updated.
234     // NOTE: A platform-specific specification may designate
235     //       certain PCR changes as not causing this counter
236     //       to increment.
237     UINT32 pcrCounter;
238 } PCR_SAVE;

```

5.9.7.2 PCR_POLICY

```

239 #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0

```

This structure holds the PCR policies, one for each group of PCR controlled by policy.

```

240 typedef struct PCR_POLICY
241 {
242     TPMI_ALG_HASH      hashAlg[NUM_POLICY_PCR_GROUP];
243     TPM2B_DIGEST       a;
244     TPM2B_DIGEST       policy[NUM_POLICY_PCR_GROUP];
245 } PCR_POLICY;
246 #endif

```

5.9.7.3 PCR_AUTHVALUE

This structure holds the PCR policies, one for each group of PCR controlled by policy.

```

247 typedef struct PCR_AUTH_VALUE
248 {
249     TPM2B_DIGEST       auth[NUM_AUTHVALUE_PCR_GROUP];
250 } PCR_AUTHVALUE;

```

5.9.8 STARTUP_TYPE

This enumeration is the possible startup types. The type is determined by the combination of TPM2_ShutDown() and TPM2_Startup.

```

251 typedef enum
252 {
253     SU_RESET,
254     SU_RESTART,
255     SU_RESUME
256 } STARTUP_TYPE;

```

5.9.9 NV

5.9.9.1 NV_INDEX

The NV_INDEX structure defines the internal format for an NV index. The *indexData* size varies according to the type of the index. In this implementation, all of the index is manipulated as a unit.

```

257 typedef struct NV_INDEX
258 {
259     TPMS_NV_PUBLIC      publicArea;
260     TPM2B_AUTH          authValue;
261 } NV_INDEX;

```

5.9.9.2 NV_REF

An NV_REF is an opaque value returned by the NV subsystem. It is used to reference and NV Index in a relatively efficient way. Rather than having to continually search for an Index, its reference value may be used. In this implementation, an NV_REF is a byte pointer that points to the copy of the NV memory that is kept in RAM.

```
262 typedef UINT32      NV_REF;
263
264 typedef BYTE        *NV_RAM_REF;
```

5.9.9.3 NV_PIN

This structure deals with the possible endianness differences between the canonical form of the TPMS_NV_PIN_COUNTER_PARAMETERS structure and the internal value. The structures allow the data in a PIN index to be read as an 8-octet value using `NvReadUINT64Data()`. That function will byte swap all the values on a little endian system. This will put the bytes with the 4-octet values in the correct order but will swap the *pinLimit* and *pinCount* values. When written, the PIN index is simply handled as a normal index with the octets in canonical order.

```
265 #if BIG_ENDIAN_TPM
266 typedef struct
267 {
268     UINT32    pinCount;
269     UINT32    pinLimit;
270 } PIN_DATA;
271 #else
272 typedef struct
273 {
274     UINT32    pinLimit;
275     UINT32    pinCount;
276 } PIN_DATA;
277 #endif
278
279 typedef union
280 {
281     UINT64    intVal;
282     PIN_DATA  pin;
283 } NV_PIN;
```

5.9.10 COMMIT_INDEX_MASK

This is the define for the mask value that is used when manipulating the bits in the commit bit array. The commit counter is a 64-bit value and the low order bits are used to index the *commitArray*. This mask value is applied to the commit counter to extract the bit number in the array.

```
284 #if ALG_ECC
285
286 #define COMMIT_INDEX_MASK ((UINT16)((sizeof(gr.commitArray)*8)-1))
287
288 #endif
```

5.9.11 RAM Global Values

5.9.11.1 Description

The values in this section are only extant in RAM or ROM as constant values.

5.9.11.2 Crypto Self-Test Values

```
289  EXTERN ALGORITHM_VECTOR    g_implementedAlgorithms;
290  EXTERN ALGORITHM_VECTOR    g_toTest;
```

5.9.11.3 g_rcIndex[]

This array is used to contain the array of values that are added to a return code when it is a parameter-, handle-, or session-related error. This is an implementation choice and the same result can be achieved by using a macro.

```
291  #define g_rcIndexInitializer {  TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,      \
292                                TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,      \
293                                TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,      \
294                                TPM_RC_D, TPM_RC_E, TPM_RC_F }
295  EXTERN const UINT16    g_rcIndex[15]  INITIALIZER(g_rcIndexInitializer);
```

5.9.11.4 g_exclusiveAuditSession

This location holds the session handle for the current exclusive audit session. If there is no exclusive audit session, the location is set to TPM_RH_UNASSIGNED.

```
296  EXTERN TPM_HANDLE    g_exclusiveAuditSession;
```

5.9.11.5 g_time

This is the value in which we keep the current command time. This is initialized at the start of each command. The time is the accumulated time since the last time that the TPM's timer was last powered up. Clock is the accumulated time since the last time that the TPM was cleared. g_time is in mS.

```
297  EXTERN UINT64    g_time;
```

5.9.11.6 g_timeEpoch

This value contains the current clock Epoch. It changes when there is a clock discontinuity. It may be necessary to place this in NV should the timer be able to run across a power down of the TPM but not in all cases (e.g. dead battery). If the nonce is placed in NV, it should go in gp because it should be changing slowly.

```
298  #if CLOCK_STOPS
299  EXTERN CLOCK_NONCE    g_timeEpoch;
300  #else
301  #define g_timeEpoch    gp.timeEpoch
302  #endif
```

5.9.11.7 g_phEnable

This is the platform hierarchy control and determines if the platform hierarchy is available. This value is SET on each TPM2_Startup(). The default value is SET.

```
303  EXTERN BOOL    g_phEnable;
```

5.9.11.8 *g_pcrReConfig*

This value is SET if a TPM2_PCR_Allocate command successfully executed since the last TPM2_Startup(). If so, then the next shutdown is required to be Shutdown(CLEAR).

```
304  EXTERN  BOOL                g_pcrReConfig;
```

5.9.11.9 *g_DRTMHandle*

This location indicates the sequence object handle that holds the DRTM sequence data. When not used, it is set to TPM_RH_UNASSIGNED. A sequence DRTM sequence is started on either _TPM_Init or _TPM_Hash_Start.

```
305  EXTERN  TPMI_DH_OBJECT     g_DRTMHandle;
```

5.9.11.10 *g_DrtmPreStartup*

This value indicates that an H-CRTM occurred after _TPM_Init but before TPM2_Startup(). The define for PRE_STARTUP_FLAG is used to add the *g_DrtmPreStartup* value to *gp_orderlyState* at shutdown. This hack is to avoid adding another NV variable.

```
306  EXTERN  BOOL                g_DrtmPreStartup;
```

5.9.11.11 *g_StartupLocality3*

This value indicates that a TPM2_Startup() occurred at locality 3. Otherwise, it at locality 0. The define for STARTUP_LOCALITY_3 is to indicate that the startup was not at locality 0. This hack is to avoid adding another NV variable.

```
307  EXTERN  BOOL                g_StartupLocality3;
```

5.9.11.12 *TPM_SU_NONE*

Part 2 defines the two shutdown/startup types that may be used in TPM2_Shutdown() and TPM2_Startup(). This additional define is used by the TPM to indicate that no shutdown was received.

NOTE This is a reserved value.

```
308  #define  SU_NONE_VALUE      (0xFFFF)
309  #define  TPM_SU_NONE        (TPM_SU) (SU_NONE_VALUE)
```

5.9.11.13 *TPM_SU_DA_USED*

As with TPM_SU_NONE, this value is added to allow indication that the shutdown was not orderly and that a DA=protected object was reference during the previous cycle.

```
310  #define  SU_DA_USED_VALUE    (SU_NONE_VALUE - 1)
311  #define  TPM_SU_DA_USED      (TPM_SU) (SU_DA_USED_VALUE)
```

5.9.11.14 *Startup Flags*

These flags are included in *gp.orderlyState*. These are hacks and are being used to avoid having to change the layout of *gp*. The PRE_STARTUP_FLAG indicates that a _TPM_Hash_Start/_Data/_End sequence was received after _TPM_Init but before TPM2_StartUp(). STARTUP_LOCALITY_3 indicates

that the last TPM2_Startup() was received at locality 3. These flags are only relevant if after a TPM2_Shutdown(STATE).

```

312 #define PRE_STARTUP_FLAG      0x8000
313 #define STARTUP_LOCALITY_3    0x4000
314
315 #if USE_DA_USED

```

5.9.11.15 *g_daUsed*

This location indicates if a DA-protected value is accessed during a boot cycle. If none has, then there is no need to increment *failedTries* on the next non-orderly startup. This bit is merged with *gp.orderlyState* when *gp.orderly* is set to SU_NONE_VALUE

```

316 EXTERN  BOOL                g_daUsed;
317 #endif

```

5.9.11.16 *g_updateNV*

This flag indicates if NV should be updated at the end of a command. This flag is set to UT_NONE at the beginning of each command in ExecuteCommand(). This flag is checked in ExecuteCommand() after the detailed actions of a command complete. If the command execution was successful and this flag is not UT_NONE, any pending NV writes will be committed to NV. UT_ORDERLY causes any RAM data to be written to the orderly space for staging the write to NV.

```

318 typedef BYTE                UPDATE_TYPE;
319 #define UT_NONE              (UPDATE_TYPE) 0
320 #define UT_NV                 (UPDATE_TYPE) 1
321 #define UT_ORDERLY           (UPDATE_TYPE) (UT_NV + 2)
322 EXTERN UPDATE_TYPE          g_updateNV;

```

5.9.11.17 *g_powerWasLost*

This flag is used to indicate if the power was lost. It is SET in _TPM__Init. This flag is cleared by TPM2_Startup() after all power-lost activities are completed.

NOTE When power is applied, this value can come up as anything. However, _plat__WasPowerLost() will provide the proper indication in that case. So, when power is actually lost, we get the correct answer. When power was not lost, but the power-lost processing has not been completed before the next _TPM__Init(), then the TPM still does the correct thing.

```

323 EXTERN  BOOL                g_powerWasLost;

```

5.9.11.18 *g_clearOrderly*

This flag indicates if the execution of a command should cause the orderly state to be cleared. This flag is set to FALSE at the beginning of each command in ExecuteCommand() and is checked in ExecuteCommand() after the detailed actions of a command complete but before the check of *g_updateNV*. If this flag is TRUE, and the orderly state is not SU_NONE_VALUE, then the orderly state in NV memory will be changed to SU_NONE_VALUE or SU_DA_USED_VALUE.

```

324 EXTERN  BOOL                g_clearOrderly;

```

5.9.11.19 *g_prevOrderlyState*

This location indicates how the TPM was shut down before the most recent TPM2_Startup(). This value, along with the startup type, determines if the TPM should do a TPM Reset, TPM Restart, or TPM Resume.

```
325  EXTERN TPM_SU          g_prevOrderlyState;
```

5.9.11.20 *g_nvOk*

This value indicates if the NV integrity check was successful or not. If not and the failure was severe, then the TPM would have been put into failure mode after it had been re-manufactured. If the NV failure was in the area where the state-save data is kept, then this variable will have a value of FALSE indicating that a TPM2_Startup(CLEAR) is required.

```
326  EXTERN BOOL          g_nvOk;
```

NV availability is sampled as the start of each command and stored here so that its value remains consistent during the command execution

```
327  EXTERN TPM_RC        g_NvStatus;
```

5.9.11.21 *g_platformUnique*

This location contains the unique value(s) used to identify the TPM. It is loaded on every TPM2_Startup(). The first value is used to seed the RNG. The second value is used as a vendor *authValue*. The value used by the RNG would be the value derived from the chip unique value (such as fused) with a dependency on the authorities of the code in the TPM boot path. The second would be derived from the chip unique value with a dependency on the details of the code in the boot path. That is, the first value depends on the various signers of the code and the second depends on what was signed. The TPM vendor should not be able to know the first value but they are expected to know the second.

```
328  EXTERN TPM2B_AUTH      g_platformUniqueAuthorities; // Reserved for RNG
329
330  EXTERN TPM2B_AUTH      g_platformUniqueDetails;    // referenced by VENDOR_PERMANENT
```

5.9.12 Persistent Global Values

5.9.12.1 Description

The values in this section are global values that are persistent across power events. The lifetime of the values determines the structure in which the value is placed.

5.9.12.2 PERSISTENT_DATA

This structure holds the persistent values that only change as a consequence of a specific Protected Capability and are not affected by TPM power events (TPM2_Startup() or TPM2_Shutdown()).

```
331  typedef struct
332  {
333  //*****
334  //      Hierarchy
335  //*****
336  // The values in this section are related to the hierarchies.
337
338      BOOL          disableClear;    // TRUE if TPM2_Clear() using
```



```

339                                     // lockoutAuth is disabled
340
341     // Hierarchy authPolicies
342     TPMI_ALG_HASH      ownerAlg;
343     TPMI_ALG_HASH      endorsementAlg;
344     TPMI_ALG_HASH      lockoutAlg;
345     TPM2B_DIGEST       ownerPolicy;
346     TPM2B_DIGEST       endorsementPolicy;
347     TPM2B_DIGEST       lockoutPolicy;
348
349     // Hierarchy authValues
350     TPM2B_AUTH          ownerAuth;
351     TPM2B_AUTH          endorsementAuth;
352     TPM2B_AUTH          lockoutAuth;
353
354     // Primary Seeds
355     TPM2B_SEED          EPSeed;
356     TPM2B_SEED          SPSeed;
357     TPM2B_SEED          PPSeed;
358     // Note there is a nullSeed in the state_reset memory.
359
360     // Hierarchy proofs
361     TPM2B_PROOF         phProof;
362     TPM2B_PROOF         shProof;
363     TPM2B_PROOF         ehProof;
364     // Note there is a nullProof in the state_reset memory.
365
366     //*****
367     //      Reset Events
368     //*****
369     // A count that increments at each TPM reset and never get reset during the life
370     // time of TPM. The value of this counter is initialized to 1 during TPM
371     // manufacture process. It is used to invalidate all saved contexts after a TPM
372     // Reset.
373     UINT64               totalResetCount;
374
375     // This counter increments on each TPM Reset. The counter is reset by
376     // TPM2_Clear().
377     UINT32               resetCount;
378
379     //*****
380     //      PCR
381     //*****
382     // This structure hold the policies for those PCR that have an update policy.
383     // This implementation only supports a single group of PCR controlled by
384     // policy. If more are required, then this structure would be changed to
385     // an array.
386     #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
387         PCR_POLICY       pcrPolicies;
388     #endif
389
390     // This structure indicates the allocation of PCR. The structure contains a
391     // list of PCR allocations for each implemented algorithm. If no PCR are
392     // allocated for an algorithm, a list entry still exists but the bit map
393     // will contain no SET bits.
394     TPML_PCR_SELECTION  pcrAllocated;
395
396     //*****
397     //      Physical Presence
398     //*****
399     // The PP_LIST type contains a bit map of the commands that require physical
400     // to be asserted when the authorization is evaluated. Physical presence will be
401     // checked if the corresponding bit in the array is SET and if the authorization
402     // handle is TPM_RH_PLATFORM.
403     //
404     // These bits may be changed with TPM2_PP_Commands().

```

```

405     BYTE                ppList[(COMMAND_COUNT + 7) / 8];
406
407 //*****
408 //         Dictionary attack values
409 //*****
410 // These values are used for dictionary attack tracking and control.
411     UINT32              failedTries;           // the current count of unexpired
412                                           // authorization failures
413
414     UINT32              maxTries;             // number of unexpired authorization
415                                           // failures before the TPM is in
416                                           // lockout
417
418     UINT32              recoveryTime;         // time between authorization failures
419                                           // before failedTries is decremented
420
421     UINT32              lockoutRecovery;      // time that must expire between
422                                           // authorization failures associated
423                                           // with lockoutAuth
424
425     BOOL                lockOutAuthEnabled;   // TRUE if use of lockoutAuth is
426                                           // allowed
427
428 //*****
429 //         Orderly State
430 //*****
431 // The orderly state for current cycle
432     TPM_SU              orderlyState;
433
434 //*****
435 //         Command audit values.
436 //*****
437     BYTE                auditCommands[((COMMAND_COUNT + 1) + 7) / 8];
438     TPMI_ALG_HASH        auditHashAlg;
439     UINT64               auditCounter;
440
441 //*****
442 //         Algorithm selection
443 //*****
444 //
445 // The 'algorithmSet' value indicates the collection of algorithms that are
446 // currently in used on the TPM. The interpretation of value is vendor dependent.
447     UINT32              algorithmSet;
448
449 //*****
450 //         Firmware version
451 //*****
452 // The firmwareV1 and firmwareV2 values are instantiated in TimeStamp.c. This is
453 // a scheme used in development to allow determination of the linker build time
454 // of the TPM. An actual implementation would implement these values in a way that
455 // is consistent with vendor needs. The values are maintained in RAM for simplified
456 // access with a master version in NV. These values are modified in a
457 // vendor-specific way.
458
459 // g_firmwareV1 contains the more significant 32-bits of the vendor version number.
460 // In the reference implementation, if this value is printed as a hex
461 // value, it will have the format of YYYYMMDD
462     UINT32              firmwareV1;
463
464 // g_firmwareV1 contains the less significant 32-bits of the vendor version number.
465 // In the reference implementation, if this value is printed as a hex
466 // value, it will have the format of 00 HH MM SS
467     UINT32              firmwareV2;
468 //*****
469 //         Timer Epoch
470 //*****

```

```

471 // timeEpoch contains a nonce that has a vendor-specific size (should not be
472 // less than 8 bytes. This nonce changes when the clock epoch changes. The clock
473 // epoch changes when there is a discontinuity in the timing of the TPM.
474 #if !CLOCK_STOPS
475     CLOCK_NONCE        timeEpoch;
476 #endif
477
478 } PERSISTENT_DATA;
479
480 EXTERN PERSISTENT_DATA gp;

```

5.9.12.3 ORDERLY_DATA

The data in this structure is saved to NV on each TPM2_Shutdown().

```

481 typedef struct orderly_data
482 {
483 //*****
484 //          TIME
485 //*****
486
487 // Clock has two parts. One is the state save part and one is the NV part. The
488 // state save version is updated on each command. When the clock rolls over, the
489 // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
490 // orderly way, then the sClock value is used to initialize the clock. If the
491 // TPM shutdown was not orderly, then the persistent value is used and the safe
492 // attribute is clear.
493
494     UINT64        clock;           // The orderly version of clock
495     TPMI_YES_NO   clockSafe;      // Indicates if the clock value is
496                                     // safe.
497
498     // In many implementations, the quality of the entropy available is not that
499     // high. To compensate, the current value of the drbgState can be saved and
500     // restored on each power cycle. This prevents the internal state from reverting
501     // to the initial state on each power cycle and starting with a limited amount
502     // of entropy. By keeping the old state and adding entropy, the entropy will
503     // accumulate.
504     DRBG_STATE    drbgState;
505
506 // These values allow the accumulation of self-healing time across orderly shutdown
507 // of the TPM.
508 #if ACCUMULATE_SELF_HEAL_TIMER
509     UINT64        selfHealTimer;   // current value of s_selfHealTimer
510     UINT64        lockoutTimer;    // current value of s_lockoutTimer
511     UINT64        time;            // current value of g_time at shutdown
512 #endif // ACCUMULATE_SELF_HEAL_TIMER
513
514 // These are the ACT Timeout values. They are saved with the other timers
515 #define DefineActData(N)  ACT_STATE    ACT_#N;
516     FOR_EACH_ACT(DefineActData)
517
518 // this is the 'signaled' attribute data for all the ACT. It is done this way so
519 // that they can be manipulated by ACT number rather than having to access a
520 // structure.
521     UINT16        signaledACT;
522     UINT16        preservedSignaled;
523 } ORDERLY_DATA;
524
525 #if ACCUMULATE_SELF_HEAL_TIMER
526 #define s_selfHealTimer    go.selfHealTimer
527 #define s_lockoutTimer    go.lockoutTimer
528 #endif // ACCUMULATE_SELF_HEAL_TIMER
529

```

```

530 # define drbgDefault go.drbgState
531
532 EXTERN ORDERLY_DATA    go;

```

5.9.12.4 STATE_CLEAR_DATA

This structure contains the data that is saved on Shutdown(STATE) and restored on Startup(STATE). The values are set to their default settings on any Startup(Clear). In other words, the data is only persistent across TPM Resume. If the comments associated with a parameter indicate a default reset value, the value is applied on each Startup(CLEAR).

```

533 typedef struct state_clear_data
534 {
535 //*****
536 //          Hierarchy Control
537 //*****
538     BOOL          shEnable;           // default reset is SET
539     BOOL          ehEnable;           // default reset is SET
540     BOOL          phEnableNV;         // default reset is SET
541     TPMI_ALG_HASH platformAlg;        // default reset is TPM_ALG_NULL
542     TPM2B_DIGEST  platformPolicy;     // default reset is an Empty Buffer
543     TPM2B_AUTH    platformAuth;       // default reset is an Empty Buffer
544
545 //*****
546 //          PCR
547 //*****
548 // The set of PCR to be saved on Shutdown(STATE)
549     PCR_SAVE      pcrSave;            // default reset is 0...0
550
551 // This structure hold the authorization values for those PCR that have an
552 // update authorization.
553 // This implementation only supports a single group of PCR controlled by
554 // authorization. If more are required, then this structure would be changed to
555 // an array.
556     PCR_AUTHVALUE pcrAuthValues;
557
558 //*****
559 //          ACT
560 //*****
561 #define DefineActPolicySpace(N)      TPMT_HA    act_##N;
562     FOR_EACH_ACT(DefineActPolicySpace)
563
564 } STATE_CLEAR_DATA;
565
566 EXTERN STATE_CLEAR_DATA gc;

```

5.9.12.5 State Reset Data

This structure contains data is that is saved on Shutdown(STATE) and restored on the subsequent Startup(ANY). That is, the data is preserved across TPM Resume and TPM Restart.

If a default value is specified in the comments this value is applied on TPM Reset.

```

567 typedef struct state_reset_data
568 {
569 //*****
570 //          Hierarchy Control
571 //*****
572     TPM2B_PROOF    nullProof;          // The proof value associated with
573                                           // the TPM_RH_NULL hierarchy. The
574                                           // default reset value is from the RNG.
575
576     TPM2B_SEED     nullSeed;           // The seed value for the TPM_RN_NULL

```

```

577                                     // hierarchy. The default reset value
578                                     // is from the RNG.
579
580 //*****
581 //      Context
582 //*****
583 // The 'clearCount' counter is incremented each time the TPM successfully executes
584 // a TPM Resume. The counter is included in each saved context that has 'stClear'
585 // SET (including descendants of keys that have 'stClear' SET). This prevents these
586 // objects from being loaded after a TPM Resume.
587 // If 'clearCount' is at its maximum value when the TPM receives a Shutdown(STATE),
588 // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
589     UINT32          clearCount;          // The default reset value is 0.
590
591     UINT64          objectContextID;     // This is the context ID for a saved
592                                         // object context. The default reset
593                                         // value is 0.
594     CONTEXT_SLOT    contextArray[MAX_ACTIVE_SESSIONS]; // This array contains
595                                         // contains the values used to track
596                                         // the version numbers of saved
597                                         // contexts (see
598                                         // Session.c in for details). The
599                                         // default reset value is {0}.
600
601     CONTEXT_COUNTER contextCounter;      // This is the value from which the
602                                         // 'contextID' is derived. The
603                                         // default reset value is {0}.
604
605 //*****
606 //      Command Audit
607 //*****
608 // When an audited command completes, ExecuteCommand() checks the return
609 // value. If it is TPM_RC_SUCCESS, and the command is an audited command, the
610 // TPM will extend the cpHash and rpHash for the command to this value. If this
611 // digest was the Zero Digest before the cpHash was extended, the audit counter
612 // is incremented.
613
614     TPM2B_DIGEST    commandAuditDigest; // This value is set to an Empty Digest
615                                         // by TPM2_GetCommandAuditDigest() or a
616                                         // TPM Reset.
617
618 //*****
619 //      Boot counter
620 //*****
621
622     UINT32          restartCount;        // This counter counts TPM Restarts.
623                                         // The default reset value is 0.
624
625 //*****
626 //      PCR
627 //*****
628 // This counter increments whenever the PCR are updated. This counter is preserved
629 // across TPM Resume even though the PCR are not preserved. This is because
630 // sessions remain active across TPM Restart and the count value in the session
631 // is compared to this counter so this counter must have values that are unique
632 // as long as the sessions are active.
633 // NOTE: A platform-specific specification may designate that certain PCR changes
634 // do not increment this counter to increment.
635     UINT32          pcrCounter;          // The default reset value is 0.
636
637 #if      ALG_ECC
638
639 //*****
640 //      ECDSA
641 //*****
642     UINT64          commitCounter;       // This counter increments each time

```

```

643                                     // TPM2_Commit() returns
644                                     // TPM_RC_SUCCESS. The default reset
645                                     // value is 0.
646
647     TPM2B_NONCE          commitNonce;    // This random value is used to compute
648                                     // the commit values. The default reset
649                                     // value is from the RNG.
650
651     // This implementation relies on the number of bits in g_commitArray being a
652     // power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
653     BYTE                  commitArray[16]; // The default reset value is {0}.
654
655 #endif // ALG_ECC
656 } STATE_RESET_DATA;
657
658 EXTERN STATE_RESET_DATA gr;

```

5.9.13 NV Layout

The NV data organization is

- 1) a PERSISTENT_DATA structure
- 2) a STATE_RESET_DATA structure
- 3) a STATE_CLEAR_DATA structure
- 4) an ORDERLY_DATA structure
- 5) the user defined NV index space

```

659 #define NV_PERSISTENT_DATA (0)
660 #define NV_STATE_RESET_DATA (NV_PERSISTENT_DATA + sizeof(PERSISTENT_DATA))
661 #define NV_STATE_CLEAR_DATA (NV_STATE_RESET_DATA + sizeof(STATE_RESET_DATA))
662 #define NV_ORDERLY_DATA (NV_STATE_CLEAR_DATA + sizeof(STATE_CLEAR_DATA))
663 #define NV_INDEX_RAM_DATA (NV_ORDERLY_DATA + sizeof(ORDERLY_DATA))
664 #define NV_USER_DYNAMIC (NV_INDEX_RAM_DATA + sizeof(s_indexOrderlyRam))
665 #define NV_USER_DYNAMIC_END NV_MEMORY_SIZE

```

5.9.14 Global Macro Definitions

The NV_READ_PERSISTENT and NV_WRITE_PERSISTENT macros are used to access members of the PERSISTENT_DATA structure in NV.

```

666 #define NV_READ_PERSISTENT(to, from) \
667     NvRead(&to, offsetof(PERSISTENT_DATA, from), sizeof(to))
668
669 #define NV_WRITE_PERSISTENT(to, from) \
670     NvWrite(offsetof(PERSISTENT_DATA, to), sizeof(gp.to), &from)
671
672 #define CLEAR_PERSISTENT(item) \
673     NvClearPersistent(offsetof(PERSISTENT_DATA, item), sizeof(gp.item))
674
675 #define NV_SYNC_PERSISTENT(item) NV_WRITE_PERSISTENT(item, gp.item)

```

At the start of command processing, the index of the command is determined. This index value is used to access the various data tables that contain per-command information. There are multiple options for how the per-command tables can be implemented. This is resolved in GetClosestCommandIndex().

```

676 typedef UINT16      COMMAND_INDEX;
677 #define UNIMPLEMENTED_COMMAND_INDEX ((COMMAND_INDEX) (~0))
678
679 typedef struct _COMMAND_FLAGS_

```



```

680 {
681     unsigned    trialPolicy : 1;    //1) If SET, one of the handles references a
682                                     // trial policy and authorization may be
683                                     // skipped. This is only allowed for a policy
684                                     // command.
685 } COMMAND_FLAGS;

```

This structure is used to avoid having to manage a large number of parameters being passed through various levels of the command input processing.

The following macros are used to define the space for the CP and RP hashes. Space, is provided for each implemented hash algorithm because it is not known what the caller may use.

```

686 #define CP_HASH(HASH, Hash)          TPM2B_##HASH##_DIGEST    Hash##CpHash;
687 #define RP_HASH(HASH, Hash)          TPM2B_##HASH##_DIGEST    Hash##RpHash;
688
689 typedef struct COMMAND
690 {
691     TPM_ST          tag;                // the parsed command tag
692     TPM_CC          code;              // the parsed command code
693     COMMAND_INDEX   index;            // the computed command index
694     UINT32          handleNum;        // the number of entity handles in the
695                                     // handle area of the command
696     TPM_HANDLE      handles[MAX_HANDLE_NUM]; // the parsed handle values
697     UINT32          sessionNum;        // the number of sessions found
698     INT32           parameterSize;    // starts out with the parsed command size
699                                     // and is reduced and values are
700                                     // unmarshaled. Just before calling the
701                                     // command actions, this should be zero.
702                                     // After the command actions, this number
703                                     // should grow as values are marshaled
704                                     // in to the response buffer.
705     INT32           authSize;         // this is initialized with the parsed size
706                                     // of authorizationSize field and should
707                                     // be zero when the authorizations are
708                                     // parsed.
709     BYTE            *parameterBuffer; // input to ExecuteCommand
710     BYTE            *responseBuffer;  // input to ExecuteCommand
711     FOR_EACH_HASH(CP_HASH)           // space for the CP hashes
712     FOR_EACH_HASH(RP_HASH)           // space for the RP hashes
713 } COMMAND;

```

Global string constants for consistency in KDF function calls. These string constants are shared across functions to make sure that they are all using consistent string values.

```

714 #define STRING_INITIALIZER(value)    {{sizeof(value), {value}}}
715 #define TPM2B_STRING(name, value)
716 typedef union name##_ {
717     struct {
718         UINT16  size;
719         BYTE    buffer[sizeof(value)];
720     } t;
721     TPM2B      b;
722 } TPM2B_##name##_;
723 EXTERN const TPM2B_##name##_ name##__INITIALIZER(STRING_INITIALIZER(value));
724 EXTERN const TPM2B_##name##_ *name##_INITIALIZER(&name##_._b)
725
726 TPM2B_STRING(PRIMARY_OBJECT_CREATION, "Primary Object Creation");
727 TPM2B_STRING(CFB_KEY, "CFB");
728 TPM2B_STRING(CONTEXT_KEY, "CONTEXT");
729 TPM2B_STRING(INTEGRITY_KEY, "INTEGRITY");
730 TPM2B_STRING(SECRET_KEY, "SECRET");
731 TPM2B_STRING(SESSION_KEY, "ATH");
732 TPM2B_STRING(STORAGE_KEY, "STORAGE");
733 TPM2B_STRING(XOR_KEY, "XOR");

```



```

734 TPM2B_STRING(COMMIT_STRING, "ECDAA Commit");
735 TPM2B_STRING(DUPLICATE_STRING, "DUPLICATE");
736 TPM2B_STRING(IDENTITY_STRING, "IDENTITY");
737 TPM2B_STRING(OBFUSCATE_STRING, "OBFUSCATE");
738 #if SELF_TEST
739 TPM2B_STRING(OAEP_TEST_STRING, "OAEP Test Value");
740 #endif // SELF_TEST

```

5.9.15 From CryptTest.c

This structure contains the self-test state values for the cryptographic modules.

```

741 EXTERN CRYPTO_SELF_TEST_STATE    g_cryptoSelfTestState;

```

5.9.16 From Manufacture.c

```

742 EXTERN BOOL                      g_manufactured_INITIALIZER(FALSE);

```

This value indicates if a TPM2_Startup commands has been receive since the power on event. This flag is maintained in power simulation module because this is the only place that may reliably set this flag to FALSE.

```

743 EXTERN BOOL                      g_initialized;

```

5.9.17 Private data

5.9.17.1 From SessionProcess.c

```

744 #if defined SESSION_PROCESS_C || defined GLOBAL_C || defined MANUFACTURE_C

```

The following arrays are used to save command sessions information so that the command handle/session buffer does not have to be preserved for the duration of the command. These arrays are indexed by the session index in accordance with the order of sessions in the session area of the command.

Array of the authorization session handles

```

745 EXTERN TPM_HANDLE                s_sessionHandles[MAX_SESSION_NUM];

```

Array of authorization session attributes

```

746 EXTERN TPMA_SESSION             s_attributes[MAX_SESSION_NUM];

```

Array of handles authorized by the corresponding authorization sessions; and if none, then TPM_RH_UNASSIGNED value is used

```

747 EXTERN TPM_HANDLE                s_associatedHandles[MAX_SESSION_NUM];

```

Array of nonces provided by the caller for the corresponding sessions

```

748 EXTERN TPM2B_NONCE              s_nonceCaller[MAX_SESSION_NUM];

```

Array of authorization values (HMAC's or passwords) for the corresponding sessions

```

749 EXTERN TPM2B_AUTH                s_inputAuthValues[MAX_SESSION_NUM];

```

Array of pointers to the SESSION structures for the sessions in a command

```
750  EXTERN SESSION          *s_usedSessions[MAX_SESSION_NUM];
```

Special value to indicate an undefined session index

```
751  #define                UNDEFINED_INDEX      (0xFFFF)
```

Index of the session used for encryption of a response parameter

```
752  EXTERN UINT32          s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```
753  EXTERN UINT32          s_decryptSessionIndex;
```

Index of a session used for audit

```
754  EXTERN UINT32          s_auditSessionIndex;
```

The *cpHash* for command audit

```
755  #ifdef  TPM_CC_GetCommandAuditDigest
756  EXTERN TPM2B_DIGEST      s_cpHashForCommandAudit;
757  #endif
```

Flag indicating if NV update is pending for the *lockOutAuthEnabled* or *failedTries* DA parameter

```
758  EXTERN BOOL            s_DAPendingOnNV;
759
760  #endif // SESSION_PROCESS_C
```

5.9.17.2 From DA.c

```
761  #if defined DA_C || defined GLOBAL_C || defined MANUFACTURE_C
```

This variable holds the accumulated time since the last time that *failedTries* was decremented. This value is in millisecond.

```
762  #if !ACCUMULATE_SELF_HEAL_TIMER
763  EXTERN UINT64            s_selfHealTimer;
```

This variable holds the accumulated time that the *lockoutAuth* has been blocked.

```
764  EXTERN UINT64            s_lockoutTimer;
765  #endif // ACCUMULATE_SELF_HEAL_TIMER
766
767  #endif // DA_C
```

5.9.17.3 From NV.c

```
768  #if defined NV_C || defined GLOBAL_C
```

This marks the end of the NV area. This is a run-time variable as it might not be compile-time constant.

```
769  EXTERN NV_REF            s_evictNvEnd;
```

This space is used to hold the index data for an orderly Index. It also contains the attributes for the index.

```
770  EXTERN BYTE              s_indexOrderlyRam[RAM_INDEX_SPACE]; // The orderly NV Index data
```

This value contains the current max counter value. It is written to the end of allocatable NV space each time an index is deleted or added. This value is initialized on Startup. The indices are searched and the maximum of all the current counter indices and this value is the initial value for this.

```
771  EXTERN UINT64      s_maxCounter;
```

This is space used for the NV Index cache. As with a persistent object, the contents of a referenced index are copied into the cache so that the NV Index memory scanning and data copying can be reduced. Only code that operates on NV Index data should use this cache directly. When that action code runs, *s_lastNvIndex* will contain the index header information. It will have been loaded when the handles were verified.

NOTE An NV index handle can appear in many commands that do not operate on the NV data (e.g. TPM2_StartAuthSession()). However, only one NV Index at a time is ever directly referenced by any command. If that changes, then the NV Index caching needs to be changed to accommodate that. Currently, the code will verify that only one NV Index is referenced by the handles of the command.

```
772  EXTERN      NV_INDEX      s_cachedNvIndex;
773  EXTERN      NV_REF        s_cachedNvRef;
774  EXTERN      BYTE          *s_cachedNvRamRef;
```

Initial NV Index/evict object iterator value

```
775  #define      NV_REF_INIT      (NV_REF) 0xFFFFFFFF
776
777  #endif
```

5.9.17.4 From Object.c

```
778  #if defined OBJECT_C || defined GLOBAL_C
```

This type is the container for an object.

```
779  EXTERN OBJECT      s_objects[MAX_LOADED_OBJECTS];
780
781  #endif // OBJECT_C
```

5.9.17.5 From PCR.c

```
782  #if defined PCR_C || defined GLOBAL_C
```

The following macro is used to define the per-implemented-hash space. This implementation reserves space for all implemented hashes.

```
783  #define PCR_ALL_HASH(HASH, Hash)      BYTE      Hash##Pcr[HASH##_DIGEST_SIZE];
784
785  typedef struct
786  {
787      FOR_EACH_HASH(PCR_ALL_HASH)
788  } PCR;
789
790  typedef struct
791  {
792      unsigned int      stateSave : 1;          // if the PCR value should be
793                                              // saved in state save
794      unsigned int      resetLocality : 5;      // The locality that the PCR
795                                              // can be reset
796      unsigned int      extendLocality : 5;     // The locality that the PCR
797                                              // can be extend
798  } PCR_Attributes;
```

```

799
800 EXTERN PCR          s_pcrs[IMPLEMENTATION_PCR];
801
802 #endif // PCR_C

```

5.9.17.6 From Session.c

```

803 #if defined SESSION_C || defined GLOBAL_C

```

Container for HMAC or policy session tracking information

```

804 typedef struct
805 {
806     BOOL          occupied;
807     SESSION       session;      // session structure
808 } SESSION_SLOT;
809
810 EXTERN SESSION_SLOT s_sessions[MAX_LOADED_SESSIONS];

```

The index in *contextArray* that has the value of the oldest saved session context. When no context is saved, this will have a value that is greater than or equal to MAX_ACTIVE_SESSIONS.

```

811 EXTERN UINT32      s_oldestSavedSession;

```

The number of available session slot openings. When this is 1, a session can't be created or loaded if the GAP is maxed out. The exception is that the oldest saved session context can always be loaded (assuming that there is a space in memory to put it)

```

812 EXTERN int         s_freeSessionSlots;
813
814 #endif // SESSION_C

```

5.9.17.7 From IoBuffers.c

```

815 #if defined IO_BUFFER_C || defined GLOBAL_C

```

Each command function is allowed a structure for the inputs to the function and a structure for the outputs. The command dispatch code unmarshals the input buffer to the command action input structure starting at the first byte of *s_actionIoBuffer*. The value of *s_actionIoAllocation* is the number of UINT64 values allocated. It is used to set the pointer for the response structure. The command dispatch code will marshal the response values into the final output buffer.

```

816 EXTERN UINT64      s_actionIoBuffer[768];      // action I/O buffer
817 EXTERN UINT32      s_actionIoAllocation;      // number of UIN64 allocated for the
818                                                    // action input structure
819 #endif // IO_BUFFER_C

```

5.9.17.8 From TPMFail.c

This value holds the address of the string containing the name of the function in which the failure occurred. This address value is not useful for anything other than helping the vendor to know in which file the failure occurred.

```

820 EXTERN BOOL        g_inFailureMode;           // Indicates that the TPM is in failure mode
821 #if SIMULATION
822 EXTERN BOOL        g_forceFailureMode;       // flag to force failure mode during test
823 #endif
824
825 typedef void (FailFunction) (const char *function, int line, int code);

```

```

826
827 #if defined TPM_FAIL_C || defined GLOBAL_C
828 EXTERN UINT32      s_failFunction;
829 EXTERN UINT32      s_failLine;           // the line in the file at which
830                                           // the error was signaled
831 EXTERN UINT32      s_failCode;          // the error code used
832
833 EXTERN FailFunction *LibFailCallback;
834
835 #endif // TPM_FAIL_C

```

5.9.17.9 From ACT_spt.c

This value is used to indicate if an ACT has been updated since the last TPM2_Startup() (one bit for each ACT). If the ACT is not updated (TPM2_ACT_SetTimeout()) after a startup, then on each TPM2_Shutdown() the TPM will save 1/2 of the current timer value. This prevents an attack on the ACT by saving the counter and then running for a long period of time before doing a TPM Restart. A quick TPM2_Shutdown() after each

```

836 EXTERN UINT16      s_ActUpdated;

```

5.9.17.10 From CommandCodeAttributes.c

This array is instantiated in CommandCodeAttributes.c when it includes CommandCodeAttributes.h. Don't change the extern to EXTERN.

```

837 extern const TPMA_CC      s_ccAttr[];
838 extern const COMMAND_ATTRIBUTES s_commandAttributes[];
839
840 #endif // GLOBAL_H

```

5.10 GpMacros.h

5.10.1 Introduction

This file is a collection of miscellaneous macros.

```

1  #ifndef GP_MACROS_H
2  #define GP_MACROS_H
3
4  #ifndef NULL
5  #define NULL 0
6  #endif
7
8  #include "swap.h"
9  #include "VendorString.h"

```

5.10.2 For Self-test

These macros are used in CryptUtil() to invoke the incremental self test.

```

10 #if SELF_TEST
11 #   define TEST(alg) if(TEST_BIT(alg, g_toTest)) CryptTestAlgorithm(alg, NULL)

```

Use of TPM_ALG_NULL is reserved for RSAEP/RSADP testing. If someone is wanting to test a hash with that value, don't do it.

```

12 #   define TEST_HASH(alg)                                     \
13     if(TEST_BIT(alg, g_toTest)                               \
14       && (alg != TPM_ALG_NULL))                               \
15       CryptTestAlgorithm(alg, NULL)                           \
16 #else                                                         \
17 #   define TEST(alg)                                          \
18 #   define TEST_HASH(alg)                                     \
19 #endif // SELF_TEST

```

5.10.3 For Failures

```

20 #if defined _POSIX
21 #   define FUNCTION_NAME 0
22 #else
23 #   define FUNCTION_NAME __FUNCTION__
24 #endif
25
26 #if !FAIL_TRACE
27 #   define FAIL(errorCode) (TpmFail(errorCode))
28 #   define LOG_FAILURE(errorCode) (TpmLogFailure(errorCode))
29 #else
30 #   define FAIL(errorCode) TpmFail(FUNCTION_NAME, __LINE__, errorCode)
31 #   define LOG_FAILURE(errorCode) TpmLogFailure(FUNCTION_NAME, __LINE__, errorCode)
32 #endif

```

If implementation is using longjmp, then the call to TpmFail() does not return and the compiler will complain about unreachable code that comes after. To allow for not having longjmp, TpmFail() will return and the subsequent code will be executed. This macro accounts for the difference.

```

33 #ifndef NO_LONGJMP
34 #   define FAIL_RETURN(returnCode)
35 #   define TPM_FAIL_RETURN NORETURN void
36 #else
37 #   define FAIL_RETURN(returnCode) return (returnCode)

```

```

38 # define TPM_FAIL_RETURN void
39 #endif

```

This macro tests that a condition is TRUE and puts the TPM into failure mode if it is not. If longjmp is being used, then the FAIL(FATAL_ERROR_) macro makes a call from which there is no return. Otherwise, it returns and the function will exit with the appropriate return code.

```

40 #define REQUIRE(condition, errorCode, returnCode) \
41 { \
42     if(!(condition)) \
43     { \
44         FAIL(FATAL_ERROR_errorCode); \
45         FAIL_RETURN(returnCode); \
46     } \
47 } \
48 #define PARAMETER_CHECK(condition, returnCode) \
49     REQUIRE((condition), PARAMETER, returnCode) \
50 \
51 #if (defined EMPTY_ASSERT) && (EMPTY_ASSERT != NO) \
52 # define pAssert(a) ((void)0) \
53 #else \
54 # define pAssert(a) {if(!(a)) FAIL(FATAL_ERROR_PARAMETER);} \
55 #endif

```

5.10.4 Derived from Vendor-specific values

Values derived from vendor specific settings in TpmProfile.h

```

56 #define PCR_SELECT_MIN ((PLATFORM_PCR+7)/8) \
57 #define PCR_SELECT_MAX ((IMPLEMENTATION_PCR+7)/8) \
58 #define MAX_ORDERLY_COUNT ((1 << ORDERLY_BITS) - 1) \
59 #define RSA_MAX_PRIME (MAX_RSA_KEY_BYTES / 2) \
60 #define RSA_PRIVATE_SIZE (RSA_MAX_PRIME * 5)

```

5.10.5 Compile-time Checks

In some cases, the relationship between two values may be dependent on things that change based on various selections like the chosen cryptographic libraries. It is possible that these selections will result in incompatible settings. These are often detectable by the compiler but it is not always possible to do the check in the preprocessor code. For example, when the check requires use of **sizeof** then the preprocessor can't do the comparison. For these cases, we include a special macro that, depending on the compiler will generate a warning to indicate if the check always passes or always fails because it involves fixed constants. To run these checks, define COMPILER_CHECKS in TpmBuildSwitches.h

```

61 #if COMPILER_CHECKS \
62 # define cAssert pAssert \
63 #else \
64 # define cAssert(value) \
65 #endif

```

This is used commonly in the **Crypt** code as a way to keep listings from getting too long. This is not to save paper but to allow one to see more useful stuff on the screen at any given time.

```

66 #define ERROR_RETURN(returnCode) \
67 { \
68     retVal = returnCode; \
69     goto Exit; \
70 } \
71 #ifndef MAX \
72 # define MAX(a, b) ((a) > (b) ? (a) : (b))

```



```

73 #endif
74 #ifndef MIN
75 # define MIN(a, b) ((a) < (b) ? (a) : (b))
76 #endif
77 #ifndef IsOdd
78 # define IsOdd(a)      ((a) & 1) != 0)
79 #endif
80
81 #ifndef BITS_TO_BYTES
82 # define BITS_TO_BYTES(bits) (((bits) + 7) >> 3)
83 #endif

```

These are defined for use when the size of the vector being checked is known at compile time.

```

84 #define TEST_BIT(bit, vector)    TestBit((bit), (BYTE *)&(vector), sizeof(vector))
85 #define SET_BIT(bit, vector)    SetBit((bit), (BYTE *)&(vector), sizeof(vector))
86 #define CLEAR_BIT(bit, vector)  ClearBit((bit), (BYTE *)&(vector), sizeof(vector))

```

The following definitions are used if they have not already been defined. The defaults for these settings are compatible with ISO/IEC 9899:2011 (E)

```

87 #ifndef LIB_EXPORT
88 # define LIB_EXPORT
89 # define LIB_IMPORT
90 #endif
91 #ifndef NORETURN
92 # define NORETURN _Noreturn
93 #endif
94 #ifndef NOT_REFERENCED
95 # define NOT_REFERENCED(x = x)    ((void) (x))
96 #endif
97
98 #define STD_RESPONSE_HEADER (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC))
99
100 #define JOIN(x, y) x##y
101 #define JOIN3(x, y, z) x##y##z
102 #define CONCAT(x, y) JOIN(x, y)
103 #define CONCAT3(x, y, z) JOIN3(x, y, z)

```

If CONTEXT_INTEGRITY_HASH_ALG is defined, then the vendor is using the old style table. Otherwise, pick the **strongest** implemented hash algorithm as the context hash.

```

104 #ifndef CONTEXT_HASH_ALGORITHM
105 # if defined ALG_SHA3_512 && ALG_SHA3_512 == YES
106 #   define CONTEXT_HASH_ALGORITHM    SHA3_512
107 # elif defined ALG_SHA512 && ALG_SHA512 == YES
108 #   define CONTEXT_HASH_ALGORITHM    SHA512
109 # elif defined ALG_SHA3_384 && ALG_SHA3_384 == YES
110 #   define CONTEXT_HASH_ALGORITHM    SHA3_384
111 # elif defined ALG_SHA384 && ALG_SHA384 == YES
112 #   define CONTEXT_HASH_ALGORITHM    SHA384
113 # elif defined ALG_SHA3_256 && ALG_SHA3_256 == YES
114 #   define CONTEXT_HASH_ALGORITHM    SHA3_256
115 # elif defined ALG_SHA256 && ALG_SHA256 == YES
116 #   define CONTEXT_HASH_ALGORITHM    SHA256
117 # elif defined ALG_SM3_256 && ALG_SM3_256 == YES
118 #   define CONTEXT_HASH_ALGORITHM    SM3_256
119 # elif defined ALG_SHA1 && ALG_SHA1 == YES
120 #   define CONTEXT_HASH_ALGORITHM    SHA1
121 #   endif
122 # define CONTEXT_INTEGRITY_HASH_ALG  CONCAT(TPM_ALG_, CONTEXT_HASH_ALGORITHM)
123 #endif
124
125 #ifndef CONTEXT_INTEGRITY_HASH_SIZE

```

```

126 #define CONTEXT_INTEGRITY_HASH_SIZE CONCAT(CONTEXT_HASH_ALGORITHM, _DIGEST_SIZE)
127 #endif
128 #if ALG_RSA
129 #define RSA_SECURITY_STRENGTH (MAX_RSA_KEY_BITS >= 15360 ? 256 : \
130 (MAX_RSA_KEY_BITS >= 7680 ? 192 : \
131 (MAX_RSA_KEY_BITS >= 3072 ? 128 : \
132 (MAX_RSA_KEY_BITS >= 2048 ? 112 : \
133 (MAX_RSA_KEY_BITS >= 1024 ? 80 : 0))))
134 #else
135 #define RSA_SECURITY_STRENGTH 0
136 #endif // ALG_RSA
137
138 #if ALG_ECC
139 #define ECC_SECURITY_STRENGTH (MAX_ECC_KEY_BITS >= 521 ? 256 : \
140 (MAX_ECC_KEY_BITS >= 384 ? 192 : \
141 (MAX_ECC_KEY_BITS >= 256 ? 128 : 0)))
142 #else
143 #define ECC_SECURITY_STRENGTH 0
144 #endif // ALG_ECC
145
146 #define MAX_ASYM_SECURITY_STRENGTH \
147 MAX(RSA_SECURITY_STRENGTH, ECC_SECURITY_STRENGTH)
148
149 #define MAX_HASH_SECURITY_STRENGTH ((CONTEXT_INTEGRITY_HASH_SIZE * 8) / 2)

```

Unless some algorithm is broken...

```

150 #define MAX_SYM_SECURITY_STRENGTH MAX_SYM_KEY_BITS
151
152 #define MAX_SECURITY_STRENGTH_BITS \
153 MAX(MAX_ASYM_SECURITY_STRENGTH, \
154 MAX(MAX_SYM_SECURITY_STRENGTH, \
155 MAX_HASH_SECURITY_STRENGTH))

```

This is the size that was used before the 1.38 errata requiring that P1.14.4 be followed

```

156 #define PROOF_SIZE CONTEXT_INTEGRITY_HASH_SIZE

```

As required by P1.14.4

```

157 #define COMPLIANT_PROOF_SIZE \
158 (MAX(CONTEXT_INTEGRITY_HASH_SIZE, (2 * MAX_SYM_KEY_BYTES)))

```

As required by P1.14.3.1

```

159 #define COMPLIANT_PRIMARY_SEED_SIZE \
160 BITS_TO_BYTES(MAX_SECURITY_STRENGTH_BITS * 2)

```

This is the pre-errata version

```

161 #ifndef PRIMARY_SEED_SIZE
162 # define PRIMARY_SEED_SIZE PROOF_SIZE
163 #endif
164
165 #if USE_SPEC_COMPLIANT_PROOFS
166 # undef PROOF_SIZE
167 # define PROOF_SIZE COMPLIANT_PROOF_SIZE
168 # undef PRIMARY_SEED_SIZE
169 # define PRIMARY_SEED_SIZE COMPLIANT_PRIMARY_SEED_SIZE
170 #endif // USE_SPEC_COMPLIANT_PROOFS
171
172 #if !SKIP_PROOF_ERRORS
173 # if PROOF_SIZE < COMPLIANT_PROOF_SIZE

```

```

174 #     error "PROOF_SIZE is not compliant with TPM specification"
175 # endif
176 # if PRIMARY_SEED_SIZE < COMPLIANT_PRIMARY_SEED_SIZE
177 #     error Non-compliant PRIMARY_SEED_SIZE
178 # endif
179 #endif // !SKIP_PROOF_ERRORS

```

If CONTEXT_ENCRYPT_ALG is defined, then the vendor is using the old style table

```

180 #if defined CONTEXT_ENCRYPT_ALG
181 # undef CONTEXT_ENCRYPT_ALGORITHM
182 # if CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
183 #     define CONTEXT_ENCRYPT_ALGORITHM    AES
184 # elif CONTEXT_ENCRYPT_ALG == ALG_SM4_VALUE
185 #     define CONTEXT_ENCRYPT_ALGORITHM    SM4
186 # elif CONTEXT_ENCRYPT_ALG == ALG_CAMELLIA_VALUE
187 #     define CONTEXT_ENCRYPT_ALGORITHM    CAMELLIA
188 # elif CONTEXT_ENCRYPT_ALG == ALG_TDES_VALUE
189 #     error Are you kidding?
190 # else
191 #     error Unknown value for CONTEXT_ENCRYPT_ALG
192 # endif // CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
193 #else
194 # define CONTEXT_ENCRYPT_ALG              \
195     CONCAT3(ALG_, CONTEXT_ENCRYPT_ALGORITHM, _VALUE)
196 #endif // CONTEXT_ENCRYPT_ALG
197 #define CONTEXT_ENCRYPT_KEY_BITS          \
198     CONCAT(CONTEXT_ENCRYPT_ALGORITHM, _MAX_KEY_SIZE_BITS)
199 #define CONTEXT_ENCRYPT_KEY_BYTES        ((CONTEXT_ENCRYPT_KEY_BITS+7)/8)

```

This is updated to follow the requirement of P2 that the label not be larger than 32 bytes.

```

200 #ifndef LABEL_MAX_BUFFER
201 #define LABEL_MAX_BUFFER MIN(32, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE))
202 #endif

```

This bit is used to indicate that an authorization ticket expires on TPM Reset and TPM Restart. It is added to the timeout value returned by TPM2_PolicySigned() and TPM2_PolicySecret() and used by TPM2_PolicyTicket(). The timeout value is relative to Time (g_time). Time is reset whenever the TPM loses power and cannot be moved forward by the user (as can Clock). g_time is a 64-bit value expressing time in ms. Stealing the MSb for a flag means that the TPM needs to be reset at least once every 292,471,208 years rather than once every 584,942,417 years.

```

203 #define EXPIRATION_BIT ((UINT64)1 << 63)

```

Check for consistency of the bit ordering of bit fields

```

204 #if BIG_ENDIAN_TPM && MOST_SIGNIFICANT_BIT_0 && USE_BIT_FIELD_STRUCTURES
205 #     error "Settings not consistent"
206 #endif

```

These macros are used to handle the variation in handling of bit fields. If

```

207 #if USE_BIT_FIELD_STRUCTURES // The default, old version, with bit fields
208 # define IS_ATTRIBUTE(a, type, b)      ((a.b) != 0)
209 # define SET_ATTRIBUTE(a, type, b)     (a.b = SET)
210 # define CLEAR_ATTRIBUTE(a, type, b)   (a.b = CLEAR)
211 # define GET_ATTRIBUTE(a, type, b)     (a.b)
212 # define TPMA_ZERO_INITIALIZER()      {0}
213 #else
214 # define IS_ATTRIBUTE(a, type, b)      ((a & type##_##b) != 0)
215 # define SET_ATTRIBUTE(a, type, b)     (a |= type##_##b)

```

```

216 #   define CLEAR_ATTRIBUTE(a, type, b)      (a &= ~type##_##b)
217 #   define GET_ATTRIBUTE(a, type, b)        \
218     (type)((a & type##_##b) >> type##_##b##_SHIFT)
219 #   define TPMA_ZERO_INITIALIZER()          (0)
220 #endif
221
222 #define VERIFY(_X) if(!(_X)) goto Error

```

These macros determine if the values in this file are referenced or instanced. Global.c defines GLOBAL_C so all the values in this file will be instanced in Global.obj. For all other files that include this file, the values will simply be external references. For constants, there can be an initializer.

```

223 #ifndef GLOBAL_C
224 #define EXTERN
225 #define INITIALIZER(_value_) = _value_
226 #else
227 #define EXTERN extern
228 #define INITIALIZER(_value_)
229 #endif

```

This macro will create an OID. All OIDs are in DER form with a first octet of 0x06 indicating an OID followed by an octet indicating the number of octets in the rest of the OID. This allows a user of this OID to know how much/little to copy.

```

230 #define MAKE_OID(NAME) \
231     EXTERN const BYTE OID##NAME[] INITIALIZER({OID##NAME##_VALUE})

```

This definition is moved from TpmProfile.h because it is not actually vendor- specific. It has to be the same size as the *sequence* parameter of a TPMS_CONTEXT and that is a UINT64. So, this is an invariant value

```

232 #define CONTEXT_COUNTER      UINT64
233
234 #endif // GP_MACROS_H

```

5.11 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

```

DRTM functions

```

7  #include "_TPM_Hash_Start_fp.h"
8  #include "_TPM_Hash_Data_fp.h"
9  #include "_TPM_Hash_End_fp.h"

```

Internal subsystem functions

```

10 #include "Object_fp.h"
11 #include "Context_spt_fp.h"
12 #include "Object_spt_fp.h"
13 #include "Entity_fp.h"
14 #include "Session_fp.h"
15 #include "Hierarchy_fp.h"
16 #include "NvReserved_fp.h"
17 #include "NvDynamic_fp.h"
18 #include "NV_spt_fp.h"
19 #include "ACT_spt_fp.h"
20 #include "PCR_fp.h"
21 #include "DA_fp.h"
22 #include "TpmFail_fp.h"
23 #include "SessionProcess_fp.h"

```

Internal support functions

```

24 #include "CommandCodeAttributes_fp.h"
25 #include "Marshal.h"
26 #include "Time_fp.h"
27 #include "Locality_fp.h"
28 #include "PP_fp.h"
29 #include "CommandAudit_fp.h"
30 #include "Manufacture_fp.h"
31 #include "Handle_fp.h"
32 #include "Power_fp.h"
33 #include "Response_fp.h"
34 #include "CommandDispatcher_fp.h"
35
36 #ifdef CC_AC_Send
37 #   include "AC_spt_fp.h"
38 #endif // CC_AC_Send

```

Miscellaneous

```

39 #include "Bits_fp.h"
40 #include "AlgorithmCap_fp.h"
41 #include "PropertyCap_fp.h"
42 #include "IoBuffers_fp.h"
43 #include "Memory_fp.h"
44 #include "ResponseCodeProcessing_fp.h"

```

Internal cryptographic functions

```

45 #include "BnConvert_fp.h"

```

```
46 #include "BnMath_fp.h"
47 #include "BnMemory_fp.h"
48 #include "Ticket_fp.h"
49 #include "CryptUtil_fp.h"
50 #include "CryptHash_fp.h"
51 #include "CryptSym_fp.h"
52 #include "CryptDes_fp.h"
53 #include "CryptPrime_fp.h"
54 #include "CryptRand_fp.h"
55 #include "CryptSelfTest_fp.h"
56 #include "MathOnByteBuffers_fp.h"
57 #include "CryptSym_fp.h"
58 #include "AlgorithmTests_fp.h"
59
60 #if ALG_RSA
61 #include "CryptRsa_fp.h"
62 #include "CryptPrimeSieve_fp.h"
63 #endif
64
65 #if ALG_ECC
66 #include "CryptEccMain_fp.h"
67 #include "CryptEccSignature_fp.h"
68 #include "CryptEccKeyExchange_fp.h"
69 #include "CryptEccCrypt_fp.h"
70 #endif
71
72 #if CC_MAC || CC_MAC_Start
73 #   include "CryptSmac_fp.h"
74 #   if ALG_CMAC
75 #       include "CryptCmac_fp.h"
76 #   endif
77 #endif
78
79 Support library
80
81 #include "SupportLibraryFunctionPrototypes_fp.h"
82
83 Linkage to platform functions
84
85 #include "Platform_fp.h"
86 #endif
```

5.12 LibSupport.h

This header file is used to select the library code that gets included in the TPM build.

```

1  #ifndef _LIB_SUPPORT_H_
2  #define _LIB_SUPPORT_H_
3
4  #ifndef RADIX_BITS
5  #   if defined(__x86_64__) || defined(__x86_64)           \
6      || defined(__amd64__) || defined(__amd64)           \
7      || defined(__WIN64__) || defined(__M_X64)           \
8      || defined(__M_ARM64__) || defined(__aarch64__)      \
9      || defined(__PPC64__) || defined(__s390x__)          \
10     || defined(__powerpc64__) || defined(__ppc64__)       \
11     #       define RADIX_BITS 64
12     #   elif defined(__i386__) || defined(__i386) || defined(i386) \
13         || defined(__WIN32__) || defined(__M_I386)           \
14         || defined(__M_ARM) || defined(__arm__) || defined(__thumb__) \
15     #       define RADIX_BITS 32
16     #   else
17     #       error Unable to determine RADIX_BITS from compiler environment
18     #   endif
19 #endif // RADIX_BITS

```

These macros use the selected libraries to the proper include files.

```

20 #define LIB_QUOTE(_STRING_) #_STRING_
21 #define LIB_INCLUDE2(_LIB_, _TYPE_) LIB_QUOTE(_LIB_/TpmTo##_LIB_##_TYPE_.h)
22 #define LIB_INCLUDE(_LIB_, _TYPE_) LIB_INCLUDE2(_LIB_, _TYPE_)

```

Include the options for hashing and symmetric. Defer the load of the math package Until the bignum parameters are defined.

```

23 #include LIB_INCLUDE(SYM_LIB, Sym)
24 #include LIB_INCLUDE(HASH_LIB, Hash)
25
26 #undef MIN
27 #undef MAX
28
29 #endif // _LIB_SUPPORT_H_

```

5.13 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 #endif // _MIN_MAX_H_

```


5.14 NV.h

5.14.1 Index Type Definitions

These definitions allow the same code to be used pre and post 1.21. The main action is to redefine the index type values from the bit values. Use TPM_NT_ORDINARY to indicate if the TPM_NT type is defined

```
1  #ifndef    _NV_H_
2  #define    _NV_H_
3
4  #ifdef    TPM_NT_ORDINARY
```

If TPM_NT_ORDINARY is defined, then the TPM_NT field is present in a TPMA_NV

```
5  #   define GET_TPM_NT(attributes) GET_ATTRIBUTE(attributes, TPMA_NV, TPM_NT)
6  #else
```

If TPM_NT_ORDINARY is not defined, then need to synthesize it from the attributes

```
7  #   define GetNv_TPM_NV(attributes) \
8      ( IS_ATTRIBUTE(attributes, TPMA_NV, COUNTER) \
9      + (IS_ATTRIBUTE(attributes, TPMA_NV, BITS) << 1) \
10     + (IS_ATTRIBUTE(attributes, TPMA_NV, EXTEND) << 2) \
11     )
12 #   define TPM_NT_ORDINARY (0)
13 #   define TPM_NT_COUNTER (1)
14 #   define TPM_NT_BITS (2)
15 #   define TPM_NT_EXTEND (4)
16 #endif
```

5.14.2 Attribute Macros

These macros are used to isolate the differences in the way that the index type changed in version 1.21 of the specification

```
17 #   define IsNvOrdinaryIndex(attributes) \
18     (GET_TPM_NT(attributes) == TPM_NT_ORDINARY)
19
20 #   define IsNvCounterIndex(attributes) \
21     (GET_TPM_NT(attributes) == TPM_NT_COUNTER)
22
23 #   define IsNvBitsIndex(attributes) \
24     (GET_TPM_NT(attributes) == TPM_NT_BITS)
25
26 #   define IsNvExtendIndex(attributes) \
27     (GET_TPM_NT(attributes) == TPM_NT_EXTEND)
28
29 #ifdef TPM_NT_PIN_PASS
30 #   define IsNvPinPassIndex(attributes) \
31     (GET_TPM_NT(attributes) == TPM_NT_PIN_PASS)
32 #endif
33
34 #ifdef TPM_NT_PIN_FAIL
35 #   define IsNvPinFailIndex(attributes) \
36     (GET_TPM_NT(attributes) == TPM_NT_PIN_FAIL)
37 #endif
38
39 typedef struct {
40     UINT32    size;
41     TPM_HANDLE handle;
```

```

42 } NV_ENTRY_HEADER;
43
44 #define NV_EVICT_OBJECT_SIZE \
45     (sizeof(UINT32) + sizeof(TPM_HANDLE) + sizeof(OBJECT))
46
47 #define NV_INDEX_COUNTER_SIZE \
48     (sizeof(UINT32) + sizeof(NV_INDEX) + sizeof(UINT64))
49
50 #define NV_RAM_INDEX_COUNTER_SIZE \
51     (sizeof(NV_RAM_HEADER) + sizeof(UINT64))
52
53 typedef struct {
54     UINT32      size;
55     TPM_HANDLE  handle;
56     TPMA_NV     attributes;
57 } NV_RAM_HEADER;

```

Defines the end-of-list marker for NV. The list terminator is a UINT32 of zero, followed by the current value of `s_maxCounter` which is a 64-bit value. The structure is defined as an array of 3 UINT32 values so that there is no padding between the UINT32 list end marker and the UINT64 `maxCounter` value.

```

58 typedef UINT32 NV_LIST_TERMINATOR[3];

```

5.14.3 Orderly RAM Values

The following defines are for accessing orderly RAM values.

This is the initialize for the RAM reference iterator.

```

59 #define NV_RAM_REF_INIT 0

```

This is the starting address of the RAM space used for orderly data

```

60 #define RAM_ORDERLY_START \
61     (&s_indexOrderlyRam[0])

```

This is the offset within NV that is used to save the orderly data on an orderly shutdown.

```

62 #define NV_ORDERLY_START \
63     (NV_INDEX_RAM_DATA)

```

This is the end of the orderly RAM space. It is actually the first byte after the last byte of orderly RAM data

```

64 #define RAM_ORDERLY_END \
65     (RAM_ORDERLY_START + sizeof(s_indexOrderlyRam))

```

This is the end of the orderly space in NV memory. As with `RAM_ORDERLY_END`, it is actually the offset of the first byte after the end of the NV orderly data.

```

66 #define NV_ORDERLY_END \
67     (NV_ORDERLY_START + sizeof(s_indexOrderlyRam))

```

Macro to check that an orderly RAM address is with range.

```

68 #define ORDERLY_RAM_ADDRESS_OK(start, offset) \
69     ((start >= RAM_ORDERLY_START) && ((start + offset - 1) < RAM_ORDERLY_END))
70
71 #define RETURN_IF_NV_IS_NOT_AVAILABLE \
72 { \
73     if(g_NvStatus != TPM_RC_SUCCESS) \
74         return g_NvStatus; \

```

75 }

Routinely have to clear the orderly flag and fail if the NV is not available so that it can be cleared.

```

76 #define RETURN_IF_ORDERLY          \
77 {                                  \
78     if(NvClearOrderly() != TPM_RC_SUCCESS) \
79         return g_NvStatus;          \
80 }
81 #define NV_IS_AVAILABLE      (g_NvStatus == TPM_RC_SUCCESS)
82
83 #define IS_ORDERLY(value)    (value < SU_DA_USED_VALUE)
84
85 #define NV_IS_ORDERLY      (IS_ORDERLY(gp.orderlyState))

```

Macro to set the NV_UPDATE_TYPE. This deals with the fact that the update is possibly a combination of UT_NV and UT_ORDERLY.

```

86 #define SET_NV_UPDATE(type)      g_updateNV |= (type)
87
88 #endif // _NV_H_

```

5.15 TPMB.h

This file contains extra TPM2B structures

```
1  #ifndef _TPMB_H
2  #define _TPMB_H
```

TPM2B Types

```
3  typedef struct {
4      UINT16      size;
5      BYTE        buffer[1];
6  } TPM2B, *P2B;
7  typedef const TPM2B      *PC2B;
```

This macro helps avoid having to type in the structure in order to create a new TPM2B type that is used in a function.

```
8  #define TPM2B_TYPE(name, bytes)
9      typedef union {
10         struct {
11             UINT16  size;
12             BYTE    buffer[ (bytes) ];
13         } t;
14         TPM2B      b;
15     } TPM2B_##name
```

This macro defines a TPM2B with a constant character value. This macro sets the size of the string to the size minus the terminating zero byte. This lets the user of the label add their terminating 0. This method is chosen so that existing code that provides a label will continue to work correctly.

Macro to instance and initialize a TPM2B value

```
16 #define TPM2B_INIT(TYPE, name) \
17     TPM2B_##TYPE      name = {sizeof(name.t.buffer), {0}}
18 #define TPM2B_BYTE_VALUE(bytes) TPM2B_TYPE(bytes##_BYTE_VALUE, bytes)
19
20 #endif
```

5.16 Tpm.h

Root header file for building any TPM.lib code

```
1  #ifndef      _TPM_H_
2  #define      _TPM_H_
3
4  #include "TpmBuildSwitches.h"
5  #include "BaseTypes.h"
6  #include "TPMB.h"
7  #include "MinMax.h"
8
9  #include "TpmProfile.h"
10 #include "TpmAlgorithmDefines.h"
11 #include "LibSupport.h"           // Types from the library. These need to come before
12                                   // Global.h because some of the structures in
13                                   // that file depend on the structures used by the
14                                   // cryptographic libraries.
15 #include "GpMacros.h"             // Define additional macros
16 #include "Global.h"               // Define other TPM types
17 #include "InternalRoutines.h"     // Function prototypes
18
19 #endif // _TPM_H_
```

5.17 TpmBuildSwitches.h

This file contains the build switches. This contains switches for multiple versions of the crypto-library so some may not apply to your environment.

The switches are guarded so that they can either be set on the command line or set here. If the switch is listed on the command line (-DSOME_SWITCH) with NO setting, then the switch will be set to YES. If the switch setting is not on the command line or if the setting is other than YES or NO, then the switch will be set to the default value. The default can either be YES or NO as indicated on each line where the default is selected.

A caution: do not try to test these macros by inserting #defines in this file. For some curious reason, a variable set on the command line with no setting will have a value of 1. An **#if SOME_VARIABLE** will work if the variable is not defined or is defined on the command line with no initial setting. However, a **#define SOME_VARIABLE** is a null string and when used in **#if SOME_VARIABLE** will not be a proper expression. If you want to test various switches, either use the command line or change the default.

```
1  #ifndef _TPM_BUILD_SWITCHES_H_
2  #define _TPM_BUILD_SWITCHES_H_
3
4  #undef YES
5  #define YES 1
6  #undef NO
7  #define NO 0
```

Allow the command line to specify a **profile** file

```
8  #ifdef PROFILE
9  #   define PROFILE_QUOTE(a) #a
10 #   define PROFILE_INCLUDE(a) PROFILE_QUOTE(a)
11 #   include PROFILE_INCLUDE(PROFILE)
12 #endif
```

Need an unambiguous definition for DEBUG. Do not change this

```
13 #ifndef DEBUG
14 #   ifdef NDEBUG
15 #       define DEBUG    NO
16 #   else
17 #       define DEBUG    YES
18 #   endif
19 #elif (DEBUG != NO) && (DEBUG != YES)
20 #   undef  DEBUG
21 #   define DEBUG                YES    // Default: Either YES or NO
22 #endif
23
24 #include "CompilerDependencies.h"
```

This definition is required for the re-factored code

```
25 #if (!defined USE_BN_ECC_DATA)
26     || ((USE_BN_ECC_DATA != NO) && (USE_BN_ECC_DATA != YES))
27 #   undef  USE_BN_ECC_DATA
28 #   define USE_BN_ECC_DATA                YES    // Default: Either YES or NO
29 #endif
```

The SIMULATION switch allows certain other macros to be enabled. The things that can be enabled in a simulation include key caching, reproducible **random** sequences, instrumentation of the RSA key generation process, and certain other debug code. SIMULATION Needs to be defined as either YES or NO. This grouping of macros will make sure that it is set correctly. A simulated TPM would include a

Virtual TPM. The interfaces for a Virtual TPM should be modified from the standard ones in the Simulator project.

If SIMULATION is in the compile parameters without modifiers, make SIMULATION == YES

```

30 #if !(defined SIMULATION) || ((SIMULATION != NO) && (SIMULATION != YES))
31 #   undef   SIMULATION
32 #   define  SIMULATION          YES      // Default: Either YES or NO
33 #endif

```

Define this to run the function that checks the compatibility between the chosen big number math library and the TPM code. Not all ports use this.

```

34 #if !(defined LIBRARY_COMPATIBILITY_CHECK)                                \
35     || (( LIBRARY_COMPATIBILITY_CHECK != NO)                             \
36         && (LIBRARY_COMPATIBILITY_CHECK != YES))                          \
37 #   undef   LIBRARY_COMPATIBILITY_CHECK
38 #   define  LIBRARY_COMPATIBILITY_CHECK YES      // Default: Either YES or NO
39 #endif
40
41 #if !(defined FIPS_COMPLIANT) || ((FIPS_COMPLIANT != NO) && (FIPS_COMPLIANT != YES))
42 #   undef   FIPS_COMPLIANT
43 #   define  FIPS_COMPLIANT          YES      // Default: Either YES or NO
44 #endif

```

Definition to allow alternate behavior for non-orderly startup. If there is a chance that the TPM could not update *failedTries*

```

45 #if !(defined USE_DA_USED) || ((USE_DA_USED != NO) && (USE_DA_USED != YES))
46 #   undef   USE_DA_USED
47 #   define  USE_DA_USED          YES      // Default: Either YES or NO
48 #endif

```

Define TABLE_DRIVEN_DISPATCH to use tables rather than case statements for command dispatch and handle unmarshaling

```

49 #if !(defined TABLE_DRIVEN_DISPATCH)                                \
50     || ((TABLE_DRIVEN_DISPATCH != NO) && (TABLE_DRIVEN_DISPATCH != YES))      \
51 #   undef   TABLE_DRIVEN_DISPATCH
52 #   define  TABLE_DRIVEN_DISPATCH YES      // Default: Either YES or NO
53 #endif

```

This switch is used to enable the self-test capability in AlgorithmTests.c

```

54 #if !(defined SELF_TEST) || ((SELF_TEST != NO) && (SELF_TEST != YES))
55 #   undef   SELF_TEST
56 #   define  SELF_TEST          YES      // Default: Either YES or NO
57 #endif

```

Enable the generation of RSA primes using a sieve.

```

58 #if !(defined RSA_KEY_SIEVE) || ((RSA_KEY_SIEVE != NO) && (RSA_KEY_SIEVE != YES))
59 #   undef   RSA_KEY_SIEVE
60 #   define  RSA_KEY_SIEVE          YES      // Default: Either YES or NO
61 #endif

```

Enable the instrumentation of the sieve process. This is used to tune the sieve variables.

```

62 #if RSA_KEY_SIEVE && SIMULATION
63 #   if !(defined RSA_INSTRUMENT)
64 #       || ((RSA_INSTRUMENT != NO) && (RSA_INSTRUMENT != YES))
65 #       undef   RSA_INSTRUMENT

```



```

66 # define RSA_INSTRUMENT NO // Default: Either YES or NO
67 # endif
68 #endif

```

This switch enables the RNG state save and restore

```

69 #if !(defined _DRBG_STATE_SAVE) \
70 || ((_DRBG_STATE_SAVE != NO) && (_DRBG_STATE_SAVE != YES))
71 # undef _DRBG_STATE_SAVE
72 # define _DRBG_STATE_SAVE YES // Default: Either YES or NO
73 #endif

```

Switch added to support packed lists that leave out space associated with unimplemented commands. Comment this out to use linear lists.

NOTE if vendor specific commands are present, the associated list is always in compressed form.

```

74 #if !(defined COMPRESSED_LISTS) \
75 || ((COMPRESSED_LISTS != NO) && (COMPRESSED_LISTS != YES))
76 # undef COMPRESSED_LISTS
77 # define COMPRESSED_LISTS YES // Default: Either YES or NO
78 #endif

```

This switch indicates where clock epoch value should be stored. If this value defined, then it is assumed that the timer will change at any time so the nonce should be a random number kept in RAM. When it is not defined, then the timer only stops during power outages.

```

79 #if !(defined CLOCK_STOPS) || ((CLOCK_STOPS != NO) && (CLOCK_STOPS != YES))
80 # undef CLOCK_STOPS
81 # define CLOCK_STOPS NO // Default: Either YES or NO
82 #endif

```

This switch allows use of #defines in place of pass-through marshaling or unmarshaling code. A pass-through function just calls another function to do the required function and does no parameter checking of its own. The table-driven dispatcher calls directly to the lowest level marshaling/unmarshaling code and by-passes any pass-through functions.

```

83 #if (defined USE_MARSHALING_DEFINES) && (USE_MARSHALING_DEFINES != NO)
84 # undef USE_MARSHALING_DEFINES
85 # define USE_MARSHALING_DEFINES YES
86 #else
87 # define USE_MARSHALING_DEFINES YES // Default: Either YES or NO
88 #endif

```

The switches in this group can only be enabled when doing debug during simulation

```

89 #if SIMULATION && DEBUG

```

This forces the use of a smaller context slot size. This reduction reduces the range of the epoch allowing the tester to force the epoch to occur faster than the normal defined in TpmProfile.h

```

90 # if !(defined CONTEXT_SLOT)
91 # define CONTEXT_SLOT UINT8
92 # endif

```

Enables use of the key cache. Default is YES

```

93 # if !(defined USE_RSA_KEY_CACHE) \
94 || ((USE_RSA_KEY_CACHE != NO) && (USE_RSA_KEY_CACHE != YES))
95 # undef USE_RSA_KEY_CACHE
96 # define USE_RSA_KEY_CACHE YES // Default: Either YES or NO

```

```
97 # endif
```

Enables use of a file to store the key cache values so that the TPM will start faster during debug. Default for this is YES

```
98 # if USE_RSA_KEY_CACHE
99 #     if !(defined USE_KEY_CACHE_FILE)
100         || ((USE_KEY_CACHE_FILE != NO) && (USE_KEY_CACHE_FILE != YES))
101 #         undef USE_KEY_CACHE_FILE
102 #         define USE_KEY_CACHE_FILE YES // Default: Either YES or NO
103 #     endif
104 # else
105 #         undef USE_KEY_CACHE_FILE
106 #         define USE_KEY_CACHE_FILE NO
107 #     endif // USE_RSA_KEY_CACHE
```

This provides fixed seeding of the RNG when doing debug on a simulator. This should allow consistent results on test runs as long as the input parameters to the functions remains the same. There is no default value.

```
108 # if !(defined USE_DEBUG_RNG) || ((USE_DEBUG_RNG != NO) && (USE_DEBUG_RNG != YES))
109 #     undef USE_DEBUG_RNG
110 #     define USE_DEBUG_RNG YES // Default: Either YES or NO
111 # endif
```

Do not change these. They are the settings needed when not doing a simulation and not doing debug. Can't use the key cache except during debug. Otherwise, all of the key values end up being the same

```
112 #else
113 # define USE_RSA_KEY_CACHE NO
114 # define USE_RSA_KEY_CACHE_FILE NO
115 # define USE_DEBUG_RNG NO
116 #endif // DEBUG && SIMULATION
117
118 #if DEBUG
```

In some cases, the relationship between two values may be dependent on things that change based on various selections like the chosen cryptographic libraries. It is possible that these selections will result in incompatible settings. These are often detectable by the compiler but it is not always possible to do the check in the preprocessor code. For example, when the check requires use of `sizeof()` then the preprocessor can't do the comparison. For these cases, we include a special macro that, depending on the compiler will generate a warning to indicate if the check always passes or always fails because it involves fixed constants. To run these checks, define `COMPILER_CHECKS`.

```
119 # if !(defined COMPILER_CHECKS)
120     || ((COMPILER_CHECKS != NO) && (COMPILER_CHECKS != YES))
121 #     undef COMPILER_CHECKS
122 #     define COMPILER_CHECKS NO // Default: Either YES or NO
123 # endif
```

Some of the values (such as sizes) are the result of different options set in `TpmProfile.h`. The combination might not be consistent. A function is defined (`TpmSizeChecks()`) that is used to verify the sizes at run time. To enable the function, define this parameter.

```
124 # if !(defined RUNTIME_SIZE_CHECKS)
125     || ((RUNTIME_SIZE_CHECKS != NO) && (RUNTIME_SIZE_CHECKS != YES))
126 #     undef RUNTIME_SIZE_CHECKS
127 #     define RUNTIME_SIZE_CHECKS YES // Default: Either YES or NO
128 # endif
```

If doing debug, can set the DRBG to print out the intermediate test values. Before enabling this, make sure that the dbgDumpMemBlock() function has been added someplace (preferably, somewhere in CryptRand.c)

```

129 #   if !(defined DRBG_DEBUG_PRINT)                                \
130     || ((DRBG_DEBUG_PRINT != NO) && (DRBG_DEBUG_PRINT != YES))
131 #       undef    DRBG_DEBUG_PRINT
132 #       define   DRBG_DEBUG_PRINT          NO        // Default: Either YES or NO
133 #   endif

```

If an assertion event it not going to produce any trace information (function and line number) then make FAIL_TRACE == NO

```

134 #   if !(defined FAIL_TRACE) || ((FAIL_TRACE != NO) && (FAIL_TRACE != YES))
135 #       undef    FAIL_TRACE
136 #       define   FAIL_TRACE                YES        // Default: Either YES or NO
137 #   endif
138
139 #endif // DEBUG

```

Indicate if the implementation is going to give lockout time credit for time up to the last orderly shutdown.

```

140 #if !(defined ACCUMULATE_SELF_HEAL_TIMER)                            \
141     || ((ACCUMULATE_SELF_HEAL_TIMER != NO) && (ACCUMULATE_SELF_HEAL_TIMER != YES))
142 #   undef    ACCUMULATE_SELF_HEAL_TIMER
143 #   define   ACCUMULATE_SELF_HEAL_TIMER  YES        // Default: Either YES or NO
144 #endif

```

Indicates if the implementation is to compute the sizes of the proof and primary seed size values based on the implemented algorithms.

```

145 #if !(defined USE_SPEC_COMPLIANT_PROOFS)                            \
146     || ((USE_SPEC_COMPLIANT_PROOFS != NO) && (USE_SPEC_COMPLIANT_PROOFS != YES))
147 #   undef    USE_SPEC_COMPLIANT_PROOFS
148 #   define   USE_SPEC_COMPLIANT_PROOFS  YES        // Default: Either YES or NO
149 #endif

```

Comment this out to allow compile to continue even though the chosen proof values do not match the compliant values. This is written so that someone would have to proactively ignore errors.

```

150 #if !(defined SKIP_PROOF_ERRORS)                                    \
151     || ((SKIP_PROOF_ERRORS != NO) && (SKIP_PROOF_ERRORS != YES))
152 #   undef    SKIP_PROOF_ERRORS
153 #   define   SKIP_PROOF_ERRORS          NO        // Default: Either YES or NO
154 #endif

```

This define is used to eliminate the use of bit-fields. It can be enabled for big- or little-endian machines. For big-endian architectures that numbers bits in registers from left to right (MSb0) this must be enabled. Little-endian machines number from right to left with the least significant bit having assigned a bit number of 0. These are LSb0 machines (they are also little-endian so they are also least-significant byte 0 (LSB0) machines. Big-endian (MSB0) machines may number in either direction (MSb0 or LSb0). For an MSB0+MSb0 machine this value is required to be NO

```

155 #if !(defined USE_BIT_FIELD_STRUCTURES)                            \
156     || ((USE_BIT_FIELD_STRUCTURES != NO) && (USE_BIT_FIELD_STRUCTURES != YES))
157 #   undef    USE_BIT_FIELD_STRUCTURES
158 #   define   USE_BIT_FIELD_STRUCTURES    NO        // Default: Either YES or NO
159 #endif

```

This define is used to control the debug for the CertifyX509() command.

```

160  #if !(defined CERTIFYX509_DEBUG)                                \
161      || ((CERTIFYX509_DEBUG != NO) && (CERTIFYX509_DEBUG != YES))
162  #   undef CERTIFYX509_DEBUG
163  #   define CERTIFYX509_DEBUG YES                                // Default: Either YES or NO
164  #endif

```

This define is used to enable the new table-driven marshaling code.

```

165  #if !(defined TABLE_DRIVEN_MARSHAL)                            \
166      || ((TABLE_DRIVEN_MARSHAL != NO) && (TABLE_DRIVEN_MARSHAL != YES))
167  #   undef TABLE_DRIVEN_MARSHAL
168  #   define TABLE_DRIVEN_MARSHAL NO    // Default: Either YES or NO
169  #endif

```

Change these definitions to turn all algorithms or commands ON or OFF. That is, to turn all algorithms on, set ALG_NO to YES. This is mostly useful as a debug feature.

```

170  #define      ALG_YES      YES
171  #define      ALG_NO      NO
172  #define      CC_YES      YES
173  #define      CC_NO      NO
174
175  #endif // _TPM_BUILD_SWITCHES_H_

```

5.18 TpmError.h

```
1  #ifndef _TPM_ERROR_H
2  #define _TPM_ERROR_H
3
4  #define FATAL_ERROR_ALLOCATION (1)
5  #define FATAL_ERROR_DIVIDE_ZERO (2)
6  #define FATAL_ERROR_INTERNAL (3)
7  #define FATAL_ERROR_PARAMETER (4)
8  #define FATAL_ERROR_ENTROPY (5)
9  #define FATAL_ERROR_SELF_TEST (6)
10 #define FATAL_ERROR_CRYPT0 (7)
11 #define FATAL_ERROR_NV_UNRECOVERABLE (8)
12 #define FATAL_ERROR_REMANUFACTURED (9) // indicates that the TPM has
13                                         // been re-manufactured after an
14                                         // unrecoverable NV error
15 #define FATAL_ERROR_DRBG (10)
16 #define FATAL_ERROR_MOVE_SIZE (11)
17 #define FATAL_ERROR_COUNTER_OVERFLOW (12)
18 #define FATAL_ERROR_SUBTRACT (13)
19 #define FATAL_ERROR_MATHLIBRARY (14)
20 #define FATAL_ERROR_FORCED (666)
21
22 #endif // _TPM_ERROR_H
```

5.19 TpmTypes.h

```

1  #ifndef _TPM_TYPES_H_
2  #define _TPM_TYPES_H_

```

TCG Algorithm Registry: Table 2 - Definition of TPM_ALG_ID Constants

```

3  typedef UINT16 TPM_ALG_ID;
4  #define TYPE_OF TPM_ALG_ID      UINT16
5  #define ALG_ERROR_VALUE        0x0000
6  #define TPM_ALG_ERROR          (TPM_ALG_ID) (ALG_ERROR_VALUE)
7  #define ALG_RSA_VALUE          0x0001
8  #define TPM_ALG_RSA            (TPM_ALG_ID) (ALG_RSA_VALUE)
9  #define ALG_TDES_VALUE         0x0003
10 #define TPM_ALG_TDES           (TPM_ALG_ID) (ALG_TDES_VALUE)
11 #define ALG_SHA_VALUE          0x0004
12 #define TPM_ALG_SHA            (TPM_ALG_ID) (ALG_SHA_VALUE)
13 #define ALG_SHA1_VALUE         0x0004
14 #define TPM_ALG_SHA1           (TPM_ALG_ID) (ALG_SHA1_VALUE)
15 #define ALG_HMAC_VALUE         0x0005
16 #define TPM_ALG_HMAC           (TPM_ALG_ID) (ALG_HMAC_VALUE)
17 #define ALG_AES_VALUE          0x0006
18 #define TPM_ALG_AES            (TPM_ALG_ID) (ALG_AES_VALUE)
19 #define ALG_MGF1_VALUE         0x0007
20 #define TPM_ALG_MGF1           (TPM_ALG_ID) (ALG_MGF1_VALUE)
21 #define ALG_KEYEDHASH_VALUE    0x0008
22 #define TPM_ALG_KEYEDHASH      (TPM_ALG_ID) (ALG_KEYEDHASH_VALUE)
23 #define ALG_XOR_VALUE          0x000A
24 #define TPM_ALG_XOR            (TPM_ALG_ID) (ALG_XOR_VALUE)
25 #define ALG_SHA256_VALUE       0x000B
26 #define TPM_ALG_SHA256         (TPM_ALG_ID) (ALG_SHA256_VALUE)
27 #define ALG_SHA384_VALUE       0x000C
28 #define TPM_ALG_SHA384         (TPM_ALG_ID) (ALG_SHA384_VALUE)
29 #define ALG_SHA512_VALUE       0x000D
30 #define TPM_ALG_SHA512         (TPM_ALG_ID) (ALG_SHA512_VALUE)
31 #define ALG_NULL_VALUE         0x0010
32 #define TPM_ALG_NULL           (TPM_ALG_ID) (ALG_NULL_VALUE)
33 #define ALG_SM3_256_VALUE       0x0012
34 #define TPM_ALG_SM3_256        (TPM_ALG_ID) (ALG_SM3_256_VALUE)
35 #define ALG_SM4_VALUE          0x0013
36 #define TPM_ALG_SM4            (TPM_ALG_ID) (ALG_SM4_VALUE)
37 #define ALG_RSASSA_VALUE       0x0014
38 #define TPM_ALG_RSASSA         (TPM_ALG_ID) (ALG_RSASSA_VALUE)
39 #define ALG_RSAES_VALUE        0x0015
40 #define TPM_ALG_RSAES           (TPM_ALG_ID) (ALG_RSAES_VALUE)
41 #define ALG_RSAPSS_VALUE       0x0016
42 #define TPM_ALG_RSAPSS         (TPM_ALG_ID) (ALG_RSAPSS_VALUE)
43 #define ALG_OAEP_VALUE         0x0017
44 #define TPM_ALG_OAEP           (TPM_ALG_ID) (ALG_OAEP_VALUE)
45 #define ALG_ECDSA_VALUE        0x0018
46 #define TPM_ALG_ECDSA           (TPM_ALG_ID) (ALG_ECDSA_VALUE)
47 #define ALG_ECDH_VALUE         0x0019
48 #define TPM_ALG_ECDH           (TPM_ALG_ID) (ALG_ECDH_VALUE)
49 #define ALG_ECDSA_VALUE        0x001A
50 #define TPM_ALG_ECDSA           (TPM_ALG_ID) (ALG_ECDSA_VALUE)
51 #define ALG_SM2_VALUE          0x001B
52 #define TPM_ALG_SM2            (TPM_ALG_ID) (ALG_SM2_VALUE)
53 #define ALG_ECSCNORR_VALUE     0x001C
54 #define TPM_ALG_ECSCNORR       (TPM_ALG_ID) (ALG_ECSCNORR_VALUE)
55 #define ALG_ECMQV_VALUE        0x001D
56 #define TPM_ALG_ECMQV          (TPM_ALG_ID) (ALG_ECMQV_VALUE)
57 #define ALG_KDF1_SP800_56A_VALUE 0x0020
58 #define TPM_ALG_KDF1_SP800_56A (TPM_ALG_ID) (ALG_KDF1_SP800_56A_VALUE)
59 #define ALG_KDF2_VALUE         0x0021

```

```

60 #define TPM_ALG_KDF2 (TPM_ALG_ID) (ALG_KDF2_VALUE)
61 #define ALG_KDF1_SP800_108_VALUE 0x0022
62 #define TPM_ALG_KDF1_SP800_108 (TPM_ALG_ID) (ALG_KDF1_SP800_108_VALUE)
63 #define ALG_ECC_VALUE 0x0023
64 #define TPM_ALG_ECC (TPM_ALG_ID) (ALG_ECC_VALUE)
65 #define ALG_SYMCIPHER_VALUE 0x0025
66 #define TPM_ALG_SYMCIPHER (TPM_ALG_ID) (ALG_SYMCIPHER_VALUE)
67 #define ALG_CAMELLIA_VALUE 0x0026
68 #define TPM_ALG_CAMELLIA (TPM_ALG_ID) (ALG_CAMELLIA_VALUE)
69 #define ALG_SHA3_256_VALUE 0x0027
70 #define TPM_ALG_SHA3_256 (TPM_ALG_ID) (ALG_SHA3_256_VALUE)
71 #define ALG_SHA3_384_VALUE 0x0028
72 #define TPM_ALG_SHA3_384 (TPM_ALG_ID) (ALG_SHA3_384_VALUE)
73 #define ALG_SHA3_512_VALUE 0x0029
74 #define TPM_ALG_SHA3_512 (TPM_ALG_ID) (ALG_SHA3_512_VALUE)
75 #define ALG_CMAC_VALUE 0x003F
76 #define TPM_ALG_CMAC (TPM_ALG_ID) (ALG_CMAC_VALUE)
77 #define ALG_CTR_VALUE 0x0040
78 #define TPM_ALG_CTR (TPM_ALG_ID) (ALG_CTR_VALUE)
79 #define ALG_OFB_VALUE 0x0041
80 #define TPM_ALG_OFB (TPM_ALG_ID) (ALG_OFB_VALUE)
81 #define ALG_CBC_VALUE 0x0042
82 #define TPM_ALG_CBC (TPM_ALG_ID) (ALG_CBC_VALUE)
83 #define ALG_CFB_VALUE 0x0043
84 #define TPM_ALG_CFB (TPM_ALG_ID) (ALG_CFB_VALUE)
85 #define ALG_ECB_VALUE 0x0044
86 #define TPM_ALG_ECB (TPM_ALG_ID) (ALG_ECB_VALUE)

```

Values derived from Table 1:2

```

87 #define ALG_FIRST_VALUE 0x0001
88 #define TPM_ALG_FIRST (TPM_ALG_ID) (ALG_FIRST_VALUE)
89 #define ALG_LAST_VALUE 0x0044
90 #define TPM_ALG_LAST (TPM_ALG_ID) (ALG_LAST_VALUE)

```

TCG Algorithm Registry: Table 4 - Definition of TPM_ECC_CURVE Constants

```

91 typedef UINT16 TPM_ECC_CURVE;
92 #define TYPE_OF_TPM_ECC_CURVE UINT16
93 #define TPM_ECC_NONE (TPM_ECC_CURVE) (0x0000)
94 #define TPM_ECC_NIST_P192 (TPM_ECC_CURVE) (0x0001)
95 #define TPM_ECC_NIST_P224 (TPM_ECC_CURVE) (0x0002)
96 #define TPM_ECC_NIST_P256 (TPM_ECC_CURVE) (0x0003)
97 #define TPM_ECC_NIST_P384 (TPM_ECC_CURVE) (0x0004)
98 #define TPM_ECC_NIST_P521 (TPM_ECC_CURVE) (0x0005)
99 #define TPM_ECC_BN_P256 (TPM_ECC_CURVE) (0x0010)
100 #define TPM_ECC_BN_P638 (TPM_ECC_CURVE) (0x0011)
101 #define TPM_ECC_SM2_P256 (TPM_ECC_CURVE) (0x0020)

```

TPM 2.0 Part 2: Table 12 - Definition of TPM_CC Constants

```

102 typedef UINT32 TPM_CC;
103 #define TYPE_OF_TPM_CC UINT32
104 #define TPM_CC_NV_UndefineSpaceSpecial (TPM_CC) (0x0000011F)
105 #define TPM_CC_EvictControl (TPM_CC) (0x00000120)
106 #define TPM_CC_HierarchyControl (TPM_CC) (0x00000121)
107 #define TPM_CC_NV_UndefineSpace (TPM_CC) (0x00000122)
108 #define TPM_CC_ChangeEPS (TPM_CC) (0x00000124)
109 #define TPM_CC_ChangePPS (TPM_CC) (0x00000125)
110 #define TPM_CC_Clear (TPM_CC) (0x00000126)
111 #define TPM_CC_ClearControl (TPM_CC) (0x00000127)
112 #define TPM_CC_ClockSet (TPM_CC) (0x00000128)
113 #define TPM_CC_HierarchyChangeAuth (TPM_CC) (0x00000129)
114 #define TPM_CC_NV_DefineSpace (TPM_CC) (0x0000012A)

```



```
115 #define TPM_CC_PCR_Allocate (TPM_CC) (0x0000012B)
116 #define TPM_CC_PCR_SetAuthPolicy (TPM_CC) (0x0000012C)
117 #define TPM_CC_PP_Commands (TPM_CC) (0x0000012D)
118 #define TPM_CC_SetPrimaryPolicy (TPM_CC) (0x0000012E)
119 #define TPM_CC_FieldUpgradeStart (TPM_CC) (0x0000012F)
120 #define TPM_CC_ClockRateAdjust (TPM_CC) (0x00000130)
121 #define TPM_CC_CreatePrimary (TPM_CC) (0x00000131)
122 #define TPM_CC_NV_GlobalWriteLock (TPM_CC) (0x00000132)
123 #define TPM_CC_GetCommandAuditDigest (TPM_CC) (0x00000133)
124 #define TPM_CC_NV_Increment (TPM_CC) (0x00000134)
125 #define TPM_CC_NV_SetBits (TPM_CC) (0x00000135)
126 #define TPM_CC_NV_Extend (TPM_CC) (0x00000136)
127 #define TPM_CC_NV_Write (TPM_CC) (0x00000137)
128 #define TPM_CC_NV_WriteLock (TPM_CC) (0x00000138)
129 #define TPM_CC_DictionaryAttackLockReset (TPM_CC) (0x00000139)
130 #define TPM_CC_DictionaryAttackParameters (TPM_CC) (0x0000013A)
131 #define TPM_CC_NV_ChangeAuth (TPM_CC) (0x0000013B)
132 #define TPM_CC_PCR_Event (TPM_CC) (0x0000013C)
133 #define TPM_CC_PCR_Reset (TPM_CC) (0x0000013D)
134 #define TPM_CC_SequenceComplete (TPM_CC) (0x0000013E)
135 #define TPM_CC_SetAlgorithmSet (TPM_CC) (0x0000013F)
136 #define TPM_CC_SetCommandCodeAuditStatus (TPM_CC) (0x00000140)
137 #define TPM_CC_FieldUpgradeData (TPM_CC) (0x00000141)
138 #define TPM_CC_IncrementalSelfTest (TPM_CC) (0x00000142)
139 #define TPM_CC_SelfTest (TPM_CC) (0x00000143)
140 #define TPM_CC_Startup (TPM_CC) (0x00000144)
141 #define TPM_CC_Shutdown (TPM_CC) (0x00000145)
142 #define TPM_CC_StirRandom (TPM_CC) (0x00000146)
143 #define TPM_CC_ActivateCredential (TPM_CC) (0x00000147)
144 #define TPM_CC_Certify (TPM_CC) (0x00000148)
145 #define TPM_CC_PolicyNV (TPM_CC) (0x00000149)
146 #define TPM_CC_CertifyCreation (TPM_CC) (0x0000014A)
147 #define TPM_CC_Duplicate (TPM_CC) (0x0000014B)
148 #define TPM_CC_GetTime (TPM_CC) (0x0000014C)
149 #define TPM_CC_GetSessionAuditDigest (TPM_CC) (0x0000014D)
150 #define TPM_CC_NV_Read (TPM_CC) (0x0000014E)
151 #define TPM_CC_NV_ReadLock (TPM_CC) (0x0000014F)
152 #define TPM_CC_ObjectChangeAuth (TPM_CC) (0x00000150)
153 #define TPM_CC_PolicySecret (TPM_CC) (0x00000151)
154 #define TPM_CC_Rewrap (TPM_CC) (0x00000152)
155 #define TPM_CC_Create (TPM_CC) (0x00000153)
156 #define TPM_CC_ECDH_ZGen (TPM_CC) (0x00000154)
157 #define TPM_CC_HMAC (TPM_CC) (0x00000155)
158 #define TPM_CC_MAC (TPM_CC) (0x00000155)
159 #define TPM_CC_Import (TPM_CC) (0x00000156)
160 #define TPM_CC_Load (TPM_CC) (0x00000157)
161 #define TPM_CC_Quote (TPM_CC) (0x00000158)
162 #define TPM_CC_RSA_Decrypt (TPM_CC) (0x00000159)
163 #define TPM_CC_HMAC_Start (TPM_CC) (0x0000015B)
164 #define TPM_CC_MAC_Start (TPM_CC) (0x0000015B)
165 #define TPM_CC_SequenceUpdate (TPM_CC) (0x0000015C)
166 #define TPM_CC_Sign (TPM_CC) (0x0000015D)
167 #define TPM_CC_Unseal (TPM_CC) (0x0000015E)
168 #define TPM_CC_PolicySigned (TPM_CC) (0x00000160)
169 #define TPM_CC_ContextLoad (TPM_CC) (0x00000161)
170 #define TPM_CC_ContextSave (TPM_CC) (0x00000162)
171 #define TPM_CC_ECDH_KeyGen (TPM_CC) (0x00000163)
172 #define TPM_CC_EncryptDecrypt (TPM_CC) (0x00000164)
173 #define TPM_CC_FlushContext (TPM_CC) (0x00000165)
174 #define TPM_CC_LoadExternal (TPM_CC) (0x00000167)
175 #define TPM_CC_MakeCredential (TPM_CC) (0x00000168)
176 #define TPM_CC_NV_ReadPublic (TPM_CC) (0x00000169)
177 #define TPM_CC_PolicyAuthorize (TPM_CC) (0x0000016A)
178 #define TPM_CC_PolicyAuthValue (TPM_CC) (0x0000016B)
179 #define TPM_CC_PolicyCommandCode (TPM_CC) (0x0000016C)
180 #define TPM_CC_PolicyCounterTimer (TPM_CC) (0x0000016D)
```

```

181 #define TPM_CC_PolicyCpHash (TPM_CC) (0x0000016E)
182 #define TPM_CC_PolicyLocality (TPM_CC) (0x0000016F)
183 #define TPM_CC_PolicyNameHash (TPM_CC) (0x00000170)
184 #define TPM_CC_PolicyOR (TPM_CC) (0x00000171)
185 #define TPM_CC_PolicyTicket (TPM_CC) (0x00000172)
186 #define TPM_CC_ReadPublic (TPM_CC) (0x00000173)
187 #define TPM_CC_RSA_Encrypt (TPM_CC) (0x00000174)
188 #define TPM_CC_StartAuthSession (TPM_CC) (0x00000176)
189 #define TPM_CC_VerifySignature (TPM_CC) (0x00000177)
190 #define TPM_CC_ECC_Parameters (TPM_CC) (0x00000178)
191 #define TPM_CC_FirmwareRead (TPM_CC) (0x00000179)
192 #define TPM_CC_GetCapability (TPM_CC) (0x0000017A)
193 #define TPM_CC_GetRandom (TPM_CC) (0x0000017B)
194 #define TPM_CC_GetTestResult (TPM_CC) (0x0000017C)
195 #define TPM_CC_Hash (TPM_CC) (0x0000017D)
196 #define TPM_CC_PCR_Read (TPM_CC) (0x0000017E)
197 #define TPM_CC_PolicyPCR (TPM_CC) (0x0000017F)
198 #define TPM_CC_PolicyRestart (TPM_CC) (0x00000180)
199 #define TPM_CC_ReadClock (TPM_CC) (0x00000181)
200 #define TPM_CC_PCR_Extend (TPM_CC) (0x00000182)
201 #define TPM_CC_PCR_SetAuthValue (TPM_CC) (0x00000183)
202 #define TPM_CC_NV_Certify (TPM_CC) (0x00000184)
203 #define TPM_CC_EventSequenceComplete (TPM_CC) (0x00000185)
204 #define TPM_CC_HashSequenceStart (TPM_CC) (0x00000186)
205 #define TPM_CC_PolicyPhysicalPresence (TPM_CC) (0x00000187)
206 #define TPM_CC_PolicyDuplicationSelect (TPM_CC) (0x00000188)
207 #define TPM_CC_PolicyGetDigest (TPM_CC) (0x00000189)
208 #define TPM_CC_TestParms (TPM_CC) (0x0000018A)
209 #define TPM_CC_Commit (TPM_CC) (0x0000018B)
210 #define TPM_CC_PolicyPassword (TPM_CC) (0x0000018C)
211 #define TPM_CC_ZGen_2Phase (TPM_CC) (0x0000018D)
212 #define TPM_CC_EC_Ephemeral (TPM_CC) (0x0000018E)
213 #define TPM_CC_PolicyNvWritten (TPM_CC) (0x0000018F)
214 #define TPM_CC_PolicyTemplate (TPM_CC) (0x00000190)
215 #define TPM_CC_CreateLoaded (TPM_CC) (0x00000191)
216 #define TPM_CC_PolicyAuthorizeNV (TPM_CC) (0x00000192)
217 #define TPM_CC_EncryptDecrypt2 (TPM_CC) (0x00000193)
218 #define TPM_CC_AC_GetCapability (TPM_CC) (0x00000194)
219 #define TPM_CC_AC_Send (TPM_CC) (0x00000195)
220 #define TPM_CC_Policy_AC_SendSelect (TPM_CC) (0x00000196)
221 #define TPM_CC_CertifyX509 (TPM_CC) (0x00000197)
222 #define TPM_CC_ACT_SetTimeout (TPM_CC) (0x00000198)
223 #define TPM_CC_ECC_Encrypt (TPM_CC) (0x00000199)
224 #define TPM_CC_ECC_Decrypt (TPM_CC) (0x0000019A)
225 #define CC_VEND 0x20000000
226 #define TPM_CC_Vendor_TCG_Test (TPM_CC) (0x20000000)

```

TPM 2.0 Part 2: Table 5 - Definition of Types for Documentation Clarity

```

227 typedef UINT32 TPM_ALGORITHM_ID;
228 #define TYPE_OF_TPM_ALGORITHM_ID UINT32
229 typedef UINT32 TPM_MODIFIER_INDICATOR;
230 #define TYPE_OF_TPM_MODIFIER_INDICATOR UINT32
231 typedef UINT32 TPM_AUTHORIZATION_SIZE;
232 #define TYPE_OF_TPM_AUTHORIZATION_SIZE UINT32
233 typedef UINT32 TPM_PARAMETER_SIZE;
234 #define TYPE_OF_TPM_PARAMETER_SIZE UINT32
235 typedef UINT16 TPM_KEY_SIZE;
236 #define TYPE_OF_TPM_KEY_SIZE UINT16
237 typedef UINT16 TPM_KEY_BITS;
238 #define TYPE_OF_TPM_KEY_BITS UINT16

```

TPM 2.0 Part 2: Table 6 - Definition of TPM_SPEC Constants

```

239 typedef UINT32 TPM_SPEC;

```

```

240 #define TYPE_OF_TPM_SPEC      UINT32
241 #define SPEC_FAMILY           0x322E3000
242 #define TPM_SPEC_FAMILY      (TPM_SPEC) (SPEC_FAMILY)
243 #define SPEC_LEVEL           00
244 #define TPM_SPEC_LEVEL      (TPM_SPEC) (SPEC_LEVEL)
245 #define SPEC_VERSION         162
246 #define TPM_SPEC_VERSION     (TPM_SPEC) (SPEC_VERSION)
247 #define SPEC_YEAR            2020
248 #define TPM_SPEC_YEAR        (TPM_SPEC) (SPEC_YEAR)
249 #define SPEC_DAY_OF_YEAR     53
250 #define TPM_SPEC_DAY_OF_YEAR (TPM_SPEC) (SPEC_DAY_OF_YEAR)

```

TPM 2.0 Part 2: Table 7 - Definition of TPM_CONSTANTS32 Constants

```

251 typedef UINT32      TPM_CONSTANTS32;
252 #define TYPE_OF_TPM_CONSTANTS32  UINT32
253 #define TPM_GENERATED_VALUE      (TPM_CONSTANTS32) (0xFF544347)
254 #define TPM_MAX_DERIVATION_BITS  (TPM_CONSTANTS32) (8192)

```

TPM 2.0 Part 2: Table 16 - Definition of TPM_RC Constants

```

255 typedef UINT32      TPM_RC;
256 #define TYPE_OF_TPM_RC      UINT32
257 #define TPM_RC_SUCCESS      (TPM_RC) (0x000)
258 #define TPM_RC_BAD_TAG      (TPM_RC) (0x01E)
259 #define RC_VER1             (TPM_RC) (0x100)
260 #define TPM_RC_INITIALIZE   (TPM_RC) (RC_VER1+0x000)
261 #define TPM_RC_FAILURE      (TPM_RC) (RC_VER1+0x001)
262 #define TPM_RC_SEQUENCE     (TPM_RC) (RC_VER1+0x003)
263 #define TPM_RC_PRIVATE      (TPM_RC) (RC_VER1+0x00B)
264 #define TPM_RC_HMAC         (TPM_RC) (RC_VER1+0x019)
265 #define TPM_RC_DISABLED     (TPM_RC) (RC_VER1+0x020)
266 #define TPM_RC_EXCLUSIVE    (TPM_RC) (RC_VER1+0x021)
267 #define TPM_RC_AUTH_TYPE    (TPM_RC) (RC_VER1+0x024)
268 #define TPM_RC_AUTH_MISSING (TPM_RC) (RC_VER1+0x025)
269 #define TPM_RC_POLICY       (TPM_RC) (RC_VER1+0x026)
270 #define TPM_RC_PCR          (TPM_RC) (RC_VER1+0x027)
271 #define TPM_RC_PCR_CHANGED  (TPM_RC) (RC_VER1+0x028)
272 #define TPM_RC_UPGRADE      (TPM_RC) (RC_VER1+0x02D)
273 #define TPM_RC_TOO_MANY_CONTEXTS (TPM_RC) (RC_VER1+0x02E)
274 #define TPM_RC_AUTH_UNAVAILABLE (TPM_RC) (RC_VER1+0x02F)
275 #define TPM_RC_REBOOT       (TPM_RC) (RC_VER1+0x030)
276 #define TPM_RC_UNBALANCED   (TPM_RC) (RC_VER1+0x031)
277 #define TPM_RC_COMMAND_SIZE (TPM_RC) (RC_VER1+0x042)
278 #define TPM_RC_COMMAND_CODE (TPM_RC) (RC_VER1+0x043)
279 #define TPM_RC_AUTHSIZE     (TPM_RC) (RC_VER1+0x044)
280 #define TPM_RC_AUTH_CONTEXT (TPM_RC) (RC_VER1+0x045)
281 #define TPM_RC_NV_RANGE     (TPM_RC) (RC_VER1+0x046)
282 #define TPM_RC_NV_SIZE      (TPM_RC) (RC_VER1+0x047)
283 #define TPM_RC_NV_LOCKED    (TPM_RC) (RC_VER1+0x048)
284 #define TPM_RC_NV_AUTHORIZATION (TPM_RC) (RC_VER1+0x049)
285 #define TPM_RC_NV_UNINITIALIZED (TPM_RC) (RC_VER1+0x04A)
286 #define TPM_RC_NV_SPACE     (TPM_RC) (RC_VER1+0x04B)
287 #define TPM_RC_NV_DEFINED   (TPM_RC) (RC_VER1+0x04C)
288 #define TPM_RC_BAD_CONTEXT  (TPM_RC) (RC_VER1+0x050)
289 #define TPM_RC_CPHASH       (TPM_RC) (RC_VER1+0x051)
290 #define TPM_RC_PARENT        (TPM_RC) (RC_VER1+0x052)
291 #define TPM_RC_NEEDS_TEST   (TPM_RC) (RC_VER1+0x053)
292 #define TPM_RC_NO_RESULT    (TPM_RC) (RC_VER1+0x054)
293 #define TPM_RC_SENSITIVE    (TPM_RC) (RC_VER1+0x055)
294 #define RC_MAX_FMO          (TPM_RC) (RC_VER1+0x07F)
295 #define RC_FMT1             (TPM_RC) (0x080)
296 #define TPM_RC_ASYMMETRIC   (TPM_RC) (RC_FMT1+0x001)
297 #define TPM_RCS_ASYMMETRIC  (TPM_RC) (RC_FMT1+0x001)
298 #define TPM_RC_ATTRIBUTES   (TPM_RC) (RC_FMT1+0x002)

```

```
299 #define TPM_RCS_ATTRIBUTES (TPM_RC) (RC_FMT1+0x002)
300 #define TPM_RC_HASH (TPM_RC) (RC_FMT1+0x003)
301 #define TPM_RCS_HASH (TPM_RC) (RC_FMT1+0x003)
302 #define TPM_RC_VALUE (TPM_RC) (RC_FMT1+0x004)
303 #define TPM_RCS_VALUE (TPM_RC) (RC_FMT1+0x004)
304 #define TPM_RC_HIERARCHY (TPM_RC) (RC_FMT1+0x005)
305 #define TPM_RCS_HIERARCHY (TPM_RC) (RC_FMT1+0x005)
306 #define TPM_RC_KEY_SIZE (TPM_RC) (RC_FMT1+0x007)
307 #define TPM_RCS_KEY_SIZE (TPM_RC) (RC_FMT1+0x007)
308 #define TPM_RC_MGF (TPM_RC) (RC_FMT1+0x008)
309 #define TPM_RCS_MGF (TPM_RC) (RC_FMT1+0x008)
310 #define TPM_RC_MODE (TPM_RC) (RC_FMT1+0x009)
311 #define TPM_RCS_MODE (TPM_RC) (RC_FMT1+0x009)
312 #define TPM_RC_TYPE (TPM_RC) (RC_FMT1+0x00A)
313 #define TPM_RCS_TYPE (TPM_RC) (RC_FMT1+0x00A)
314 #define TPM_RC_HANDLE (TPM_RC) (RC_FMT1+0x00B)
315 #define TPM_RCS_HANDLE (TPM_RC) (RC_FMT1+0x00B)
316 #define TPM_RC_KDF (TPM_RC) (RC_FMT1+0x00C)
317 #define TPM_RCS_KDF (TPM_RC) (RC_FMT1+0x00C)
318 #define TPM_RC_RANGE (TPM_RC) (RC_FMT1+0x00D)
319 #define TPM_RCS_RANGE (TPM_RC) (RC_FMT1+0x00D)
320 #define TPM_RC_AUTH_FAIL (TPM_RC) (RC_FMT1+0x00E)
321 #define TPM_RCS_AUTH_FAIL (TPM_RC) (RC_FMT1+0x00E)
322 #define TPM_RC_NONCE (TPM_RC) (RC_FMT1+0x00F)
323 #define TPM_RCS_NONCE (TPM_RC) (RC_FMT1+0x00F)
324 #define TPM_RC_PP (TPM_RC) (RC_FMT1+0x010)
325 #define TPM_RCS_PP (TPM_RC) (RC_FMT1+0x010)
326 #define TPM_RC_SCHEME (TPM_RC) (RC_FMT1+0x012)
327 #define TPM_RCS_SCHEME (TPM_RC) (RC_FMT1+0x012)
328 #define TPM_RC_SIZE (TPM_RC) (RC_FMT1+0x015)
329 #define TPM_RCS_SIZE (TPM_RC) (RC_FMT1+0x015)
330 #define TPM_RC_SYMMETRIC (TPM_RC) (RC_FMT1+0x016)
331 #define TPM_RCS_SYMMETRIC (TPM_RC) (RC_FMT1+0x016)
332 #define TPM_RC_TAG (TPM_RC) (RC_FMT1+0x017)
333 #define TPM_RCS_TAG (TPM_RC) (RC_FMT1+0x017)
334 #define TPM_RC_SELECTOR (TPM_RC) (RC_FMT1+0x018)
335 #define TPM_RCS_SELECTOR (TPM_RC) (RC_FMT1+0x018)
336 #define TPM_RC_INSUFFICIENT (TPM_RC) (RC_FMT1+0x01A)
337 #define TPM_RCS_INSUFFICIENT (TPM_RC) (RC_FMT1+0x01A)
338 #define TPM_RC_SIGNATURE (TPM_RC) (RC_FMT1+0x01B)
339 #define TPM_RCS_SIGNATURE (TPM_RC) (RC_FMT1+0x01B)
340 #define TPM_RC_KEY (TPM_RC) (RC_FMT1+0x01C)
341 #define TPM_RCS_KEY (TPM_RC) (RC_FMT1+0x01C)
342 #define TPM_RC_POLICY_FAIL (TPM_RC) (RC_FMT1+0x01D)
343 #define TPM_RCS_POLICY_FAIL (TPM_RC) (RC_FMT1+0x01D)
344 #define TPM_RC_INTEGRITY (TPM_RC) (RC_FMT1+0x01F)
345 #define TPM_RCS_INTEGRITY (TPM_RC) (RC_FMT1+0x01F)
346 #define TPM_RC_TICKET (TPM_RC) (RC_FMT1+0x020)
347 #define TPM_RCS_TICKET (TPM_RC) (RC_FMT1+0x020)
348 #define TPM_RC_RESERVED_BITS (TPM_RC) (RC_FMT1+0x021)
349 #define TPM_RCS_RESERVED_BITS (TPM_RC) (RC_FMT1+0x021)
350 #define TPM_RC_BAD_AUTH (TPM_RC) (RC_FMT1+0x022)
351 #define TPM_RCS_BAD_AUTH (TPM_RC) (RC_FMT1+0x022)
352 #define TPM_RC_EXPIRED (TPM_RC) (RC_FMT1+0x023)
353 #define TPM_RCS_EXPIRED (TPM_RC) (RC_FMT1+0x023)
354 #define TPM_RC_POLICY_CC (TPM_RC) (RC_FMT1+0x024)
355 #define TPM_RCS_POLICY_CC (TPM_RC) (RC_FMT1+0x024)
356 #define TPM_RC_BINDING (TPM_RC) (RC_FMT1+0x025)
357 #define TPM_RCS_BINDING (TPM_RC) (RC_FMT1+0x025)
358 #define TPM_RC_CURVE (TPM_RC) (RC_FMT1+0x026)
359 #define TPM_RCS_CURVE (TPM_RC) (RC_FMT1+0x026)
360 #define TPM_RC_ECC_POINT (TPM_RC) (RC_FMT1+0x027)
361 #define TPM_RCS_ECC_POINT (TPM_RC) (RC_FMT1+0x027)
362 #define RC_WARN (TPM_RC) (0x900)
363 #define TPM_RC_CONTEXT_GAP (TPM_RC) (RC_WARN+0x001)
364 #define TPM_RC_OBJECT_MEMORY (TPM_RC) (RC_WARN+0x002)
```



```

365 #define TPM_RC_SESSION_MEMORY (TPM_RC) (RC_WARN+0x003)
366 #define TPM_RC_MEMORY (TPM_RC) (RC_WARN+0x004)
367 #define TPM_RC_SESSION_HANDLES (TPM_RC) (RC_WARN+0x005)
368 #define TPM_RC_OBJECT_HANDLES (TPM_RC) (RC_WARN+0x006)
369 #define TPM_RC_LOCALITY (TPM_RC) (RC_WARN+0x007)
370 #define TPM_RC_YIELDED (TPM_RC) (RC_WARN+0x008)
371 #define TPM_RC_CANCELED (TPM_RC) (RC_WARN+0x009)
372 #define TPM_RC_TESTING (TPM_RC) (RC_WARN+0x00A)
373 #define TPM_RC_REFERENCE_H0 (TPM_RC) (RC_WARN+0x010)
374 #define TPM_RC_REFERENCE_H1 (TPM_RC) (RC_WARN+0x011)
375 #define TPM_RC_REFERENCE_H2 (TPM_RC) (RC_WARN+0x012)
376 #define TPM_RC_REFERENCE_H3 (TPM_RC) (RC_WARN+0x013)
377 #define TPM_RC_REFERENCE_H4 (TPM_RC) (RC_WARN+0x014)
378 #define TPM_RC_REFERENCE_H5 (TPM_RC) (RC_WARN+0x015)
379 #define TPM_RC_REFERENCE_H6 (TPM_RC) (RC_WARN+0x016)
380 #define TPM_RC_REFERENCE_S0 (TPM_RC) (RC_WARN+0x018)
381 #define TPM_RC_REFERENCE_S1 (TPM_RC) (RC_WARN+0x019)
382 #define TPM_RC_REFERENCE_S2 (TPM_RC) (RC_WARN+0x01A)
383 #define TPM_RC_REFERENCE_S3 (TPM_RC) (RC_WARN+0x01B)
384 #define TPM_RC_REFERENCE_S4 (TPM_RC) (RC_WARN+0x01C)
385 #define TPM_RC_REFERENCE_S5 (TPM_RC) (RC_WARN+0x01D)
386 #define TPM_RC_REFERENCE_S6 (TPM_RC) (RC_WARN+0x01E)
387 #define TPM_RC_NV_RATE (TPM_RC) (RC_WARN+0x020)
388 #define TPM_RC_LOCKOUT (TPM_RC) (RC_WARN+0x021)
389 #define TPM_RC_RETRY (TPM_RC) (RC_WARN+0x022)
390 #define TPM_RC_NV_UNAVAILABLE (TPM_RC) (RC_WARN+0x023)
391 #define TPM_RC_NOT_USED (TPM_RC) (RC_WARN+0x7F)
392 #define TPM_RC_H (TPM_RC) (0x000)
393 #define TPM_RC_P (TPM_RC) (0x040)
394 #define TPM_RC_S (TPM_RC) (0x800)
395 #define TPM_RC_1 (TPM_RC) (0x100)
396 #define TPM_RC_2 (TPM_RC) (0x200)
397 #define TPM_RC_3 (TPM_RC) (0x300)
398 #define TPM_RC_4 (TPM_RC) (0x400)
399 #define TPM_RC_5 (TPM_RC) (0x500)
400 #define TPM_RC_6 (TPM_RC) (0x600)
401 #define TPM_RC_7 (TPM_RC) (0x700)
402 #define TPM_RC_8 (TPM_RC) (0x800)
403 #define TPM_RC_9 (TPM_RC) (0x900)
404 #define TPM_RC_A (TPM_RC) (0xA00)
405 #define TPM_RC_B (TPM_RC) (0xB00)
406 #define TPM_RC_C (TPM_RC) (0xC00)
407 #define TPM_RC_D (TPM_RC) (0xD00)
408 #define TPM_RC_E (TPM_RC) (0xE00)
409 #define TPM_RC_F (TPM_RC) (0xF00)
410 #define TPM_RC_N_MASK (TPM_RC) (0xF00)

```

TPM 2.0 Part 2: Table 17 - Definition of TPM_CLOCK_ADJUST Constants

```

411 typedef INT8 TPM_CLOCK_ADJUST;
412 #define TYPE_OF_TPM_CLOCK_ADJUST UINT8
413 #define TPM_CLOCK_COARSE_SLOWER (TPM_CLOCK_ADJUST) (-3)
414 #define TPM_CLOCK_MEDIUM_SLOWER (TPM_CLOCK_ADJUST) (-2)
415 #define TPM_CLOCK_FINE_SLOWER (TPM_CLOCK_ADJUST) (-1)
416 #define TPM_CLOCK_NO_CHANGE (TPM_CLOCK_ADJUST) (0)
417 #define TPM_CLOCK_FINE_FASTER (TPM_CLOCK_ADJUST) (1)
418 #define TPM_CLOCK_MEDIUM_FASTER (TPM_CLOCK_ADJUST) (2)
419 #define TPM_CLOCK_COARSE_FASTER (TPM_CLOCK_ADJUST) (3)

```

TPM 2.0 Part 2: Table 18 - Definition of TPM_EO Constants

```

420 typedef UINT16 TPM_EO;
421 #define TYPE_OF_TPM_EO UINT16
422 #define TPM_EO_EQ (TPM_EO) (0x0000)
423 #define TPM_EO_NEQ (TPM_EO) (0x0001)

```

```

424 #define TPM_EO_SIGNED_GT      (TPM_EO) (0x0002)
425 #define TPM_EO_UNSIGNED_GT    (TPM_EO) (0x0003)
426 #define TPM_EO_SIGNED_LT      (TPM_EO) (0x0004)
427 #define TPM_EO_UNSIGNED_LT    (TPM_EO) (0x0005)
428 #define TPM_EO_SIGNED_GE      (TPM_EO) (0x0006)
429 #define TPM_EO_UNSIGNED_GE    (TPM_EO) (0x0007)
430 #define TPM_EO_SIGNED_LE      (TPM_EO) (0x0008)
431 #define TPM_EO_UNSIGNED_LE    (TPM_EO) (0x0009)
432 #define TPM_EO_BITSET          (TPM_EO) (0x000A)
433 #define TPM_EO_BITCLEAR       (TPM_EO) (0x000B)

```

TPM 2.0 Part 2: Table 19 - Definition of TPM_ST Constants

```

434 typedef UINT16          TPM_ST;
435 #define TYPE_OF_TPM_ST  UINT16
436 #define TPM_ST_RSP_COMMAND      (TPM_ST) (0x00C4)
437 #define TPM_ST_NULL             (TPM_ST) (0x8000)
438 #define TPM_ST_NO_SESSIONS      (TPM_ST) (0x8001)
439 #define TPM_ST_SESSIONS        (TPM_ST) (0x8002)
440 #define TPM_ST_ATTEST_NV        (TPM_ST) (0x8014)
441 #define TPM_ST_ATTEST_COMMAND_AUDIT (TPM_ST) (0x8015)
442 #define TPM_ST_ATTEST_SESSION_AUDIT (TPM_ST) (0x8016)
443 #define TPM_ST_ATTEST_CERTIFY   (TPM_ST) (0x8017)
444 #define TPM_ST_ATTEST_QUOTE     (TPM_ST) (0x8018)
445 #define TPM_ST_ATTEST_TIME      (TPM_ST) (0x8019)
446 #define TPM_ST_ATTEST_CREATION  (TPM_ST) (0x801A)
447 #define TPM_ST_ATTEST_NV_DIGEST (TPM_ST) (0x801C)
448 #define TPM_ST_CREATION         (TPM_ST) (0x8021)
449 #define TPM_ST_VERIFIED         (TPM_ST) (0x8022)
450 #define TPM_ST_AUTH_SECRET      (TPM_ST) (0x8023)
451 #define TPM_ST_HASHCHECK        (TPM_ST) (0x8024)
452 #define TPM_ST_AUTH_SIGNED      (TPM_ST) (0x8025)
453 #define TPM_ST_FU_MANIFEST      (TPM_ST) (0x8029)

```

TPM 2.0 Part 2: Table 20 - Definition of TPM_SU Constants

```

454 typedef UINT16          TPM_SU;
455 #define TYPE_OF_TPM_SU  UINT16
456 #define TPM_SU_CLEAR     (TPM_SU) (0x0000)
457 #define TPM_SU_STATE     (TPM_SU) (0x0001)

```

TPM 2.0 Part 2: Table 21 - Definition of TPM_SE Constants

```

458 typedef UINT8          TPM_SE;
459 #define TYPE_OF_TPM_SE  UINT8
460 #define TPM_SE_HMAC      (TPM_SE) (0x00)
461 #define TPM_SE_POLICY     (TPM_SE) (0x01)
462 #define TPM_SE_TRIAL     (TPM_SE) (0x03)

```

TPM 2.0 Part 2: Table 22 - Definition of TPM_CAP Constants

```

463 typedef UINT32          TPM_CAP;
464 #define TYPE_OF_TPM_CAP  UINT32
465 #define TPM_CAP_FIRST    (TPM_CAP) (0x00000000)
466 #define TPM_CAP_ALGS      (TPM_CAP) (0x00000000)
467 #define TPM_CAP_HANDLES   (TPM_CAP) (0x00000001)
468 #define TPM_CAP_COMMANDS  (TPM_CAP) (0x00000002)
469 #define TPM_CAP_PP_COMMANDS (TPM_CAP) (0x00000003)
470 #define TPM_CAP_AUDIT_COMMANDS (TPM_CAP) (0x00000004)
471 #define TPM_CAP_PCERS     (TPM_CAP) (0x00000005)
472 #define TPM_CAP_TPM_PROPERTIES (TPM_CAP) (0x00000006)
473 #define TPM_CAP_PCR_PROPERTIES (TPM_CAP) (0x00000007)
474 #define TPM_CAP_ECC_CURVES (TPM_CAP) (0x00000008)
475 #define TPM_CAP_AUTH_POLICIES (TPM_CAP) (0x00000009)

```

```

476 #define TPM_CAP_ACT (TPM_CAP) (0x0000000A)
477 #define TPM_CAP_LAST (TPM_CAP) (0x0000000A)
478 #define TPM_CAP_VENDOR_PROPERTY (TPM_CAP) (0x00000100)

```

TPM 2.0 Part 2: Table 23 - Definition of TPM_PT Constants

```

479 typedef UINT32 TPM_PT;
480 #define TYPE_OF_TPM_PT UINT32
481 #define TPM_PT_NONE (TPM_PT) (0x00000000)
482 #define PT_GROUP (TPM_PT) (0x00000100)
483 #define PT_FIXED (TPM_PT) (PT_GROUP*1)
484 #define TPM_PT_FAMILY_INDICATOR (TPM_PT) (PT_FIXED+0)
485 #define TPM_PT_LEVEL (TPM_PT) (PT_FIXED+1)
486 #define TPM_PT_REVISION (TPM_PT) (PT_FIXED+2)
487 #define TPM_PT_DAY_OF_YEAR (TPM_PT) (PT_FIXED+3)
488 #define TPM_PT_YEAR (TPM_PT) (PT_FIXED+4)
489 #define TPM_PT_MANUFACTURER (TPM_PT) (PT_FIXED+5)
490 #define TPM_PT_VENDOR_STRING_1 (TPM_PT) (PT_FIXED+6)
491 #define TPM_PT_VENDOR_STRING_2 (TPM_PT) (PT_FIXED+7)
492 #define TPM_PT_VENDOR_STRING_3 (TPM_PT) (PT_FIXED+8)
493 #define TPM_PT_VENDOR_STRING_4 (TPM_PT) (PT_FIXED+9)
494 #define TPM_PT_VENDOR_TPM_TYPE (TPM_PT) (PT_FIXED+10)
495 #define TPM_PT_FIRMWARE_VERSION_1 (TPM_PT) (PT_FIXED+11)
496 #define TPM_PT_FIRMWARE_VERSION_2 (TPM_PT) (PT_FIXED+12)
497 #define TPM_PT_INPUT_BUFFER (TPM_PT) (PT_FIXED+13)
498 #define TPM_PT_HR_TRANSIENT_MIN (TPM_PT) (PT_FIXED+14)
499 #define TPM_PT_HR_PERSISTENT_MIN (TPM_PT) (PT_FIXED+15)
500 #define TPM_PT_HR_LOADED_MIN (TPM_PT) (PT_FIXED+16)
501 #define TPM_PT_ACTIVE_SESSIONS_MAX (TPM_PT) (PT_FIXED+17)
502 #define TPM_PT_PCR_COUNT (TPM_PT) (PT_FIXED+18)
503 #define TPM_PT_PCR_SELECT_MIN (TPM_PT) (PT_FIXED+19)
504 #define TPM_PT_CONTEXT_GAP_MAX (TPM_PT) (PT_FIXED+20)
505 #define TPM_PT_NV_COUNTERS_MAX (TPM_PT) (PT_FIXED+22)
506 #define TPM_PT_NV_INDEX_MAX (TPM_PT) (PT_FIXED+23)
507 #define TPM_PT_MEMORY (TPM_PT) (PT_FIXED+24)
508 #define TPM_PT_CLOCK_UPDATE (TPM_PT) (PT_FIXED+25)
509 #define TPM_PT_CONTEXT_HASH (TPM_PT) (PT_FIXED+26)
510 #define TPM_PT_CONTEXT_SYM (TPM_PT) (PT_FIXED+27)
511 #define TPM_PT_CONTEXT_SYM_SIZE (TPM_PT) (PT_FIXED+28)
512 #define TPM_PT_ORDERLY_COUNT (TPM_PT) (PT_FIXED+29)
513 #define TPM_PT_MAX_COMMAND_SIZE (TPM_PT) (PT_FIXED+30)
514 #define TPM_PT_MAX_RESPONSE_SIZE (TPM_PT) (PT_FIXED+31)
515 #define TPM_PT_MAX_DIGEST (TPM_PT) (PT_FIXED+32)
516 #define TPM_PT_MAX_OBJECT_CONTEXT (TPM_PT) (PT_FIXED+33)
517 #define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+34)
518 #define TPM_PT_PS_FAMILY_INDICATOR (TPM_PT) (PT_FIXED+35)
519 #define TPM_PT_PS_LEVEL (TPM_PT) (PT_FIXED+36)
520 #define TPM_PT_PS_REVISION (TPM_PT) (PT_FIXED+37)
521 #define TPM_PT_PS_DAY_OF_YEAR (TPM_PT) (PT_FIXED+38)
522 #define TPM_PT_PS_YEAR (TPM_PT) (PT_FIXED+39)
523 #define TPM_PT_SPLIT_MAX (TPM_PT) (PT_FIXED+40)
524 #define TPM_PT_TOTAL_COMMANDS (TPM_PT) (PT_FIXED+41)
525 #define TPM_PT_LIBRARY_COMMANDS (TPM_PT) (PT_FIXED+42)
526 #define TPM_PT_VENDOR_COMMANDS (TPM_PT) (PT_FIXED+43)
527 #define TPM_PT_NV_BUFFER_MAX (TPM_PT) (PT_FIXED+44)
528 #define TPM_PT_MODES (TPM_PT) (PT_FIXED+45)
529 #define TPM_PT_MAX_CAP_BUFFER (TPM_PT) (PT_FIXED+46)
530 #define PT_VAR (TPM_PT) (PT_GROUP*2)
531 #define TPM_PT_PERMANENT (TPM_PT) (PT_VAR+0)
532 #define TPM_PT_STARTUP_CLEAR (TPM_PT) (PT_VAR+1)
533 #define TPM_PT_HR_NV_INDEX (TPM_PT) (PT_VAR+2)
534 #define TPM_PT_HR_LOADED (TPM_PT) (PT_VAR+3)
535 #define TPM_PT_HR_LOADED_AVAIL (TPM_PT) (PT_VAR+4)
536 #define TPM_PT_HR_ACTIVE (TPM_PT) (PT_VAR+5)
537 #define TPM_PT_HR_ACTIVE_AVAIL (TPM_PT) (PT_VAR+6)

```



```

538 #define TPM_PT_HR_TRANSIENT_AVAIL (TPM_PT) (PT_VAR+7)
539 #define TPM_PT_HR_PERSISTENT (TPM_PT) (PT_VAR+8)
540 #define TPM_PT_HR_PERSISTENT_AVAIL (TPM_PT) (PT_VAR+9)
541 #define TPM_PT_NV_COUNTERS (TPM_PT) (PT_VAR+10)
542 #define TPM_PT_NV_COUNTERS_AVAIL (TPM_PT) (PT_VAR+11)
543 #define TPM_PT_ALGORITHM_SET (TPM_PT) (PT_VAR+12)
544 #define TPM_PT_LOADED_CURVES (TPM_PT) (PT_VAR+13)
545 #define TPM_PT_LOCKOUT_COUNTER (TPM_PT) (PT_VAR+14)
546 #define TPM_PT_MAX_AUTH_FAIL (TPM_PT) (PT_VAR+15)
547 #define TPM_PT_LOCKOUT_INTERVAL (TPM_PT) (PT_VAR+16)
548 #define TPM_PT_LOCKOUT_RECOVERY (TPM_PT) (PT_VAR+17)
549 #define TPM_PT_NV_WRITE_RECOVERY (TPM_PT) (PT_VAR+18)
550 #define TPM_PT_AUDIT_COUNTER_0 (TPM_PT) (PT_VAR+19)
551 #define TPM_PT_AUDIT_COUNTER_1 (TPM_PT) (PT_VAR+20)

```

TPM 2.0 Part 2: Table 24 - Definition of TPM_PT_PCR Constants

```

552 typedef UINT32 TPM_PT_PCR;
553 #define TYPE_OF_TPM_PT_PCR UINT32
554 #define TPM_PT_PCR_FIRST (TPM_PT_PCR) (0x00000000)
555 #define TPM_PT_PCR_SAVE (TPM_PT_PCR) (0x00000000)
556 #define TPM_PT_PCR_EXTEND_L0 (TPM_PT_PCR) (0x00000001)
557 #define TPM_PT_PCR_RESET_L0 (TPM_PT_PCR) (0x00000002)
558 #define TPM_PT_PCR_EXTEND_L1 (TPM_PT_PCR) (0x00000003)
559 #define TPM_PT_PCR_RESET_L1 (TPM_PT_PCR) (0x00000004)
560 #define TPM_PT_PCR_EXTEND_L2 (TPM_PT_PCR) (0x00000005)
561 #define TPM_PT_PCR_RESET_L2 (TPM_PT_PCR) (0x00000006)
562 #define TPM_PT_PCR_EXTEND_L3 (TPM_PT_PCR) (0x00000007)
563 #define TPM_PT_PCR_RESET_L3 (TPM_PT_PCR) (0x00000008)
564 #define TPM_PT_PCR_EXTEND_L4 (TPM_PT_PCR) (0x00000009)
565 #define TPM_PT_PCR_RESET_L4 (TPM_PT_PCR) (0x0000000A)
566 #define TPM_PT_PCR_NO_INCREMENT (TPM_PT_PCR) (0x00000011)
567 #define TPM_PT_PCR_DRMT_RESET (TPM_PT_PCR) (0x00000012)
568 #define TPM_PT_PCR_POLICY (TPM_PT_PCR) (0x00000013)
569 #define TPM_PT_PCR_AUTH (TPM_PT_PCR) (0x00000014)
570 #define TPM_PT_PCR_LAST (TPM_PT_PCR) (0x00000014)

```

TPM 2.0 Part 2: Table 25 - Definition of TPM_PS Constants

```

571 typedef UINT32 TPM_PS;
572 #define TYPE_OF_TPM_PS UINT32
573 #define TPM_PS_MAIN (TPM_PS) (0x00000000)
574 #define TPM_PS_PC (TPM_PS) (0x00000001)
575 #define TPM_PS_PDA (TPM_PS) (0x00000002)
576 #define TPM_PS_CELL_PHONE (TPM_PS) (0x00000003)
577 #define TPM_PS_SERVER (TPM_PS) (0x00000004)
578 #define TPM_PS_PERIPHERAL (TPM_PS) (0x00000005)
579 #define TPM_PS_TSS (TPM_PS) (0x00000006)
580 #define TPM_PS_STORAGE (TPM_PS) (0x00000007)
581 #define TPM_PS_AUTHENTICATION (TPM_PS) (0x00000008)
582 #define TPM_PS_EMBEDDED (TPM_PS) (0x00000009)
583 #define TPM_PS_HARDCOPY (TPM_PS) (0x0000000A)
584 #define TPM_PS_INFRASTRUCTURE (TPM_PS) (0x0000000B)
585 #define TPM_PS_VIRTUALIZATION (TPM_PS) (0x0000000C)
586 #define TPM_PS_TNC (TPM_PS) (0x0000000D)
587 #define TPM_PS_MULTI_TENANT (TPM_PS) (0x0000000E)
588 #define TPM_PS_TC (TPM_PS) (0x0000000F)

```

TPM 2.0 Part 2: Table 26 - Definition of Types for Handles

```

589 typedef UINT32 TPM_HANDLE;
590 #define TYPE_OF_TPM_HANDLE UINT32

```

TPM 2.0 Part 2: Table 27 - Definition of TPM_HT Constants

```

591 typedef UINT8 TPM_HT;
592 #define TYPE_OF_TPM_HT UINT8
593 #define TPM_HT_PCR (TPM_HT) (0x00)
594 #define TPM_HT_NV_INDEX (TPM_HT) (0x01)
595 #define TPM_HT_HMAC_SESSION (TPM_HT) (0x02)
596 #define TPM_HT_LOADED_SESSION (TPM_HT) (0x02)
597 #define TPM_HT_POLICY_SESSION (TPM_HT) (0x03)
598 #define TPM_HT_SAVED_SESSION (TPM_HT) (0x03)
599 #define TPM_HT_PERMANENT (TPM_HT) (0x40)
600 #define TPM_HT_TRANSIENT (TPM_HT) (0x80)
601 #define TPM_HT_PERSISTENT (TPM_HT) (0x81)
602 #define TPM_HT_AC (TPM_HT) (0x90)

```

TPM 2.0 Part 2: Table 28 - Definition of TPM_RH Constants

```

603 typedef TPM_HANDLE TPM_RH;
604 #define TPM_RH_FIRST (TPM_RH) (0x40000000)
605 #define TPM_RH_SRK (TPM_RH) (0x40000000)
606 #define TPM_RH_OWNER (TPM_RH) (0x40000001)
607 #define TPM_RH_REVOKE (TPM_RH) (0x40000002)
608 #define TPM_RH_TRANSPORT (TPM_RH) (0x40000003)
609 #define TPM_RH_OPERATOR (TPM_RH) (0x40000004)
610 #define TPM_RH_ADMIN (TPM_RH) (0x40000005)
611 #define TPM_RH_EK (TPM_RH) (0x40000006)
612 #define TPM_RH_NULL (TPM_RH) (0x40000007)
613 #define TPM_RH_UNASSIGNED (TPM_RH) (0x40000008)
614 #define TPM_RS_PW (TPM_RH) (0x40000009)
615 #define TPM_RH_LOCKOUT (TPM_RH) (0x4000000A)
616 #define TPM_RH_ENDORSEMENT (TPM_RH) (0x4000000B)
617 #define TPM_RH_PLATFORM (TPM_RH) (0x4000000C)
618 #define TPM_RH_PLATFORM_NV (TPM_RH) (0x4000000D)
619 #define TPM_RH_AUTH_00 (TPM_RH) (0x40000010)
620 #define TPM_RH_AUTH_FF (TPM_RH) (0x4000010F)
621 #define TPM_RH_ACT_0 (TPM_RH) (0x40000110)
622 #define TPM_RH_ACT_F (TPM_RH) (0x4000011F)
623 #define TPM_RH_LAST (TPM_RH) (0x4000011F)

```

TPM 2.0 Part 2: Table 29 - Definition of TPM_HC Constants

```

624 typedef TPM_HANDLE TPM_HC;
625 #define HR_HANDLE_MASK (TPM_HC) (0x00FFFFFF)
626 #define HR_RANGE_MASK (TPM_HC) (0xFF000000)
627 #define HR_SHIFT (TPM_HC) (24)
628 #define HR_PCR (TPM_HC) ((TPM_HT_PCR<<HR_SHIFT))
629 #define HR_HMAC_SESSION (TPM_HC) ((TPM_HT_HMAC_SESSION<<HR_SHIFT))
630 #define HR_POLICY_SESSION (TPM_HC) ((TPM_HT_POLICY_SESSION<<HR_SHIFT))
631 #define HR_TRANSIENT (TPM_HC) ((TPM_HT_TRANSIENT<<HR_SHIFT))
632 #define HR_PERSISTENT (TPM_HC) ((TPM_HT_PERSISTENT<<HR_SHIFT))
633 #define HR_NV_INDEX (TPM_HC) ((TPM_HT_NV_INDEX<<HR_SHIFT))
634 #define HR_PERMANENT (TPM_HC) ((TPM_HT_PERMANENT<<HR_SHIFT))
635 #define PCR_FIRST (TPM_HC) ((HR_PCR+0))
636 #define PCR_LAST (TPM_HC) ((PCR_FIRST+IMPLEMENTATION_PCR-1))
637 #define HMAC_SESSION_FIRST (TPM_HC) ((HR_HMAC_SESSION+0))
638 #define HMAC_SESSION_LAST (TPM_HC) ((HMAC_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
639 #define LOADED_SESSION_FIRST (TPM_HC) (HMAC_SESSION_FIRST)
640 #define LOADED_SESSION_LAST (TPM_HC) (HMAC_SESSION_LAST)
641 #define POLICY_SESSION_FIRST (TPM_HC) ((HR_POLICY_SESSION+0))
642 #define POLICY_SESSION_LAST \
643     (TPM_HC) ((POLICY_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
644 #define TRANSIENT_FIRST (TPM_HC) ((HR_TRANSIENT+0))
645 #define ACTIVE_SESSION_FIRST (TPM_HC) (POLICY_SESSION_FIRST)
646 #define ACTIVE_SESSION_LAST (TPM_HC) (POLICY_SESSION_LAST)
647 #define TRANSIENT_LAST (TPM_HC) ((TRANSIENT_FIRST+MAX_LOADED_OBJECTS-1))
648 #define PERSISTENT_FIRST (TPM_HC) ((HR_PERSISTENT+0))
649 #define PERSISTENT_LAST (TPM_HC) ((PERSISTENT_FIRST+0x00FFFFFF))

```

```

650 #define PLATFORM_PERSISTENT (TPM_HC) ((PERSISTENT_FIRST+0x00800000))
651 #define NV_INDEX_FIRST (TPM_HC) ((HR_NV_INDEX+0))
652 #define NV_INDEX_LAST (TPM_HC) ((NV_INDEX_FIRST+0x00FFFFFF))
653 #define PERMANENT_FIRST (TPM_HC) (TPM_RH_FIRST)
654 #define PERMANENT_LAST (TPM_HC) (TPM_RH_LAST)
655 #define HR_NV_AC (TPM_HC) (((TPM_HT_NV_INDEX<<HR_SHIFT)+0xD00000))
656 #define NV_AC_FIRST (TPM_HC) ((HR_NV_AC+0))
657 #define NV_AC_LAST (TPM_HC) ((HR_NV_AC+0x0000FFFF))
658 #define HR_AC (TPM_HC) ((TPM_HT_AC<<HR_SHIFT))
659 #define AC_FIRST (TPM_HC) ((HR_AC+0))
660 #define AC_LAST (TPM_HC) ((HR_AC+0x0000FFFF))
661
662 #define TYPE_OF_TPMA_ALGORITHM UINT32
663 #define TPMA_ALGORITHM_TO_UINT32(a) (*(UINT32 *)&(a))
664 #define UINT32_TO_TPMA_ALGORITHM(a) (*(TPMA_ALGORITHM *)&(a))
665 #define TPMA_ALGORITHM_TO_BYTE_ARRAY(i, a) \
666     UINT32_TO_BYTE_ARRAY(TPMA_ALGORITHM_TO_UINT32(i), (a))
667 #define BYTE_ARRAY_TO_TPMA_ALGORITHM(i, a) \
668     {UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
669     i = UINT32_TO_TPMA_ALGORITHM(x); \
670     }
671 #if USE_BIT_FIELD_STRUCTURES
672 typedef struct TPMA_ALGORITHM { // Table 2:30
673     unsigned asymmetric : 1;
674     unsigned symmetric : 1;
675     unsigned hash : 1;
676     unsigned object : 1;
677     unsigned Reserved_bits_at_4 : 4;
678     unsigned signing : 1;
679     unsigned encrypting : 1;
680     unsigned method : 1;
681     unsigned Reserved_bits_at_11 : 21;
682 } TPMA_ALGORITHM;

```

This is the initializer for a TPMA_ALGORITHM structure

```

683 #define TPMA_ALGORITHM_INITIALIZER( \
684     asymmetric, symmetric, hash, object, bits_at_4, \
685     signing, encrypting, method, bits_at_11) \
686     {asymmetric, symmetric, hash, object, bits_at_4, \
687     signing, encrypting, method, bits_at_11}
688 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:30 TPMA_ALGORITHM using bit masking

```

689 typedef UINT32 TPMA_ALGORITHM;
690 #define TYPE_OF_TPMA_ALGORITHM UINT32
691 #define TPMA_ALGORITHM_asymmetric ((TPMA_ALGORITHM)1 << 0)
692 #define TPMA_ALGORITHM_symmetric ((TPMA_ALGORITHM)1 << 1)
693 #define TPMA_ALGORITHM_hash ((TPMA_ALGORITHM)1 << 2)
694 #define TPMA_ALGORITHM_object ((TPMA_ALGORITHM)1 << 3)
695 #define TPMA_ALGORITHM_signing ((TPMA_ALGORITHM)1 << 8)
696 #define TPMA_ALGORITHM_encrypting ((TPMA_ALGORITHM)1 << 9)
697 #define TPMA_ALGORITHM_method ((TPMA_ALGORITHM)1 << 10)

```

This is the initializer for a TPMA_ALGORITHM bit array.

```

698 #define TPMA_ALGORITHM_INITIALIZER( \
699     asymmetric, symmetric, hash, object, bits_at_4, \
700     signing, encrypting, method, bits_at_11) \
701     (TPMA_ALGORITHM) ( \
702     (asymmetric << 0) + (symmetric << 1) + (hash << 2) + \
703     (object << 3) + (signing << 8) + (encrypting << 9) + \
704     (method << 10))

```

```

705 #endif // USE_BIT_FIELD_STRUCTURES
706
707 #define TYPE_OF_TPMA_OBJECT UINT32
708 #define TPMA_OBJECT_TO_UINT32(a)      (*(UINT32 *)&(a))
709 #define UINT32_TO_TPMA_OBJECT(a)      (*(TPMA_OBJECT *)&(a))
710 #define TPMA_OBJECT_TO_BYTE_ARRAY(i, a) \
711     UINT32_TO_BYTE_ARRAY(TPMA_OBJECT_TO_UINT32(i)), (a) \
712 #define BYTE_ARRAY_TO_TPMA_OBJECT(i, a) \
713     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_OBJECT(x); }
714 #if USE_BIT_FIELD_STRUCTURES
715 typedef struct TPMA_OBJECT { // Table 2:31
716     unsigned Reserved_bit_at_0 : 1;
717     unsigned fixedTPM : 1;
718     unsigned stClear : 1;
719     unsigned Reserved_bit_at_3 : 1;
720     unsigned fixedParent : 1;
721     unsigned sensitiveDataOrigin : 1;
722     unsigned userWithAuth : 1;
723     unsigned adminWithPolicy : 1;
724     unsigned Reserved_bits_at_8 : 2;
725     unsigned noDA : 1;
726     unsigned encryptedDuplication : 1;
727     unsigned Reserved_bits_at_12 : 4;
728     unsigned restricted : 1;
729     unsigned decrypt : 1;
730     unsigned sign : 1;
731     unsigned x509sign : 1;
732     unsigned Reserved_bits_at_20 : 12;
733 } TPMA_OBJECT;

```

This is the initializer for a TPMA_OBJECT structure

```

734 #define TPMA_OBJECT_INITIALIZER( \
735     bit_at_0, fixedtpm, stclear, \
736     bit_at_3, fixedparent, sensitivedataorigin, \
737     userwithauth, adminwithpolicy, bits_at_8, \
738     noda, encryptedduplication, bits_at_12, \
739     restricted, decrypt, sign, \
740     x509sign, bits_at_20) \
741 {bit_at_0, fixedtpm, stclear, \
742     bit_at_3, fixedparent, sensitivedataorigin, \
743     userwithauth, adminwithpolicy, bits_at_8, \
744     noda, encryptedduplication, bits_at_12, \
745     restricted, decrypt, sign, \
746     x509sign, bits_at_20}
747 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:31 TPMA_OBJECT using bit masking

```

748 typedef UINT32 TPMA_OBJECT;
749 #define TYPE_OF_TPMA_OBJECT UINT32
750 #define TPMA_OBJECT_fixedTPM ((TPMA_OBJECT)1 << 1)
751 #define TPMA_OBJECT_stClear ((TPMA_OBJECT)1 << 2)
752 #define TPMA_OBJECT_fixedParent ((TPMA_OBJECT)1 << 4)
753 #define TPMA_OBJECT_sensitiveDataOrigin ((TPMA_OBJECT)1 << 5)
754 #define TPMA_OBJECT_userWithAuth ((TPMA_OBJECT)1 << 6)
755 #define TPMA_OBJECT_adminWithPolicy ((TPMA_OBJECT)1 << 7)
756 #define TPMA_OBJECT_noDA ((TPMA_OBJECT)1 << 10)
757 #define TPMA_OBJECT_encryptedDuplication ((TPMA_OBJECT)1 << 11)
758 #define TPMA_OBJECT_restricted ((TPMA_OBJECT)1 << 16)
759 #define TPMA_OBJECT_decrypt ((TPMA_OBJECT)1 << 17)
760 #define TPMA_OBJECT_sign ((TPMA_OBJECT)1 << 18)
761 #define TPMA_OBJECT_x509sign ((TPMA_OBJECT)1 << 19)

```

This is the initializer for a TPMA_OBJECT bit array.

```

762 #define TPMA_OBJECT_INITIALIZER(                                \
763     bit_at_0,          fixedtpm,          stclear,          \
764     bit_at_3,          fixedparent,        sensitivedataorigin, \
765     userwithauth,      adminwithpolicy,    bits_at_8,        \
766     noda,              encryptedduplication, bits_at_12,       \
767     restricted,        decrypt,            sign,             \
768     x509sign,          bits_at_20)        \
769 (TPMA_OBJECT) (
770     (fixedtpm << 1)          + (stclear << 2)          +
771     (fixedparent << 4)       + (sensitivedataorigin << 5) +
772     (userwithauth << 6)      + (adminwithpolicy << 7)    +
773     (noda << 10)             + (encryptedduplication << 11) +
774     (restricted << 16)       + (decrypt << 17)          +
775     (sign << 18)             + (x509sign << 19))
776 #endif // USE_BIT_FIELD_STRUCTURES
777
778 #define TYPE_OF_TPMA_SESSION      UINT8
779 #define TPMA_SESSION_TO_UINT8(a)  (*(UINT8 *)&(a))
780 #define UINT8_TO_TPMA_SESSION(a)  (*(TPMA_SESSION *)&(a))
781 #define TPMA_SESSION_TO_BYTE_ARRAY(i, a) \
782     UINT8_TO_BYTE_ARRAY((TPMA_SESSION_TO_UINT8(i)), (a))
783 #define BYTE_ARRAY_TO_TPMA_SESSION(i, a) \
784     { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_SESSION(x); }
785 #if USE_BIT_FIELD_STRUCTURES
786 typedef struct TPMA_SESSION { // Table 2:32
787     unsigned continueSession : 1;
788     unsigned auditExclusive   : 1;
789     unsigned auditReset       : 1;
790     unsigned Reserved_bits_at_3 : 2;
791     unsigned decrypt          : 1;
792     unsigned encrypt          : 1;
793     unsigned audit            : 1;
794 } TPMA_SESSION;

```

This is the initializer for a TPMA_SESSION structure

```

795 #define TPMA_SESSION_INITIALIZER(                                \
796     continuesession, auditexclusive, auditreset, bits_at_3, \
797     decrypt,          encrypt,          audit)          \
798 { continuesession, auditexclusive, auditreset, bits_at_3, \
799     decrypt,          encrypt,          audit}
800 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:32 TPMA_SESSION using bit masking

```

801 typedef UINT8      TPMA_SESSION;
802 #define TYPE_OF_TPMA_SESSION      UINT8
803 #define TPMA_SESSION_continueSession ((TPMA_SESSION)1 << 0)
804 #define TPMA_SESSION_auditExclusive ((TPMA_SESSION)1 << 1)
805 #define TPMA_SESSION_auditReset     ((TPMA_SESSION)1 << 2)
806 #define TPMA_SESSION_decrypt        ((TPMA_SESSION)1 << 5)
807 #define TPMA_SESSION_encrypt        ((TPMA_SESSION)1 << 6)
808 #define TPMA_SESSION_audit          ((TPMA_SESSION)1 << 7)

```

This is the initializer for a TPMA_SESSION bit array.

```

809 #define TPMA_SESSION_INITIALIZER(                                \
810     continuesession, auditexclusive, auditreset, bits_at_3, \
811     decrypt,          encrypt,          audit)          \
812 (TPMA_SESSION) (
813     (continuesession << 0) + (auditexclusive << 1) +
814     (auditreset << 2)      + (decrypt << 5)          +

```



```

815         (encrypt << 6)          + (audit << 7))
816 #endif // USE_BIT_FIELD_STRUCTURES
817
818 #define TYPE_OF_TPMA_LOCALITY    UINT8
819 #define TPMA_LOCALITY_TO_UINT8(a)  (*(UINT8 *)&(a))
820 #define UINT8_TO_TPMA_LOCALITY(a)  (*(TPMA_LOCALITY *)&(a))
821 #define TPMA_LOCALITY_TO_BYTE_ARRAY(i, a) \
822     UINT8_TO_BYTE_ARRAY((TPMA_LOCALITY_TO_UINT8(i)), (a)) \
823 #define BYTE_ARRAY_TO_TPMA_LOCALITY(i, a) \
824     { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_LOCALITY(x); }
825 #if USE_BIT_FIELD_STRUCTURES
826 typedef struct TPMA_LOCALITY {           // Table 2:33
827     unsigned    TPM_LOC_ZERO           : 1;
828     unsigned    TPM_LOC_ONE            : 1;
829     unsigned    TPM_LOC_TWO            : 1;
830     unsigned    TPM_LOC_THREE          : 1;
831     unsigned    TPM_LOC_FOUR           : 1;
832     unsigned    Extended                : 3;
833 } TPMA_LOCALITY;

```

This is the initializer for a TPMA_LOCALITY structure

```

834 #define TPMA_LOCALITY_INITIALIZER( \
835     tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three, \
836     tpm_loc_four, extended) \
837     {tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three, \
838     tpm_loc_four, extended}
839 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:33 TPMA_LOCALITY using bit masking

```

840 typedef UINT8 TPMA_LOCALITY;
841 #define TYPE_OF_TPMA_LOCALITY    UINT8
842 #define TPMA_LOCALITY_TPM_LOC_ZERO ((TPMA_LOCALITY)1 << 0)
843 #define TPMA_LOCALITY_TPM_LOC_ONE  ((TPMA_LOCALITY)1 << 1)
844 #define TPMA_LOCALITY_TPM_LOC_TWO  ((TPMA_LOCALITY)1 << 2)
845 #define TPMA_LOCALITY_TPM_LOC_THREE ((TPMA_LOCALITY)1 << 3)
846 #define TPMA_LOCALITY_TPM_LOC_FOUR ((TPMA_LOCALITY)1 << 4)
847 #define TPMA_LOCALITY_Extended_SHIFT 5
848 #define TPMA_LOCALITY_Extended ((TPMA_LOCALITY)0x7 << 5)

```

This is the initializer for a TPMA_LOCALITY bit array.

```

849 #define TPMA_LOCALITY_INITIALIZER( \
850     tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three, \
851     tpm_loc_four, extended) \
852     (TPMA_LOCALITY) ( \
853     (tpm_loc_zero << 0) + (tpm_loc_one << 1) + (tpm_loc_two << 2) + \
854     (tpm_loc_three << 3) + (tpm_loc_four << 4) + (extended << 5))
855 #endif // USE_BIT_FIELD_STRUCTURES
856
857 #define TYPE_OF_TPMA_PERMANENT    UINT32
858 #define TPMA_PERMANENT_TO_UINT32(a)  (*(UINT32 *)&(a))
859 #define UINT32_TO_TPMA_PERMANENT(a)  (*(TPMA_PERMANENT *)&(a))
860 #define TPMA_PERMANENT_TO_BYTE_ARRAY(i, a) \
861     UINT32_TO_BYTE_ARRAY((TPMA_PERMANENT_TO_UINT32(i)), (a)) \
862 #define BYTE_ARRAY_TO_TPMA_PERMANENT(i, a) \
863     {UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
864     i = UINT32_TO_TPMA_PERMANENT(x); \
865     }
866 #if USE_BIT_FIELD_STRUCTURES
867 typedef struct TPMA_PERMANENT {           // Table 2:34
868     unsigned    ownerAuthSet           : 1;
869     unsigned    endorsementAuthSet     : 1;

```

```

870     unsigned    lockoutAuthSet      : 1;
871     unsigned    Reserved_bits_at_3  : 5;
872     unsigned    disableClear        : 1;
873     unsigned    inLockout           : 1;
874     unsigned    tpmGeneratedEPS     : 1;
875     unsigned    Reserved_bits_at_11 : 21;
876 } TPMA_PERMANENT;

```

This is the initializer for a TPMA_PERMANENT structure

```

877 #define TPMA_PERMANENT_INITIALIZER(                                     \
878     ownerauthset,      endorsementauthset, lockoutauthset,           \
879     bits_at_3,         disableclear,      inlockout,                \
880     tpmgenerateddeps,  bits_at_11)                                     \
881 {ownerauthset,      endorsementauthset, lockoutauthset,           \
882     bits_at_3,         disableclear,      inlockout,                \
883     tpmgenerateddeps,  bits_at_11}                                     \
884 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:34 TPMA_PERMANENT using bit masking

```

885 typedef UINT32      TPMA_PERMANENT;
886 #define TYPE_OF_TPMA_PERMANENT      UINT32
887 #define TPMA_PERMANENT_ownerAuthSet ((TPMA_PERMANENT)1 << 0)
888 #define TPMA_PERMANENT_endorsementAuthSet ((TPMA_PERMANENT)1 << 1)
889 #define TPMA_PERMANENT_lockoutAuthSet ((TPMA_PERMANENT)1 << 2)
890 #define TPMA_PERMANENT_disableClear ((TPMA_PERMANENT)1 << 8)
891 #define TPMA_PERMANENT_inLockout ((TPMA_PERMANENT)1 << 9)
892 #define TPMA_PERMANENT_tpmGeneratedEPS ((TPMA_PERMANENT)1 << 10)

```

This is the initializer for a TPMA_PERMANENT bit array.

```

893 #define TPMA_PERMANENT_INITIALIZER(                                     \
894     ownerauthset,      endorsementauthset, lockoutauthset,           \
895     bits_at_3,         disableclear,      inlockout,                \
896     tpmgenerateddeps,  bits_at_11)                                     \
897 (TPMA_PERMANENT) (                                                    \
898     (ownerauthset << 0)      + (endorsementauthset << 1) +          \
899     (lockoutauthset << 2)    + (disableclear << 8)      +          \
900     (inlockout << 9)        + (tpmgenerateddeps << 10))              \
901 #endif // USE_BIT_FIELD_STRUCTURES
902
903 #define TYPE_OF_TPMA_STARTUP_CLEAR      UINT32
904 #define TPMA_STARTUP_CLEAR_TO_UINT32(a) (*(UINT32 *)&(a))
905 #define UINT32_TO_TPMA_STARTUP_CLEAR(a) (*(TPMA_STARTUP_CLEAR *)&(a))
906 #define TPMA_STARTUP_CLEAR_TO_BYTE_ARRAY(i, a)                        \
907     UINT32_TO_BYTE_ARRAY(TPMA_STARTUP_CLEAR_TO_UINT32(i), (a))
908 #define BYTE_ARRAY_TO_TPMA_STARTUP_CLEAR(i, a)                        \
909     {UINT32 x = BYTE_ARRAY_TO_UINT32(a);                               \
910     i = UINT32_TO_TPMA_STARTUP_CLEAR(x);                               \
911     }
912 #if USE_BIT_FIELD_STRUCTURES
913 typedef struct TPMA_STARTUP_CLEAR {                                     // Table 2:35
914     unsigned    phEnable          : 1;
915     unsigned    shEnable          : 1;
916     unsigned    ehEnable          : 1;
917     unsigned    phEnableNV        : 1;
918     unsigned    Reserved_bits_at_4 : 27;
919     unsigned    orderly           : 1;
920 } TPMA_STARTUP_CLEAR;

```

This is the initializer for a TPMA_STARTUP_CLEAR structure

```

921 #define TPMA_STARTUP_CLEAR_INITIALIZER(

```



```

922         phenable, shenable, ehenable, phenablenv, bits_at_4, orderly) \
923     {phenable, shenable, ehenable, phenablenv, bits_at_4, orderly}
924 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:35 TPMA_STARTUP_CLEAR using bit masking

```

925 typedef UINT32 TPMA_STARTUP_CLEAR;
926 #define TYPE_OF_TPMA_STARTUP_CLEAR UINT32
927 #define TPMA_STARTUP_CLEAR_phEnable ((TPMA_STARTUP_CLEAR)1 << 0)
928 #define TPMA_STARTUP_CLEAR_shEnable ((TPMA_STARTUP_CLEAR)1 << 1)
929 #define TPMA_STARTUP_CLEAR_ehEnable ((TPMA_STARTUP_CLEAR)1 << 2)
930 #define TPMA_STARTUP_CLEAR_phEnableNV ((TPMA_STARTUP_CLEAR)1 << 3)
931 #define TPMA_STARTUP_CLEAR_orderly ((TPMA_STARTUP_CLEAR)1 << 31)

```

This is the initializer for a TPMA_STARTUP_CLEAR bit array.

```

932 #define TPMA_STARTUP_CLEAR_INITIALIZER( \
933     phenable, shenable, ehenable, phenablenv, bits_at_4, orderly) \
934     (TPMA_STARTUP_CLEAR) ( \
935         (phenable << 0) + (shenable << 1) + (ehenable << 2) + \
936         (phenablenv << 3) + (orderly << 31))
937 #endif // USE_BIT_FIELD_STRUCTURES
938
939 #define TYPE_OF_TPMA_MEMORY UINT32
940 #define TPMA_MEMORY_TO_UINT32(a) (*(UINT32 *)&(a))
941 #define UINT32_TO_TPMA_MEMORY(a) (*(TPMA_MEMORY *)&(a))
942 #define TPMA_MEMORY_TO_BYTE_ARRAY(i, a) \
943     UINT32_TO_BYTE_ARRAY((TPMA_MEMORY_TO_UINT32(i)), (a))
944 #define BYTE_ARRAY_TO_TPMA_MEMORY(i, a) \
945     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_MEMORY(x); }
946 #if USE_BIT_FIELD_STRUCTURES
947 typedef struct TPMA_MEMORY { // Table 2:36
948     unsigned sharedRAM : 1;
949     unsigned sharedNV : 1;
950     unsigned objectCopiedToRam : 1;
951     unsigned Reserved_bits_at_3 : 29;
952 } TPMA_MEMORY;

```

This is the initializer for a TPMA_MEMORY structure

```

953 #define TPMA_MEMORY_INITIALIZER( \
954     sharedram, sharednv, objectcopiedtoram, bits_at_3) \
955     {sharedram, sharednv, objectcopiedtoram, bits_at_3}
956 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:36 TPMA_MEMORY using bit masking

```

957 typedef UINT32 TPMA_MEMORY;
958 #define TYPE_OF_TPMA_MEMORY UINT32
959 #define TPMA_MEMORY_sharedRAM ((TPMA_MEMORY)1 << 0)
960 #define TPMA_MEMORY_sharedNV ((TPMA_MEMORY)1 << 1)
961 #define TPMA_MEMORY_objectCopiedToRam ((TPMA_MEMORY)1 << 2)

```

This is the initializer for a TPMA_MEMORY bit array.

```

962 #define TPMA_MEMORY_INITIALIZER( \
963     sharedram, sharednv, objectcopiedtoram, bits_at_3) \
964     (TPMA_MEMORY) ( \
965         (sharedram << 0) + (sharednv << 1) + (objectcopiedtoram << 2))
966 #endif // USE_BIT_FIELD_STRUCTURES
967
968 #define TYPE_OF_TPMA_CC UINT32
969 #define TPMA_CC_TO_UINT32(a) (*(UINT32 *)&(a))

```

```

970 #define UINT32_TO_TPMA_CC(a)      (*((TPMA_CC *)&(a)))
971 #define TPMA_CC_TO_BYTE_ARRAY(i, a) \
972     UINT32_TO_BYTE_ARRAY((TPMA_CC_TO_UINT32(i)), (a)) \
973 #define BYTE_ARRAY_TO_TPMA_CC(i, a) \
974     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_CC(x); }
975 #if USE_BIT_FIELD_STRUCTURES
976 typedef struct TPMA_CC {           // Table 2:37
977     unsigned    commandIndex      : 16;
978     unsigned    Reserved_bits_at_16 : 6;
979     unsigned    nv                 : 1;
980     unsigned    extensive          : 1;
981     unsigned    flushed            : 1;
982     unsigned    cHandles           : 3;
983     unsigned    rHandle            : 1;
984     unsigned    v                  : 1;
985     unsigned    Reserved_bits_at_30 : 2;
986 } TPMA_CC;

```

This is the initializer for a TPMA_CC structure

```

987 #define TPMA_CC_INITIALIZER( \
988     commandindex, bits_at_16, nv, extensive, flushed, \
989     chandles, rhandle, v, bits_at_30) \
990     {commandindex, bits_at_16, nv, extensive, flushed, \
991     chandles, rhandle, v, bits_at_30}
992 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:37 TPMA_CC using bit masking

```

993 typedef UINT32 TPMA_CC;
994 #define TYPE_OF_TPMA_CC      UINT32
995 #define TPMA_CC_commandIndex_SHIFT 0
996 #define TPMA_CC_commandIndex      ((TPMA_CC)0xffff << 0)
997 #define TPMA_CC_nv                 ((TPMA_CC)1 << 22)
998 #define TPMA_CC_extensive          ((TPMA_CC)1 << 23)
999 #define TPMA_CC_flushed            ((TPMA_CC)1 << 24)
1000 #define TPMA_CC_cHandles_SHIFT    25
1001 #define TPMA_CC_cHandles           ((TPMA_CC)0x7 << 25)
1002 #define TPMA_CC_rHandle            ((TPMA_CC)1 << 28)
1003 #define TPMA_CC_v                  ((TPMA_CC)1 << 29)

```

This is the initializer for a TPMA_CC bit array.

```

1004 #define TPMA_CC_INITIALIZER( \
1005     commandindex, bits_at_16, nv, extensive, flushed, \
1006     chandles, rhandle, v, bits_at_30) \
1007     (TPMA_CC) ( \
1008         (commandindex << 0) + (nv << 22) + (extensive << 23) + \
1009         (flushed << 24) + (chandles << 25) + (rhandle << 28) + \
1010         (v << 29))
1011 #endif // USE_BIT_FIELD_STRUCTURES
1012
1013 #define TYPE_OF_TPMA_MODES      UINT32
1014 #define TPMA_MODES_TO_UINT32(a) (*((UINT32 *)&(a)))
1015 #define UINT32_TO_TPMA_MODES(a) (*((TPMA_MODES *)&(a)))
1016 #define TPMA_MODES_TO_BYTE_ARRAY(i, a) \
1017     UINT32_TO_BYTE_ARRAY((TPMA_MODES_TO_UINT32(i)), (a)) \
1018 #define BYTE_ARRAY_TO_TPMA_MODES(i, a) \
1019     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_MODES(x); }
1020 #if USE_BIT_FIELD_STRUCTURES
1021 typedef struct TPMA_MODES {       // Table 2:38
1022     unsigned    FIPS_140_2        : 1;
1023     unsigned    Reserved_bits_at_1 : 31;
1024 } TPMA_MODES;

```

This is the initializer for a TPMA_MODES structure

```
1025 #define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {fips_140_2, bits_at_1}
1026 #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:38 TPMA_MODES using bit masking

```
1027 typedef UINT32          TPMA_MODES;
1028 #define TYPE_OF_TPMA_MODES  UINT32
1029 #define TPMA_MODES_FIPS_140_2 ((TPMA_MODES)1 << 0)
```

This is the initializer for a TPMA_MODES bit array.

```
1030 #define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) \
1031     (TPMA_MODES) ( \
1032         (fips_140_2 << 0)) \
1033 #endif // USE_BIT_FIELD_STRUCTURES \
1034 \
1035 #define TYPE_OF_TPMA_X509_KEY_USAGE UINT32
1036 #define TPMA_X509_KEY_USAGE_TO_UINT32(a) ((UINT32 *)&(a))
1037 #define UINT32_TO_TPMA_X509_KEY_USAGE(a) ((TPMA_X509_KEY_USAGE *)&(a))
1038 #define TPMA_X509_KEY_USAGE_TO_BYTE_ARRAY(i, a) \
1039     UINT32_TO_BYTE_ARRAY((TPMA_X509_KEY_USAGE_TO_UINT32(i)), (a)) \
1040 #define BYTE_ARRAY_TO_TPMA_X509_KEY_USAGE(i, a) \
1041     {UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
1042     i = UINT32_TO_TPMA_X509_KEY_USAGE(x); \
1043     } \
1044 #if USE_BIT_FIELD_STRUCTURES
1045 typedef struct TPMA_X509_KEY_USAGE { // Table 2:39
1046     unsigned Reserved_bits_at_0 : 23;
1047     unsigned decipherOnly       : 1;
1048     unsigned encipherOnly       : 1;
1049     unsigned cRLSign            : 1;
1050     unsigned keyCertSign        : 1;
1051     unsigned keyAgreement       : 1;
1052     unsigned dataEncipherment   : 1;
1053     unsigned keyEncipherment    : 1;
1054     unsigned nonrepudiation     : 1;
1055     unsigned digitalSignature    : 1;
1056 } TPMA_X509_KEY_USAGE;
```

This is the initializer for a TPMA_X509_KEY_USAGE structure

```
1057 #define TPMA_X509_KEY_USAGE_INITIALIZER( \
1058     bits_at_0, decipheronly, encipheronly, \
1059     crlsign, keycertsign, keyagreement, \
1060     dataencipherment, keyencipherment, nonrepudiation, \
1061     digitalsignature) \
1062     {bits_at_0, decipheronly, encipheronly, \
1063     crlsign, keycertsign, keyagreement, \
1064     dataencipherment, keyencipherment, nonrepudiation, \
1065     digitalsignature} \
1066 #else // USE_BIT_FIELD_STRUCTURES
```

This implements Table 2:39 TPMA_X509_KEY_USAGE using bit masking

```
1067 typedef UINT32          TPMA_X509_KEY_USAGE;
1068 #define TYPE_OF_TPMA_X509_KEY_USAGE  UINT32
1069 #define TPMA_X509_KEY_USAGE_decipherOnly ((TPMA_X509_KEY_USAGE)1 << 23)
1070 #define TPMA_X509_KEY_USAGE_encipherOnly ((TPMA_X509_KEY_USAGE)1 << 24)
1071 #define TPMA_X509_KEY_USAGE_cRLSign      ((TPMA_X509_KEY_USAGE)1 << 25)
1072 #define TPMA_X509_KEY_USAGE_keyCertSign  ((TPMA_X509_KEY_USAGE)1 << 26)
1073 #define TPMA_X509_KEY_USAGE_keyAgreement ((TPMA_X509_KEY_USAGE)1 << 27)
```

```

1074 #define TPMA_X509_KEY_USAGE_dataEncipherment ((TPMA_X509_KEY_USAGE)1 << 28)
1075 #define TPMA_X509_KEY_USAGE_keyEncipherment ((TPMA_X509_KEY_USAGE)1 << 29)
1076 #define TPMA_X509_KEY_USAGE_nonrepudiation ((TPMA_X509_KEY_USAGE)1 << 30)
1077 #define TPMA_X509_KEY_USAGE_digitalSignature ((TPMA_X509_KEY_USAGE)1 << 31)

```

This is the initializer for a TPMA_X509_KEY_USAGE bit array.

```

1078 #define TPMA_X509_KEY_USAGE_INITIALIZER(                                     \
1079     bits_at_0,      decipheronly,      encipheronly,                     \
1080     crlsign,        keycertsign,       keyagreement,                     \
1081     dataencipherment, keyencipherment,  nonrepudiation,                   \
1082     digitalsignature)                                                     \
1083     (TPMA_X509_KEY_USAGE) (                                              \
1084         (decipheronly << 23)      + (encipheronly << 24)      +          \
1085         (crlsign << 25)           + (keycertsign << 26)       +          \
1086         (keyagreement << 27)      + (dataencipherment << 28) +          \
1087         (keyencipherment << 29)   + (nonrepudiation << 30)   +          \
1088         (digitalsignature << 31))                                         \
1089 #endif // USE_BIT_FIELD_STRUCTURES
1090
1091 #define TYPE_OF TPMA_ACT          UINT32
1092 #define TPMA_ACT_TO_UINT32(a)     (*((UINT32 *)&(a)))
1093 #define UINT32_TO_TPMA_ACT(a)     (*((TPMA_ACT *)&(a)))
1094 #define TPMA_ACT_TO_BYTE_ARRAY(i, a)                                     \
1095     UINT32_TO_BYTE_ARRAY((TPMA_ACT_TO_UINT32(i)), (a))
1096 #define BYTE_ARRAY_TO_TPMA_ACT(i, a)                                     \
1097     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_ACT(x); }
1098 #if USE_BIT_FIELD_STRUCTURES
1099 typedef struct TPMA_ACT {                                                // Table 2:40
1100     unsigned signaled : 1;
1101     unsigned preserveSignaled : 1;
1102     unsigned Reserved_bits_at_2 : 30;
1103 } TPMA_ACT;

```

This is the initializer for a TPMA_ACT structure

```

1104 #define TPMA_ACT_INITIALIZER(signaled, preservesignaled, bits_at_2)      \
1105     {signaled, preservesignaled, bits_at_2}
1106 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:40 TPMA_ACT using bit masking

```

1107 typedef UINT32          TPMA_ACT;
1108 #define TYPE_OF TPMA_ACT          UINT32
1109 #define TPMA_ACT_signaled ((TPMA_ACT)1 << 0)
1110 #define TPMA_ACT_preserveSignaled ((TPMA_ACT)1 << 1)

```

This is the initializer for a TPMA_ACT bit array.

```

1111 #define TPMA_ACT_INITIALIZER(signaled, preservesignaled, bits_at_2)      \
1112     (TPMA_ACT) (                                                         \
1113         (signaled << 0) + (preservesignaled << 1))
1114 #endif // USE_BIT_FIELD_STRUCTURES
1115
1116 typedef BYTE          TPMI_YES_NO; // Table 2:41
1117
1118 typedef TPM_HANDLE    TPMI_DH_OBJECT; // Table 2:42
1119
1120 typedef TPM_HANDLE    TPMI_DH_PARENT; // Table 2:43
1121
1122 typedef TPM_HANDLE    TPMI_DH_PERSISTENT; // Table 2:44
1123
1124 typedef TPM_HANDLE    TPMI_DH_ENTITY; // Table 2:45
1125

```

```

1126 typedef TPM_HANDLE TPMI_DH_PCR; // Table 2:46
1127
1128 typedef TPM_HANDLE TPMI_SH_AUTH_SESSION; // Table 2:47
1129
1130 typedef TPM_HANDLE TPMI_SH_HMAC; // Table 2:48
1131
1132 typedef TPM_HANDLE TPMI_SH_POLICY; // Table 2:49
1133
1134 typedef TPM_HANDLE TPMI_DH_CONTEXT; // Table 2:50
1135
1136 typedef TPM_HANDLE TPMI_DH_SAVED; // Table 2:51
1137
1138 typedef TPM_HANDLE TPMI_RH_HIERARCHY; // Table 2:52
1139
1140 typedef TPM_HANDLE TPMI_RH_ENABLES; // Table 2:53
1141
1142 typedef TPM_HANDLE TPMI_RH_HIERARCHY_AUTH; // Table 2:54
1143
1144 typedef TPM_HANDLE TPMI_RH_HIERARCHY_POLICY;
1145
1146 typedef TPM_HANDLE TPMI_RH_PLATFORM; // Table 2:56
1147
1148 typedef TPM_HANDLE TPMI_RH_OWNER; // Table 2:57
1149
1150 typedef TPM_HANDLE TPMI_RH_ENDORSEMENT; // Table 2:58
1151
1152 typedef TPM_HANDLE TPMI_RH_PROVISION; // Table 2:59
1153
1154 typedef TPM_HANDLE TPMI_RH_CLEAR; // Table 2:60
1155
1156 typedef TPM_HANDLE TPMI_RH_NV_AUTH; // Table 2:61
1157
1158 typedef TPM_HANDLE TPMI_RH_LOCKOUT; // Table 2:62
1159
1160 typedef TPM_HANDLE TPMI_RH_NV_INDEX; // Table 2:63
1161
1162 typedef TPM_HANDLE TPMI_RH_AC; // Table 2:64
1163
1164 typedef TPM_HANDLE TPMI_RH_ACT; // Table 2:65
1165
1166 typedef TPM_ALG_ID TPMI_ALG_HASH; // Table 2:66
1167
1168 typedef TPM_ALG_ID TPMI_ALG_ASYM; // Table 2:67
1169
1170 typedef TPM_ALG_ID TPMI_ALG_SYM; // Table 2:68
1171
1172 typedef TPM_ALG_ID TPMI_ALG_SYM_OBJECT; // Table 2:69
1173
1174 typedef TPM_ALG_ID TPMI_ALG_SYM_MODE; // Table 2:70
1175
1176 typedef TPM_ALG_ID TPMI_ALG_KDF; // Table 2:71
1177
1178 typedef TPM_ALG_ID TPMI_ALG_SIG_SCHEME; // Table 2:72
1179
1180 typedef TPM_ALG_ID TPMI_ECC_KEY_EXCHANGE; // Table 2:73
1181
1182 typedef TPM_ST TPMI_ST_COMMAND_TAG; // Table 2:74
1183
1184 typedef TPM_ALG_ID TPMI_ALG_MAC_SCHEME; // Table 2:75
1185
1186 typedef TPM_ALG_ID TPMI_ALG_CIPHER_MODE; // Table 2:76
1187
1188 typedef BYTE TPMS_EMPTY; // Table 2:77
1189
1190 typedef struct { // Table 2:78
1191     TPM_ALG_ID alg;

```

```

1192     TPMA_ALGORITHM          attributes;
1193 } TPMS_ALGORITHM_DESCRIPTION;
1194
1195 typedef union {                // Table 2:79
1196 #if ALG_SHA1
1197     BYTE                sha1[SHA1_DIGEST_SIZE];
1198 #endif // ALG_SHA1
1199 #if ALG_SHA256
1200     BYTE                sha256[SHA256_DIGEST_SIZE];
1201 #endif // ALG_SHA256
1202 #if ALG_SHA384
1203     BYTE                sha384[SHA384_DIGEST_SIZE];
1204 #endif // ALG_SHA384
1205 #if ALG_SHA512
1206     BYTE                sha512[SHA512_DIGEST_SIZE];
1207 #endif // ALG_SHA512
1208 #if ALG_SM3_256
1209     BYTE                sm3_256[SM3_256_DIGEST_SIZE];
1210 #endif // ALG_SM3_256
1211 #if ALG_SHA3_256
1212     BYTE                sha3_256[SHA3_256_DIGEST_SIZE];
1213 #endif // ALG_SHA3_256
1214 #if ALG_SHA3_384
1215     BYTE                sha3_384[SHA3_384_DIGEST_SIZE];
1216 #endif // ALG_SHA3_384
1217 #if ALG_SHA3_512
1218     BYTE                sha3_512[SHA3_512_DIGEST_SIZE];
1219 #endif // ALG_SHA3_512
1220 } TPMU_HA;
1221
1222 typedef struct {                // Table 2:80
1223     TPMI_ALG_HASH        hashAlg;
1224     TPMU_HA              digest;
1225 } TPMT_HA;
1226
1227 typedef union {                // Table 2:81
1228     struct {
1229         UINT16            size;
1230         BYTE              buffer[sizeof(TPMU_HA)];
1231     } t;
1232     TPM2B                b;
1233 } TPM2B_DIGEST;
1234
1235 typedef union {                // Table 2:82
1236     struct {
1237         UINT16            size;
1238         BYTE              buffer[sizeof(TPMT_HA)];
1239     } t;
1240     TPM2B                b;
1241 } TPM2B_DATA;

```

TPM 2.0 Part 2: Table 83 - Definition of Types for TPM2B_NONCE

```

1242 typedef TPM2B_DIGEST        TPM2B_NONCE;

```

TPM 2.0 Part 2: Table 84 - Definition of Types for TPM2B_AUTH

```

1243 typedef TPM2B_DIGEST        TPM2B_AUTH;

```

TPM 2.0 Part 2: Table 85 - Definition of Types for TPM2B_OPERAND

```

1244 typedef TPM2B_DIGEST        TPM2B_OPERAND;
1245
1246 typedef union {                // Table 2:86

```



```

1247     struct {
1248         UINT16      size;
1249         BYTE        buffer[1024];
1250     } t;
1251     TPM2B          b;
1252 } TPM2B_EVENT;
1253
1254 typedef union { // Table 2:87
1255     struct {
1256         UINT16      size;
1257         BYTE        buffer[MAX_DIGEST_BUFFER];
1258     } t;
1259     TPM2B          b;
1260 } TPM2B_MAX_BUFFER;
1261
1262 typedef union { // Table 2:88
1263     struct {
1264         UINT16      size;
1265         BYTE        buffer[MAX_NV_BUFFER_SIZE];
1266     } t;
1267     TPM2B          b;
1268 } TPM2B_MAX_NV_BUFFER;
1269
1270 typedef union { // Table 2:89
1271     struct {
1272         UINT16      size;
1273         BYTE        buffer[sizeof(UINT64)];
1274     } t;
1275     TPM2B          b;
1276 } TPM2B_TIMEOUT;
1277
1278 typedef union { // Table 2:90
1279     struct {
1280         UINT16      size;
1281         BYTE        buffer[MAX_SYM_BLOCK_SIZE];
1282     } t;
1283     TPM2B          b;
1284 } TPM2B_IV;
1285
1286 typedef union { // Table 2:91
1287     TPMT_HA        digest;
1288     TPM_HANDLE     handle;
1289 } TPMU_NAME;
1290
1291 typedef union { // Table 2:92
1292     struct {
1293         UINT16      size;
1294         BYTE        name[sizeof(TPMU_NAME)];
1295     } t;
1296     TPM2B          b;
1297 } TPM2B_NAME;
1298
1299 typedef struct { // Table 2:93
1300     UINT8          sizeofSelect;
1301     BYTE          pcrSelect[PCR_SELECT_MAX];
1302 } TPMS_PCR_SELECT;
1303
1304 typedef struct { // Table 2:94
1305     TPMT_ALG_HASH  hash;
1306     UINT8          sizeofSelect;
1307     BYTE          pcrSelect[PCR_SELECT_MAX];
1308 } TPMS_PCR_SELECTION;
1309
1310 typedef struct { // Table 2:97
1311     TPM_ST          tag;
1312     TPMT_RH_HIERARCHY hierarchy;

```



```

1313     TPM2B_DIGEST                digest;
1314 } TPMT_TK_CREATION;
1315
1316 typedef struct {                // Table 2:98
1317     TPM_ST                tag;
1318     TPMI_RH_HIERARCHY     hierarchy;
1319     TPM2B_DIGEST          digest;
1320 } TPMT_TK_VERIFIED;
1321
1322 typedef struct {                // Table 2:99
1323     TPM_ST                tag;
1324     TPMI_RH_HIERARCHY     hierarchy;
1325     TPM2B_DIGEST          digest;
1326 } TPMT_TK_AUTH;
1327
1328 typedef struct {                // Table 2:100
1329     TPM_ST                tag;
1330     TPMI_RH_HIERARCHY     hierarchy;
1331     TPM2B_DIGEST          digest;
1332 } TPMT_TK_HASHCHECK;
1333
1334 typedef struct {                // Table 2:101
1335     TPM_ALG_ID            alg;
1336     TPMA_ALGORITHM        algProperties;
1337 } TPMS_ALG_PROPERTY;
1338
1339 typedef struct {                // Table 2:102
1340     TPM_PT                property;
1341     UINT32                value;
1342 } TPMS_TAGGED_PROPERTY;
1343
1344 typedef struct {                // Table 2:103
1345     TPM_PT_PCR            tag;
1346     UINT8                sizeofSelect;
1347     BYTE                  pcrSelect[PCR_SELECT_MAX];
1348 } TPMS_TAGGED_PCR_SELECT;
1349
1350 typedef struct {                // Table 2:104
1351     TPM_HANDLE            handle;
1352     TPMT_HA                policyHash;
1353 } TPMS_TAGGED_POLICY;
1354
1355 typedef struct {                // Table 2:105
1356     TPM_HANDLE            handle;
1357     UINT32                timeout;
1358     TPMA_ACT              attributes;
1359 } TPMS_ACT_DATA;
1360
1361 typedef struct {                // Table 2:106
1362     UINT32                count;
1363     TPM_CC                commandCodes[MAX_CAP_CC];
1364 } TPML_CC;
1365
1366 typedef struct {                // Table 2:107
1367     UINT32                count;
1368     TPMA_CC                commandAttributes[MAX_CAP_CC];
1369 } TPML_CCA;
1370
1371 typedef struct {                // Table 2:108
1372     UINT32                count;
1373     TPM_ALG_ID            algorithms[MAX_ALG_LIST_SIZE];
1374 } TPML_ALG;
1375
1376 typedef struct {                // Table 2:109
1377     UINT32                count;
1378     TPM_HANDLE            handle[MAX_CAP_HANDLES];

```

```

1379 } TPML_HANDLE;
1380
1381 typedef struct { // Table 2:110
1382     UINT32 count;
1383     TPM2B_DIGEST digests[8];
1384 } TPML_DIGEST;
1385
1386 typedef struct { // Table 2:111
1387     UINT32 count;
1388     TPMT_HA digests[HASH_COUNT];
1389 } TPML_DIGEST_VALUES;
1390
1391 typedef struct { // Table 2:112
1392     UINT32 count;
1393     TPMS_PCR_SELECTION pcrSelections[HASH_COUNT];
1394 } TPML_PCR_SELECTION;
1395
1396 typedef struct { // Table 2:113
1397     UINT32 count;
1398     TPMS_ALG_PROPERTY algProperties[MAX_CAP_ALGS];
1399 } TPML_ALG_PROPERTY;
1400
1401 typedef struct { // Table 2:114
1402     UINT32 count;
1403     TPMS_TAGGED_PROPERTY tpmProperty[MAX_TPM_PROPERTIES];
1404 } TPML_TAGGED_TPM_PROPERTY;
1405
1406 typedef struct { // Table 2:115
1407     UINT32 count;
1408     TPMS_TAGGED_PCR_SELECT pcrProperty[MAX_PCR_PROPERTIES];
1409 } TPML_TAGGED_PCR_PROPERTY;
1410
1411 typedef struct { // Table 2:116
1412     UINT32 count;
1413     TPM_ECC_CURVE eccCurves[MAX_ECC_CURVES];
1414 } TPML_ECC_CURVE;
1415
1416 typedef struct { // Table 2:117
1417     UINT32 count;
1418     TPMS_TAGGED_POLICY policies[MAX_TAGGED_POLICIES];
1419 } TPML_TAGGED_POLICY;
1420
1421 typedef struct { // Table 2:118
1422     UINT32 count;
1423     TPMS_ACT_DATA actData[MAX_ACT_DATA];
1424 } TPML_ACT_DATA;
1425
1426 typedef union { // Table 2:119
1427     TPML_ALG_PROPERTY algorithms;
1428     TPML_HANDLE handles;
1429     TPML_CCA command;
1430     TPML_CC ppCommands;
1431     TPML_CC auditCommands;
1432     TPML_PCR_SELECTION assignedPCR;
1433     TPML_TAGGED_TPM_PROPERTY tpmProperties;
1434     TPML_TAGGED_PCR_PROPERTY pcrProperties;
1435 #if ALG_ECC
1436     TPML_ECC_CURVE eccCurves;
1437 #endif // ALG_ECC
1438     TPML_TAGGED_POLICY authPolicies;
1439     TPML_ACT_DATA actData;
1440 } TPMU_CAPABILITIES;
1441
1442 typedef struct { // Table 2:120
1443     TPM_CAP capability;
1444     TPMU_CAPABILITIES data;

```

```

1445 } TPMS_CAPABILITY_DATA;
1446
1447 typedef struct { // Table 2:121
1448     UINT64          clock;
1449     UINT32          resetCount;
1450     UINT32          restartCount;
1451     TPMI_YES_NO     safe;
1452 } TPMS_CLOCK_INFO;
1453
1454 typedef struct { // Table 2:122
1455     UINT64          time;
1456     TPMS_CLOCK_INFO clockInfo;
1457 } TPMS_TIME_INFO;
1458
1459 typedef struct { // Table 2:123
1460     TPMS_TIME_INFO  time;
1461     UINT64          firmwareVersion;
1462 } TPMS_TIME_ATTEST_INFO;
1463
1464 typedef struct { // Table 2:124
1465     TPM2B_NAME      name;
1466     TPM2B_NAME      qualifiedName;
1467 } TPMS_CERTIFY_INFO;
1468
1469 typedef struct { // Table 2:125
1470     TPML_PCR_SELECTION pcrSelect;
1471     TPM2B_DIGEST       pcrDigest;
1472 } TPMS_QUOTE_INFO;
1473
1474 typedef struct { // Table 2:126
1475     UINT64          auditCounter;
1476     TPM_ALG_ID      digestAlg;
1477     TPM2B_DIGEST    auditDigest;
1478     TPM2B_DIGEST    commandDigest;
1479 } TPMS_COMMAND_AUDIT_INFO;
1480
1481 typedef struct { // Table 2:127
1482     TPMI_YES_NO     exclusiveSession;
1483     TPM2B_DIGEST    sessionDigest;
1484 } TPMS_SESSION_AUDIT_INFO;
1485
1486 typedef struct { // Table 2:128
1487     TPM2B_NAME      objectName;
1488     TPM2B_DIGEST    creationHash;
1489 } TPMS_CREATION_INFO;
1490
1491 typedef struct { // Table 2:129
1492     TPM2B_NAME      indexName;
1493     UINT16          offset;
1494     TPM2B_MAX_NV_BUFFER nvContents;
1495 } TPMS_NV_CERTIFY_INFO;
1496
1497 typedef struct { // Table 2:130
1498     TPM2B_NAME      indexName;
1499     TPM2B_DIGEST    nvDigest;
1500 } TPMS_NV_DIGEST_CERTIFY_INFO;
1501
1502 typedef TPM_ST      TPMI_ST_ATTEST; // Table 2:131
1503
1504 typedef union { // Table 2:132
1505     TPMS_CERTIFY_INFO      certify;
1506     TPMS_CREATION_INFO     creation;
1507     TPMS_QUOTE_INFO        quote;
1508     TPMS_COMMAND_AUDIT_INFO commandAudit;
1509     TPMS_SESSION_AUDIT_INFO sessionAudit;
1510     TPMS_TIME_ATTEST_INFO  time;

```

```

1511     TPMS_NV_CERTIFY_INFO          nv;
1512     TPMS_NV_DIGEST_CERTIFY_INFO  nvDigest;
1513 } TPMU_ATTEST;
1514
1515 typedef struct { // Table 2:133
1516     TPM_CONSTANTS32    magic;
1517     TPMI_ST_ATTEST     type;
1518     TPM2B_NAME         qualifiedSigner;
1519     TPM2B_DATA         extraData;
1520     TPMS_CLOCK_INFO    clockInfo;
1521     UINT64             firmwareVersion;
1522     TPMU_ATTEST        attested;
1523 } TPMS_ATTEST;
1524
1525 typedef union { // Table 2:134
1526     struct {
1527         UINT16    size;
1528         BYTE      attestationData[sizeof(TPMS_ATTEST)];
1529     }            t;
1530     TPM2B        b;
1531 } TPM2B_ATTEST;
1532
1533 typedef struct { // Table 2:135
1534     TPMI_SH_AUTH_SESSION    sessionHandle;
1535     TPM2B_NONCE             nonce;
1536     TPMA_SESSION            sessionAttributes;
1537     TPM2B_AUTH              hmac;
1538 } TPMS_AUTH_COMMAND;
1539
1540 typedef struct { // Table 2:136
1541     TPM2B_NONCE    nonce;
1542     TPMA_SESSION   sessionAttributes;
1543     TPM2B_AUTH     hmac;
1544 } TPMS_AUTH_RESPONSE;
1545
1546 typedef TPM_KEY_BITS    TPMI_TDES_KEY_BITS; // Table 2:137
1547
1548 typedef TPM_KEY_BITS    TPMI_AES_KEY_BITS; // Table 2:137
1549
1550 typedef TPM_KEY_BITS    TPMI_SM4_KEY_BITS; // Table 2:137
1551
1552 typedef TPM_KEY_BITS    TPMI_CAMELLIA_KEY_BITS; // Table 2:137
1553
1554 typedef union { // Table 2:138
1555     #if ALG_TDES
1556         TPMI_TDES_KEY_BITS    tdes;
1557     #endif // ALG_TDES
1558     #if ALG_AES
1559         TPMI_AES_KEY_BITS     aes;
1560     #endif // ALG_AES
1561     #if ALG_SM4
1562         TPMI_SM4_KEY_BITS     sm4;
1563     #endif // ALG_SM4
1564     #if ALG_CAMELLIA
1565         TPMI_CAMELLIA_KEY_BITS    camellia;
1566     #endif // ALG_CAMELLIA
1567     TPM_KEY_BITS    sym;
1568     #if ALG_XOR
1569         TPMI_ALG_HASH    xor;
1570     #endif // ALG_XOR
1571 } TPMU_SYM_KEY_BITS;
1572
1573 typedef union { // Table 2:139
1574     #if ALG_TDES
1575         TPMI_ALG_SYM_MODE    tdes;
1576     #endif // ALG_TDES

```

```

1577 #if ALG_AES
1578     TPMI_ALG_SYM_MODE      aes;
1579 #endif // ALG_AES
1580 #if ALG_SM4
1581     TPMI_ALG_SYM_MODE      sm4;
1582 #endif // ALG_SM4
1583 #if ALG_CAMELLIA
1584     TPMI_ALG_SYM_MODE      camellia;
1585 #endif // ALG_CAMELLIA
1586     TPMI_ALG_SYM_MODE      sym;
1587 } TPMU_SYM_MODE;
1588
1589 typedef struct { // Table 2:141
1590     TPMI_ALG_SYM      algorithm;
1591     TPMU_SYM_KEY_BITS keyBits;
1592     TPMU_SYM_MODE     mode;
1593 } TPMT_SYM_DEF;
1594
1595 typedef struct { // Table 2:142
1596     TPMI_ALG_SYM_OBJECT algorithm;
1597     TPMU_SYM_KEY_BITS  keyBits;
1598     TPMU_SYM_MODE      mode;
1599 } TPMT_SYM_DEF_OBJECT;
1600
1601 typedef union { // Table 2:143
1602     struct {
1603         UINT16      size;
1604         BYTE        buffer[MAX_SYM_KEY_BYTES];
1605     } t;
1606     TPM2B          b;
1607 } TPM2B_SYM_KEY;
1608
1609 typedef struct { // Table 2:144
1610     TPMT_SYM_DEF_OBJECT sym;
1611 } TPMS_SYMCIPHER_PARMS;
1612
1613 typedef union { // Table 2:145
1614     struct {
1615         UINT16      size;
1616         BYTE        buffer[LABEL_MAX_BUFFER];
1617     } t;
1618     TPM2B          b;
1619 } TPM2B_LABEL;
1620
1621 typedef struct { // Table 2:146
1622     TPM2B_LABEL label;
1623     TPM2B_LABEL context;
1624 } TPMS_DERIVE;
1625
1626 typedef union { // Table 2:147
1627     struct {
1628         UINT16      size;
1629         BYTE        buffer[sizeof(TPMS_DERIVE)];
1630     } t;
1631     TPM2B          b;
1632 } TPM2B_DERIVE;
1633
1634 typedef union { // Table 2:148
1635     BYTE        create[MAX_SYM_DATA];
1636     TPMS_DERIVE derive;
1637 } TPMU_SENSITIVE_CREATE;
1638
1639 typedef union { // Table 2:149
1640     struct {
1641         UINT16      size;
1642         BYTE        buffer[sizeof(TPMU_SENSITIVE_CREATE)];

```

```

1643     }                t;
1644     TPM2B            b;
1645 } TPM2B_SENSITIVE_DATA;
1646
1647 typedef struct {                                // Table 2:150
1648     TPM2B_AUTH        userAuth;
1649     TPM2B_SENSITIVE_DATA data;
1650 } TPMS_SENSITIVE_CREATE;
1651
1652 typedef struct {                                // Table 2:151
1653     UINT16            size;
1654     TPMS_SENSITIVE_CREATE sensitive;
1655 } TPM2B_SENSITIVE_CREATE;
1656
1657 typedef struct {                                // Table 2:152
1658     TPMI_ALG_HASH      hashAlg;
1659 } TPMS_SCHEME_HASH;
1660
1661 typedef struct {                                // Table 2:153
1662     TPMI_ALG_HASH      hashAlg;
1663     UINT16            count;
1664 } TPMS_SCHEME_ECDA;
1665
1666 typedef TPM_ALG_ID      TPMI_ALG_KEYEDHASH_SCHEME;

```

TPM 2.0 Part 2: Table 155 - Definition of Types for HMAC_SIG_SCHEME

```

1667 typedef TPMS_SCHEME_HASH      TPMS_SCHEME_HMAC;
1668
1669 typedef struct {                                // Table 2:156
1670     TPMI_ALG_HASH      hashAlg;
1671     TPMI_ALG_KDF        kdf;
1672 } TPMS_SCHEME_XOR;
1673
1674 typedef union {                                // Table 2:157
1675     #if ALG_HMAC
1676         TPMS_SCHEME_HMAC      hmac;
1677     #endif // ALG_HMAC
1678     #if ALG_XOR
1679         TPMS_SCHEME_XOR      xor;
1680     #endif // ALG_XOR
1681 } TPMU_SCHEME_KEYEDHASH;
1682
1683 typedef struct {                                // Table 2:158
1684     TPMI_ALG_KEYEDHASH_SCHEME      scheme;
1685     TPMU_SCHEME_KEYEDHASH      details;
1686 } TPMT_KEYEDHASH_SCHEME;

```

TPM 2.0 Part 2: Table 159 - Definition of Types for RSA Signature Schemes

```

1687 typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_RSASSA;
1688 typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_RSAPSS;

```

TPM 2.0 Part 2: Table 160 - Definition of Types for ECC Signature Schemes

```

1689 typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_ECDSA;
1690 typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_SM2;
1691 typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_ECSCHNORR;
1692 typedef TPMS_SCHEME_ECDA      TPMS_SIG_SCHEME_ECDA;
1693
1694 typedef union {                                // Table 2:161
1695     #if ALG_ECC
1696         TPMS_SIG_SCHEME_ECDA      ecda;
1697     #endif // ALG_ECC

```

```

1698 #if ALG_RSASSA
1699     TPMS_SIG_SCHEME_RSASSA          rsassa;
1700 #endif // ALG_RSASSA
1701 #if ALG_RSAPSS
1702     TPMS_SIG_SCHEME_RSAPSS          rsapss;
1703 #endif // ALG_RSAPSS
1704 #if ALG_ECDSA
1705     TPMS_SIG_SCHEME_ECDSA           ecdsa;
1706 #endif // ALG_ECDSA
1707 #if ALG_SM2
1708     TPMS_SIG_SCHEME_SM2             sm2;
1709 #endif // ALG_SM2
1710 #if ALG_ECSCHNORR
1711     TPMS_SIG_SCHEME_ECSCHNORR       ecschnorr;
1712 #endif // ALG_ECSCHNORR
1713 #if ALG_HMAC
1714     TPMS_SCHEME_HMAC                hmac;
1715 #endif // ALG_HMAC
1716     TPMS_SCHEME_HASH                any;
1717 } TPMU_SIG_SCHEME;
1718
1719 typedef struct {                                // Table 2:162
1720     TPMI_ALG_SIG_SCHEME              scheme;
1721     TPMU_SIG_SCHEME                  details;
1722 } TPMT_SIG_SCHEME;

```

TPM 2.0 Part 2: Table 163 - Definition of Types for Encryption Schemes

```

1723 typedef TPMS_SCHEME_HASH    TPMS_ENC_SCHEME_OAEP;
1724 typedef TPMS_EMPTY          TPMS_ENC_SCHEME_RSAES;

```

TPM 2.0 Part 2: Table 164 - Definition of Types for ECC Key Exchange

```

1725 typedef TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECDH;
1726 typedef TPMS_SCHEME_HASH    TPMS_KEY_SCHEME_ECMQV;

```

TPM 2.0 Part 2: Table 165 - Definition of Types for KDF Schemes

```

1727 typedef TPMS_SCHEME_HASH    TPMS_KDF_SCHEME_MGF1;
1728 typedef TPMS_SCHEME_HASH    TPMS_KDF_SCHEME_KDF1_SP800_56A;
1729 typedef TPMS_SCHEME_HASH    TPMS_KDF_SCHEME_KDF2;
1730 typedef TPMS_SCHEME_HASH    TPMS_KDF_SCHEME_KDF1_SP800_108;
1731
1732 typedef union {                                // Table 2:166
1733     #if ALG_MGF1
1734         TPMS_KDF_SCHEME_MGF1          mgf1;
1735     #endif // ALG_MGF1
1736     #if ALG_KDF1_SP800_56A
1737         TPMS_KDF_SCHEME_KDF1_SP800_56A kdf1_sp800_56a;
1738     #endif // ALG_KDF1_SP800_56A
1739     #if ALG_KDF2
1740         TPMS_KDF_SCHEME_KDF2          kdf2;
1741     #endif // ALG_KDF2
1742     #if ALG_KDF1_SP800_108
1743         TPMS_KDF_SCHEME_KDF1_SP800_108 kdf1_sp800_108;
1744     #endif // ALG_KDF1_SP800_108
1745     TPMS_SCHEME_HASH                anyKdf;
1746 } TPMU_KDF_SCHEME;
1747
1748 typedef struct {                                // Table 2:167
1749     TPMI_ALG_KDF                    scheme;
1750     TPMU_KDF_SCHEME                  details;
1751 } TPMT_KDF_SCHEME;
1752

```



```

1753 typedef TPM_ALG_ID          TPMI_ALG_ASYM_SCHEME;    // Table 2:168
1754
1755 typedef union {                                           // Table 2:169
1756 #if ALG_ECDH
1757     TPMS_KEY_SCHEME_ECDH          ecdh;
1758 #endif // ALG_ECDH
1759 #if ALG_ECMQV
1760     TPMS_KEY_SCHEME_ECMQV        ecmqv;
1761 #endif // ALG_ECMQV
1762 #if ALG_ECC
1763     TPMS_SIG_SCHEME_ECDA         ecdaa;
1764 #endif // ALG_ECC
1765 #if ALG_RSASSA
1766     TPMS_SIG_SCHEME_RSASSA       rsassa;
1767 #endif // ALG_RSASSA
1768 #if ALG_RSAPSS
1769     TPMS_SIG_SCHEME_RSAPSS       rsapss;
1770 #endif // ALG_RSAPSS
1771 #if ALG_ECDSA
1772     TPMS_SIG_SCHEME_ECDSA        ecdsa;
1773 #endif // ALG_ECDSA
1774 #if ALG_SM2
1775     TPMS_SIG_SCHEME_SM2          sm2;
1776 #endif // ALG_SM2
1777 #if ALG_ECSCHNORR
1778     TPMS_SIG_SCHEME_ECSCHNORR    ecschnorr;
1779 #endif // ALG_ECSCHNORR
1780 #if ALG_RSAES
1781     TPMS_ENC_SCHEME_RSAES        rsaes;
1782 #endif // ALG_RSAES
1783 #if ALG_OAEP
1784     TPMS_ENC_SCHEME_OAEP         oaep;
1785 #endif // ALG_OAEP
1786     TPMS_SCHEME_HASH             anySig;
1787 } TPMU_ASYM_SCHEME;
1788
1789 typedef struct {                                           // Table 2:170
1790     TPMI_ALG_ASYM_SCHEME          scheme;
1791     TPMU_ASYM_SCHEME              details;
1792 } TPMT_ASYM_SCHEME;
1793
1794 typedef TPM_ALG_ID          TPMI_ALG_RSA_SCHEME;    // Table 2:171
1795
1796 typedef struct {                                           // Table 2:172
1797     TPMI_ALG_RSA_SCHEME          scheme;
1798     TPMU_ASYM_SCHEME              details;
1799 } TPMT_RSA_SCHEME;
1800
1801 typedef TPM_ALG_ID          TPMI_ALG_RSA_DECRYPT;    // Table 2:173
1802
1803 typedef struct {                                           // Table 2:174
1804     TPMI_ALG_RSA_DECRYPT          scheme;
1805     TPMU_ASYM_SCHEME              details;
1806 } TPMT_RSA_DECRYPT;
1807
1808 typedef union {                                           // Table 2:175
1809     struct {
1810         UINT16                    size;
1811         BYTE                      buffer[MAX_RSA_KEY_BYTES];
1812     } t;
1813     TPM2B                        b;
1814 } TPM2B_PUBLIC_KEY_RSA;
1815
1816 typedef TPM_KEY_BITS          TPMI_RSA_KEY_BITS;    // Table 2:176
1817
1818 typedef union {                                           // Table 2:177

```

```

1819     struct {
1820         UINT16          size;
1821         BYTE            buffer[RSA_PRIVATE_SIZE];
1822     } t;
1823     TPM2B              b;
1824 } TPM2B_PRIVATE_KEY_RSA;
1825
1826 typedef union {
1827     struct {
1828         UINT16          size;
1829         BYTE            buffer[MAX_ECC_KEY_BYTES];
1830     } t;
1831     TPM2B              b;
1832 } TPM2B_ECC_PARAMETER;
1833
1834 typedef struct {
1835     TPM2B_ECC_PARAMETER x;
1836     TPM2B_ECC_PARAMETER y;
1837 } TPMS_ECC_POINT;
1838
1839 typedef struct {
1840     UINT16          size;
1841     TPMS_ECC_POINT  point;
1842 } TPM2B_ECC_POINT;
1843
1844 typedef TPM_ALG_ID  TPMI_ALG_ECC_SCHEME;
1845
1846 typedef TPM_ECC_CURVE  TPMI_ECC_CURVE;
1847
1848 typedef struct {
1849     TPMI_ALG_ECC_SCHEME  scheme;
1850     TPMU_ASYM_SCHEME     details;
1851 } TPMT_ECC_SCHEME;
1852
1853 typedef struct {
1854     TPM_ECC_CURVE        curveID;
1855     UINT16               keySize;
1856     TPMT_KDF_SCHEME       kdf;
1857     TPMT_ECC_SCHEME       sign;
1858     TPM2B_ECC_PARAMETER   p;
1859     TPM2B_ECC_PARAMETER   a;
1860     TPM2B_ECC_PARAMETER   b;
1861     TPM2B_ECC_PARAMETER   gX;
1862     TPM2B_ECC_PARAMETER   gY;
1863     TPM2B_ECC_PARAMETER   n;
1864     TPM2B_ECC_PARAMETER   h;
1865 } TPMS_ALGORITHM_DETAIL_ECC;
1866
1867 typedef struct {
1868     TPMI_ALG_HASH          hash;
1869     TPM2B_PUBLIC_KEY_RSA   sig;
1870 } TPMS_SIGNATURE_RSA;

```

TPM 2.0 Part 2: Table 186 - Definition of Types for Signature

```

1871 typedef TPMS_SIGNATURE_RSA  TPMS_SIGNATURE_RSASSA;
1872 typedef TPMS_SIGNATURE_RSA  TPMS_SIGNATURE_RSAPSS;
1873
1874 typedef struct {
1875     TPMI_ALG_HASH          hash;
1876     TPM2B_ECC_PARAMETER    signatureR;
1877     TPM2B_ECC_PARAMETER    signatureS;
1878 } TPMS_SIGNATURE_ECC;

```

TPM 2.0 Part 2: Table 188 - Definition of Types for TPMS_SIGNATURE_ECC

```

1879 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECDA;
1880 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECDSA;
1881 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_SM2;
1882 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECSCHNORR;
1883
1884 typedef union { // Table 2:189
1885     #if ALG_ECC
1886         TPMS_SIGNATURE_ECDA          ecda;
1887     #endif // ALG_ECC
1888     #if ALG_RSA
1889         TPMS_SIGNATURE_RSASSA        rsassa;
1890     #endif // ALG_RSA
1891     #if ALG_RSA
1892         TPMS_SIGNATURE_RSAPSS        rsapss;
1893     #endif // ALG_RSA
1894     #if ALG_ECC
1895         TPMS_SIGNATURE_ECDSA          ecdsa;
1896     #endif // ALG_ECC
1897     #if ALG_ECC
1898         TPMS_SIGNATURE_SM2            sm2;
1899     #endif // ALG_ECC
1900     #if ALG_ECC
1901         TPMS_SIGNATURE_ECSCHNORR      ecschnorr;
1902     #endif // ALG_ECC
1903     #if ALG_HMAC
1904         TPMT_HA                        hmac;
1905     #endif // ALG_HMAC
1906         TPMS_SCHEME_HASH                any;
1907 } TPMU_SIGNATURE;
1908
1909 typedef struct { // Table 2:190
1910     TPMI_ALG_SIG_SCHEME                sigAlg;
1911     TPMU_SIGNATURE                      signature;
1912 } TPMT_SIGNATURE;
1913
1914 typedef union { // Table 2:191
1915     #if ALG_ECC
1916         BYTE                                ecc[sizeof(TPMS_ECC_POINT)];
1917     #endif // ALG_ECC
1918     #if ALG_RSA
1919         BYTE                                rsa[MAX_RSA_KEY_BYTES];
1920     #endif // ALG_RSA
1921     #if ALG_SYMCIPHER
1922         BYTE                                symmetric[sizeof(TPM2B_DIGEST)];
1923     #endif // ALG_SYMCIPHER
1924     #if ALG_KEYEDHASH
1925         BYTE                                keyedHash[sizeof(TPM2B_DIGEST)];
1926     #endif // ALG_KEYEDHASH
1927 } TPMU_ENCRYPTED_SECRET;
1928
1929 typedef union { // Table 2:192
1930     struct {
1931         UINT16                                size;
1932         BYTE                                secret[sizeof(TPMU_ENCRYPTED_SECRET)];
1933     } t;
1934     TPM2B                                b;
1935 } TPM2B_ENCRYPTED_SECRET;
1936
1937 typedef TPM_ALG_ID TPMI_ALG_PUBLIC; // Table 2:193
1938
1939 typedef union { // Table 2:194
1940     #if ALG_KEYEDHASH
1941         TPM2B_DIGEST                                keyedHash;
1942     #endif // ALG_KEYEDHASH
1943     #if ALG_SYMCIPHER
1944         TPM2B_DIGEST                                sym;

```

```

1945 #endif // ALG_SYMCIPHER
1946 #if ALG_RSA
1947     TPM2B_PUBLIC_KEY_RSA      rsa;
1948 #endif // ALG_RSA
1949 #if ALG_ECC
1950     TPMS_ECC_POINT            ecc;
1951 #endif // ALG_ECC
1952     TPMS_DERIVE                derive;
1953 } TPMU_PUBLIC_ID;
1954
1955 typedef struct { // Table 2:195
1956     TPMT_KEYEDHASH_SCHEME      scheme;
1957 } TPMS_KEYEDHASH_PARMS;
1958
1959 typedef struct { // Table 2:196
1960     TPMT_SYM_DEF_OBJECT        symmetric;
1961     TPMT_ASYM_SCHEME           scheme;
1962 } TPMS_ASYM_PARMS;
1963
1964 typedef struct { // Table 2:197
1965     TPMT_SYM_DEF_OBJECT        symmetric;
1966     TPMT_RSA_SCHEME            scheme;
1967     TPMI_RSA_KEY_BITS          keyBits;
1968     UINT32                     exponent;
1969 } TPMS_RSA_PARMS;
1970
1971 typedef struct { // Table 2:198
1972     TPMT_SYM_DEF_OBJECT        symmetric;
1973     TPMT_ECC_SCHEME            scheme;
1974     TPMI_ECC_CURVE             curveID;
1975     TPMT_KDF_SCHEME            kdf;
1976 } TPMS_ECC_PARMS;
1977
1978 typedef union { // Table 2:199
1979     #if ALG_KEYEDHASH
1980         TPMS_KEYEDHASH_PARMS      keyedHashDetail;
1981     #endif // ALG_KEYEDHASH
1982     #if ALG_SYMCIPHER
1983         TPMS_SYMCIPHER_PARMS      symDetail;
1984     #endif // ALG_SYMCIPHER
1985     #if ALG_RSA
1986         TPMS_RSA_PARMS            rsaDetail;
1987     #endif // ALG_RSA
1988     #if ALG_ECC
1989         TPMS_ECC_PARMS            eccDetail;
1990     #endif // ALG_ECC
1991     TPMS_ASYM_PARMS              asymDetail;
1992 } TPMU_PUBLIC_PARMS;
1993
1994 typedef struct { // Table 2:200
1995     TPMI_ALG_PUBLIC             type;
1996     TPMU_PUBLIC_PARMS           parameters;
1997 } TPMT_PUBLIC_PARMS;
1998
1999 typedef struct { // Table 2:201
2000     TPMI_ALG_PUBLIC             type;
2001     TPMI_ALG_HASH               nameAlg;
2002     TPMA_OBJECT                 objectAttributes;
2003     TPM2B_DIGEST                authPolicy;
2004     TPMU_PUBLIC_PARMS           parameters;
2005     TPMU_PUBLIC_ID              unique;
2006 } TPMT_PUBLIC;
2007
2008 typedef struct { // Table 2:202
2009     UINT16                      size;
2010     TPMT_PUBLIC                 publicArea;

```

```

2011 } TPM2B_PUBLIC;
2012
2013 typedef union { // Table 2:203
2014     struct {
2015         UINT16 size;
2016         BYTE buffer[sizeof(TPMT_PUBLIC)];
2017     } t;
2018     TPM2B b;
2019 } TPM2B_TEMPLATE;
2020
2021 typedef union { // Table 2:204
2022     struct {
2023         UINT16 size;
2024         BYTE buffer[PRIVATE_VENDOR_SPECIFIC_BYTES];
2025     } t;
2026     TPM2B b;
2027 } TPM2B_PRIVATE_VENDOR_SPECIFIC;
2028
2029 typedef union { // Table 2:205
2030 #if ALG_RSA
2031     TPM2B_PRIVATE_KEY_RSA rsa;
2032 #endif // ALG_RSA
2033 #if ALG_ECC
2034     TPM2B_ECC_PARAMETER ecc;
2035 #endif // ALG_ECC
2036 #if ALG_KEYEDHASH
2037     TPM2B_SENSITIVE_DATA bits;
2038 #endif // ALG_KEYEDHASH
2039 #if ALG_SYMCIPHER
2040     TPM2B_SYM_KEY sym;
2041 #endif // ALG_SYMCIPHER
2042     TPM2B_PRIVATE_VENDOR_SPECIFIC any;
2043 } TPMU_SENSITIVE_COMPOSITE;
2044
2045 typedef struct { // Table 2:206
2046     TPMI_ALG_PUBLIC sensitiveType;
2047     TPM2B_AUTH authValue;
2048     TPM2B_DIGEST seedValue;
2049     TPMU_SENSITIVE_COMPOSITE sensitive;
2050 } TPMT_SENSITIVE;
2051
2052 typedef struct { // Table 2:207
2053     UINT16 size;
2054     TPMT_SENSITIVE sensitiveArea;
2055 } TPM2B_SENSITIVE;
2056
2057 typedef struct { // Table 2:208
2058     TPM2B_DIGEST integrityOuter;
2059     TPM2B_DIGEST integrityInner;
2060     TPM2B_SENSITIVE sensitive;
2061 } _PRIVATE;
2062
2063 typedef union { // Table 2:209
2064     struct {
2065         UINT16 size;
2066         BYTE buffer[sizeof(_PRIVATE)];
2067     } t;
2068     TPM2B b;
2069 } TPM2B_PRIVATE;
2070
2071 typedef struct { // Table 2:210
2072     TPM2B_DIGEST integrityHMAC;
2073     TPM2B_DIGEST encIdentity;
2074 } TPMS_ID_OBJECT;
2075
2076 typedef union { // Table 2:211

```

```

2077     struct {
2078         UINT16          size;
2079         BYTE            credential[sizeof(TPMS_ID_OBJECT)];
2080     } t;
2081     TPM2B              b;
2082 } TPM2B_ID_OBJECT;
2083
2084 #define TYPE_OF_TPM_NV_INDEX    UINT32
2085 #define TPM_NV_INDEX_TO_UINT32(a)    (*(UINT32 *)&(a))
2086 #define UINT32_TO_TPM_NV_INDEX(a)    (*(TPM_NV_INDEX *)&(a))
2087 #define TPM_NV_INDEX_TO_BYTE_ARRAY(i, a) \
2088     UINT32_TO_BYTE_ARRAY((TPM_NV_INDEX_TO_UINT32(i)), (a)) \
2089 #define BYTE_ARRAY_TO_TPM_NV_INDEX(i, a) \
2090     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPM_NV_INDEX(x); }
2091 #if USE_BIT_FIELD_STRUCTURES
2092 typedef struct TPM_NV_INDEX { // Table 2:212
2093     unsigned index : 24;
2094     unsigned RH_NV : 8;
2095 } TPM_NV_INDEX;

```

This is the initializer for a TPM_NV_INDEX structure

```

2096 #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) {index, rh_nv}
2097 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:212 TPM_NV_INDEX using bit masking

```

2098 typedef UINT32 TPM_NV_INDEX;
2099 #define TYPE_OF_TPM_NV_INDEX    UINT32
2100 #define TPM_NV_INDEX_index_SHIFT    0
2101 #define TPM_NV_INDEX_index    ((TPM_NV_INDEX)0xffffffff << 0)
2102 #define TPM_NV_INDEX_RH_NV_SHIFT    24
2103 #define TPM_NV_INDEX_RH_NV    ((TPM_NV_INDEX)0xff << 24)

```

This is the initializer for a TPM_NV_INDEX bit array.

```

2104 #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) \
2105     (TPM_NV_INDEX) ( \
2106     (index << 0) + (rh_nv << 24)) \
2107 #endif // USE_BIT_FIELD_STRUCTURES

```

TPM 2.0 Part 2: Table 213 - Definition of TPM_NT Constants

```

2108 typedef UINT32 TPM_NT;
2109 #define TYPE_OF_TPM_NT    UINT32
2110 #define TPM_NT_ORDINARY    (TPM_NT) (0x0)
2111 #define TPM_NT_COUNTER    (TPM_NT) (0x1)
2112 #define TPM_NT_BITS    (TPM_NT) (0x2)
2113 #define TPM_NT_EXTEND    (TPM_NT) (0x4)
2114 #define TPM_NT_PIN_FAIL    (TPM_NT) (0x8)
2115 #define TPM_NT_PIN_PASS    (TPM_NT) (0x9)
2116
2117 typedef struct { // Table 2:214
2118     UINT32 pinCount;
2119     UINT32 pinLimit;
2120 } TPMS_NV_PIN_COUNTER_PARAMETERS;
2121
2122 #define TYPE_OF_TPMA_NV    UINT32
2123 #define TPMA_NV_TO_UINT32(a)    (*(UINT32 *)&(a))
2124 #define UINT32_TO_TPMA_NV(a)    (*(TPMA_NV *)&(a))
2125 #define TPMA_NV_TO_BYTE_ARRAY(i, a) \
2126     UINT32_TO_BYTE_ARRAY((TPMA_NV_TO_UINT32(i)), (a)) \
2127 #define BYTE_ARRAY_TO_TPMA_NV(i, a) \
2128     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_NV(x); }

```

```

2129 #if USE_BIT_FIELD_STRUCTURES
2130 typedef struct TPMA_NV { // Table 2:215
2131     unsigned PPWRITE : 1;
2132     unsigned OWNERWRITE : 1;
2133     unsigned AUTHWRITE : 1;
2134     unsigned POLICYWRITE : 1;
2135     unsigned TPM_NT : 4;
2136     unsigned Reserved_bits_at_8 : 2;
2137     unsigned POLICY_DELETE : 1;
2138     unsigned WRITELOCKED : 1;
2139     unsigned WRITEALL : 1;
2140     unsigned WRITEDEFINE : 1;
2141     unsigned WRITE_STCLEAR : 1;
2142     unsigned GLOBALLOCK : 1;
2143     unsigned PPREAD : 1;
2144     unsigned OWNERREAD : 1;
2145     unsigned AUTHREAD : 1;
2146     unsigned POLICYREAD : 1;
2147     unsigned Reserved_bits_at_20 : 5;
2148     unsigned NO_DA : 1;
2149     unsigned ORDERLY : 1;
2150     unsigned CLEAR_STCLEAR : 1;
2151     unsigned READLOCKED : 1;
2152     unsigned WRITTEN : 1;
2153     unsigned PLATFORMCREATE : 1;
2154     unsigned READ_STCLEAR : 1;
2155 } TPMA_NV;

```

This is the initializer for a TPMA_NV structure

```

2156 #define TPMA_NV_INITIALIZER( \
2157     ppwrite, ownerwrite, authwrite, policywrite, \
2158     tpm_nt, bits_at_8, policy_delete, writelocked, \
2159     writeall, writedefine, write_stclear, globallock, \
2160     ppread, ownerread, authread, policyread, \
2161     bits_at_20, no_da, orderly, clear_stclear, \
2162     readlocked, written, platformcreate, read_stclear) \
2163 {ppwrite, ownerwrite, authwrite, policywrite, \
2164     tpm_nt, bits_at_8, policy_delete, writelocked, \
2165     writeall, writedefine, write_stclear, globallock, \
2166     ppread, ownerread, authread, policyread, \
2167     bits_at_20, no_da, orderly, clear_stclear, \
2168     readlocked, written, platformcreate, read_stclear} \
2169 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:215 TPMA_NV using bit masking

```

2170 typedef UINT32 TPMA_NV;
2171 #define TYPE_OF_TPMA_NV UINT32
2172 #define TPMA_NV_PPWRITE ((TPMA_NV)1 << 0)
2173 #define TPMA_NV_OWNERWRITE ((TPMA_NV)1 << 1)
2174 #define TPMA_NV_AUTHWRITE ((TPMA_NV)1 << 2)
2175 #define TPMA_NV_POLICYWRITE ((TPMA_NV)1 << 3)
2176 #define TPMA_NV_TPM_NT_SHIFT 4
2177 #define TPMA_NV_TPM_NT ((TPMA_NV)0xf << 4)
2178 #define TPMA_NV_POLICY_DELETE ((TPMA_NV)1 << 10)
2179 #define TPMA_NV_WRITELOCKED ((TPMA_NV)1 << 11)
2180 #define TPMA_NV_WRITEALL ((TPMA_NV)1 << 12)
2181 #define TPMA_NV_WRITEDEFINE ((TPMA_NV)1 << 13)
2182 #define TPMA_NV_WRITE_STCLEAR ((TPMA_NV)1 << 14)
2183 #define TPMA_NV_GLOBALLOCK ((TPMA_NV)1 << 15)
2184 #define TPMA_NV_PPREAD ((TPMA_NV)1 << 16)
2185 #define TPMA_NV_OWNERREAD ((TPMA_NV)1 << 17)
2186 #define TPMA_NV_AUTHREAD ((TPMA_NV)1 << 18)
2187 #define TPMA_NV_POLICYREAD ((TPMA_NV)1 << 19)

```



```

2188 #define TPMA_NV_NO_DA          ((TPMA_NV)1 << 25)
2189 #define TPMA_NV_ORDERLY        ((TPMA_NV)1 << 26)
2190 #define TPMA_NV_CLEAR_STCLEAR  ((TPMA_NV)1 << 27)
2191 #define TPMA_NV_READLOCKED     ((TPMA_NV)1 << 28)
2192 #define TPMA_NV_WRITTEN        ((TPMA_NV)1 << 29)
2193 #define TPMA_NV_PLATFORMCREATE ((TPMA_NV)1 << 30)
2194 #define TPMA_NV_READ_STCLEAR   ((TPMA_NV)1 << 31)

```

This is the initializer for a TPMA_NV bit array.

```

2195 #define TPMA_NV_INITIALIZER(                                     \
2196     ppwrite,          ownerwrite,      authwrite,      policywrite,      \
2197     tpm_nt,           bits_at_8,       policy_delete, writelocked,      \
2198     writeall,         writedefine,     write_stclear, globallock,      \
2199     ppread,           ownerread,       authread,       policyread,      \
2200     bits_at_20,       no_da,           orderly,        clear_stclear,      \
2201     readlocked,       written,         platformcreate, read_stclear)    \
2202 (TPMA_NV) (                                                    \
2203     (ppwrite << 0)          + (ownerwrite << 1)          +          \
2204     (authwrite << 2)        + (policywrite << 3)          +          \
2205     (tpm_nt << 4)           + (policy_delete << 10)       +          \
2206     (writelocked << 11)     + (writeall << 12)            +          \
2207     (writedefine << 13)     + (write_stclear << 14)       +          \
2208     (globallock << 15)      + (ppread << 16)              +          \
2209     (ownerread << 17)       + (authread << 18)            +          \
2210     (policyread << 19)      + (no_da << 25)               +          \
2211     (orderly << 26)         + (clear_stclear << 27)       +          \
2212     (readlocked << 28)      + (written << 29)             +          \
2213     (platformcreate << 30) + (read_stclear << 31))          \
2214 #endif // USE_BIT_FIELD_STRUCTURES
2215
2216 typedef struct {                                               // Table 2:216
2217     TPMI_RH_NV_INDEX    nvIndex;
2218     TPMI_ALG_HASH        nameAlg;
2219     TPMA_NV              attributes;
2220     TPM2B_DIGEST         authPolicy;
2221     UINT16               dataSize;
2222 } TPMS_NV_PUBLIC;
2223
2224 typedef struct {                                               // Table 2:217
2225     UINT16               size;
2226     TPMS_NV_PUBLIC       nvPublic;
2227 } TPM2B_NV_PUBLIC;
2228
2229 typedef union {                                                // Table 2:218
2230     struct {
2231         UINT16           size;
2232         BYTE             buffer[MAX_CONTEXT_SIZE];
2233     }                    t;
2234     TPM2B                b;
2235 } TPM2B_CONTEXT_SENSITIVE;
2236
2237 typedef struct {                                               // Table 2:219
2238     TPM2B_DIGEST         integrity;
2239     TPM2B_CONTEXT_SENSITIVE encrypted;
2240 } TPMS_CONTEXT_DATA;
2241
2242 typedef union {                                                // Table 2:220
2243     struct {
2244         UINT16           size;
2245         BYTE             buffer[sizeof(TPMS_CONTEXT_DATA)];
2246     }                    t;
2247     TPM2B                b;
2248 } TPM2B_CONTEXT_DATA;
2249

```

```

2250 typedef struct {                                     // Table 2:221
2251     UINT64 sequence;
2252     TPMI_DH_SAVED savedHandle;
2253     TPMI_RH_HIERARCHY hierarchy;
2254     TPM2B_CONTEXT_DATA contextBlob;
2255 } TPMS_CONTEXT;
2256
2257 typedef struct {                                     // Table 2:223
2258     TPML_PCR_SELECTION pcrSelect;
2259     TPM2B_DIGEST pcrDigest;
2260     TPMA_LOCALITY locality;
2261     TPM_ALG_ID parentNameAlg;
2262     TPM2B_NAME parentName;
2263     TPM2B_NAME parentQualifiedName;
2264     TPM2B_DATA outsideInfo;
2265 } TPMS_CREATION_DATA;
2266
2267 typedef struct {                                     // Table 2:224
2268     UINT16 size;
2269     TPMS_CREATION_DATA creationData;
2270 } TPM2B_CREATION_DATA;

```

TPM 2.0 Part 2: Table 225 - Definition of TPM_AT Constants

```

2271 typedef UINT32 TPM_AT;
2272 #define TYPE_OF_TPM_AT UINT32
2273 #define TPM_AT_ANY (TPM_AT) (0x00000000)
2274 #define TPM_AT_ERROR (TPM_AT) (0x00000001)
2275 #define TPM_AT_PV1 (TPM_AT) (0x00000002)
2276 #define TPM_AT_VEND (TPM_AT) (0x80000000)

```

TPM 2.0 Part 2: Table 226 - Definition of TPM_AE Constants

```

2277 typedef UINT32 TPM_AE;
2278 #define TYPE_OF_TPM_AE UINT32
2279 #define TPM_AE_NONE (TPM_AE) (0x00000000)
2280
2281 typedef struct {                                     // Table 2:227
2282     TPM_AT tag;
2283     UINT32 data;
2284 } TPMS_AC_OUTPUT;
2285
2286 typedef struct {                                     // Table 2:228
2287     UINT32 count;
2288     TPMS_AC_OUTPUT acCapabilities[MAX_AC_CAPABILITIES];
2289 } TPML_AC_CAPABILITIES;
2290
2291 #endif // _TPM_TYPES_H_

```

5.20 VendorString.h

```
1  #ifndef      _VENDOR_STRING_H
2  #define      _VENDOR_STRING_H
```

Define up to 4-byte values for MANUFACTURER. This value defines the response for TPM_PT_MANUFACTURER in TPM2_GetCapability(). The following line should be un-commented and a vendor specific string should be provided here.

```
3  #define      MANUFACTURER      "MSFT"
```

The following #if macro may be deleted after a proper MANUFACTURER is provided.

```
4  #ifndef MANUFACTURER
5  #error MANUFACTURER is not provided. \
6  Please modify include/VendorString.h to provide a specific \
7  manufacturer name.
8  #endif
```

Define up to 4, 4-byte values. The values must each be 4 bytes long and the last value used may contain trailing zeros. These values define the response for TPM_PT_VENDOR_STRING_(1-4) in TPM2_GetCapability(). The following line should be un-commented and a vendor specific string should be provided here. The vendor strings 2-4 may also be defined as appropriate.

```
9  #define      VENDOR_STRING_1      "xCG "
10 #define      VENDOR_STRING_2      "fTPM"
11 // #define      VENDOR_STRING_3
12 // #define      VENDOR_STRING_4
13
```

The following #if macro may be deleted after a proper VENDOR_STRING_1 is provided.

```
14 #ifndef VENDOR_STRING_1
15 #error VENDOR_STRING_1 is not provided. \
16 Please modify include/VendorString.h to provide a vendor-specific string.
17 #endif
```

the more significant 32-bits of a vendor-specific value indicating the version of the firmware The following line should be un-commented and a vendor specific firmware V1 should be provided here. The FIRMWARE_V2 may also be defined as appropriate.

```
18 #define      FIRMWARE_V1      (0x20170619)
```

the less significant 32-bits of a vendor-specific value indicating the version of the firmware

```
19 #define      FIRMWARE_V2      (0x00163636)
```

The following #if macro may be deleted after a proper FIRMWARE_V1 is provided.

```
20 #ifndef FIRMWARE_V1
21 #error FIRMWARE_V1 is not provided. \
22 Please modify include/VendorString.h to provide a vendor-specific firmware \
23 version
24 #endif
25
26 #endif
```

5.21 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

```

The aggregation macros for machines that do not allow unaligned access or for little-endian machines.

Aggregate bytes into a UINT

```

21 #define BYTE_ARRAY_TO_UINT8(b)    (uint8_t)((b)[0])
22 #define BYTE_ARRAY_TO_UINT16(b)   ByteArrayToUint16((BYTE *) (b))
23 #define BYTE_ARRAY_TO_UINT32(b)   ByteArrayToUint32((BYTE *) (b))
24 #define BYTE_ARRAY_TO_UINT64(b)   ByteArrayToUint64((BYTE *) (b))
25 #define UINT8_TO_BYTE_ARRAY(i, b) ((b)[0] = (uint8_t)(i))
26 #define UINT16_TO_BYTE_ARRAY(i, b) Uint16ToByteArray((i), (BYTE *) (b))
27 #define UINT32_TO_BYTE_ARRAY(i, b) Uint32ToByteArray((i), (BYTE *) (b))
28 #define UINT64_TO_BYTE_ARRAY(i, b) Uint64ToByteArray((i), (BYTE *) (b))
29
30 #else // AUTO_ALIGN
31
32 #if BIG_ENDIAN_TPM

```

the big-endian macros for machines that allow unaligned memory access Aggregate a byte array into a UINT

```

33 #define BYTE_ARRAY_TO_UINT8(b)    *((uint8_t *) (b))
34 #define BYTE_ARRAY_TO_UINT16(b)   *((uint16_t *) (b))
35 #define BYTE_ARRAY_TO_UINT32(b)   *((uint32_t *) (b))
36 #define BYTE_ARRAY_TO_UINT64(b)   *((uint64_t *) (b))

```

Disaggregate a UINT into a byte array

```

37 #define UINT8_TO_BYTE_ARRAY(i, b) {*((uint8_t *) (b)) = (i);}
38 #define UINT16_TO_BYTE_ARRAY(i, b) {*((uint16_t *) (b)) = (i);}
39 #define UINT32_TO_BYTE_ARRAY(i, b) {*((uint32_t *) (b)) = (i);}
40 #define UINT64_TO_BYTE_ARRAY(i, b) {*((uint64_t *) (b)) = (i);}
41 #else

```

the little endian macros for machines that allow unaligned memory access the big-endian macros for machines that allow unaligned memory access Aggregate a byte array into a UINT

```

42 #define BYTE_ARRAY_TO_UINT8(b)    *((uint8_t *) (b))
43 #define BYTE_ARRAY_TO_UINT16(b)   REVERSE_ENDIAN_16(*((uint16_t *) (b)))
44 #define BYTE_ARRAY_TO_UINT32(b)   REVERSE_ENDIAN_32(*((uint32_t *) (b)))
45 #define BYTE_ARRAY_TO_UINT64(b)   REVERSE_ENDIAN_64(*((uint64_t *) (b)))

```

Disaggregate a UINT into a byte array

```
46 #define UINT8_TO_BYTE_ARRAY(i, b)  {*((uint8_t *) (b)) = (i);}
47 #define UINT16_TO_BYTE_ARRAY(i, b) {*((uint16_t *) (b)) = REVERSE_ENDIAN_16(i);}
48 #define UINT32_TO_BYTE_ARRAY(i, b) {*((uint32_t *) (b)) = REVERSE_ENDIAN_32(i);}
49 #define UINT64_TO_BYTE_ARRAY(i, b) {*((uint64_t *) (b)) = REVERSE_ENDIAN_64(i);}
50 #endif // BIG_ENDIAN_TPM
51
52 #endif // AUTO_ALIGN == NO
53
54 #endif // _SWAP_H
```

5.22 ACT.h

```

1  #ifndef _ACT_H_
2  #define _ACT_H_
3
4  #include "TpmProfile.h"
5
6  #if !(defined RH_ACT_0) || (RH_ACT_0 != YES)
7  #   undef RH_ACT_0
8  #   define RH_ACT_0 NO
9  #   define IF_ACT_0_IMPLEMENTED(op)
10 #else
11 #   define IF_ACT_0_IMPLEMENTED(op) op(0)
12 #endif
13 #if !(defined RH_ACT_1) || (RH_ACT_1 != YES)
14 #   undef RH_ACT_1
15 #   define RH_ACT_1 NO
16 #   define IF_ACT_1_IMPLEMENTED(op)
17 #else
18 #   define IF_ACT_1_IMPLEMENTED(op) op(1)
19 #endif
20 #if !(defined RH_ACT_2) || (RH_ACT_2 != YES)
21 #   undef RH_ACT_2
22 #   define RH_ACT_2 NO
23 #   define IF_ACT_2_IMPLEMENTED(op)
24 #else
25 #   define IF_ACT_2_IMPLEMENTED(op) op(2)
26 #endif
27 #if !(defined RH_ACT_3) || (RH_ACT_3 != YES)
28 #   undef RH_ACT_3
29 #   define RH_ACT_3 NO
30 #   define IF_ACT_3_IMPLEMENTED(op)
31 #else
32 #   define IF_ACT_3_IMPLEMENTED(op) op(3)
33 #endif
34 #if !(defined RH_ACT_4) || (RH_ACT_4 != YES)
35 #   undef RH_ACT_4
36 #   define RH_ACT_4 NO
37 #   define IF_ACT_4_IMPLEMENTED(op)
38 #else
39 #   define IF_ACT_4_IMPLEMENTED(op) op(4)
40 #endif
41 #if !(defined RH_ACT_5) || (RH_ACT_5 != YES)
42 #   undef RH_ACT_5
43 #   define RH_ACT_5 NO
44 #   define IF_ACT_5_IMPLEMENTED(op)
45 #else
46 #   define IF_ACT_5_IMPLEMENTED(op) op(5)
47 #endif
48 #if !(defined RH_ACT_6) || (RH_ACT_6 != YES)
49 #   undef RH_ACT_6
50 #   define RH_ACT_6 NO
51 #   define IF_ACT_6_IMPLEMENTED(op)
52 #else
53 #   define IF_ACT_6_IMPLEMENTED(op) op(6)
54 #endif
55 #if !(defined RH_ACT_7) || (RH_ACT_7 != YES)
56 #   undef RH_ACT_7
57 #   define RH_ACT_7 NO
58 #   define IF_ACT_7_IMPLEMENTED(op)
59 #else
60 #   define IF_ACT_7_IMPLEMENTED(op) op(7)
61 #endif
62 #if !(defined RH_ACT_8) || (RH_ACT_8 != YES)
63 #   undef RH_ACT_8

```

```

64 # define RH_ACT_8 NO
65 # define IF_ACT_8_IMPLEMENTED(op)
66 #else
67 # define IF_ACT_8_IMPLEMENTED(op) op(8)
68 #endif
69 #if !(defined RH_ACT_9) || (RH_ACT_9 != YES)
70 # undef RH_ACT_9
71 # define RH_ACT_9 NO
72 # define IF_ACT_9_IMPLEMENTED(op)
73 #else
74 # define IF_ACT_9_IMPLEMENTED(op) op(9)
75 #endif
76 #if !(defined RH_ACT_A) || (RH_ACT_A != YES)
77 # undef RH_ACT_A
78 # define RH_ACT_A NO
79 # define IF_ACT_A_IMPLEMENTED(op)
80 #else
81 # define IF_ACT_A_IMPLEMENTED(op) op(A)
82 #endif
83 #if !(defined RH_ACT_B) || (RH_ACT_B != YES)
84 # undef RH_ACT_B
85 # define RH_ACT_B NO
86 # define IF_ACT_B_IMPLEMENTED(op)
87 #else
88 # define IF_ACT_B_IMPLEMENTED(op) op(B)
89 #endif
90 #if !(defined RH_ACT_C) || (RH_ACT_C != YES)
91 # undef RH_ACT_C
92 # define RH_ACT_C NO
93 # define IF_ACT_C_IMPLEMENTED(op)
94 #else
95 # define IF_ACT_C_IMPLEMENTED(op) op(C)
96 #endif
97 #if !(defined RH_ACT_D) || (RH_ACT_D != YES)
98 # undef RH_ACT_D
99 # define RH_ACT_D NO
100 # define IF_ACT_D_IMPLEMENTED(op)
101 #else
102 # define IF_ACT_D_IMPLEMENTED(op) op(D)
103 #endif
104 #if !(defined RH_ACT_E) || (RH_ACT_E != YES)
105 # undef RH_ACT_E
106 # define RH_ACT_E NO
107 # define IF_ACT_E_IMPLEMENTED(op)
108 #else
109 # define IF_ACT_E_IMPLEMENTED(op) op(E)
110 #endif
111 #if !(defined RH_ACT_F) || (RH_ACT_F != YES)
112 # undef RH_ACT_F
113 # define RH_ACT_F NO
114 # define IF_ACT_F_IMPLEMENTED(op)
115 #else
116 # define IF_ACT_F_IMPLEMENTED(op) op(F)
117 #endif
118
119 #ifndef TPM_RH_ACT_0
120 #error Need numeric definition for TPM_RH_ACT_0
121 #endif
122
123 #ifndef TPM_RH_ACT_1
124 # define TPM_RH_ACT_1 (TPM_RH_ACT_0 + 1)
125 #endif
126 #ifndef TPM_RH_ACT_2
127 # define TPM_RH_ACT_2 (TPM_RH_ACT_0 + 2)
128 #endif
129 #ifndef TPM_RH_ACT_3

```



```

130 #   define TPM_RH_ACT_3      (TPM_RH_ACT_0 + 3)
131 #endif
132 #ifndef TPM_RH_ACT_4
133 #   define TPM_RH_ACT_4      (TPM_RH_ACT_0 + 4)
134 #endif
135 #ifndef TPM_RH_ACT_5
136 #   define TPM_RH_ACT_5      (TPM_RH_ACT_0 + 5)
137 #endif
138 #ifndef TPM_RH_ACT_6
139 #   define TPM_RH_ACT_6      (TPM_RH_ACT_0 + 6)
140 #endif
141 #ifndef TPM_RH_ACT_7
142 #   define TPM_RH_ACT_7      (TPM_RH_ACT_0 + 7)
143 #endif
144 #ifndef TPM_RH_ACT_8
145 #   define TPM_RH_ACT_8      (TPM_RH_ACT_0 + 8)
146 #endif
147 #ifndef TPM_RH_ACT_9
148 #   define TPM_RH_ACT_9      (TPM_RH_ACT_0 + 9)
149 #endif
150 #ifndef TPM_RH_ACT_A
151 #   define TPM_RH_ACT_A      (TPM_RH_ACT_0 + 0xA)
152 #endif
153 #ifndef TPM_RH_ACT_B
154 #   define TPM_RH_ACT_B      (TPM_RH_ACT_0 + 0xB)
155 #endif
156 #ifndef TPM_RH_ACT_C
157 #   define TPM_RH_ACT_C      (TPM_RH_ACT_0 + 0xC)
158 #endif
159 #ifndef TPM_RH_ACT_D
160 #   define TPM_RH_ACT_D      (TPM_RH_ACT_0 + 0xD)
161 #endif
162 #ifndef TPM_RH_ACT_E
163 #   define TPM_RH_ACT_E      (TPM_RH_ACT_0 + 0xE)
164 #endif
165 #ifndef TPM_RH_ACT_F
166 #   define TPM_RH_ACT_F      (TPM_RH_ACT_0 + 0xF)
167 #endif
168
169 #define FOR_EACH_ACT(op)
170     IF_ACT_0_IMPLEMENTED(op)
171     IF_ACT_1_IMPLEMENTED(op)
172     IF_ACT_2_IMPLEMENTED(op)
173     IF_ACT_3_IMPLEMENTED(op)
174     IF_ACT_4_IMPLEMENTED(op)
175     IF_ACT_5_IMPLEMENTED(op)
176     IF_ACT_6_IMPLEMENTED(op)
177     IF_ACT_7_IMPLEMENTED(op)
178     IF_ACT_8_IMPLEMENTED(op)
179     IF_ACT_9_IMPLEMENTED(op)
180     IF_ACT_A_IMPLEMENTED(op)
181     IF_ACT_B_IMPLEMENTED(op)
182     IF_ACT_C_IMPLEMENTED(op)
183     IF_ACT_D_IMPLEMENTED(op)
184     IF_ACT_E_IMPLEMENTED(op)
185     IF_ACT_F_IMPLEMENTED(op)

```

This is the mask for ACT that are implemented

```

186 // #define ACT_MASK(N)      | (1 << 0x##N)
187 // #define ACT_IMPLEMENTED_MASK (0 FOR_EACH_ACT(ACT_MASK))
188
189 #define CASE_ACT_HANDLE(N)      case TPM_RH_ACT_##N:
190 #define CASE_ACT_NUMBER(N)     case 0x##N:
191

```

```
192 typedef struct ACT_STATE
193 {
194     UINT32      remaining;
195     TPM_ALG_ID   hashAlg;
196     TPM2B_DIGEST authPolicy;
197 } ACT_STATE, *P_ACT_STATE;
198
199 #endif // _ACT_H_
```

DRAFT

6 Main

6.1 Introduction

The files in this clause are the main processing blocks for the TPM. `ExecuteCommand.c` contains the entry point into the TPM code and the parsing of the command header. `SessionProcess.c` handles the parsing of the session area and the authorization checks, and `CommandDispatch.c` does the parameter unmarshaling and command dispatch.

6.2 ExecCommand.c

6.2.1 Introduction

This file contains the entry function `ExecuteCommand()` which provides the main control flow for TPM command execution.

6.2.2 Includes

```
1 #include "Tpm.h"
2 #include "ExecCommand_fp.h"
```

Uncomment this next `#include` if doing static command/response buffer sizing

```
3 // #include "CommandResponseSizes_fp.h"
4
```

6.2.3 ExecuteCommand()

The function performs the following steps.

- a) Parses the command header from input buffer.
- b) Calls `ParseHandleBuffer()` to parse the handle area of the command.
- c) Validates that each of the handles references a loaded entity.
- d) Calls `ParseSessionBuffer()` () to:
 - 1) unmarshal and parse the session area;
 - 2) check the authorizations; and
 - 3) when necessary, decrypt a parameter.
- e) Calls `CommandDispatcher()` to:
 - 1) unmarshal the command parameters from the command buffer;
 - 2) call the routine that performs the command actions; and
 - 3) marshal the responses into the response buffer.
- f) If any error occurs in any of the steps above create the error response and return.
- g) Calls `BuildResponseSession()` to:
 - 1) when necessary, encrypt a parameter
 - 2) build the response authorization sessions
 - 3) update the audit sessions and nonces

h) Calls BuildResponseHeader() to complete the construction of the response.

responseSize is set by the caller to the maximum number of bytes available in the output buffer. ExecuteCommand() will adjust the value and return the number of bytes placed in the buffer.

response is also set by the caller to indicate the buffer into which ExecuteCommand() is to place the response.

request and *response* may point to the same buffer

NOTE As of February, 2016, the failure processing has been moved to the platform-specific code. When the TPM code encounters an unrecoverable failure, it will SET *g_inFailureMode* and call *_plat_Fail()*. That function should not return but may call ExecuteCommand().

```

5  LIB_EXPORT void
6  ExecuteCommand(
7      uint32_t      requestSize,    // IN: command buffer size
8      unsigned char *request,      // IN: command buffer
9      uint32_t      *responseSize, // IN/OUT: response buffer size
10     unsigned char **response     // IN/OUT: response buffer
11 )
12 {
13     // Command local variables
14     UINT32      commandSize;
15     COMMAND     command;
16
17     // Response local variables
18     UINT32      maxResponse = *responseSize;
19     TPM_RC      result;      // return code for the command
20
21     // This next function call is used in development to size the command and response
22     // buffers. The values printed are the sizes of the internal structures and
23     // not the sizes of the canonical forms of the command response structures. Also,
24     // the sizes do not include the tag, command.code, requestSize, or the authorization
25     // fields.
26     //CommandResponseSizes();
27     // Set flags for NV access state. This should happen before any other
28     // operation that may require a NV write. Note, that this needs to be done
29     // even when in failure mode. Otherwise, g_updateNV would stay SET while in
30     // Failure mode and the NV would be written on each call.
31     g_updateNV = UT_NONE;
32     g_clearOrderly = FALSE;
33     if(g_inFailureMode)
34     {
35         // Do failure mode processing
36         TpmFailureMode(requestSize, request, responseSize, response);
37         return;
38     }
39     // Query platform to get the NV state. The result state is saved internally
40     // and will be reported by NvIsAvailable(). The reference code requires that
41     // accessibility of NV does not change during the execution of a command.
42     // Specifically, if NV is available when the command execution starts and then
43     // is not available later when it is necessary to write to NV, then the TPM
44     // will go into failure mode.
45     NvCheckState();
46
47     // Due to the limitations of the simulation, TPM clock must be explicitly
48     // synchronized with the system clock whenever a command is received.
49     // This function call is not necessary in a hardware TPM. However, taking
50     // a snapshot of the hardware timer at the beginning of the command allows
51     // the time value to be consistent for the duration of the command execution.
52     TimeUpdateToCurrent();
53
54     // Any command through this function will unceremoniously end the
55     // _TPM_Hash_Data/ TPM_Hash_End sequence.
56     if(g_DRTMHandle != TPM_RH_UNASSIGNED)

```

```

57     ObjectTerminateEvent();
58
59     // Get command buffer size and command buffer.
60     command.parameterBuffer = request;
61     command.parameterSize = requestSize;
62
63     // Parse command header: tag, commandSize and command.code.
64     // First parse the tag. The unmarshaling routine will validate
65     // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
66     result = TPMI_ST_COMMAND_TAG_Unmarshal(&command.tag,
67                                           &command.parameterBuffer,
68                                           &command.parameterSize);
69
70     if(result != TPM_RC_SUCCESS)
71         goto Cleanup;
72     // Unmarshal the commandSize indicator.
73     result = UINT32_Unmarshal(&commandSize,
74                             &command.parameterBuffer,
75                             &command.parameterSize);
76
77     if(result != TPM_RC_SUCCESS)
78         goto Cleanup;
79     // On a TPM that receives bytes on a port, the number of bytes that were
80     // received on that port is requestSize it must be identical to commandSize.
81     // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
82     // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
83     // as the input processing (the function that receives the command bytes and
84     // places them in the input buffer) would likely have the input truncated when
85     // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
86     if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
87     {
88         result = TPM_RC_COMMAND_SIZE;
89         goto Cleanup;
90     }
91     // Unmarshal the command code.
92     result = TPM_CC_Unmarshal(&command.code, &command.parameterBuffer,
93                             &command.parameterSize);
94
95     if(result != TPM_RC_SUCCESS)
96         goto Cleanup;
97     // Check to see if the command is implemented.
98     command.index = CommandCodeToCommandIndex(command.code);
99     if(UNIMPLEMENTED_COMMAND_INDEX == command.index)
100     {
101         result = TPM_RC_COMMAND_CODE;
102         goto Cleanup;
103     }
104 #if FIELD_UPGRADE_IMPLEMENTED == YES
105     // If the TPM is in FUM, then the only allowed command is
106     // TPM_CC_FieldUpgradeData.
107     if(IsFieldUpgradeMode() && (command.code != TPM_CC_FieldUpgradeData))
108     {
109         result = TPM_RC_UPGRADE;
110         goto Cleanup;
111     }
112 else
113 #endif
114     // Excepting FUM, the TPM only accepts TPM2_Startup() after
115     // TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
116     // is no longer allowed.
117     if(!TPMIsStarted() && command.code != TPM_CC_Startup)
118         || (TPMIsStarted() && command.code == TPM_CC_Startup)
119     {
120         result = TPM_RC_INITIALIZE;
121         goto Cleanup;
122     }
123 // Start regular command process.
124 NvIndexCacheInit();
125 // Parse Handle buffer.

```

```

123     result = ParseHandleBuffer(&command);
124     if(result != TPM_RC_SUCCESS)
125         goto Cleanup;
126     // All handles in the handle area are required to reference TPM-resident
127     // entities.
128     result = EntityGetLoadStatus(&command);
129     if(result != TPM_RC_SUCCESS)
130         goto Cleanup;
131     // Authorization session handling for the command.
132     ClearCpRpHashes(&command);
133     if(command.tag == TPM_ST_SESSIONS)
134     {
135         // Find out session buffer size.
136         result = UINT32_Unmarshal((UINT32 *)&command.authSize,
137                                 &command.parameterBuffer,
138                                 &command.parameterSize);
139         if(result != TPM_RC_SUCCESS)
140             goto Cleanup;
141         // Perform sanity check on the unmarshaled value. If it is smaller than
142         // the smallest possible session or larger than the remaining size of
143         // the command, then it is an error. NOTE: This check could pass but the
144         // session size could still be wrong. That will be determined after the
145         // sessions are unmarshaled.
146         if(command.authSize < 9
147            || command.authSize > command.parameterSize)
148         {
149             result = TPM_RC_SIZE;
150             goto Cleanup;
151         }
152         command.parameterSize -= command.authSize;
153
154         // The actions of ParseSessionBuffer() are described in the introduction.
155         // As the sessions are parsed command.parameterBuffer is advanced so, on a
156         // successful return, command.parameterBuffer should be pointing at the
157         // first byte of the parameters.
158         result = ParseSessionBuffer(&command);
159         if(result != TPM_RC_SUCCESS)
160             goto Cleanup;
161     }
162     else
163     {
164         command.authSize = 0;
165         // The command has no authorization sessions.
166         // If the command requires authorizations, then CheckAuthNoSession() will
167         // return an error.
168         result = CheckAuthNoSession(&command);
169         if(result != TPM_RC_SUCCESS)
170             goto Cleanup;
171     }
172     // Set up the response buffer pointers. CommandDispatch will marshal the
173     // response parameters starting at the address in command.responseBuffer.
174     /*response = MemoryGetResponseBuffer(command.index);
175     // leave space for the command header
176     command.responseBuffer = *response + STD_RESPONSE_HEADER;
177
178     // leave space for the parameter size field if needed
179     if(command.tag == TPM_ST_SESSIONS)
180         command.responseBuffer += sizeof(UINT32);
181     if(IsHandleInResponse(command.index))
182         command.responseBuffer += sizeof(TPM_HANDLE);
183
184     // CommandDispatcher returns a response handle buffer and a response parameter
185     // buffer if it succeeds. It will also set the parameterSize field in the
186     // buffer if the tag is TPM_RC_SESSIONS.
187     result = CommandDispatcher(&command);
188     if(result != TPM_RC_SUCCESS)

```

```

189         goto Cleanup;
190
191     // Build the session area at the end of the parameter area.
192     BuildResponseSession(&command);
193
194 Cleanup:
195     if(g_clearOrderly == TRUE
196         && NV_IS_ORDERLY)
197     {
198         #if USE_DA_USED
199             gp.orderlyState = g_daUsed ? SU_DA_USED_VALUE : SU_NONE_VALUE;
200         #else
201             gp.orderlyState = SU_NONE_VALUE;
202         #endif
203         NV_SYNC_PERSISTENT(orderlyState);
204     }
205     // This implementation loads an "evict" object to a transient object slot in
206     // RAM whenever an "evict" object handle is used in a command so that the
207     // access to any object is the same. These temporary objects need to be
208     // cleared from RAM whether the command succeeds or fails.
209     ObjectCleanupEvict();
210
211     // The parameters and sessions have been marshaled. Now tack on the header and
212     // set the sizes
213     BuildResponseHeader(&command, *response, result);
214
215     // Try to commit all the writes to NV if any NV write happened during this
216     // command execution. This check should be made for both succeeded and failed
217     // commands, because a failed one may trigger a NV write in DA logic as well.
218     // This is the only place in the command execution path that may call the NV
219     // commit. If the NV commit fails, the TPM should be put in failure mode.
220     if((g_updateNV != UT_NONE) && !g_inFailureMode)
221     {
222         if(g_updateNV == UT_ORDERLY)
223             NvUpdateIndexOrderlyData();
224         if(!NvCommit())
225             FAIL(FATAL_ERROR_INTERNAL);
226         g_updateNV = UT_NONE;
227     }
228     pAssert((UINT32)command.parameterSize <= maxResponse);
229
230     // Clear unused bits in response buffer.
231     MemorySet(*response + *responseSize, 0, maxResponse - *responseSize);
232
233     // as a final act, and not before, update the response size.
234     *responseSize = (UINT32)command.parameterSize;
235
236     return;
237 }

```


6.3 CommandDispatcher.c

6.3.1 Introduction

CommandDispatcher() performs the following operations:

- unmarshals command parameters from the input buffer;

NOTE 1 Unlike other unmarshaling functions, *parmBufferStart* does not advance but *parmBufferSize* is reduced.

- invokes the function that performs the command actions;
- marshals the returned handles, if any; and
- marshals the returned parameters, if any, into the output buffer putting in the *parameterSize* field if authorization sessions are present.

NOTE 2 The output buffer is the return from the MemoryGetResponseBuffer() function. It includes the header, handles, response parameters, and authorization area. *respParamSize* is the response parameter size, and does not include the header, handles, or authorization area.

NOTE 3 The reference implementation is permitted to do compare operations over a union as a byte array. Therefore, the command parameter *in* structure must be initialized (e.g., zeroed) before unmarshaling so that the compare operation is valid in cases where some bytes are unused.

6.3.2 Includes and Typedefs

```

1  #include "Tpm.h"
2  #include "Marshal.h"
3
4  #if TABLE_DRIVEN_DISPATCH
5
6  typedef TPM_RC (NoFlagFunction) (void *target, BYTE **buffer, INT32 *size);
7  typedef TPM_RC (FlagFunction) (void *target, BYTE **buffer, INT32 *size, BOOL flag);
8
9  typedef FlagFunction *UNMARSHAL_t;
10
11 typedef INT16 (MarshalFunction) (void *source, BYTE **buffer, INT32 *size);
12 typedef MarshalFunction *MARSHAL_t;
13
14 typedef TPM_RC (COMMAND_NO_ARGS) (void);
15 typedef TPM_RC (COMMAND_IN_ARG) (void *in);
16 typedef TPM_RC (COMMAND_OUT_ARG) (void *out);
17 typedef TPM_RC (COMMAND_INOUT_ARG) (void *in, void *out);
18
19 typedef union COMMAND_t
20 {
21     COMMAND_NO_ARGS      *noArgs;
22     COMMAND_IN_ARG       *inArg;
23     COMMAND_OUT_ARG      *outArg;
24     COMMAND_INOUT_ARG    *inOutArg;
25 } COMMAND_t;

```

This structure is used by ParseHandleBuffer() and CommandDispatcher(). The parameters in this structure are unique for each command. The parameters are:

command holds the address of the command processing function that is called by Command Dispatcher

inSize This is the size of the command-dependent input structure. The input structure holds the unmarshaled handles and command parameters. If the command takes no arguments (handles or parameters) then *inSize* will have a value of 0.

- outSize* This is the size of the command-dependent output structure. The output structure holds the results of the command in an unmarshaled form. When command processing is completed, these values are marshaled into the output buffer. It is always the case that the unmarshaled version of an output structure is larger than the marshaled version. This is because the marshaled version contains the exact same number of significant bytes but with padding removed.
- typesOffsets* This parameter points to the list of data types that are to be marshaled or unmarshaled. The list of types follows the *offsets* array. The *offsets* array is variable sized so the *typesOffset* field is necessary for the handle and command processing to be able to find the types that are being handled. The *offsets* array may be empty. The *types* structure is described below.
- offsets* This is an array of offsets of each of the parameters in the command or response. When processing the command parameters (not handles) the list contains the offset of the next parameter. For example, if the first command parameter has a size of 4 and there is a second command parameter, then the offset would be 4, indicating that the second parameter starts at 4. If the second parameter has a size of 8, and there is a third parameter, then the second entry in *offsets* is 12 (4 for the first parameter and 8 for the second). An offset value of 0 in the list indicates the start of the response parameter list. When *CommandDispatcher()* hits this value, it will stop unmarshaling the parameters and call *command*. If a command has no response parameters and only one command parameter, then *offsets* can be an empty list.

```

26 typedef struct COMMAND_DESCRIPTOR_t
27 {
28     COMMAND_t      command;           // Address of the command
29     UINT16         inSize;            // Maximum size of the input structure
30     UINT16         outSize;           // Maximum size of the output structure
31     UINT16         typesOffset;       // address of the types field
32     UINT16         offsets[1];
33 } COMMAND_DESCRIPTOR_t;

```

The *types* list is an encoded byte array. The byte value has two parts. The most significant bit is used when a parameter takes a flag and indicates if the flag should be SET or not. The remaining 7 bits are an index into an array of addresses of marshaling and unmarshaling functions. The array of functions is divided into 6 sections with a value assigned to denote the start of that section (and the end of the previous section). The defined offset values for each section are:

0	unmarshaling for handles that do not take flags
HANDLE_FIRST_FLAG_TYPE	unmarshaling for handles that take flags
PARAMETER_FIRST_TYPE	unmarshaling for parameters that do not take flags
PARAMETER_FIRST_FLAG_TYPE	unmarshaling for parameters that take flags
PARAMETER_LAST_TYPE + 1	marshaling for handles
RESPONSE_PARAMETER_FIRST_TYPE	marshaling for parameters
RESPONSE_PARAMETER_LAST_TYPE	is the last value in the list of marshaling and unmarshaling functions. The <i>types</i> list is constructed with a byte of 0xff at the end of the command parameters and with an 0xff at the end of the response parameters.

```

34 #if COMPRESSED_LISTS
35 #   define PAD_LIST 0
36 #else
37 #   define PAD_LIST 1
38 #endif

```

```

39 #define _COMMAND_TABLE_DISPATCH_
40 #include "CommandDispatchData.h"
41
42 #define TEST_COMMAND    TPM_CC_Startup
43
44 #define NEW_CC
45
46 #else
47
48 #include "Commands.h"
49
50 #endif

```

6.3.3 Marshal/Unmarshal Functions

6.3.3.1 ParseHandleBuffer()

This is the table-driven version of the handle buffer unmarshaling code

```

51 TPM_RC
52 ParseHandleBuffer(
53     COMMAND                *command
54 )
55 {
56     TPM_RC                result;
57 #if TABLE_DRIVEN_DISPATCH
58     COMMAND_DESCRIPTOR_t  *desc;
59     BYTE                  *types;
60     BYTE                  type;
61     BYTE                  dType;
62
63     // Make sure that nothing strange has happened
64     pAssert(command->index
65             < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
66     // Get the address of the descriptor for this command
67     desc = s_CommandDataArray[command->index];
68
69     pAssert(desc != NULL);
70     // Get the associated list of unmarshaling data types.
71     types = &((BYTE *)desc)[desc->typesOffset];
72
73     // if(s_ccAttr[commandIndex].commandIndex == TEST_COMMAND)
74     //     commandIndex = commandIndex;
75     // No handles yet
76     command->handleNum = 0;
77
78     // Get the first type value
79     for(type = *types++;
80         // check each byte to make sure that we have not hit the start
81         // of the parameters
82         (dType = (type & 0x7F)) < PARAMETER_FIRST_TYPE;
83         // get the next type
84         type = *types++)
85     {
86 #if TABLE_DRIVEN_MARSHAL
87         marshalIndex_t    index;
88         index = unmarshalArray[dType] | ((type & 0x80) ? NULL_FLAG : 0);
89         result = Unmarshal(index, &(command->handles[command->handleNum]),
90                           &command->parameterBuffer, &command->parameterSize);
91 #else
92 #else
93         // See if unmarshaling of this handle type requires a flag
94         if(dType < HANDLE_FIRST_FLAG_TYPE)

```

```

95     {
96         // Look up the function to do the unmarshaling
97         NoFlagFunction *f = (NoFlagFunction *)unmarshalArray[dType];
98         // call it
99         result = f(&(command->handles[command->handleNum]),
100                  &command->parameterBuffer,
101                  &command->parameterSize);
102     }
103     else
104     {
105         // Look up the function
106         FlagFunction *f = unmarshalArray[dType];
107
108         // Call it setting the flag to the appropriate value
109         result = f(&(command->handles[command->handleNum]),
110                  &command->parameterBuffer,
111                  &command->parameterSize, (type & 0x80) != 0);
112     }
113 #endif
114
115     // Got a handle
116     // We do this first so that the match for the handle offset of the
117     // response code works correctly.
118     command->handleNum += 1;
119     if(result != TPM_RC_SUCCESS)
120         // if the unmarshaling failed, return the response code with the
121         // handle indication set
122         return result + TPM_RC_H + (command->handleNum * TPM_RC_1);
123 }
124 #else
125 BYTE          **handleBufferStart = &command->parameterBuffer;
126 INT32         *bufferRemainingSize = &command->parameterSize;
127 TPM_HANDLE    *handles = &command->handles[0];
128 UINT32        *handleCount = &command->handleNum;
129 *handleCount = 0;
130 switch(command->code)
131 {
132 #include "HandleProcess.h"
133 #undef handles
134     default:
135         FAIL(FATAL_ERROR_INTERNAL);
136         break;
137 }
138 #endif
139 return TPM_RC_SUCCESS;
140 }

```

6.3.3.2 CommandDispatcher()

Function to unmarshal the command parameters, call the selected action code, and marshal the response parameters.

```

141 TPM_RC
142 CommandDispatcher(
143     COMMAND          *command
144 )
145 {
146 #if !TABLE_DRIVEN_DISPATCH
147     TPM_RC result;
148     BYTE    **paramBuffer = &command->parameterBuffer;
149     INT32    *paramBufferSize = &command->parameterSize;
150     BYTE    **responseBuffer = &command->responseBuffer;
151     INT32    *respParamSize = &command->parameterSize;
152     INT32    rSize;

```

```

153     TPM_HANDLE *handles = &command->handles[0];
154 //
155     command->handleNum = 0;                                // The command-specific code knows how
156                                                            // many handles there are. This is for
157                                                            // cataloging the number of response
158                                                            // handles
159     MemoryIoBufferAllocationReset();                       // Initialize so that allocation will
160                                                            // work properly
161     switch (GetCommandCode (command->index))
162     {
163 #include "CommandDispatcher.h"
164
165         default:
166             FAIL(FATAL_ERROR_INTERNAL);
167             break;
168     }
169 Exit:
170     MemoryIoBufferZero();
171     return result;
172 #else
173     COMMAND_DESCRIPTOR_t *desc;
174     BYTE *types;
175     BYTE type;
176     UINT16 *offsets;
177     UINT16 offset = 0;
178     UINT32 maxInSize;
179     BYTE *commandIn;
180     INT32 maxOutSize;
181     BYTE *commandOut;
182     COMMAND_t cmd;
183     TPM_HANDLE *handles;
184     UINT32 hasInParameters = 0;
185     BOOL hasOutParameters = FALSE;
186     UINT32 pNum = 0;
187     BYTE dType; // dispatch type
188     TPM_RC result;
189 //
190 // Get the address of the descriptor for this command
191 pAssert(command->index
192         < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
193 desc = s_CommandDataArray[command->index];
194
195 // Get the list of parameter types for this command
196 pAssert(desc != NULL);
197 types = &((BYTE *)desc)[desc->typesOffset];
198
199 // Get a pointer to the list of parameter offsets
200 offsets = &desc->offsets[0];
201 // pointer to handles
202 handles = command->handles;
203
204 // Get the size required to hold all the unmarshaled parameters for this command
205 maxInSize = desc->inSize;
206 // and the size of the output parameter structure returned by this command
207 maxOutSize = desc->outSize;
208
209 MemoryIoBufferAllocationReset();
210 // Get a buffer for the input parameters
211 commandIn = MemoryGetInBuffer(maxInSize);
212 // And the output parameters
213 commandOut = (BYTE *)MemoryGetOutBuffer((UINT32)maxOutSize);
214
215 // Get the address of the action code dispatch
216 cmd = desc->command;
217
218 // Copy any handles into the input buffer

```

```

219     for(type = *types++; (type & 0x7F) < PARAMETER_FIRST_TYPE; type = *types++)
220     {
221         // 'offset' was initialized to zero so the first unmarshaling will always
222         // be to the start of the data structure
223         *(TPM_HANDLE *)&(commandIn[offset]) = *handles++;
224         // This check is used so that we don't have to add an additional offset
225         // value to the offsets list to correspond to the stop value in the
226         // command parameter list.
227         if(*types != 0xFF)
228             offset = *offsets++;
229     //     maxInSize -= sizeof(TPM_HANDLE);
230     hasInParameters++;
231     }
232     // Exit loop with type containing the last value read from types
233     // maxInSize has the amount of space remaining in the command action input
234     // buffer. Make sure that we don't have more data to unmarshal than is going to
235     // fit.
236
237     // type contains the last value read from types so it is not necessary to
238     // reload it, which is good because *types now points to the next value
239     for(; (dType = (type & 0x7F)) <= PARAMETER_LAST_TYPE; type = *types++)
240     {
241         pNum++;
242     #if TABLE_DRIVEN_UNMARSHAL
243         {
244             marshalIndex_t    index = unmarshalArray[dType];
245             index |= (type & 0x80) ? NULL_FLAG : 0;
246             result = Unmarshal(index, &commandIn[offset], &command->parameterBuffer,
247                               &command->parameterSize);
248         }
249     #else
250         if(dType < PARAMETER_FIRST_FLAG_TYPE)
251         {
252             NoFlagFunction    *f = (NoFlagFunction *)unmarshalArray[dType];
253             result = f(&commandIn[offset], &command->parameterBuffer,
254                       &command->parameterSize);
255         }
256         else
257         {
258             FlagFunction      *f = unmarshalArray[dType];
259             result = f(&commandIn[offset], &command->parameterBuffer,
260                       &command->parameterSize,
261                       (type & 0x80) != 0);
262         }
263     #endif
264     if(result != TPM_RC_SUCCESS)
265     {
266         result += TPM_RC_P + (TPM_RC_1 * pNum);
267         goto Exit;
268     }
269     // This check is used so that we don't have to add an additional offset
270     // value to the offsets list to correspond to the stop value in the
271     // command parameter list.
272     if(*types != 0xFF)
273         offset = *offsets++;
274     hasInParameters++;
275 }
276 // Should have used all the bytes in the input
277 if(command->parameterSize != 0)
278 {
279     result = TPM_RC_SIZE;
280     goto Exit;
281 }
282 // The command parameter unmarshaling stopped when it hit a value that was out
283 // of range for unmarshaling values and left *types pointing to the first
284 // marshaling type. If that type happens to be the STOP value, then there

```

```

285 // are no response parameters. So, set the flag to indicate if there are
286 // output parameters.
287 hasOutParameters = *types != 0xFF;
288
289 // There are four cases for calling, with and without input parameters and with
290 // and without output parameters.
291 if(hasInParameters > 0)
292 {
293     if(hasOutParameters)
294         result = cmd.inOutArg(commandIn, commandOut);
295     else
296         result = cmd.inArg(commandIn);
297 }
298 else
299 {
300     if(hasOutParameters)
301         result = cmd.outArg(commandOut);
302     else
303         result = cmd.noArgs();
304 }
305 if(result != TPM_RC_SUCCESS)
306     goto Exit;
307
308 // Offset in the marshaled output structure
309 offset = 0;
310
311 // Process the return handles, if any
312 command->handleNum = 0;
313
314 // Could make this a loop to process output handles but there is only ever
315 // one handle in the outputs (for now).
316 type = *types++;
317 if((dType = (type & 0x7F)) < RESPONSE_PARAMETER_FIRST_TYPE)
318 {
319     // The out->handle value was referenced as TPM HANDLE in the
320     // action code so it has to be properly aligned.
321     command->handles[command->handleNum++] =
322         *((TPM_HANDLE *)&(commandOut[offset]));
323     maxOutSize -= sizeof(UINT32);
324     type = *types++;
325     offset = *offsets++;
326 }
327 // Use the size of the command action output buffer as the maximum for the
328 // number of bytes that can get marshaled. Since the marshaling code has
329 // no pointers to data, all of the data being returned has to be in the
330 // command action output buffer. If we try to marshal more bytes than
331 // could fit into the output buffer, we need to fail.
332 for(; (dType = (type & 0x7F)) <= RESPONSE_PARAMETER_LAST_TYPE
333     && !g_inFailureMode; type = *types++)
334 {
335 #if TABLE_DRIVEN MARSHAL
336     marshalIndex_t index = marshalArray[dType];
337     command->parameterSize += Marshal(index, &commandOut[offset],
338                                     &command->responseBuffer,
339                                     &maxOutSize);
340 #else
341     const MARSHAL_t f = marshalArray[dType];
342
343     command->parameterSize += f(&commandOut[offset],
344                               &command->responseBuffer,
345                               &maxOutSize);
346 #endif
347     offset = *offsets++;
348 }
349 result = (maxOutSize < 0) ? TPM_RC_FAILURE : TPM_RC_SUCCESS;
350 Exit:

```



```
351     MemoryIoBufferZero();  
352     return result;  
353 #endif  
354 }
```

DRAFT

6.4 SessionProcess.c

6.4.1 Introduction

This file contains the subsystem that process the authorization sessions including implementation of the Dictionary Attack logic. ExecCommand() uses ParseSessionBuffer() to process the authorization session area of a command and BuildResponseSession() to create the authorization session area of a response.

6.4.2 Includes and Data Definitions

```

1  #define SESSION_PROCESS_C
2
3  #include "Tpm.h"
4  #include "ACT.h"
5

```

6.4.3 Authorization Support Functions

6.4.3.1 IsDAExempted()

This function indicates if a handle is exempted from DA logic. A handle is exempted if it is:

- a) a primary seed handle;
- b) an object with *noDA* bit SET;
- c) an NV Index with TPMA_NV_NO_DA bit SET; or
- d) a PCR handle.

Return Value	Meaning
TRUE(1)	handle is exempted from DA logic
FALSE(0)	handle is not exempted from DA logic

```

6  BOOL
7  IsDAExempted(
8      TPM_HANDLE    handle        // IN: entity handle
9  )
10 {
11     BOOL    result = FALSE;
12     //
13     switch(HandleGetType(handle))
14     {
15         case TPM_HT_PERMANENT:
16             // All permanent handles, other than TPM_RH_LOCKOUT, are exempt from
17             // DA protection.
18             result = (handle != TPM_RH_LOCKOUT);
19             break;
20             // When this function is called, a persistent object will have been loaded
21             // into an object slot and assigned a transient handle.
22         case TPM_HT_TRANSIENT:
23             {
24                 TPMA_OBJECT    attributes = ObjectGetPublicAttributes(handle);
25                 result = IS_ATTRIBUTE(attributes, TPMA_OBJECT, noDA);
26                 break;
27             }
28         case TPM_HT_NV_INDEX:
29             {
30                 NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);

```

```

31         result = IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, NO_DA);
32         break;
33     }
34     case TPM_HT_PCR:
35         // PCRs are always exempted from DA.
36         result = TRUE;
37         break;
38     default:
39         break;
40 }
41 return result;
42 }

```

6.4.3.2 IncrementLockout()

This function is called after an authorization failure that involves use of an *authValue*. If the entity referenced by the handle is not exempt from DA protection, then the *failedTries* counter will be incremented.

Error Return	Meaning
TPM_RC_AUTH_FAIL	authorization failure that caused DA lockout to increment
TPM_RC_BAD_AUTH	authorization failure did not cause DA lockout to increment

```

43 static TPM_RC
44 IncrementLockout(
45     UINT32      sessionIndex
46 )
47 {
48     TPM_HANDLE   handle = s_associatedHandles[sessionIndex];
49     TPM_HANDLE   sessionHandle = s_sessionHandles[sessionIndex];
50     SESSION      *session = NULL;
51     //
52     // Don't increment lockout unless the handle associated with the session
53     // is DA protected or the session is bound to a DA protected entity.
54     if(sessionHandle == TPM_RS_PW)
55     {
56         if(IsDAExempted(handle))
57             return TPM_RC_BAD_AUTH;
58     }
59     else
60     {
61         session = SessionGet(sessionHandle);
62         // If the session is bound to lockout, then use that as the relevant
63         // handle. This means that an authorization failure with a bound session
64         // bound to lockoutAuth will take precedence over any other
65         // lockout check
66         if(session->attributes.isLockoutBound == SET)
67             handle = TPM_RH_LOCKOUT;
68         if(session->attributes.isDaBound == CLEAR
69             && (IsDAExempted(handle) || session->attributes.includeAuth == CLEAR))
70             // If the handle was changed to TPM_RH_LOCKOUT, this will not return
71             // TPM_RC_BAD_AUTH
72             return TPM_RC_BAD_AUTH;
73     }
74     if(handle == TPM_RH_LOCKOUT)
75     {
76         pAssert(gp.lockOutAuthEnabled == TRUE);
77
78         // lockout is no longer enabled
79         gp.lockOutAuthEnabled = FALSE;
80
81         // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since

```

```

82     // the lockout authorization will be reset at startup.
83     if(gp.lockoutRecovery != 0)
84     {
85         if(NV_IS_AVAILABLE)
86             // Update NV.
87             NV_SYNC_PERSISTENT(lockOutAuthEnabled);
88         else
89             // No NV access for now. Put the TPM in pending mode.
90             s_DAPendingOnNV = TRUE;
91     }
92 }
93 else
94 {
95     if(gp.recoveryTime != 0)
96     {
97         gp.failedTries++;
98         if(NV_IS_AVAILABLE)
99             // Record changes to NV. NvWrite will SET g_updateNV
100             NV_SYNC_PERSISTENT(failedTries);
101         else
102             // No NV access for now. Put the TPM in pending mode.
103             s_DAPendingOnNV = TRUE;
104     }
105 }
106 // Register a DA failure and reset the timers.
107 DARegisterFailure(handle);
108
109 return TPM_RC_AUTH_FAIL;
110 }

```

6.4.3.3 IsSessionBindEntity()

This function indicates if the entity associated with the handle is the entity, to which this session is bound. The binding would occur by making the **bind** parameter in TPM2_StartAuthSession() not equal to TPM_RH_NULL. The binding only occurs if the session is an HMAC session. The bind value is a combination of the Name and the *authValue* of the entity.

Return Value	Meaning
TRUE(1)	handle points to the session start entity
FALSE(0)	handle does not point to the session start entity

```

111 static BOOL
112 IsSessionBindEntity(
113     TPM_HANDLE    associatedHandle, // IN: handle to be authorized
114     SESSION       *session         // IN: associated session
115 )
116 {
117     TPM2B_NAME    entity;          // The bind value for the entity
118     //
119     // If the session is not bound, return FALSE.
120     if(session->attributes.isBound)
121     {
122         // Compute the bind value for the entity.
123         SessionComputeBoundEntity(associatedHandle, &entity);
124
125         // Compare to the bind value in the session.
126         return MemoryEqual2B(&entity.b, &session->u1.boundEntity.b);
127     }
128     return FALSE;
129 }

```

6.4.3.4 IsPolicySessionRequired()

Checks if a policy session is required for a command. If a command requires DUP or ADMIN role authorization, then the handle that requires that role is the first handle in the command. This simplifies this checking. If a new command is created that requires multiple ADMIN role authorizations, then it will have to be special-cased in this function. A policy session is required if:

- a) the command requires the DUP role;
- b) the command requires the ADMIN role and the authorized entity is an object and its *adminWithPolicy* bit is SET;
- c) the command requires the ADMIN role and the authorized entity is a permanent handle or an NV Index; or
- d) the authorized entity is a PCR belonging to a policy group, and has its policy initialized

Return Value	Meaning
TRUE(1)	policy session is required
FALSE(0)	policy session is not required

```

130 static BOOL
131 IsPolicySessionRequired(
132     COMMAND_INDEX    commandIndex, // IN: command index
133     UINT32            sessionIndex  // IN: session index
134 )
135 {
136     AUTH_ROLE    role = CommandAuthRole(commandIndex, sessionIndex);
137     TPM_HT       type = HandleGetType(s_associatedHandles[sessionIndex]);
138     //
139     if(role == AUTH_DUP)
140         return TRUE;
141     if(role == AUTH_ADMIN)
142     {
143         // We allow an exception for ADMIN role in a transient object. If the object
144         // allows ADMIN role actions with authorization, then policy is not
145         // required. For all other cases, there is no way to override the command
146         // requirement that a policy be used
147         if(type == TPM_HT_TRANSIENT)
148         {
149             OBJECT    *object = HandleToObject(s_associatedHandles[sessionIndex]);
150             if(!IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT,
151                             adminWithPolicy))
152                 return FALSE;
153             return TRUE;
154         }
155         return TRUE;
156     }
157     if(type == TPM_HT_PCR)
158     {
159         if(PCRPolicyIsAvailable(s_associatedHandles[sessionIndex]))
160         {
161             TPM2B_DIGEST    policy;
162             TPMI_ALG_HASH    policyAlg;
163             policyAlg = PCRGetAuthPolicy(s_associatedHandles[sessionIndex],
164                                         &policy);
165             if(policyAlg != TPM_ALG_NULL)
166                 return TRUE;
167         }
168     }
169     return FALSE;
170 }

```

6.4.3.5 IsAuthValueAvailable()

This function indicates if *authValue* is available and allowed for USER role authorization of an entity.

This function is similar to *IsAuthPolicyAvailable()* except that it does not check the size of the *authValue* as *IsAuthPolicyAvailable()* does (a null *authValue* is a valid authorization, but a null policy is not a valid policy).

This function does not check that the handle reference is valid or if the entity is in an enabled hierarchy. Those checks are assumed to have been performed during the handle unmarshaling.

Return Value	Meaning
TRUE(1)	<i>authValue</i> is available
FALSE(0)	<i>authValue</i> is not available

```

171 static BOOL
172 IsAuthValueAvailable(
173     TPM_HANDLE      handle,          // IN: handle of entity
174     COMMAND_INDEX   commandIndex,   // IN: command index
175     UINT32          sessionIndex    // IN: session index
176 )
177 {
178     BOOL            result = FALSE;
179     //
180     switch(HandleGetType(handle))
181     {
182         case TPM_HT_PERMANENT:
183             switch(handle)
184             {
185                 // At this point hierarchy availability has already been
186                 // checked so primary seed handles are always available here
187                 case TPM_RH_OWNER:
188                 case TPM_RH_ENDORSEMENT:
189                 case TPM_RH_PLATFORM:
190 #ifdef VENDOR_PERMANENT
191                 // This vendor defined handle associated with the
192                 // manufacturer's shared secret
193                 case VENDOR_PERMANENT:
194 #endif
195                 // The DA checking has been performed on LockoutAuth but we
196                 // bypass the DA logic if we are using lockout policy. The
197                 // policy would allow execution to continue an lockoutAuth
198                 // could be used, even if direct use of lockoutAuth is disabled
199                 case TPM_RH_LOCKOUT:
200                 // NullAuth is always available.
201                 case TPM_RH_NULL:
202                     result = TRUE;
203                     break;
204                 FOR_EACH_ACT(CASE_ACT_HANDLE)
205                 {
206                     // The ACT auth value is not available if the platform is disabled
207                     result = g_phEnable == SET;
208                     break;
209                 }
210                 default:
211                     // Otherwise authValue is not available.
212                     break;
213             }
214             break;
215         case TPM_HT_TRANSIENT:
216             // A persistent object has already been loaded and the internal
217             // handle changed.
218     }

```

```

219         OBJECT          *object;
220         TPMA_OBJECT      attributes;
221     //
222     object = HandleToObject(handle);
223     attributes = object->publicArea.objectAttributes;
224
225     // authValue is always available for a sequence object.
226     // An alternative for this is to
227     // SET_ATTRIBUTE(object->publicArea, TPMA_OBJECT, userWithAuth) when the
228     // sequence is started.
229     if(ObjectIsSequence(object))
230     {
231         result = TRUE;
232         break;
233     }
234     // authValue is available for an object if it has its sensitive
235     // portion loaded and
236     // a) userWithAuth bit is SET, or
237     // b) ADMIN role is required
238     if(object->attributes.publicOnly == CLEAR
239        && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, userWithAuth)
240           || (CommandAuthRole(commandIndex, sessionIndex) == AUTH_ADMIN
241              && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, adminWithPolicy))))
242         result = TRUE;
243     }
244     break;
245     case TPM_HT_NV_INDEX:
246         // NV Index.
247     {
248         NV_REF          locator;
249         NV_INDEX         *nvIndex = NvGetIndexInfo(handle, &locator);
250         TPMA_NV          nvAttributes;
251     //
252         pAssert(nvIndex != 0);
253
254         nvAttributes = nvIndex->publicArea.attributes;
255
256         if(IsWriteOperation(commandIndex))
257         {
258             // AuthWrite can't be set for a PIN index
259             if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHWRITE))
260                 result = TRUE;
261         }
262         else
263         {
264             // A "read" operation
265             // For a PIN Index, the authValue is available as long as the
266             // Index has been written and the pinCount is less than pinLimit
267             if(IsNvPinFailIndex(nvAttributes)
268                || IsNvPinPassIndex(nvAttributes))
269             {
270                 NV_PIN      pin;
271                 if(!IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))
272                     break; // return false
273                 // get the index values
274                 pin.intVal = NvGetUINT64Data(nvIndex, locator);
275                 if(pin.pin.pinCount < pin.pin.pinLimit)
276                     result = TRUE;
277             }
278             // For non-PIN Indexes, need to allow use of the authValue
279             else if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHREAD))
280                 result = TRUE;
281         }
282     }
283     break;
284     case TPM_HT_PCR:

```



```

285         // PCR handle.
286         // authValue is always allowed for PCR
287         result = TRUE;
288         break;
289     default:
290         // Otherwise, authValue is not available
291         break;
292 }
293 return result;
294 }

```

6.4.3.6 IsAuthPolicyAvailable()

This function indicates if an *authPolicy* is available and allowed.

This function does not check that the handle reference is valid or if the entity is in an enabled hierarchy. Those checks are assumed to have been performed during the handle unmarshaling.

Return Value	Meaning
TRUE(1)	<i>authPolicy</i> is available
FALSE(0)	<i>authPolicy</i> is not available

```

295 static BOOL
296 IsAuthPolicyAvailable(
297     TPM_HANDLE      handle,          // IN: handle of entity
298     COMMAND_INDEX   commandIndex,   // IN: command index
299     UINT32          sessionIndex    // IN: session index
300 )
301 {
302     BOOL            result = FALSE;
303     //
304     switch(HandleGetType(handle))
305     {
306     case TPM_HT_PERMANENT:
307         switch(handle)
308         {
309             // At this point hierarchy availability has already been checked.
310             case TPM_RH_OWNER:
311                 if(gp.ownerPolicy.t.size != 0)
312                     result = TRUE;
313                 break;
314             case TPM_RH_ENDORSEMENT:
315                 if(gp.endorsementPolicy.t.size != 0)
316                     result = TRUE;
317                 break;
318             case TPM_RH_PLATFORM:
319                 if(gc.platformPolicy.t.size != 0)
320                     result = TRUE;
321                 break;
322             #define ACT_GET_POLICY(N)
323                 case TPM_RH_ACT_##N:
324                     if(go.ACT_##N.authPolicy.t.size != 0)
325                         result = TRUE;
326                     break;
327             //
328             FOR_EACH_ACT(ACT_GET_POLICY)
329             case TPM_RH_LOCKOUT:
330                 if(gp.lockoutPolicy.t.size != 0)
331                     result = TRUE;
332                 break;
333             default:
334

```

```

335         break;
336     }
337     break;
338 case TPM_HT_TRANSIENT:
339 {
340     // Object handle.
341     // An evict object would already have been loaded and given a
342     // transient object handle by this point.
343     OBJECT *object = HandleToObject(handle);
344     // Policy authorization is not available for an object with only
345     // public portion loaded.
346     if(object->attributes.publicOnly == CLEAR)
347     {
348         // Policy authorization is always available for an object but
349         // is never available for a sequence.
350         if(!ObjectIsSequence(object))
351             result = TRUE;
352     }
353     break;
354 }
355 case TPM_HT_NV_INDEX:
356     // An NV Index.
357 {
358     NV_INDEX      *nvIndex = NvGetIndexInfo(handle, NULL);
359     TPMA_NV        nvAttributes = nvIndex->publicArea.attributes;
360     //
361     // If the policy size is not zero, check if policy can be used.
362     if(nvIndex->publicArea.authPolicy.t.size != 0)
363     {
364         // If policy session is required for this handle, always
365         // uses policy regardless of the attributes bit setting
366         if(IsPolicySessionRequired(commandIndex, sessionIndex))
367             result = TRUE;
368         // Otherwise, the presence of the policy depends on the NV
369         // attributes.
370         else if(IsWriteOperation(commandIndex))
371         {
372             if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYWRITE))
373                 result = TRUE;
374         }
375         else
376         {
377             if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYREAD))
378                 result = TRUE;
379         }
380     }
381 }
382 break;
383 case TPM_HT_PCR:
384     // PCR handle.
385     if(PCRPolicyIsAvailable(handle))
386         result = TRUE;
387     break;
388 default:
389     break;
390 }
391 return result;
392 }

```

6.4.4 Session Parsing Functions

6.4.4.1 ClearCpRpHashes()

```
393 void
```

```

394 ClearCpRphashes (
395     COMMAND          *command
396 )
397 {
398     // The macros expand according to the implemented hash algorithms. An IDE may
399     // complain that COMMAND does not contain SHA1CpHash or SHA1RpHash because of the
400     // complexity of the macro expansion where the data space is defined; but, if SHA1
401     // is implemented, it actually does and the compiler is happy.
402     #define CLEAR_CP_HASH(HASH, Hash)      command->Hash##CpHash.b.size = 0;
403     FOR_EACH_HASH(CLEAR_CP_HASH)
404     #define CLEAR_RP_HASH(HASH, Hash)      command->Hash##RpHash.b.size = 0;
405     FOR_EACH_HASH(CLEAR_RP_HASH)
406 }

```

6.4.4.2 GetCpHashPointer()

Function to get a pointer to the *cpHash* of the command

```

407 static TPM2B_DIGEST *
408 GetCpHashPointer(
409     COMMAND          *command,
410     TPMI_ALG_HASH    hashAlg
411 )
412 {
413     TPM2B_DIGEST      *retVal;
414     //
415     // Define the macro that will expand for each implemented algorithm in the switch
416     // statement below.
417     #define GET_CP_HASH_POINTER(HASH, Hash)
418     \
419         case ALG_###HASH##_VALUE:
420             retVal = (TPM2B_DIGEST *) &command->Hash##CpHash;
421             break;
422     \
423     switch(hashAlg)
424     {
425         // For each implemented hash, this will expand as defined above
426         // by GET_CP_HASH_POINTER. Your IDE may complain that
427         // 'struct COMMAND' has no field "SHA1CpHash" but the compiler says
428         // it does, so...
429         FOR_EACH_HASH(GET_CP_HASH_POINTER)
430         default:
431             retVal = NULL;
432             break;
433     }
434     return retVal;
435 }

```

6.4.4.3 GetRpHashPointer()

Function to get a pointer to the *RpHash()* of the command

```

435 static TPM2B_DIGEST *
436 GetRpHashPointer(
437     COMMAND          *command,
438     TPMI_ALG_HASH    hashAlg
439 )
440 {
441     TPM2B_DIGEST      *retVal;
442     //
443     // Define the macro that will expand for each implemented algorithm in the switch
444     // statement below.
445     #define GET_RP_HASH_POINTER(HASH, Hash)

```

```

446     case ALG_ ##HASH##_VALUE:
447         retVal = (TPM2B_DIGEST *) &command->Hash##RpHash;
448         break;
449
450     switch(hashAlg)
451     {
452         // For each implemented hash, this will expand as defined above
453         // by GET_RP_HASH_POINTER. Your IDE may complain that
454         // 'struct "COMMAND" has no field 'SHA1RpHash'" but the compiler says
455         // it does, so...
456         FOR_EACH_HASH(GET_RP_HASH_POINTER)
457         default:
458             retVal = NULL;
459             break;
460     }
461     return retVal;
462 }

```

6.4.4.4 ComputeCpHash()

This function computes the *cpHash* as defined in Part 2 and described in Part 1.

```

463 static TPM2B_DIGEST *
464 ComputeCpHash(
465     COMMAND          *command,           // IN: command parsing structure
466     TPMI_ALG_HASH    hashAlg            // IN: hash algorithm
467 )
468 {
469     UINT32            i;
470     HASH_STATE        hashState;
471     TPM2B_NAME        name;
472     TPM2B_DIGEST      *cpHash;
473     //
474     // cpHash = hash(commandCode [ || authName1
475     //                  [ || authName2
476     //                  [ || authName 3 ]]]
477     //                  [ || parameters])
478     // A cpHash can contain just a commandCode only if the lone session is
479     // an audit session.
480     // Get pointer to the hash value
481     cpHash = GetCpHashPointer(command, hashAlg);
482     if(cpHash->t.size == 0)
483     {
484         cpHash->t.size = CryptHashStart(&hashState, hashAlg);
485         // Add commandCode.
486         CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
487         // Add authNames for each of the handles.
488         for(i = 0; i < command->handleNum; i++)
489             CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
490                                                         &name)->b);
491         // Add the parameters.
492         CryptDigestUpdate(&hashState, command->parameterSize,
493                         command->parameterBuffer);
494         // Complete the hash.
495         CryptHashEnd2B(&hashState, &cpHash->b);
496     }
497     return cpHash;
498 }

```

6.4.4.5 GetCpHash()

This function is used to access a precomputed *cpHash*.

```

499 static TPM2B_DIGEST *
500 GetCpHash(
501     COMMAND      *command,
502     TPMI_ALG_HASH hashAlg
503 )
504 {
505     TPM2B_DIGEST *cpHash = GetCpHashPointer(command, hashAlg);
506     //
507     pAssert(cpHash->t.size != 0);
508     return cpHash;
509 }

```

6.4.4.6 CompareTemplateHash()

This function computes the template hash and compares it to the session *templateHash*. It is the hash of the second parameter assuming that the command is TPM2_Create(), TPM2_CreatePrimary(), or TPM2_CreateLoaded()

Return Value	Meaning
TRUE(1)	template hash equal to <i>session→templateHash</i>
FALSE(0)	template hash not equal to <i>session→templateHash</i>

```

510 static BOOL
511 CompareTemplateHash(
512     COMMAND      *command,           // IN: parsing structure
513     SESSION      *session           // IN: session data
514 )
515 {
516     BYTE          *pBuffer = command->parameterBuffer;
517     INT32          pSize = command->parameterSize;
518     TPM2B_DIGEST  tHash;
519     UINT16         size;
520     //
521     // Only try this for the three commands for which it is intended
522     if(command->code != TPM_CC_Create
523         && command->code != TPM_CC_CreatePrimary
524 #if CC_CreateLoaded
525         && command->code != TPM_CC_CreateLoaded
526 #endif
527     )
528         return FALSE;
529     // Assume that the first parameter is a TPM2B and unmarshal the size field
530     // Note: this will not affect the parameter buffer and size in the calling
531     // function.
532     if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
533         return FALSE;
534     // reduce the space in the buffer.
535     // NOTE: this could make pSize go negative if the parameters are not correct but
536     // the unmarshaling code does not try to unmarshal if the remaining size is
537     // negative.
538     pSize -= size;
539
540     // Advance the pointer
541     pBuffer += size;
542
543     // Get the size of what should be the template
544     if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
545         return FALSE;
546     // See if this is reasonable
547     if(size > pSize)
548         return FALSE;
549     // Hash the template data

```

```

550     tHash.t.size = CryptHashBlock(session->authHashAlg, size, pBuffer,
551                                   sizeof(tHash.t.buffer), tHash.t.buffer);
552     return(MemoryEqual2B(&session->ul.templateHash.b, &tHash.b));
553 }

```

6.4.4.7 CompareNameHash()

This function computes the name hash and compares it to the *nameHash* in the session data.

```

554 BOOL
555 CompareNameHash(
556     COMMAND      *command,          // IN: main parsing structure
557     SESSION      *session           // IN: session structure with nameHash
558 )
559 {
560     HASH_STATE    hashState;
561     TPM2B_DIGEST  nameHash;
562     UINT32        i;
563     TPM2B_NAME     name;
564     //
565     nameHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
566     // Add names.
567     for(i = 0; i < command->handleNum; i++)
568         CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
569                                                         &name)->b);
570     // Complete hash.
571     CryptHashEnd2B(&hashState, &nameHash.b);
572     // and compare
573     return MemoryEqual(session->ul.nameHash.t.buffer, nameHash.t.buffer,
574                        nameHash.t.size);
575 }

```

6.4.4.8 CheckPWAuthSession()

This function validates the authorization provided in a PWAP session. It compares the input value to *authValue* of the authorized entity. Argument *sessionIndex* is used to get handles handle of the referenced entities from *s_inputAuthValues[]* and *s_associatedHandles[]*.

Error Return	Meaning
TPM_RC_AUTH_FAIL	authorization fails and increments DA failure count
TPM_RC_BAD_AUTH	authorization fails but DA does not apply

```

576 static TPM_RC
577 CheckPWAuthSession(
578     UINT32        sessionIndex      // IN: index of session to be processed
579 )
580 {
581     TPM2B_AUTH     authValue;
582     TPM_HANDLE     associatedHandle = s_associatedHandles[sessionIndex];
583     //
584     // Strip trailing zeros from the password.
585     MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
586
587     // Get the authValue with trailing zeros removed
588     EntityGetAuthValue(associatedHandle, &authValue);
589
590     // Success if the values are identical.
591     if(MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &authValue.b))
592     {
593         return TPM_RC_SUCCESS;
594     }

```

```

595     else                                // if the digests are not identical
596     {
597         // Invoke DA protection if applicable.
598         return IncrementLockout(sessionIndex);
599     }
600 }

```

6.4.4.9 ComputeCommandHMAC()

This function computes the HMAC for an authorization session in a command.

```

601 static TPM2B_DIGEST *
602 ComputeCommandHMAC(
603     COMMAND          *command,           // IN: primary control structure
604     UINT32            sessionIndex,       // IN: index of session to be processed
605     TPM2B_DIGEST      *hmac              // OUT: authorization HMAC
606 )
607 {
608     TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
609     TPM2B_KEY          key;
610     BYTE               marshalBuffer[sizeof(TPMA_SESSION)];
611     BYTE               *buffer;
612     UINT32              marshalSize;
613     HMAC_STATE          hmacState;
614     TPM2B_NONCE         *nonceDecrypt;
615     TPM2B_NONCE         *nonceEncrypt;
616     SESSION             *session;
617     //
618     nonceDecrypt = NULL;
619     nonceEncrypt = NULL;
620
621     // Determine if extra nonceTPM values are going to be required.
622     // If this is the first session (sessionIndex = 0) and it is an authorization
623     // session that uses an HMAC, then check if additional session nonces are to be
624     // included.
625     if(sessionIndex == 0
626         && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
627     {
628         // If there is a decrypt session and if this is not the decrypt session,
629         // then an extra nonce may be needed.
630         if(s_decryptSessionIndex != UNDEFINED_INDEX
631             && s_decryptSessionIndex != sessionIndex)
632         {
633             // Will add the nonce for the decrypt session.
634             SESSION *decryptSession
635                 = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
636             nonceDecrypt = &decryptSession->nonceTPM;
637         }
638         // Now repeat for the encrypt session.
639         if(s_encryptSessionIndex != UNDEFINED_INDEX
640             && s_encryptSessionIndex != sessionIndex
641             && s_encryptSessionIndex != s_decryptSessionIndex)
642         {
643             // Have to have the nonce for the encrypt session.
644             SESSION *encryptSession
645                 = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
646             nonceEncrypt = &encryptSession->nonceTPM;
647         }
648     }
649     // Continue with the HMAC processing.
650     session = SessionGet(s_sessionHandles[sessionIndex]);
651
652     // Generate HMAC key.
653     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));

```



```

654
655 // Check if the session has an associated handle and if the associated entity
656 // is the one to which the session is bound. If not, add the authValue of
657 // this entity to the HMAC key.
658 // If the session is bound to the object or the session is a policy session
659 // with no authValue required, do not include the authValue in the HMAC key.
660 // Note: For a policy session, its isBound attribute is CLEARED.
661 //
662 // Include the entity authValue if it is needed
663 if(session->attributes.includeAuth == SET)
664 {
665     TPM2B_AUTH authValue;
666     // Get the entity authValue with trailing zeros removed
667     EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
668     // add the authValue to the HMAC key
669     MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
670 }
671 // if the HMAC key size is 0, a NULL string HMAC is allowed
672 if(key.t.size == 0
673    && s_inputAuthValues[sessionIndex].t.size == 0)
674 {
675     hmac->t.size = 0;
676     return hmac;
677 }
678 // Start HMAC
679 hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
680
681 // Add cpHash
682 CryptDigestUpdate2B(&hmacState.hashState,
683                    &ComputeCpHash(command, session->authHashAlg)->b);
684 // Add nonces as required
685 CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
686 CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
687 if(nonceDecrypt != NULL)
688     CryptDigestUpdate2B(&hmacState.hashState, &nonceDecrypt->b);
689 if(nonceEncrypt != NULL)
690     CryptDigestUpdate2B(&hmacState.hashState, &nonceEncrypt->b);
691 // Add sessionAttributes
692 buffer = marshalBuffer;
693 marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
694                                   &buffer, NULL);
695 CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
696 // Complete the HMAC computation
697 CryptHmacEnd2B(&hmacState, &hmac->b);
698
699 return hmac;
700 }

```

6.4.4.10 CheckSessionHMAC()

This function checks the HMAC of in a session. It uses ComputeCommandHMAC() to compute the expected HMAC value and then compares the result with the HMAC in the authorization session. The authorization is successful if they are the same.

If the authorizations are not the same, IncrementLockout() is called. It will return TPM_RC_AUTH_FAIL if the failure caused the *failureCount* to increment. Otherwise, it will return TPM_RC_BAD_AUTH.

Error Return	Meaning
TPM_RC_AUTH_FAIL	authorization failure caused <i>failureCount</i> increment
TPM_RC_BAD_AUTH	authorization failure did not cause <i>failureCount</i> increment

701 static TPM_RC

```

702 CheckSessionHMAC (
703     COMMAND      *command,          // IN: primary control structure
704     UINT32        sessionIndex      // IN: index of session to be processed
705 )
706 {
707     TPM2B_DIGEST    hmac;           // authHMAC for comparing
708     //
709     // Compute authHMAC
710     ComputeCommandHMAC(command, sessionIndex, &hmac);
711
712     // Compare the input HMAC with the authHMAC computed above.
713     if(!MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &hmac.b))
714     {
715         // If an HMAC session has a failure, invoke the anti-hammering
716         // if it applies to the authorized entity or the session.
717         // Otherwise, just indicate that the authorization is bad.
718         return IncrementLockout(sessionIndex);
719     }
720     return TPM_RC_SUCCESS;
721 }

```

6.4.4.11 CheckPolicyAuthSession()

This function is used to validate the authorization in a policy session. This function performs the following comparisons to see if a policy authorization is properly provided. The check are:

- compare *policyDigest* in session with *authPolicy* associated with the entity to be authorized;
- compare timeout if applicable;
- compare *commandCode* if applicable;
- compare *cpHash* if applicable; and
- see if PCR values have changed since computed.

If all the above checks succeed, the handle is authorized. The order of these comparisons is not important because any failure will result in the same error code.

Error Return	Meaning
TPM_RC_PCR_CHANGED	PCR value is not current
TPM_RC_POLICY_FAIL	policy session fails
TPM_RC_LOCALITY	command locality is not allowed
TPM_RC_POLICY_CC	CC doesn't match
TPM_RC_EXPIRED	policy session has expired
TPM_RC_PP	PP is required but not asserted
TPM_RC_NV_UNAVAILABLE	NV is not available for write
TPM_RC_NV_RATE	NV is rate limiting

```

722 static TPM_RC
723 CheckPolicyAuthSession (
724     COMMAND      *command,          // IN: primary parsing structure
725     UINT32        sessionIndex      // IN: index of session to be processed
726 )
727 {
728     SESSION        *session;
729     TPM2B_DIGEST    authPolicy;
730     TPMT_ALG_HASH    policyAlg;
731     UINT8           locality;

```

```

732 //
733 // Initialize pointer to the authorization session.
734 session = SessionGet(s_sessionHandles[sessionIndex]);
735
736 // If the command is TPM2_PolicySecret(), make sure that
737 // either password or authValue is required
738 if(command->code == TPM_CC_PolicySecret
739    && session->attributes.isPasswordNeeded == CLEAR
740    && session->attributes.isAuthValueNeeded == CLEAR)
741     return TPM_RC_MODE;
742 // See if the PCR counter for the session is still valid.
743 if(!SessionPCRValueIsCurrent(session))
744     return TPM_RC_PCR_CHANGED;
745 // Get authPolicy.
746 policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
747                                &authPolicy);
748 // Compare authPolicy.
749 if(!MemoryEqual2B(&session->u2.policyDigest.b, &authPolicy.b))
750     return TPM_RC_POLICY_FAIL;
751 // Policy is OK so check if the other factors are correct
752
753 // Compare policy hash algorithm.
754 if(policyAlg != session->authHashAlg)
755     return TPM_RC_POLICY_FAIL;
756
757 // Compare timeout.
758 if(session->timeout != 0)
759 {
760     // Cannot compare time if clock stop advancing. An TPM_RC_NV_UNAVAILABLE
761     // or TPM_RC_NV_RATE error may be returned here. This doesn't mean that
762     // a new nonce will be created just that, because TPM time can't advance
763     // we can't do time-based operations.
764     RETURN_IF_NV_IS_NOT_AVAILABLE;
765
766     if((session->timeout < g_time)
767        || (session->epoch != g_timeEpoch))
768         return TPM_RC_EXPIRED;
769 }
770 // If command code is provided it must match
771 if(session->commandCode != 0)
772 {
773     if(session->commandCode != command->code)
774         return TPM_RC_POLICY_CC;
775 }
776 else
777 {
778     // If command requires a DUP or ADMIN authorization, the session must have
779     // command code set.
780     AUTH_ROLE role = CommandAuthRole(command->index, sessionIndex);
781     if(role == AUTH_ADMIN || role == AUTH_DUP)
782         return TPM_RC_POLICY_FAIL;
783 }
784 // Check command locality.
785 {
786     BYTE sessionLocality[sizeof(TPMA_LOCALITY)];
787     BYTE *buffer = sessionLocality;
788
789     // Get existing locality setting in canonical form
790     sessionLocality[0] = 0;
791     TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
792
793     // See if the locality has been set
794     if(sessionLocality[0] != 0)
795     {
796         // If so, get the current locality
797         locality = _plat__LocalityGet();

```

```

798         if(locality < 5)
799         {
800             if(((sessionLocality[0] & (1 << locality)) == 0)
801                || sessionLocality[0] > 31)
802                 return TPM_RC_LOCALITY;
803         }
804         else if(locality > 31)
805         {
806             if(sessionLocality[0] != locality)
807                 return TPM_RC_LOCALITY;
808         }
809         else
810         {
811             // Could throw an assert here but a locality error is just
812             // as good. It just means that, whatever the locality is, it isn't
813             // the locality requested so...
814             return TPM_RC_LOCALITY;
815         }
816     }
817 } // end of locality check
818 // Check physical presence.
819 if(session->attributes.isPPRequired == SET
820    && !_plat_PhysicalPresenceAsserted())
821     return TPM_RC_PP;
822 // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
823 // DUP role for this handle.
824 if(session->ul.cpHash.b.size != 0)
825 {
826     BOOL        OK;
827     if(session->attributes.isCpHashDefined)
828         // Compare cpHash.
829         OK = MemoryEqual2B(&session->ul.cpHash.b,
830                           &ComputeCpHash(command, session->authHashAlg)->b);
831     else if(session->attributes.isTemplateSet)
832         OK = CompareTemplateHash(command, session);
833     else
834         OK = CompareNameHash(command, session);
835     if(!OK)
836         return TPM_RCS_POLICY_FAIL;
837 }
838 if(session->attributes.checkNvWritten)
839 {
840     NV_REF        locator;
841     NV_INDEX       *nvIndex;
842     //
843     // If this is not an NV index, the policy makes no sense so fail it.
844     if(HandleGetType(s_associatedHandles[sessionIndex]) != TPM_HT_NV_INDEX)
845         return TPM_RC_POLICY_FAIL;
846     // Get the index data
847     nvIndex = NvGetIndexInfo(s_associatedHandles[sessionIndex], &locator);
848
849     // Make sure that the TPMA WRITTEN ATTRIBUTE has the desired state
850     if((IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
851        != (session->attributes.nvWrittenState == SET))
852         return TPM_RC_POLICY_FAIL;
853 }
854 return TPM_RC_SUCCESS;
855 }

```

6.4.4.12 RetrieveSessionData()

This function will unmarshal the sessions in the session area of a command. The values are placed in the arrays that are defined at the beginning of this file. The normal unmarshaling errors are possible.

Error Return	Meaning
TPM_RC_SUCCSS	unmarshaled without error
TPM_RC_SIZE	the number of bytes unmarshaled is not the same as the value for <i>authorizationSize</i> in the command

```

856 static TPM_RC
857 RetrieveSessionData(
858     COMMAND      *command      // IN: main parsing structure for command
859 )
860 {
861     int          i;
862     TPM_RC       result;
863     SESSION      *session;
864     TPMA_SESSION sessionAttributes;
865     TPM_HT       sessionType;
866     INT32        sessionIndex;
867     TPM_RC       errorIndex;
868     //
869     s_decryptSessionIndex = UNDEFINED_INDEX;
870     s_encryptSessionIndex = UNDEFINED_INDEX;
871     s_auditSessionIndex = UNDEFINED_INDEX;
872
873     for(sessionIndex = 0; command->authSize > 0; sessionIndex++)
874     {
875         errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
876
877         // If maximum allowed number of sessions has been parsed, return a size
878         // error with a session number that is larger than the number of allowed
879         // sessions
880         if(sessionIndex == MAX_SESSION_NUM)
881             return TPM_RC_SIZE + errorIndex;
882         // make sure that the associated handle for each session starts out
883         // unassigned
884         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
885
886         // First parameter: Session handle.
887         result = TPMT_SH_AUTH_SESSION_Unmarshal(
888             &s_sessionHandles[sessionIndex],
889             &command->parameterBuffer,
890             &command->authSize, TRUE);
891         if(result != TPM_RC_SUCCESS)
892             return result + TPM_RC_S + g_rcIndex[sessionIndex];
893         // Second parameter: Nonce.
894         result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
895             &command->parameterBuffer,
896             &command->authSize);
897         if(result != TPM_RC_SUCCESS)
898             return result + TPM_RC_S + g_rcIndex[sessionIndex];
899         // Third parameter: sessionAttributes.
900         result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
901             &command->parameterBuffer,
902             &command->authSize);
903         if(result != TPM_RC_SUCCESS)
904             return result + TPM_RC_S + g_rcIndex[sessionIndex];
905         // Fourth parameter: authValue (PW or HMAC).
906         result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
907             &command->parameterBuffer,
908             &command->authSize);
909         if(result != TPM_RC_SUCCESS)
910             return result + errorIndex;
911
912         sessionAttributes = s_attributes[sessionIndex];
913         if(s_sessionHandles[sessionIndex] == TPM_RS_PW)

```

```

914     {
915         // A PWAP session needs additional processing.
916         // Can't have any attributes set other than continueSession bit
917         if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt)
918            || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt)
919            || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit)
920            || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
921            || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset))
922             return TPM_RCS_ATTRIBUTES + errorIndex;
923         // The nonce size must be zero.
924         if(s_nonceCaller[sessionIndex].t.size != 0)
925             return TPM_RCS_NONCE + errorIndex;
926         continue;
927     }
928     // For not password sessions...
929     // Find out if the session is loaded.
930     if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
931         return TPM_RC_REFERENCE_S0 + sessionIndex;
932     sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
933     session = SessionGet(s_sessionHandles[sessionIndex]);
934
935     // Check if the session is an HMAC/policy session.
936     if((session->attributes.isPolicy == SET
937        && sessionType == TPM_HT_HMAC_SESSION)
938        || (session->attributes.isPolicy == CLEAR
939        && sessionType == TPM_HT_POLICY_SESSION))
940         return TPM_RCS_HANDLE + errorIndex;
941     // Check that this handle has not previously been used.
942     for(i = 0; i < sessionIndex; i++)
943     {
944         if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
945             return TPM_RCS_HANDLE + errorIndex;
946     }
947     // If the session is used for parameter encryption or audit as well, set
948     // the corresponding indexes.
949
950     // First process decrypt.
951     if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt))
952     {
953         // Check if the commandCode allows command parameter encryption.
954         if(DecryptSize(command->index) == 0)
955             return TPM_RCS_ATTRIBUTES + errorIndex;
956         // Encrypt attribute can only appear in one session
957         if(s_decryptSessionIndex != UNDEFINED_INDEX)
958             return TPM_RCS_ATTRIBUTES + errorIndex;
959         // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
960         if(session->symmetric.algorithm == TPM_ALG_NULL)
961             return TPM_RCS_SYMMETRIC + errorIndex;
962         // All checks passed, so set the index for the session used to decrypt
963         // a command parameter.
964         s_decryptSessionIndex = sessionIndex;
965     }
966     // Now process encrypt.
967     if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt))
968     {
969         // Check if the commandCode allows response parameter encryption.
970         if(EncryptSize(command->index) == 0)
971             return TPM_RCS_ATTRIBUTES + errorIndex;
972         // Encrypt attribute can only appear in one session.
973         if(s_encryptSessionIndex != UNDEFINED_INDEX)
974             return TPM_RCS_ATTRIBUTES + errorIndex;
975         // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
976         if(session->symmetric.algorithm == TPM_ALG_NULL)
977             return TPM_RCS_SYMMETRIC + errorIndex;
978         // All checks passed, so set the index for the session used to encrypt
979         // a response parameter.

```



```

980         s_encryptSessionIndex = sessionIndex;
981     }
982     // At last process audit.
983     if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit))
984     {
985         // Audit attribute can only appear in one session.
986         if(s_auditSessionIndex != UNDEFINED_INDEX)
987             return TPM_RC_ATTRIBUTES + errorIndex;
988         // An audit session can not be policy session.
989         if(HandleGetType(s_sessionHandles[sessionIndex])
990            == TPM_HT_POLICY_SESSION)
991             return TPM_RC_ATTRIBUTES + errorIndex;
992         // If this is a reset of the audit session, or the first use
993         // of the session as an audit session, it doesn't matter what
994         // the exclusive state is. The session will become exclusive.
995         if(!IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset)
996            && session->attributes.isAudit == SET)
997         {
998             // Not first use or reset. If auditExclusive is SET, then this
999             // session must be the current exclusive session.
1000             if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
1001                && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
1002                 return TPM_RC_EXCLUSIVE;
1003         }
1004         s_auditSessionIndex = sessionIndex;
1005     }
1006     // Initialize associated handle as undefined. This will be changed when
1007     // the handles are processed.
1008     s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
1009 }
1010 command->sessionNum = sessionIndex;
1011 return TPM_RC_SUCCESS;
1012 }

```

6.4.4.13 CheckLockedOut()

This function checks to see if the TPM is in lockout. This function should only be called if the entity being checked is subject to DA protection. The TPM is in lockout if the NV is not available and a DA write is pending. Otherwise the TPM is locked out if checking for *lockoutAuth* (*lockoutAuthCheck* == TRUE) and use of *lockoutAuth* is disabled, or *failedTries* >= *maxTries*

Error Return	Meaning
TPM_RC_NV_RATE	NV is rate limiting
TPM_RC_NV_UNAVAILABLE	NV is not available at this time
TPM_RC_LOCKOUT	TPM is in lockout

```

1013 static TPM_RC
1014 CheckLockedOut(
1015     BOOL                lockoutAuthCheck    // IN: TRUE if checking is for lockoutAuth
1016 )
1017 {
1018     // If NV is unavailable, and current cycle state recorded in NV is not
1019     // SU_NONE_VALUE, refuse to check any authorization because we would
1020     // not be able to handle a DA failure.
1021     if(!NV_IS_AVAILABLE && NV_IS_ORDERLY)
1022         return g_NvStatus;
1023     // Check if DA info needs to be updated in NV.
1024     if(s_DAPendingOnNV)
1025     {
1026         // If NV is accessible,
1027         RETURN_IF_NV_IS_NOT_AVAILABLE;

```



```

1028
1029     // ... write the pending DA data and proceed.
1030     NV_SYNC_PERSISTENT(lockOutAuthEnabled);
1031     NV_SYNC_PERSISTENT(failedTries);
1032     s_DAPendingOnNV = FALSE;
1033 }
1034 // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
1035 // is disabled...
1036 if(lockoutAuthCheck)
1037 {
1038     if(gp.lockOutAuthEnabled == FALSE)
1039         return TPM_RC_LOCKOUT;
1040 }
1041 else
1042 {
1043     // ... or if the number of failed tries has been maxed out.
1044     if(gp.failedTries >= gp.maxTries)
1045         return TPM_RC_LOCKOUT;
1046 #if USE_DA_USED
1047     // If the daUsed flag is not SET, then no DA validation until the
1048     // daUsed state is written to NV
1049     if(!g_daUsed)
1050     {
1051         RETURN_IF_NV_IS_NOT_AVAILABLE;
1052         g_daUsed = TRUE;
1053         gp.orderlyState = SU_DA_USED_VALUE;
1054         NV_SYNC_PERSISTENT(orderlyState);
1055         return TPM_RC_RETRY;
1056     }
1057 #endif
1058 }
1059 return TPM_RC_SUCCESS;
1060 }

```

6.4.4.14 CheckAuthSession()

This function checks that the authorization session properly authorizes the use of the associated handle.

Error Return	Meaning
TPM_RC_LOCKOUT	entity is protected by DA and TPM is in lockout, or TPM is locked out on NV update pending on DA parameters
TPM_RC_PP	Physical Presence is required but not provided
TPM_RC_AUTH_FAIL	HMAC or PW authorization failed with DA side-effects (can be a policy session)
TPM_RC_BAD_AUTH	HMAC or PW authorization failed without DA side-effects (can be a policy session)
TPM_RC_POLICY_FAIL	if policy session fails
TPM_RC_POLICY_CC	command code of policy was wrong
TPM_RC_EXPIRED	the policy session has expired
TPM_RC_PCR	
TPM_RC_AUTH_UNAVAILABLE	authValue or authPolicy unavailable

```

1061 static TPM_RC
1062 CheckAuthSession(
1063     COMMAND      *command,          // IN: primary parsing structure
1064     UINT32       sessionIndex      // IN: index of session to be processed
1065 )
1066 {

```

```

1067     TPM_RC          result = TPM_RC_SUCCESS;
1068     SESSION         *session = NULL;
1069     TPM_HANDLE       sessionHandle = s_sessionHandles[sessionIndex];
1070     TPM_HANDLE       associatedHandle = s_associatedHandles[sessionIndex];
1071     TPM_HT           sessionHandleType = HandleGetType(sessionHandle);
1072     BOOL             authUsed;
1073     //
1074     pAssert(sessionHandle != TPM_RH_UNASSIGNED);
1075
1076     // Take care of physical presence
1077     if(associatedHandle == TPM_RH_PLATFORM)
1078     {
1079         // If the physical presence is required for this command, check for PP
1080         // assertion. If it isn't asserted, no point going any further.
1081         if(PhysicalPresenceIsRequired(command->index)
1082            && !_plat_PhysicalPresenceAsserted())
1083             return TPM_RC_PP;
1084     }
1085     if(sessionHandle != TPM_RS_PW)
1086     {
1087         session = SessionGet(sessionHandle);
1088
1089         // Set includeAuth to indicate if DA checking will be required and if the
1090         // authValue will be included in any HMAC.
1091         if(sessionHandleType == TPM_HT_POLICY_SESSION)
1092         {
1093             // For a policy session, will check the DA status of the entity if either
1094             // isAuthValueNeeded or isPasswordNeeded is SET.
1095             session->attributes.includeAuth =
1096                 session->attributes.isAuthValueNeeded
1097                 || session->attributes.isPasswordNeeded;
1098         }
1099         else
1100         {
1101             // For an HMAC session, need to check unless the session
1102             // is bound.
1103             session->attributes.includeAuth =
1104                 !IsSessionBindEntity(s_associatedHandles[sessionIndex], session);
1105         }
1106         authUsed = session->attributes.includeAuth;
1107     }
1108     else
1109         // Password session
1110         authUsed = TRUE;
1111     // If the authorization session is going to use an authValue, then make sure
1112     // that access to that authValue isn't locked out.
1113     if(authUsed)
1114     {
1115         // See if entity is subject to lockout.
1116         if(!IsDAExempted(associatedHandle))
1117         {
1118             // See if in lockout
1119             result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
1120             if(result != TPM_RC_SUCCESS)
1121                 return result;
1122         }
1123     }
1124     // Policy or HMAC+PW?
1125     if(sessionHandleType != TPM_HT_POLICY_SESSION)
1126     {
1127         // for non-policy session make sure that a policy session is not required
1128         if(IsPolicySessionRequired(command->index, sessionIndex))
1129             return TPM_RC_AUTH_TYPE;
1130         // The authValue must be available.
1131         // Note: The authValue is going to be "used" even if it is an EmptyAuth.
1132         // and the session is bound.

```

```

1133         if(!IsAuthValueAvailable(associatedHandle, command->index, sessionIndex))
1134             return TPM_RC_AUTH_UNAVAILABLE;
1135     }
1136     else
1137     {
1138         // ... see if the entity has a policy, ...
1139         // Note: IsAuthPolicyAvailable will return FALSE if the sensitive area of the
1140         // object is not loaded
1141         if(!IsAuthPolicyAvailable(associatedHandle, command->index, sessionIndex))
1142             return TPM_RC_AUTH_UNAVAILABLE;
1143         // ... and check the policy session.
1144         result = CheckPolicyAuthSession(command, sessionIndex);
1145         if(result != TPM_RC_SUCCESS)
1146             return result;
1147     }
1148     // Check authorization according to the type
1149     if((TPM_RS_PW == sessionHandle) || (session->attributes.isPasswordNeeded == SET))
1150         result = CheckPWAuthSession(sessionIndex);
1151     else
1152         result = CheckSessionHMAC(command, sessionIndex);
1153     // Do processing for PIN Indexes are only three possibilities for 'result' at
1154     // this point: TPM_RC_SUCCESS, TPM_RC_AUTH_FAIL, and TPM_RC_BAD_AUTH.
1155     // For all these cases, we would have to process a PIN index if the
1156     // authValue of the index was used for authorization.
1157     if((TPM_HT_NV_INDEX == HandleGetType(associatedHandle)) && authUsed)
1158     {
1159         NV_REF          locator;
1160         NV_INDEX         *nvIndex = NvGetIndexInfo(associatedHandle, &locator);
1161         NV_PIN           pinData;
1162         TPMA_NV          nvAttributes;
1163     //
1164         pAssert(nvIndex != NULL);
1165         nvAttributes = nvIndex->publicArea.attributes;
1166         // If this is a PIN FAIL index and the value has been written
1167         // then we can update the counter (increment or clear)
1168         if(IsNvPinFailIndex(nvAttributes)
1169             && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))
1170         {
1171             pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1172             if(result != TPM_RC_SUCCESS)
1173                 pinData.pin.pinCount++;
1174             else
1175                 pinData.pin.pinCount = 0;
1176             NvWriteUINT64Data(nvIndex, pinData.intVal);
1177         }
1178         // If this is a PIN PASS Index, increment if we have used the
1179         // authorization value.
1180         // NOTE: If the counter has already hit the limit, then we
1181         // would not get here because the authorization value would not
1182         // be available and the TPM would have returned before it gets here
1183         else if(IsNvPinPassIndex(nvAttributes)
1184             && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN)
1185             && result == TPM_RC_SUCCESS)
1186         {
1187             // If the access is valid, then increment the use counter
1188             pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1189             pinData.pin.pinCount++;
1190             NvWriteUINT64Data(nvIndex, pinData.intVal);
1191         }
1192     }
1193     return result;
1194 }
1195 #ifdef TPM_CC_GetCommandAuditDigest

```

6.4.4.15 CheckCommandAudit()

This function is called before the command is processed if audit is enabled for the command. It will check to see if the audit can be performed and will ensure that the *cpHash* is available for the audit.

Error Return	Meaning
TPM_RC_NV_UNAVAILABLE	NV is not available for write
TPM_RC_NV_RATE	NV is rate limiting

```

1196 static TPM_RC
1197 CheckCommandAudit(
1198     COMMAND      *command
1199 )
1200 {
1201     // If the audit digest is clear and command audit is required, NV must be
1202     // available so that TPM2_GetCommandAuditDigest() is able to increment
1203     // audit counter. If NV is not available, the function bails out to prevent
1204     // the TPM from attempting an operation that would fail anyway.
1205     if(gr.commandAuditDigest.t.size == 0
1206        || GetCommandCode(command->index) == TPM_CC_GetCommandAuditDigest)
1207     {
1208         RETURN_IF_NV_IS_NOT_AVAILABLE;
1209     }
1210     // Make sure that the cpHash is computed for the algorithm
1211     ComputeCpHash(command, gp.auditHashAlg);
1212     return TPM_RC_SUCCESS;
1213 }
1214 #endif

```

6.4.4.16 ParseSessionBuffer()

This function is the entry function for command session processing. It iterates sessions in session area and reports if the required authorization has been properly provided. It also processes audit session and passes the information of encryption sessions to parameter encryption module.

Error Return	Meaning
various	parsing failure or authorization failure

```

1215 TPM_RC
1216 ParseSessionBuffer(
1217     COMMAND      *command      // IN: the structure that contains
1218 )
1219 {
1220     TPM_RC      result;
1221     UINT32      i;
1222     INT32       size = 0;
1223     TPM2B_AUTH  extraKey;
1224     UINT32      sessionIndex;
1225     TPM_RC      errorIndex;
1226     SESSION     *session = NULL;
1227     //
1228     // Check if a command allows any session in its session area.
1229     if(!IsSessionAllowed(command->index))
1230         return TPM_RC_AUTH_CONTEXT;
1231     // Default-initialization.
1232     command->sessionNum = 0;
1233
1234     result = RetrieveSessionData(command);
1235     if(result != TPM_RC_SUCCESS)
1236         return result;

```

```

1237 // There is no command in the TPM spec that has more handles than
1238 // MAX_SESSION_NUM.
1239 pAssert(command->handleNum <= MAX_SESSION_NUM);
1240
1241 // Associate the session with an authorization handle.
1242 for(i = 0; i < command->handleNum; i++)
1243 {
1244     if(CommandAuthRole(command->index, i) != AUTH_NONE)
1245     {
1246         // If the received session number is less than the number of handles
1247         // that requires authorization, an error should be returned.
1248         // Note: for all the TPM 2.0 commands, handles requiring
1249         // authorization come first in a command input and there are only ever
1250         // two values requiring authorization
1251         if(i > (command->sessionNum - 1))
1252             return TPM_RC_AUTH_MISSING;
1253         // Record the handle associated with the authorization session
1254         s_associatedHandles[i] = command->handles[i];
1255     }
1256 }
1257 // Consistency checks are done first to avoid authorization failure when the
1258 // command will not be executed anyway.
1259 for(sessionIndex = 0; sessionIndex < command->sessionNum; sessionIndex++)
1260 {
1261     errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
1262     // PW session must be an authorization session
1263     if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1264     {
1265         if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1266             return TPM_RCS_HANDLE + errorIndex;
1267         // a password session can't be audit, encrypt or decrypt
1268         if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1269            || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1270            || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1271             return TPM_RCS_ATTRIBUTES + errorIndex;
1272         session = NULL;
1273     }
1274     else
1275     {
1276         session = SessionGet(s_sessionHandles[sessionIndex]);
1277
1278         // A trial session can not appear in session area, because it cannot
1279         // be used for authorization, audit or encrypt/decrypt.
1280         if(session->attributes.isTrialPolicy == SET)
1281             return TPM_RCS_ATTRIBUTES + errorIndex;
1282
1283         // See if the session is bound to a DA protected entity
1284         // NOTE: Since a policy session is never bound, a policy is still
1285         // usable even if the object is DA protected and the TPM is in
1286         // lockout.
1287         if(session->attributes.isDaBound == SET)
1288         {
1289             result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1290             if(result != TPM_RC_SUCCESS)
1291                 return result;
1292         }
1293         // If this session is for auditing, make sure the cpHash is computed.
1294         if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit))
1295             ComputeCpHash(command, session->authHashAlg);
1296     }
1297     // if the session has an associated handle, check the authorization
1298     if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1299     {
1300         result = CheckAuthSession(command, sessionIndex);
1301         if(result != TPM_RC_SUCCESS)
1302             return RcSafeAddToResult(result, errorIndex);
1303     }

```

```

1303     }
1304     else
1305     {
1306         // a session that is not for authorization must either be encrypt,
1307         // decrypt, or audit
1308         if(!IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1309            && !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1310            && !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1311             return TPM_RCS_ATTRIBUTES + errorIndex;
1312
1313         // no authValue included in any of the HMAC computations
1314         pAssert(session != NULL);
1315         session->attributes.includeAuth = CLEAR;
1316
1317         // check HMAC for encrypt/decrypt/audit only sessions
1318         result = CheckSessionHMAC(command, sessionIndex);
1319         if(result != TPM_RC_SUCCESS)
1320             return RcSafeAddToResult(result, errorIndex);
1321     }
1322 }
1323 #ifdef TPM_CC_GetCommandAuditDigest
1324 // Check if the command should be audited. Need to do this before any parameter
1325 // encryption so that the cpHash for the audit is correct
1326 if(CommandAuditIsRequired(command->index))
1327 {
1328     result = CheckCommandAudit(command);
1329     if(result != TPM_RC_SUCCESS)
1330         return result; // No session number to reference
1331 }
1332 #endif
1333 // Decrypt the first parameter if applicable. This should be the last operation
1334 // in session processing.
1335 // If the encrypt session is associated with a handle and the handle's
1336 // authValue is available, then authValue is concatenated with sessionKey to
1337 // generate encryption key, no matter if the handle is the session bound entity
1338 // or not.
1339 if(s_decryptSessionIndex != UNDEFINED_INDEX)
1340 {
1341     // If this is an authorization session, include the authValue in the
1342     // generation of the decryption key
1343     if(s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED)
1344     {
1345         EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1346                           &extraKey);
1347     }
1348     else
1349     {
1350         extraKey.b.size = 0;
1351     }
1352     size = DecryptSize(command->index);
1353     result = CryptParameterDecryption(s_sessionHandles[s_decryptSessionIndex],
1354                                     &s_nonceCaller[s_decryptSessionIndex].b,
1355                                     command->parameterSize, (UINT16)size,
1356                                     &extraKey,
1357                                     command->parameterBuffer);
1358     if(result != TPM_RC_SUCCESS)
1359         return RcSafeAddToResult(result,
1360                                   TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1361 }
1362 return TPM_RC_SUCCESS;
1363 }

```


6.4.4.17 CheckAuthNoSession()

Function to process a command with no session associated. The function makes sure all the handles in the command require no authorization.

Error Return	Meaning
TPM_RC_AUTH_MISSING	failure - one or more handles require authorization

```

1364 TPM_RC
1365 CheckAuthNoSession(
1366     COMMAND      *command          // IN: command parsing structure
1367 )
1368 {
1369     UINT32 i;
1370     TPM_RC      result = TPM_RC_SUCCESS;
1371     //
1372     // Check if the command requires authorization
1373     for(i = 0; i < command->handleNum; i++)
1374     {
1375         if(CommandAuthRole(command->index, i) != AUTH_NONE)
1376             return TPM_RC_AUTH_MISSING;
1377     }
1378 #ifndef TPM_CC_GetCommandAuditDigest
1379     // Check if the command should be audited.
1380     if(CommandAuditIsRequired(command->index))
1381     {
1382         result = CheckCommandAudit(command);
1383         if(result != TPM_RC_SUCCESS)
1384             return result;
1385     }
1386 #endif
1387     // Initialize number of sessions to be 0
1388     command->sessionNum = 0;
1389     return TPM_RC_SUCCESS;
1390 }
1391
```

6.4.5 Response Session Processing

6.4.5.1 Introduction

The following functions build the session area in a response and handle the audit sessions (if present).

6.4.5.2 ComputeRpHash()

Function to compute *rpHash* (Response Parameter Hash). The *rpHash* is only computed if there is an HMAC authorization session and the return code is TPM_RC_SUCCESS.

```

1392 static TPM2B_DIGEST *
1393 ComputeRpHash(
1394     COMMAND      *command,          // IN: command structure
1395     TPM_ALG_ID    hashAlg          // IN: hash algorithm to compute rpHash
1396 )
1397 {
1398     TPM2B_DIGEST *rpHash = GetRpHashPointer(command, hashAlg);
1399     HASH_STATE    hashState;
1400     //
1401     if(rpHash->t.size == 0)
1402     {
1403         // rpHash := hash(responseCode || commandCode || parameters)
1404     }
1405
```



```

1404
1405     // Initiate hash creation.
1406     rpHash->t.size = CryptHashStart(&hashState, hashAlg);
1407
1408     // Add hash constituents.
1409     CryptDigestUpdateInt(&hashState, sizeof(TPM_RC), TPM_RC_SUCCESS);
1410     CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
1411     CryptDigestUpdate(&hashState, command->parameterSize,
1412                      command->parameterBuffer);
1413     // Complete hash computation.
1414     CryptHashEnd2B(&hashState, &rpHash->b);
1415 }
1416 return rpHash;
1417 }

```

6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```

1418 static void
1419 InitAuditSession(
1420     SESSION      *session      // session to be initialized
1421 )
1422 {
1423     // Mark session as an audit session.
1424     session->attributes.isAudit = SET;
1425
1426     // Audit session can not be bound.
1427     session->attributes.isBound = CLEAR;
1428
1429     // Size of the audit log is the size of session hash algorithm digest.
1430     session->u2.auditDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
1431
1432     // Set the original digest value to be 0.
1433     MemorySet(&session->u2.auditDigest.t.buffer,
1434              0,
1435              session->u2.auditDigest.t.size);
1436     return;
1437 }

```

6.4.5.4 UpdateAuditDigest()

Function to update an audit digest

```

1438 static void
1439 UpdateAuditDigest(
1440     COMMAND      *command,
1441     TPMI_ALG_HASH hashAlg,
1442     TPM2B_DIGEST *digest
1443 )
1444 {
1445     HASH_STATE      hashState;
1446     TPM2B_DIGEST    *cpHash = GetCpHash(command, hashAlg);
1447     TPM2B_DIGEST    *rpHash = ComputeRpHash(command, hashAlg);
1448     //
1449     pAssert(cpHash != NULL);
1450
1451     // digestNew := hash (digestOld || cpHash || rpHash)
1452     // Start hash computation.
1453     digest->t.size = CryptHashStart(&hashState, hashAlg);
1454     // Add old digest.
1455     CryptDigestUpdate2B(&hashState, &digest->b);
1456     // Add cpHash

```

```

1457     CryptDigestUpdate2B(&hashState, &cpHash->b);
1458     // Add rpHash
1459     CryptDigestUpdate2B(&hashState, &rpHash->b);
1460     // Finalize the hash.
1461     CryptHashEnd2B(&hashState, &digest->b);
1462 }

```

6.4.5.5 Audit()

This function updates the audit digest in an audit session.

```

1463 static void
1464 Audit(
1465     COMMAND      *command,      // IN: primary control structure
1466     SESSION      *auditSession  // IN: loaded audit session
1467 )
1468 {
1469     UpdateAuditDigest(command, auditSession->authHashAlg,
1470                       &auditSession->u2.auditDigest);
1471     return;
1472 }
1473 #ifdef TPM_CC_GetCommandAuditDigest

```

6.4.5.6 CommandAudit()

This function updates the command audit digest.

```

1474 static void
1475 CommandAudit(
1476     COMMAND      *command      // IN:
1477 )
1478 {
1479     // If the digest.size is one, it indicates the special case of changing
1480     // the audit hash algorithm. For this case, no audit is done on exit.
1481     // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1482     // force an update to the NV on exit so that the change in digest will
1483     // be recorded. So, it is safe to exit here without setting any flags
1484     // because the digest change will be written to NV when this code exits.
1485     if(gr.commandAuditDigest.t.size == 1)
1486     {
1487         gr.commandAuditDigest.t.size = 0;
1488         return;
1489     }
1490     // If the digest size is zero, need to start a new digest and increment
1491     // the audit counter.
1492     if(gr.commandAuditDigest.t.size == 0)
1493     {
1494         gr.commandAuditDigest.t.size = CryptHashGetDigestSize(gp.auditHashAlg);
1495         MemorySet(gr.commandAuditDigest.t.buffer,
1496                 0,
1497                 gr.commandAuditDigest.t.size);
1498
1499         // Bump the counter and save its value to NV.
1500         gp.auditCounter++;
1501         NV_SYNC_PERSISTENT(auditCounter);
1502     }
1503     UpdateAuditDigest(command, gp.auditHashAlg, &gr.commandAuditDigest);
1504     return;
1505 }
1506 #endif

```

6.4.5.7 UpdateAuditSessionStatus()

This function updates the internal audit related states of a session. It will:

- a) initialize the session as audit session and set it to be exclusive if this is the first time it is used for audit or audit reset was requested;
- b) report exclusive audit session;
- c) extend audit log; and
- d) clear exclusive audit session if no audit session found in the command.

```

1507 static void
1508 UpdateAuditSessionStatus(
1509     COMMAND      *command          // IN: primary control structure
1510 )
1511 {
1512     UINT32        i;
1513     TPM_HANDLE    auditSession = TPM_RH_UNASSIGNED;
1514 //
1515 // Iterate through sessions
1516 for(i = 0; i < command->sessionNum; i++)
1517 {
1518     SESSION      *session;
1519 //
1520 // PW session do not have a loaded session and can not be an audit
1521 // session either. Skip it.
1522 if(s_sessionHandles[i] == TPM_RS_PW)
1523     continue;
1524 session = SessionGet(s_sessionHandles[i]);
1525
1526 // If a session is used for audit
1527 if(IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, audit))
1528 {
1529     // An audit session has been found
1530     auditSession = s_sessionHandles[i];
1531
1532     // If the session has not been an audit session yet, or
1533     // the auditSetting bits indicate a reset, initialize it and set
1534     // it to be the exclusive session
1535     if(session->attributes.isAudit == CLEAR
1536        || IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditReset))
1537     {
1538         InitAuditSession(session);
1539         g_exclusiveAuditSession = auditSession;
1540     }
1541     else
1542     {
1543         // Check if the audit session is the current exclusive audit
1544         // session and, if not, clear previous exclusive audit session.
1545         if(g_exclusiveAuditSession != auditSession)
1546             g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1547     }
1548     // Report audit session exclusivity.
1549     if(g_exclusiveAuditSession == auditSession)
1550     {
1551         SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1552     }
1553     else
1554     {
1555         CLEAR_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1556     }
1557     // Extend audit log.
1558     Audit(command, session);
1559 }

```

```

1560     }
1561     // If no audit session is found in the command, and the command allows
1562     // a session then, clear the current exclusive
1563     // audit session.
1564     if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(command->index))
1565     {
1566         g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1567     }
1568     return;
1569 }

```

6.4.5.8 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```

1570 static void
1571 ComputeResponseHMAC(
1572     COMMAND          *command,           // IN: command structure
1573     UINT32           sessionIndex,       // IN: session index to be processed
1574     SESSION          *session,           // IN: loaded session
1575     TPM2B_DIGEST     *hmac,              // OUT: authHMAC
1576 )
1577 {
1578     TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1579     TPM2B_KEY         key;               // HMAC key
1580     BYTE              marshalBuffer[sizeof(TPMA_SESSION)];
1581     BYTE              *buffer;
1582     UINT32            marshalSize;
1583     HMAC_STATE        hmacState;
1584     TPM2B_DIGEST     *rpHash = ComputeRpHash(command, session->authHashAlg);
1585 //
1586     // Generate HMAC key
1587     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1588 //
1589     // Add the object authValue if required
1590     if(session->attributes.includeAuth == SET)
1591     {
1592         // Note: includeAuth may be SET for a policy that is used in
1593         // UndefinedSpaceSpecial(). At this point, the Index has been deleted
1594         // so the includeAuth will have no meaning. However, the
1595         // s_associatedHandles[] value for the session is now set to TPM_RH_NULL so
1596         // this will return the authValue associated with TPM_RH_NULL and that is
1597         // and empty buffer.
1598         TPM2B_AUTH     authValue;
1599 //
1600         // Get the authValue with trailing zeros removed
1601         EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
1602 //
1603         // Add it to the key
1604         MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
1605     }
1606     // if the HMAC key size is 0, the response HMAC is computed according to the
1607     // input HMAC
1608     if(key.t.size == 0
1609        && s_inputAuthValues[sessionIndex].t.size == 0)
1610     {
1611         hmac->t.size = 0;
1612         return;
1613     }
1614     // Start HMAC computation.
1615     hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
1616 //
1617     // Add hash components.
1618     CryptDigestUpdate2B(&hmacState.hashState, &rpHash->b);

```

```

1619     CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
1620     CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
1621
1622     // Add session attributes.
1623     buffer = marshalBuffer;
1624     marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1625     CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
1626
1627     // Finalize HMAC.
1628     CryptHmacEnd2B(&hmacState, &hmac->b);
1629
1630     return;
1631 }

```

6.4.5.9 UpdateInternalSession()

This function updates internal sessions by:

- a) restarting session time; and
- b) clearing a policy session since nonce is rolling.

```

1632 static void
1633 UpdateInternalSession(
1634     SESSION      *session,      // IN: the session structure
1635     UINT32       i              // IN: session number
1636 )
1637 {
1638     // If nonce is rolling in a policy session, the policy related data
1639     // will be re-initialized.
1640     if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION
1641         && IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1642     {
1643         // When the nonce rolls it starts a new timing interval for the
1644         // policy session.
1645         SessionResetPolicyData(session);
1646         SessionSetStartTime(session);
1647     }
1648     return;
1649 }

```

6.4.5.10 BuildSingleResponseAuth()

Function to compute response HMAC value for a policy or HMAC session.

```

1650 static TPM2B_NONCE *
1651 BuildSingleResponseAuth(
1652     COMMAND      *command,      // IN: command structure
1653     UINT32       sessionIndex,  // IN: session index to be processed
1654     TPM2B_AUTH   *auth          // OUT: authHMAC
1655 )
1656 {
1657     // Fill in policy/HMAC based session response.
1658     SESSION      *session = SessionGet(s_sessionHandles[sessionIndex]);
1659     //
1660     // If the session is a policy session with isPasswordNeeded SET, the
1661     // authorization field is empty.
1662     if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1663         && session->attributes.isPasswordNeeded == SET)
1664         auth->t.size = 0;
1665     else
1666         // Compute response HMAC.
1667         ComputeResponseHMAC(command, sessionIndex, session, auth);

```

```

1668
1669     UpdateInternalSession(session, sessionIndex);
1670     return &session->nonceTPM;
1671 }

```

6.4.5.11 UpdateAllNonceTPM()

Updates TPM nonce for all sessions in command.

```

1672 static void
1673 UpdateAllNonceTPM(
1674     COMMAND      *command          // IN: controlling structure
1675 )
1676 {
1677     UINT32      i;
1678     SESSION     *session;
1679 //
1680     for(i = 0; i < command->sessionNum; i++)
1681     {
1682         // If not a PW session, compute the new nonceTPM.
1683         if(s_sessionHandles[i] != TPM_RS_PW)
1684         {
1685             session = SessionGet(s_sessionHandles[i]);
1686             // Update nonceTPM in both internal session and response.
1687             CryptRandomGenerate(session->nonceTPM.t.size,
1688                                session->nonceTPM.t.buffer);
1689         }
1690     }
1691     return;
1692 }

```

6.4.5.12 BuildResponseSession()

Function to build Session buffer in a response. The authorization data is added to the end of *command->responseBuffer*. The size of the authorization area is accumulated in *command->authSize*. When this is called, *command->responseBuffer* is pointing at the next location in the response buffer to be filled. This is where the authorization sessions will go, if any. *command->parameterSize* is the number of bytes that have been marshaled as parameters in the output buffer.

```

1693 void
1694 BuildResponseSession(
1695     COMMAND      *command          // IN: structure that has relevant command
1696 )                                  // information
1697 {
1698     pAssert(command->authSize == 0);
1699 //
1700 // Reset the parameter buffer to point to the start of the parameters so that
1701 // there is a starting point for any rpHash that might be generated and so there
1702 // is a place where parameter encryption would start
1703 command->parameterBuffer = command->responseBuffer - command->parameterSize;
1704 //
1705 // Session nonces should be updated before parameter encryption
1706 if(command->tag == TPM_ST_SESSIONS)
1707 {
1708     UpdateAllNonceTPM(command);
1709 //
1710 // Encrypt first parameter if applicable. Parameter encryption should
1711 // happen after nonce update and before any rpHash is computed.
1712 // If the encrypt session is associated with a handle, the authValue of
1713 // this handle will be concatenated with sessionKey to generate
1714 // encryption key, no matter if the handle is the session bound entity
1715 }

```

```

1716         // or not. The authValue is added to sessionKey only when the authValue
1717         // is available.
1718         if(s_encryptSessionIndex != UNDEFINED_INDEX)
1719         {
1720             UINT32          size;
1721             TPM2B_AUTH      extraKey;
1722         //
1723             extraKey.b.size = 0;
1724             // If this is an authorization session, include the authValue in the
1725             // generation of the encryption key
1726             if(s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED)
1727             {
1728                 EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1729                                     &extraKey);
1730             }
1731             size = EncryptSize(command->index);
1732             CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1733                                     &s_nonceCaller[s_encryptSessionIndex].b,
1734                                     (UINT16)size,
1735                                     &extraKey,
1736                                     command->parameterBuffer);
1737         }
1738     }
1739     // Audit sessions should be processed regardless of the tag because
1740     // a command with no session may cause a change of the exclusivity state.
1741     UpdateAuditSessionStatus(command);
1742     #if CC_GetCommandAuditDigest
1743     // Command Audit
1744     if(CommandAuditIsRequired(command->index))
1745         CommandAudit(command);
1746     #endif
1747     // Process command with sessions.
1748     if(command->tag == TPM_ST_SESSIONS)
1749     {
1750         UINT32          i;
1751     //
1752         pAssert(command->sessionNum > 0);
1753
1754         // Iterate over each session in the command session area, and create
1755         // corresponding sessions for response.
1756         for(i = 0; i < command->sessionNum; i++)
1757         {
1758             TPM2B_NONCE    *nonceTPM;
1759             TPM2B_DIGEST    responseAuth;
1760             // Make sure that continueSession is SET on any Password session.
1761             // This makes it marginally easier for the management software
1762             // to keep track of the closed sessions.
1763             if(s_sessionHandles[i] == TPM_RS_PW)
1764             {
1765                 SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession);
1766                 responseAuth.t.size = 0;
1767                 nonceTPM = (TPM2B_NONCE *)&responseAuth;
1768             }
1769             else
1770             {
1771                 // Compute the response HMAC and get a pointer to the nonce used.
1772                 // This function will also update the values if needed. Note, the
1773                 nonceTPM = BuildSingleResponseAuth(command, i, &responseAuth);
1774             }
1775             command->authSize += TPM2B_NONCE_Marshal(nonceTPM,
1776                                                         &command->responseBuffer,
1777                                                         NULL);
1778             command->authSize += TPMA_SESSION_Marshal(&s_attributes[i],
1779                                                         &command->responseBuffer,
1780                                                         NULL);
1781             command->authSize += TPM2B_DIGEST_Marshal(&responseAuth,

```



```
1782                                     &command->responseBuffer,  
1783                                     NULL);  
1784     if(!IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))  
1785         SessionFlush(s_sessionHandles[i]);  
1786     }  
1787 }  
1788 return;  
1789 }
```

6.4.5.13 SessionRemoveAssociationToHandle()

This function deals with the case where an entity associated with an authorization is deleted during command processing. The primary use of this is to support UndefineSpaceSpecial().

```
1790 void  
1791 SessionRemoveAssociationToHandle(  
1792     TPM_HANDLE      handle  
1793 )  
1794 {  
1795     UINT32          i;  
1796     //   
1797     for(i = 0; i < MAX_SESSION_NUM; i++)  
1798     {  
1799         if(s_associatedHandles[i] == handle)  
1800         {  
1801             s_associatedHandles[i] = TPM_RH_NULL;  
1802         }  
1803     }  
1804 }
```

7 Command Support Functions

7.1 Introduction

This clause contains support routines that are called by the command action code in TPM 2.0 Part 3. The functions are grouped by the command group that is supported by the functions.

7.2 Attestation Command Support (Attest_spt.c)

7.2.1 Includes

```
1 #include "Tpm.h"
2 #include "Attest_spt_fp.h"
```

7.2.2 Functions

7.2.2.1 FillInAttestInfo()

Fill in common fields of TPMS_ATTEST structure.

```
3 void
4 FillInAttestInfo(
5     TPMI_DH_OBJECT      signHandle,      // IN: handle of signing object
6     TPMT_SIG_SCHEME      *scheme,        // IN/OUT: scheme to be used for signing
7     TPM2B_DATA           *data,          // IN: qualifying data
8     TPMS_ATTEST          *attest        // OUT: attest structure
9 )
10 {
11     OBJECT              *signObject = HandleToObject(signHandle);
12
13     // Magic number
14     attest->magic = TPM_GENERATED_VALUE;
15
16     if(signObject == NULL)
17     {
18         // The name for a null handle is TPM_RH_NULL
19         // This is defined because UINT32_TO_BYTE_ARRAY does a cast. If the
20         // size of the cast is smaller than a constant, the compiler warns
21         // about the truncation of a constant value.
22         TPM_HANDLE      nullHandle = TPM_RH_NULL;
23         attest->qualifiedSigner.t.size = sizeof(TPM_HANDLE);
24         UINT32_TO_BYTE_ARRAY(nullHandle, attest->qualifiedSigner.t.name);
25     }
26     else
27     {
28         // Certifying object qualified name
29         // if the scheme is anonymous, this is an empty buffer
30         if(CryptIsSchemeAnonymous(scheme->scheme))
31             attest->qualifiedSigner.t.size = 0;
32         else
33             attest->qualifiedSigner = signObject->qualifiedName;
34     }
35     // current clock in plain text
36     TimeFillInfo(&attest->clockInfo);
37
38     // Firmware version in plain text
39     attest->firmwareVersion = ((UINT64)gp.firmwareV1 << (sizeof(UINT32) * 8));
40     attest->firmwareVersion += gp.firmwareV2;
41 }
```

```

42 // Check the hierarchy of sign object. For NULL sign handle, the hierarchy
43 // will be TPM_RH_NULL
44 if((signObject == NULL)
45    || (!signObject->attributes.epsHierarchy
46        && !signObject->attributes.ppsHierarchy))
47 {
48     // For signing key that is not in platform or endorsement hierarchy,
49     // obfuscate the reset, restart and firmware version information
50     UINT64 obfuscation[2];
51     CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gp.shProof.b, OBFUSCATE_STRING,
52               &attest->qualifiedSigner.b, NULL, 128,
53               (BYTE *)&obfuscation[0], NULL, FALSE);
54     // Obfuscate data
55     attest->firmwareVersion += obfuscation[0];
56     attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
57     attest->clockInfo.restartCount += (UINT32)obfuscation[1];
58 }
59 // External data
60 if(CryptIsSchemeAnonymous(scheme->scheme))
61     attest->extraData.t.size = 0;
62 else
63 {
64     // If we move the data to the attestation structure, then it is not
65     // used in the signing operation except as part of the signed data
66     attest->extraData = *data;
67     data->t.size = 0;
68 }
69 }

```

7.2.2.2 SignAttestInfo()

Sign a TPMS_ATTEST structure. If *signHandle* is TPM_RH_NULL, a null signature is returned.

Error Return	Meaning
TPM_RC_ATTRIBUTES	<i>signHandle</i> references not a signing key
TPM_RC_SCHEME	<i>scheme</i> is not compatible with <i>signHandle</i> type
TPM_RC_VALUE	digest generated for the given <i>scheme</i> is greater than the modulus of <i>signHandle</i> (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```

70 TPM_RC
71 SignAttestInfo(
72     OBJECT          *signKey,           // IN: sign object
73     TPMT_SIG_SCHEME *scheme,           // IN: sign scheme
74     TPMS_ATTEST     *certifyInfo,       // IN: the data to be signed
75     TPM2B_DATA       *qualifyingData,   // IN: extra data for the signing
76                                     // process
77     TPM2B_ATTEST     *attest,           // OUT: marshaled attest blob to be
78                                     // signed
79     TPMT_SIGNATURE   *signature        // OUT: signature
80 )
81 {
82     BYTE          *buffer;
83     HASH_STATE     hashState;
84     TPM2B_DIGEST   digest;
85     TPM_RC         result;
86
87     // Marshal TPMS_ATTEST structure for hash
88     buffer = attest->t.attestationData;
89     attest->t.size = TPMS_ATTEST_Marshal(certifyInfo, &buffer, NULL);
90
91     if(signKey == NULL)

```

```

92     {
93         signature->sigAlg = TPM_ALG_NULL;
94         result = TPM_RC_SUCCESS;
95     }
96     else
97     {
98         TPMI_ALG_HASH      hashAlg;
99         // Compute hash
100        hashAlg = scheme->details.any.hashAlg;
101        // need to set the receive buffer to get something put in it
102        digest.t.size = sizeof(digest.t.buffer);
103        digest.t.size = CryptHashBlock(hashAlg, attest->t.size,
104                                       attest->t.attestationData,
105                                       digest.t.size, digest.t.buffer);
106        // If there is qualifying data, need to rehash the data
107        // hash(qualifyingData || hash(attestationData))
108        if(qualifyingData->t.size != 0)
109        {
110            CryptHashStart(&hashState, hashAlg);
111            CryptDigestUpdate2B(&hashState, &qualifyingData->b);
112            CryptDigestUpdate2B(&hashState, &digest.b);
113            CryptHashEnd2B(&hashState, &digest.b);
114        }
115        // Sign the hash. A TPM_RC_VALUE, TPM_RC_SCHEME, or
116        // TPM_RC_ATTRIBUTES error may be returned at this point
117        result = CryptSign(signKey, scheme, &digest, signature);
118
119        // Since the clock is used in an attestation, the state in NV is no longer
120        // "orderly" with respect to the data in RAM if the signature is valid
121        if(result == TPM_RC_SUCCESS)
122        {
123            // Command uses the clock so need to clear the orderly state if it is
124            // set.
125            result = NvClearOrderly();
126        }
127    }
128    return result;
129 }

```

7.2.2.3 IsSigningObject()

Checks to see if the object is OK for signing. This is here rather than in Object_spt.c because all the attestation commands use this file but not Object_spt.c.

Return Value	Meaning
TRUE(1)	object may sign
FALSE(0)	object may not sign

```

130  BOOL
131  IsSigningObject(
132      OBJECT      *object          // IN:
133  )
134  {
135      return ((object == NULL)
136             || ((IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, sign)
137                 && object->publicArea.type != TPM_ALG_SYMCIPHER)));
138  }

```

7.3 Context Management Command Support (Context_spt.c)

7.3.1 Includes

```
1 #include "Tpm.h"
2 #include "Context_spt_fp.h"
```

7.3.2 Functions

7.3.2.1 ComputeContextProtectionKey()

This function retrieves the symmetric protection key for context encryption. It is used by TPM2_ConextSave() and TPM2_ContextLoad() to create the symmetric encryption key and iv.

```
3 void
4 ComputeContextProtectionKey(
5     TPMS_CONTEXT *contextBlob, // IN: context blob
6     TPM2B_SYM_KEY *symKey,     // OUT: the symmetric key
7     TPM2B_IV *iv,             // OUT: the IV.
8 )
9 {
10     UINT16 symKeyBits; // number of bits in the parent's
11                        // symmetric key
12     TPM2B_PROOF *proof = NULL; // the proof value to use. Is null for
13                                // everything but a primary object in
14                                // the Endorsement Hierarchy
15
16     BYTE kdfResult[sizeof(TPMU_HA) * 2]; // Value produced by the KDF
17
18     TPM2B_DATA sequence2B, handle2B;
19
20     // Get proof value
21     proof = HierarchyGetProof(contextBlob->hierarchy);
22
23     // Get sequence value in 2B format
24     sequence2B.t.size = sizeof(contextBlob->sequence);
25     cAssert(sequence2B.t.size <= sizeof(sequence2B.t.buffer));
26     MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence, sequence2B.t.size);
27
28     // Get handle value in 2B format
29     handle2B.t.size = sizeof(contextBlob->savedHandle);
30     cAssert(handle2B.t.size <= sizeof(handle2B.t.buffer));
31     MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle, handle2B.t.size);
32
33     // Get the symmetric encryption key size
34     symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
35     symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;
36     // Get the size of the IV for the algorithm
37     iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);
38
39     // KDFa to generate symmetric key and IV value
40     CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, CONTEXT_KEY, &sequence2B.b,
41               &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL,
42               FALSE);
43
44     // Copy part of the returned value as the key
45     pAssert(symKey->t.size <= sizeof(symKey->t.buffer));
46     MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size);
47
48     // Copy the rest as the IV
49     pAssert(iv->t.size <= sizeof(iv->t.buffer));
```

```

50     MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size);
51
52     return;
53 }

```

7.3.2.2 ComputeContextIntegrity()

Generate the integrity hash for a context. It is used by TPM2_ContextSave() to create an integrity hash and by TPM2_ContextLoad() to compare an integrity hash.

```

54 void
55 ComputeContextIntegrity(
56     TPMS_CONTEXT *contextBlob, // IN: context blob
57     TPM2B_DIGEST *integrity    // OUT: integrity
58 )
59 {
60     HMAC_STATE      hmacState;
61     TPM2B_PROOF     *proof;
62     UINT16          integritySize;
63
64     // Get proof value
65     proof = HierarchyGetProof(contextBlob->hierarchy);
66
67     // Start HMAC
68     integrity->t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
69                                         &proof->b);
70
71     // Compute integrity size at the beginning of context blob
72     integritySize = sizeof(integrity->t.size) + integrity->t.size;
73
74     // Adding total reset counter so that the context cannot be
75     // used after a TPM Reset
76     CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
77                         gp.totalResetCount);
78
79     // If this is a ST_CLEAR object, add the clear count
80     // so that this context cannot be loaded after a TPM Restart
81     if(contextBlob->savedHandle == 0x80000002)
82         CryptDigestUpdateInt(&hmacState.hashState, sizeof(gr.clearCount),
83                             gr.clearCount);
84
85     // Adding sequence number to the HMAC to make sure that it doesn't
86     // get changed
87     CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->sequence),
88                         contextBlob->sequence);
89
90     // Protect the handle
91     CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->savedHandle),
92                         contextBlob->savedHandle);
93
94     // Adding sensitive contextData, skip the leading integrity area
95     CryptDigestUpdate(&hmacState.hashState,
96                     contextBlob->contextBlob.t.size - integritySize,
97                     contextBlob->contextBlob.t.buffer + integritySize);
98
99     // Complete HMAC
100    CryptHmacEnd2B(&hmacState, &integrity->b);
101
102    return;
103 }

```

7.3.2.3 SequenceDataExport();

This function is used scan through the sequence object and either modify the hash state data for export (*contextSave*) or to import it into the internal format (*contextLoad*). This function should only be called after the sequence object has been copied to the context buffer (*contextSave*) or from the context buffer into the sequence object. The presumption is that the context buffer version of the data is the same size as the internal representation so nothing outside of the hash context area gets modified.

```

104 void
105 SequenceDataExport(
106     HASH_OBJECT      *object,           // IN: an internal hash object
107     HASH_OBJECT_BUFFER *exportObject    // OUT: a sequence context in a buffer
108 )
109 {
110     // If the hash object is not an event, then only one hash context is needed
111     int count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
112
113     for(count--; count >= 0; count--)
114     {
115         HASH_STATE      *hash = &object->state.hashState[count];
116         size_t          offset = (BYTE *)hash - (BYTE *)object;
117         BYTE             *exportHash = &((BYTE *)exportObject)[offset];
118
119         CryptHashExportState(hash, (EXPORT_HASH_STATE *)exportHash);
120     }
121 }

```

7.3.2.4 SequenceDataImport();

This function is used scan through the sequence object and either modify the hash state data for export (*contextSave*) or to import it into the internal format (*contextLoad*). This function should only be called after the sequence object has been copied to the context buffer (*contextSave*) or from the context buffer into the sequence object. The presumption is that the context buffer version of the data is the same size as the internal representation so nothing outside of the hash context area gets modified.

```

122 void
123 SequenceDataImport(
124     HASH_OBJECT      *object,           // IN/OUT: an internal hash object
125     HASH_OBJECT_BUFFER *exportObject    // IN/OUT: a sequence context in a buffer
126 )
127 {
128     // If the hash object is not an event, then only one hash context is needed
129     int count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
130
131     for(count--; count >= 0; count--)
132     {
133         HASH_STATE      *hash = &object->state.hashState[count];
134         size_t          offset = (BYTE *)hash - (BYTE *)object;
135         BYTE             *importHash = &((BYTE *)exportObject)[offset];
136
137         CryptHashImportState(hash, (EXPORT_HASH_STATE *)importHash);
138     }
139 }

```


7.4 Policy Command Support (Policy_spt.c)

7.4.1 Includes

```

1  #include "Tpm.h"
2  #include "Policy_spt_fp.h"
3  #include "PolicySigned_fp.h"
4  #include "PolicySecret_fp.h"
5  #include "PolicyTicket_fp.h"

```

7.4.2 Functions

7.4.2.1 PolicyParameterChecks()

This function validates the common parameters of TPM2_PolicySigned() and TPM2_PolicySecret(). The common parameters are *nonceTPM*, *expiration*, and *cpHashA*.

```

6  TPM_RC
7  PolicyParameterChecks(
8      SESSION      *session,
9      UINT64        authTimeout,
10     TPM2B_DIGEST  *cpHashA,
11     TPM2B_NONCE   *nonce,
12     TPM_RC        blameNonce,
13     TPM_RC        blameCpHash,
14     TPM_RC        blameExpiration
15 )
16 {
17     // Validate that input nonceTPM is correct if present
18     if(nonce != NULL && nonce->t.size != 0)
19     {
20         if(!MemoryEqual2B(&nonce->b, &session->nonceTPM.b))
21             return TPM_RCS_NONCE + blameNonce;
22     }
23     // If authTimeout is set (expiration != 0...
24     if(authTimeout != 0)
25     {
26         // Validate input expiration.
27         // Cannot compare time if clock stop advancing. A TPM_RC_NV_UNAVAILABLE
28         // or TPM_RC_NV_RATE error may be returned here.
29         RETURN_IF_NV_IS_NOT_AVAILABLE;
30
31         // if the time has already passed or the time epoch has changed then the
32         // time value is no longer good.
33         if((authTimeout < g_time)
34            || (session->epoch != g_timeEpoch))
35             return TPM_RCS_EXPIRED + blameExpiration;
36     }
37     // If the cpHash is present, then check it
38     if(cpHashA != NULL && cpHashA->t.size != 0)
39     {
40         // The cpHash input has to have the correct size
41         if(cpHashA->t.size != session->u2.policyDigest.t.size)
42             return TPM_RCS_SIZE + blameCpHash;
43
44         // If the cpHash has already been set, then this input value
45         // must match the current value.
46         if(session->u1.cpHash.b.size != 0
47            && !MemoryEqual2B(&cpHashA->b, &session->u1.cpHash.b))
48             return TPM_RC_CPHASH;
49     }

```

```

50     return TPM_RC_SUCCESS;
51 }

```

7.4.2.2 PolicyContextUpdate()

Update policy hash Update the *policyDigest* in policy session by extending *policyRef* and *objectName* to it. This will also update the *cpHash* if it is present.

```

52 void
53 PolicyContextUpdate(
54     TPM_CC      commandCode,    // IN: command code
55     TPM2B_NAME  *name,          // IN: name of entity
56     TPM2B_NONCE *ref,           // IN: the reference data
57     TPM2B_DIGEST *cpHash,       // IN: the cpHash (optional)
58     UINT64      policyTimeout,  // IN: the timeout value for the policy
59     SESSION     *session        // IN/OUT: policy session to be updated
60 )
61 {
62     HASH_STATE      hashState;
63
64     // Start hash
65     CryptHashStart(&hashState, session->authHashAlg);
66
67     // policyDigest size should always be the digest size of session hash algorithm.
68     pAssert(session->u2.policyDigest.t.size
69             == CryptHashGetDigestSize(session->authHashAlg));
70
71     // add old digest
72     CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
73
74     // add commandCode
75     CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
76
77     // add name if applicable
78     if(name != NULL)
79         CryptDigestUpdate2B(&hashState, &name->b);
80
81     // Complete the digest and get the results
82     CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
83
84     // If the policy reference is not null, do a second update to the digest.
85     if(ref != NULL)
86     {
87         // Start second hash computation
88         CryptHashStart(&hashState, session->authHashAlg);
89
90         // add policyDigest
91         CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
92
93         // add policyRef
94         CryptDigestUpdate2B(&hashState, &ref->b);
95
96         // Complete second digest
97         CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
98     }
99
100    // Deal with the cpHash. If the cpHash value is present
101    // then it would have already been checked to make sure that
102    // it is compatible with the current value so all we need
103    // to do here is copy it and set the isCpHashDefined attribute
104    if(cpHash != NULL && cpHash->t.size != 0)
105    {
106        session->u1.cpHash = *cpHash;
107        session->attributes.isCpHashDefined = SET;

```

```

108     }
109     // update the timeout if it is specified
110     if(policyTimeout != 0)
111     {
112         // If the timeout has not been set, then set it to the new value
113         // than the current timeout then set it to the new value
114         if(session->timeout == 0 || session->timeout > policyTimeout)
115             session->timeout = policyTimeout;
116     }
117     return;
118 }

```

7.4.2.3 ComputeAuthTimeout()

This function is used to determine what the authorization timeout value for the session should be.

```

119 UINT64
120 ComputeAuthTimeout(
121     SESSION      *session,           // IN: the session containing the time
122                                     // values
123     INT32         expiration,        // IN: either the number of seconds from
124                                     // the start of the session or the
125                                     // time in g_timer;
126     TPM2B_NONCE  *nonce             // IN: indicator of the time base
127 )
128 {
129     UINT64        policyTime;
130     // If no expiration, policy time is 0
131     if(expiration == 0)
132         policyTime = 0;
133     else
134     {
135         if(expiration < 0)
136             expiration = -expiration;
137         if(nonce->t.size == 0)
138             // The input time is absolute Time (not Clock), but it is expressed
139             // in seconds. To make sure that we don't time out too early, take the
140             // current value of milliseconds in g_time and add that to the input
141             // seconds value.
142             policyTime = (((UINT64)expiration) * 1000) + g_time % 1000;
143         else
144             // The policy timeout is the absolute value of the expiration in seconds
145             // added to the start time of the policy.
146             policyTime = session->startTime + (((UINT64)expiration) * 1000);
147     }
148     return policyTime;
149 }
150

```

7.4.2.4 PolicyDigestClear()

Function to reset the *policyDigest* of a session

```

151 void
152 PolicyDigestClear(
153     SESSION      *session
154 )
155 {
156     session->u2.policyDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
157     MemorySet(session->u2.policyDigest.t.buffer, 0,
158               session->u2.policyDigest.t.size);
159 }

```

7.4.2.5 PolicySptCheckCondition()

Checks to see if the condition in the policy is satisfied.

```

160  BOOL
161  PolicySptCheckCondition(
162      TPM_EO      operation,
163      BYTE        *opA,
164      BYTE        *opB,
165      UINT16      size
166  )
167  {
168      // Arithmetic Comparison
169      switch(operation)
170      {
171          case TPM_EO_EQ:
172              // compare A = B
173              return (UnsignedCompareB(size, opA, size, opB) == 0);
174              break;
175          case TPM_EO_NEQ:
176              // compare A != B
177              return (UnsignedCompareB(size, opA, size, opB) != 0);
178              break;
179          case TPM_EO_SIGNED_GT:
180              // compare A > B signed
181              return (SignedCompareB(size, opA, size, opB) > 0);
182              break;
183          case TPM_EO_UNSIGNED_GT:
184              // compare A > B unsigned
185              return (UnsignedCompareB(size, opA, size, opB) > 0);
186              break;
187          case TPM_EO_SIGNED_LT:
188              // compare A < B signed
189              return (SignedCompareB(size, opA, size, opB) < 0);
190              break;
191          case TPM_EO_UNSIGNED_LT:
192              // compare A < B unsigned
193              return (UnsignedCompareB(size, opA, size, opB) < 0);
194              break;
195          case TPM_EO_SIGNED_GE:
196              // compare A >= B signed
197              return (SignedCompareB(size, opA, size, opB) >= 0);
198              break;
199          case TPM_EO_UNSIGNED_GE:
200              // compare A >= B unsigned
201              return (UnsignedCompareB(size, opA, size, opB) >= 0);
202              break;
203          case TPM_EO_SIGNED_LE:
204              // compare A <= B signed
205              return (SignedCompareB(size, opA, size, opB) <= 0);
206              break;
207          case TPM_EO_UNSIGNED_LE:
208              // compare A <= B unsigned
209              return (UnsignedCompareB(size, opA, size, opB) <= 0);
210              break;
211          case TPM_EO_BITSET:
212              // All bits SET in B are SET in A. ((A&B)=B)
213              {
214                  UINT32 i;
215                  for(i = 0; i < size; i++)
216                      if((opA[i] & opB[i]) != opB[i])
217                          return FALSE;
218              }
219              break;
220          case TPM_EO_BITCLEAR:

```

```
221         // All bits SET in B are CLEAR in A. ((A&B)=0)
222     {
223         UINT32 i;
224         for(i = 0; i < size; i++)
225             if((opA[i] & opB[i]) != 0)
226                 return FALSE;
227     }
228     break;
229 default:
230     FAIL(FATAL_ERROR_INTERNAL);
231     break;
232 }
233 return TRUE;
234 }
```

7.5 NV Command Support (NV_spt.c)

7.5.1 Includes

```
1 #include "Tpm.h"
2 #include "NV_spt_fp.h"
```

7.5.2 Functions

7.5.2.1 NvReadAccessChecks()

Common routine for validating a read Used by TPM2_NV_Read, TPM2_NV_ReadLock() and TPM2_PolicyNV()

Error Return	Meaning
TPM_RC_NV_AUTHORIZATION	<i>authHandle</i> is not allowed to authorize read of the index
TPM_RC_NV_LOCKED	Read locked
TPM_RC_NV_UNINITIALIZED	Try to read an uninitialized index

```
3 TPM_RC
4 NvReadAccessChecks(
5     TPM_HANDLE    authHandle,    // IN: the handle that provided the
6                                // authorization
7     TPM_HANDLE    nvHandle,      // IN: the handle of the NV index to be read
8     TPMA_NV       attributes     // IN: the attributes of 'nvHandle'
9 )
10 {
11     // If data is read locked, returns an error
12     if(IS_ATTRIBUTE(attributes, TPMA_NV, READLOCKED))
13         return TPM_RC_NV_LOCKED;
14     // If the authorization was provided by the owner or platform, then check
15     // that the attributes allow the read. If the authorization handle
16     // is the same as the index, then the checks were made when the authorization
17     // was checked..
18     if(authHandle == TPM_RH_OWNER)
19     {
20         // If Owner provided authorization then OWNERWRITE must be SET
21         if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERREAD))
22             return TPM_RC_NV_AUTHORIZATION;
23     }
24     else if(authHandle == TPM_RH_PLATFORM)
25     {
26         // If Platform provided authorization then PPWRITE must be SET
27         if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPREAD))
28             return TPM_RC_NV_AUTHORIZATION;
29     }
30     // If neither Owner nor Platform provided authorization, make sure that it was
31     // provided by this index.
32     else if(authHandle != nvHandle)
33         return TPM_RC_NV_AUTHORIZATION;
34
35     // If the index has not been written, then the value cannot be read
36     // NOTE: This has to come after other access checks to make sure that
37     // the proper authorization is given to TPM2_NV_ReadLock()
38     if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN))
39         return TPM_RC_NV_UNINITIALIZED;
40
41     return TPM_RC_SUCCESS;
42 }
```

7.5.2.2 NvWriteAccessChecks()

Common routine for validating a write Used by TPM2_NV_Write, TPM2_NV_Increment, TPM2_SetBits(), and TPM2_NV_WriteLock()

Error Return	Meaning
TPM_RC_NV_AUTHORIZATION	Authorization fails
TPM_RC_NV_LOCKED	Write locked

```

43  TPM_RC
44  NvWriteAccessChecks(
45      TPM_HANDLE    authHandle,    // IN: the handle that provided the
46                                  // authorization
47      TPM_HANDLE    nvHandle,      // IN: the handle of the NV index to be written
48      TPMA_NV       attributes     // IN: the attributes of 'nvHandle'
49  )
50  {
51      // If data is write locked, returns an error
52      if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED))
53          return TPM_RC_NV_LOCKED;
54      // If the authorization was provided by the owner or platform, then check
55      // that the attributes allow the write. If the authorization handle
56      // is the same as the index, then the checks were made when the authorization
57      // was checked..
58      if(authHandle == TPM_RH_OWNER)
59      {
60          // If Owner provided authorization then ONWERWRITE must be SET
61          if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERWRITE))
62              return TPM_RC_NV_AUTHORIZATION;
63      }
64      else if(authHandle == TPM_RH_PLATFORM)
65      {
66          // If Platform provided authorization then PPWRITE must be SET
67          if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPWRITE))
68              return TPM_RC_NV_AUTHORIZATION;
69      }
70      // If neither Owner nor Platform provided authorization, make sure that it was
71      // provided by this index.
72      else if(authHandle != nvHandle)
73          return TPM_RC_NV_AUTHORIZATION;
74      return TPM_RC_SUCCESS;
75  }

```

7.5.2.3 NvClearOrderly()

This function is used to cause *gp.orderlyState* to be cleared to the non-orderly state.

```

76  TPM_RC
77  NvClearOrderly(
78      void
79  )
80  {
81      if(gp.orderlyState < SU_DA_USED_VALUE)
82          RETURN_IF_NV_IS_NOT_AVAILABLE;
83      g_clearOrderly = TRUE;
84      return TPM_RC_SUCCESS;
85  }

```


7.5.2.4 NvIsPinPassIndex()

Function to check to see if an NV index is a PIN Pass Index

Return Value	Meaning
TRUE(1)	is pin pass
FALSE(0)	is not pin pass

```
86  BOOL
87  NvIsPinPassIndex(
88      TPM_HANDLE      index      // IN: Handle to check
89  )
90  {
91      if(HandleGetType(index) == TPM_HT_NV_INDEX)
92      {
93          NV_INDEX      *nvIndex = NvGetIndexInfo(index, NULL);
94
95          return IsNvPinPassIndex(nvIndex->publicArea.attributes);
96      }
97      return FALSE;
98  }
```

7.6 Object Command Support (Object_spt.c)

7.6.1 Includes

```
1 #include "Tpm.h"
2 #include "Object_spt_fp.h"
```

7.6.2 Local Functions

7.6.2.1 GetIV2BSize()

Get the size of TPM2B_IV in canonical form that will be append to the start of the sensitive data. It includes both size of size field and size of iv data

```
3 static UINT16
4 GetIV2BSize(
5     OBJECT          *protector          // IN: the protector handle
6 )
7 {
8     TPM_ALG_ID      symAlg;
9     UINT16          keyBits;
10
11     // Determine the symmetric algorithm and size of key
12     if(protector == NULL)
13     {
14         // Use the context encryption algorithm and key size
15         symAlg = CONTEXT_ENCRYPT_ALG;
16         keyBits = CONTEXT_ENCRYPT_KEY_BITS;
17     }
18     else
19     {
20         symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
21         keyBits = protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
22     }
23     // The IV size is a UINT16 size field plus the block size of the symmetric
24     // algorithm
25     return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
26 }
```

7.6.2.2 ComputeProtectionKeyParms()

This function retrieves the symmetric protection key parameters for the sensitive data. The parameters retrieved from this function include encryption algorithm, key size in bit, and a TPM2B_SYM_KEY containing the key material as well as the key size in bytes. This function is used for any action that requires encrypting or decrypting of the sensitive area of an object or a credential blob.

```
27 static void
28 ComputeProtectionKeyParms(
29     OBJECT          *protector,          // IN: the protector object
30     TPM_ALG_ID      hashAlg,             // IN: hash algorithm for KDFa
31     TPM2B           *name,               // IN: name of the object
32     TPM2B           *seedIn,             // IN: optional seed for duplication blob.
33                                         // For non duplication blob, this
34                                         // parameter should be NULL
35     TPM_ALG_ID      *symAlg,             // OUT: the symmetric algorithm
36     UINT16          *keyBits,            // OUT: the symmetric key size in bits
37     TPM2B_SYM_KEY   *symKey             // OUT: the symmetric key
38 )
39 {
```

```

40     const TPM2B          *seed = seedIn;
41
42     // Determine the algorithms for the KDF and the encryption/decryption
43     // For TPM_RH_NULL, using context settings
44     if(protector == NULL)
45     {
46         // Use the context encryption algorithm and key size
47         *symAlg = CONTEXT_ENCRYPT_ALG;
48         symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
49         *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
50     }
51     else
52     {
53         TPMT_SYM_DEF_OBJECT *symDef;
54         symDef = &protector->publicArea.parameters.asymDetail.symmetric;
55         *symAlg = symDef->algorithm;
56         *keyBits = symDef->keyBits.sym;
57         symKey->t.size = (*keyBits + 7) / 8;
58     }
59     // Get seed for KDF
60     if(seed == NULL)
61         seed = GetSeedForKDF(protector);
62     // KDFa to generate symmetric key and IV value
63     CryptKDFa(hashAlg, seed, STORAGE_KEY, name, NULL,
64              symKey->t.size * 8, symKey->t.buffer, NULL, FALSE);
65     return;
66 }

```

7.6.2.3 ComputeOuterIntegrity()

The sensitive area parameter is a buffer that holds a space for the integrity value and the marshaled sensitive area. The caller should skip over the area set aside for the integrity value and compute the hash of the remainder of the object. The size field of sensitive is in unmarshaled form and the sensitive area contents is an array of bytes.

```

67 static void
68 ComputeOuterIntegrity(
69     TPM2B          *name,           // IN: the name of the object
70     OBJECT         *protector,      // IN: the object that
71                                     // provides protection. For an object,
72                                     // it is a parent. For a credential, it
73                                     // is the encrypt object. For
74                                     // a Temporary Object, it is NULL
75     TPMI_ALG_HASH  hashAlg,        // IN: algorithm to use for integrity
76     TPM2B          *seedIn,        // IN: an external seed may be provided for
77                                     // duplication blob. For non duplication
78                                     // blob, this parameter should be NULL
79     UINT32         sensitiveSize,   // IN: size of the marshaled sensitive data
80     BYTE           *sensitiveData,  // IN: sensitive area
81     TPM2B_DIGEST   *integrity,     // OUT: integrity
82 )
83 {
84     HMAC_STATE      hmacState;
85     TPM2B_DIGEST    hmacKey;
86     const TPM2B     *seed = seedIn;
87     //
88     // Get seed for KDF
89     if(seed == NULL)
90         seed = GetSeedForKDF(protector);
91     // Determine the HMAC key bits
92     hmacKey.t.size = CryptHashGetDigestSize(hashAlg);
93
94     // KDFa to generate HMAC key
95     CryptKDFa(hashAlg, seed, INTEGRITY_KEY, NULL, NULL,

```

```

96         hmacKey.t.size * 8, hmacKey.t.buffer, NULL, FALSE);
97     // Start HMAC and get the size of the digest which will become the integrity
98     integrity->t.size = CryptHmacStart2B(&hmacState, hashAlg, &hmacKey.b);
99
100    // Adding the marshaled sensitive area to the integrity value
101    CryptDigestUpdate(&hmacState.hashState, sensitiveSize, sensitiveData);
102
103    // Adding name
104    CryptDigestUpdate2B(&hmacState.hashState, name);
105
106    // Compute HMAC
107    CryptHmacEnd2B(&hmacState, &integrity->b);
108
109    return;
110 }

```

7.6.2.4 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap

```

111 static void
112 ComputeInnerIntegrity(
113     TPM_ALG_ID      hashAlg,          // IN: hash algorithm for inner wrap
114     TPM2B            *name,           // IN: the name of the object
115     UINT16           dataSize,        // IN: the size of sensitive data
116     BYTE             *sensitiveData,  // IN: sensitive data
117     TPM2B_DIGEST     *integrity       // OUT: inner integrity
118 )
119 {
120     HASH_STATE      hashState;
121     //
122     // Start hash and get the size of the digest which will become the integrity
123     integrity->t.size = CryptHashStart(&hashState, hashAlg);
124
125     // Adding the marshaled sensitive area to the integrity value
126     CryptDigestUpdate(&hashState, dataSize, sensitiveData);
127
128     // Adding name
129     CryptDigestUpdate2B(&hashState, name);
130
131     // Compute hash
132     CryptHashEnd2B(&hashState, &integrity->b);
133
134     return;
135 }

```

7.6.2.5 ProduceInnerIntegrity()

This function produces an inner integrity for regular private, credential or duplication blob. It requires the sensitive data being marshaled to the *innerBuffer*, with the leading bytes reserved for integrity hash. It assumes the sensitive data starts at address (*innerBuffer* + integrity size). This function computes the integrity at the beginning of the inner buffer. It returns the total size of buffer with the inner wrap.

```

136 static UINT16
137 ProduceInnerIntegrity(
138     TPM2B            *name,           // IN: the name of the object
139     TPM_ALG_ID      hashAlg,          // IN: hash algorithm for inner wrap
140     UINT16           dataSize,        // IN: the size of sensitive data, excluding the
141                                     // leading integrity buffer size
142     BYTE             *innerBuffer     // IN/OUT: inner buffer with sensitive data in
143                                     // it. At input, the leading bytes of this
144                                     // buffer is reserved for integrity

```

```

145     )
146 {
147     BYTE          *sensitiveData; // pointer to the sensitive data
148     TPM2B_DIGEST  integrity;
149     UINT16        integritySize;
150     BYTE          *buffer;        // Auxiliary buffer pointer
151 //
152 // sensitiveData points to the beginning of sensitive data in innerBuffer
153 integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
154 sensitiveData = innerBuffer + integritySize;
155
156 ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);
157
158 // Add integrity at the beginning of inner buffer
159 buffer = innerBuffer;
160 TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
161
162 return dataSize + integritySize;
163 }

```

7.6.2.6 CheckInnerIntegrity()

This function check integrity of inner blob

Error Return	Meaning
TPM_RC_INTEGRITY	if the outer blob integrity is bad
unmarshal errors	unmarshal errors while unmarshaling integrity

```

164 static TPM_RC
165 CheckInnerIntegrity(
166     TPM2B          *name,           // IN: the name of the object
167     TPM_ALG_ID     hashAlg,        // IN: hash algorithm for inner wrap
168     UINT16         dataSize,       // IN: the size of sensitive data, including the
169                                     // leading integrity buffer size
170     BYTE          *innerBuffer     // IN/OUT: inner buffer with sensitive data in
171                                     // it
172 )
173 {
174     TPM_RC         result;
175     TPM2B_DIGEST   integrity;
176     TPM2B_DIGEST   integrityToCompare;
177     BYTE          *buffer;          // Auxiliary buffer pointer
178     INT32          size;
179 //
180 // Unmarshal integrity
181 buffer = innerBuffer;
182 size = (INT32)dataSize;
183 result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
184 if(result == TPM_RC_SUCCESS)
185 {
186     // Compute integrity to compare
187     ComputeInnerIntegrity(hashAlg, name, (UINT16)size, buffer,
188                           &integrityToCompare);
189     // Compare outer blob integrity
190     if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
191         result = TPM_RC_INTEGRITY;
192 }
193 return result;
194 }

```

7.6.3 Public Functions

7.6.3.1 AdjustAuthSize()

This function will validate that the input *authValue* is no larger than the *digestSize* for the *nameAlg*. It will then pad with zeros to the size of the digest.

```

195  BOOL
196  AdjustAuthSize(
197      TPM2B_AUTH      *auth,          // IN/OUT: value to adjust
198      TPMI_ALG_HASH   nameAlg        // IN:
199  )
200  {
201      UINT16           digestSize;
202      //
203      // If there is no nameAlg, then this is a LoadExternal and the authVale can
204      // be any size up to the maximum allowed by the implementation
205      digestSize = (nameAlg == TPM_ALG_NULL) ? sizeof(TPMU_HA)
206          : CryptHashGetDigestSize(nameAlg);
207      if(digestSize < MemoryRemoveTrailingZeros(auth))
208          return FALSE;
209      else if(digestSize > auth->t.size)
210          MemoryPad2B(&auth->b, digestSize);
211      auth->t.size = digestSize;
212
213      return TRUE;
214  }

```

7.6.3.2 AreAttributesForParent()

This function is called by create, load, and import functions.

NOTE The *isParent* attribute is SET when an object is loaded and it has attributes that are suitable for a parent object.

Return Value	Meaning
TRUE(1)	properties are those of a parent
FALSE(0)	properties are not those of a parent

```

215  BOOL
216  ObjectIsParent(
217      OBJECT      *parentObject    // IN: parent handle
218  )
219  {
220      return parentObject->attributes.isParent;
221  }

```

7.6.3.3 CreateChecks()

Attribute checks that are unique to creation.

Error Return	Meaning
TPM_RC_ATTRIBUTES	<i>sensitiveDataOrigin</i> is not consistent with the object type
other	returns from PublicAttributesValidation()

```

222  TPM_RC
223  CreateChecks(

```

```

224     OBJECT                *parentObject,
225     TPMT_PUBLIC           *publicArea,
226     UINT16                sensitiveDataSize
227 )
228 {
229     TPMA_OBJECT            attributes = publicArea->objectAttributes;
230     TPM_RC                 result = TPM_RC_SUCCESS;
231 //
232 // If the caller indicates that they have provided the data, then make sure that
233 // they have provided some data.
234 if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
235     && (sensitiveDataSize == 0))
236     return TPM_RCS_ATTRIBUTES;
237 // For an ordinary object, data can only be provided when sensitiveDataOrigin
238 // is CLEAR
239 if((parentObject != NULL)
240     && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
241     && (sensitiveDataSize != 0))
242     return TPM_RCS_ATTRIBUTES;
243 switch(publicArea->type)
244 {
245     case TPM_ALG_KEYEDHASH:
246         // if this is a data object (sign == decrypt == CLEAR) then the
247         // TPM cannot be the data source.
248         if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
249             && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
250             && IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
251             result = TPM_RC_ATTRIBUTES;
252         // comment out the next line in order to prevent a fixedTPM derivation
253         // parent
254         break;
255 //
256     case TPM_ALG_SYMCIPHER:
257         // A restricted key symmetric key (SYMCIPHER and KEYEDHASH)
258         // must have sensitiveDataOrigin SET unless it has fixedParent and
259         // fixedTPM CLEAR.
260         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
261             if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
262                 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
263                     || IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
264                     result = TPM_RCS_ATTRIBUTES;
265         break;
266     default: // Asymmetric keys cannot have the sensitive portion provided
267         if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
268             result = TPM_RCS_ATTRIBUTES;
269         break;
270 }
271 if(TPM_RC_SUCCESS == result)
272 {
273     result = PublicAttributesValidation(parentObject, publicArea);
274 }
275 return result;
276 }

```

7.6.3.4 SchemeChecks()

This function is called by TPM2_LoadExternal() and PublicAttributesValidation(). This function validates the schemes in the public area of an object.

Error Return	Meaning
TPM_RC_HASH	non-duplicable storage key and its parent have different name algorithm
TPM_RC_KDF	incorrect KDF specified for decrypting keyed hash object
TPM_RC_KEY	invalid key size values in an asymmetric key public area
TPM_RC_SCHEME	inconsistent attributes <i>decrypt</i> , <i>sign</i> , <i>restricted</i> and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object
TPM_RC_SYMMETRIC	a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from TPM_ALG_NULL

```

276 TPM_RC
277 SchemeChecks (
278     OBJECT          *parentObject, // IN: parent (null if primary seed)
279     TPMT_PUBLIC     *publicArea    // IN: public area of the object
280 )
281 {
282     TPMT_SYM_DEF_OBJECT *symAlgs = NULL;
283     TPM_ALG_ID          scheme = TPM_ALG_NULL;
284     TPMA_OBJECT          attributes = publicArea->objectAttributes;
285     TPMU_PUBLIC_PARMS    *parms = &publicArea->parameters;
286 //
287     switch(publicArea->type)
288     {
289         case TPM_ALG_SYMCIPHER:
290             symAlgs = &parms->symDetail.sym;
291             // If this is a decrypt key, then only the block cipher modes (not
292             // SMAC) are valid. TPM_ALG_NULL is OK too. If this is a 'sign' key,
293             // then any mode that got through the unmarshaling is OK.
294             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
295                 && !CryptSymModeIsValid(symAlgs->mode.sym, TRUE))
296                 return TPM_RC_SCHEME;
297             break;
298         case TPM_ALG_KEYEDHASH:
299             scheme = parms->keyedHashDetail.scheme.scheme;
300             // if both sign and decrypt
301             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
302                 == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
303             {
304                 // if both sign and decrypt are set or clear, then need
305                 // TPM_ALG_NULL as scheme
306                 if(scheme != TPM_ALG_NULL)
307                     return TPM_RC_SCHEME;
308             }
309             else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
310                 && scheme != TPM_ALG_HMAC)
311                 return TPM_RC_SCHEME;
312             else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
313             {
314                 if(scheme != TPM_ALG_XOR)
315                     return TPM_RC_SCHEME;
316                 // If this is a derivation parent, then the KDF needs to be
317                 // SP800-108 for this implementation. This is the only derivation
318                 // supported by this implementation. Other implementations could
319                 // support additional schemes. There is no default.
320                 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
321                 {
322                     if(parms->keyedHashDetail.scheme.details.xor.kdf
323                         != TPM_ALG_KDF1_SP800_108)
324                         return TPM_RC_SCHEME;

```

```

325         // Must select a digest.
326         if(CryptHashGetDigestSize(
327             parms->keyedHashDetail.scheme.details.xor.hashAlg) == 0)
328             return TPM_RCS_HASH;
329     }
330 }
331 break;
332 default: // handling for asymmetric
333     scheme = parms->asymDetail.scheme.scheme;
334     symAlgs = &parms->asymDetail.symmetric;
335     // if the key is both sign and decrypt, then the scheme must be
336     // TPM_ALG_NULL because there is no way to specify both a sign and a
337     // decrypt scheme in the key.
338     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
339         == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
340     {
341         // scheme must be TPM_ALG_NULL
342         if(scheme != TPM_ALG_NULL)
343             return TPM_RCS_SCHEME;
344     }
345     else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
346     {
347         // If this is a signing key, see if it has a signing scheme
348         if(CryptIsAsymSignScheme(publicArea->type, scheme))
349         {
350             // if proper signing scheme then it needs a proper hash
351             if(parms->asymDetail.scheme.details.anySig.hashAlg
352                 == TPM_ALG_NULL)
353                 return TPM_RCS_SCHEME;
354         }
355         else
356         {
357             // signing key that does not have a proper signing scheme.
358             // This is OK if the key is not restricted and its scheme
359             // is TPM_ALG_NULL
360             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
361                 || scheme != TPM_ALG_NULL)
362                 return TPM_RCS_SCHEME;
363         }
364     }
365     else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
366     {
367         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
368         {
369             // for a restricted decryption key (a parent), scheme
370             // is required to be TPM_ALG_NULL
371             if(scheme != TPM_ALG_NULL)
372                 return TPM_RCS_SCHEME;
373         }
374         else
375         {
376             // For an unrestricted decryption key, the scheme has to
377             // be a valid scheme or TPM_ALG_NULL
378             if(scheme != TPM_ALG_NULL &&
379                 !CryptIsAsymDecryptScheme(publicArea->type, scheme))
380                 return TPM_RCS_SCHEME;
381         }
382     }
383     if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
384         || !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
385     {
386         // For an asymmetric key that is not a parent, the symmetric
387         // algorithms must be TPM_ALG_NULL
388         if(symAlgs->algorithm != TPM_ALG_NULL)
389             return TPM_RCS_SYMMETRIC;
390     }

```

```

391 // Special checks for an ECC key
392 #if ALG_ECC
393 if(publicArea->type == TPM_ALG_ECC)
394 {
395     TPM_ECC_CURVE        curveID;
396     const TPMT_ECC_SCHEME *curveScheme;
397
398     curveID = publicArea->parameters.eccDetail.curveID;
399     curveScheme = CryptGetCurveSignScheme(curveID);
400     // The curveID must be valid or the unmarshaling is busted.
401     pAssert(curveScheme != NULL);
402
403     // If the curveID requires a specific scheme, then the key must
404     // select the same scheme
405     if(curveScheme->scheme != TPM_ALG_NULL)
406     {
407         TPMS_ECC_PARMS *ecc = &publicArea->parameters.eccDetail;
408         if(scheme != curveScheme->scheme)
409             return TPM_RCS_SCHEME;
410         // The scheme can allow any hash, or not...
411         if(curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
412            && (ecc->scheme.details.anySig.hashAlg
413               != curveScheme->details.anySig.hashAlg))
414             return TPM_RCS_SCHEME;
415     }
416     // For now, the KDF must be TPM_ALG_NULL
417     if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
418         return TPM_RCS_KDF;
419 }
420 #endif
421 break;
422 }
423 // If this is a restricted decryption key with symmetric algorithms, then it
424 // is an ordinary parent (not a derivation parent). It needs to specific
425 // symmetric algorithms other than TPM_ALG_NULL
426 if(symAlgs != NULL
427    && IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
428    && IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
429 {
430     if(symAlgs->algorithm == TPM_ALG_NULL)
431         return TPM_RCS_SYMMETRIC;
432 #if 0
433     //??
434     // This next check is under investigation. Need to see if it will break Windows
435     // before it is enabled. If it does not, then it should be default because a
436     // the mode used with a parent is always CFB and Part 2 indicates as much.
437     if(symAlgs->mode.sym != TPM_ALG_CFB)
438         return TPM_RCS_MODE;
439 #endif
440 // If this parent is not duplicable, then the symmetric algorithms
441 // (encryption and hash) must match those of its parent
442 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
443    && (parentObject != NULL))
444 {
445     if(publicArea->nameAlg != parentObject->publicArea.nameAlg)
446         return TPM_RCS_HASH;
447     if(!MemoryEqual(symAlgs, &parentObject->publicArea.parameters,
448                     sizeof(TPMT_SYM_DEF_OBJECT)))
449         return TPM_RCS_SYMMETRIC;
450 }
451 }
452 return TPM_RC_SUCCESS;
453 }

```

7.6.3.5 PublicAttributesValidation()

This function validates the values in the public area of an object. This function is used in the processing of TPM2_Create, TPM2_CreatePrimary(), TPM2_CreateLoaded(), TPM2_Load(), TPM2_Import(), and TPM2_LoadExternal(). For TPM2_Import() this is only used if the new parent has *fixedTPM* SET. For TPM2_LoadExternal(), this is not used for a public-only key

Error Return	Meaning
TPM_RC_ATTRIBUTES	<i>fixedTPM</i> , <i>fixedParent</i> , or <i>encryptedDuplication</i> attributes are inconsistent between themselves or with those of the parent object; inconsistent <i>restricted</i> , <i>decrypt</i> and <i>sign</i> attributes; attempt to inject sensitive data for an asymmetric key; attempt to create a symmetric cipher key that is not a decryption key
TPM_RC_HASH	<i>nameAlg</i> is TPM_ALG_NULL
TPM_RC_SIZE	<i>authPolicy</i> size does not match digest size of the name algorithm in <i>publicArea</i>
other	returns from SchemeChecks()

```

453 TPM_RC
454 PublicAttributesValidation(
455     OBJECT      *parentObject, // IN: input parent object
456     TPMT_PUBLIC *publicArea    // IN: public area of the object
457 )
458 {
459     TPMA_OBJECT attributes = publicArea->objectAttributes;
460     TPMA_OBJECT parentAttributes = TPMA_ZERO_INITIALIZER();
461     //
462     if(parentObject != NULL)
463         parentAttributes = parentObject->publicArea.objectAttributes;
464     if(publicArea->nameAlg == TPM_ALG_NULL)
465         return TPM_RCS_HASH;
466     // If there is an authPolicy, it needs to be the size of the digest produced
467     // by the nameAlg of the object
468     if((publicArea->authPolicy.t.size != 0
469         && (publicArea->authPolicy.t.size
470             != CryptHashGetDigestSize(publicArea->nameAlg))))
471         return TPM_RCS_SIZE;
472     // If the parent is fixedTPM (including a Primary Object) the object must have
473     // the same value for fixedTPM and fixedParent
474     if(parentObject == NULL
475        || IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
476     {
477         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
478            != IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
479             return TPM_RCS_ATTRIBUTES;
480     }
481     else
482     {
483         // The parent is not fixedTPM so the object can't be fixedTPM
484         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
485             return TPM_RCS_ATTRIBUTES;
486     }
487     // See if sign and decrypt are the same
488     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
489        == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
490     {
491         // a restricted key cannot have both SET or both CLEAR
492         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
493             return TPM_RC_ATTRIBUTES;
494         // only a data object may have both sign and decrypt CLEAR
495         // BTW, since we know that decrypt==sign, no need to check both

```

```

496     if(publicArea->type != TPM_ALG_KEYEDHASH
497        && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
498         return TPM_RC_ATTRIBUTES;
499     }
500     // If the object can't be duplicated (directly or indirectly) then there
501     // is no justification for having encryptedDuplication SET
502     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
503        && IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
504         return TPM_RCS_ATTRIBUTES;
505     // If a parent object has fixedTPM CLEAR, the child must have the
506     // same encryptedDuplication value as its parent.
507     // Primary objects are considered to have a fixedTPM parent (the seeds).
508     if(parentObject != NULL
509        && !IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
510     {
511         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication)
512            != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, encryptedDuplication))
513             return TPM_RCS_ATTRIBUTES;
514     }
515     // Special checks for derived objects
516     if((parentObject != NULL) && (parentObject->attributes.derivation == SET))
517     {
518         // A derived object has the same settings for fixedTPM as its parent
519         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
520            != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
521             return TPM_RCS_ATTRIBUTES;
522         // A derived object is required to be fixedParent
523         if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent))
524             return TPM_RCS_ATTRIBUTES;
525     }
526     return SchemeChecks(parentObject, publicArea);
527 }

```

7.6.3.6 FillInCreationData()

Fill in creation data for an object.

```

528 void
529 FillInCreationData(
530     TPMI_DH_OBJECT      parentHandle, // IN: handle of parent
531     TPMI_ALG_HASH        nameHashAlg, // IN: name hash algorithm
532     TPMI_PCR_SELECTION   *creationPCR, // IN: PCR selection
533     TPM2B_DATA            *outsideData, // IN: outside data
534     TPM2B_CREATION_DATA   *outCreation, // OUT: creation data for output
535     TPM2B_DIGEST          *creationDigest // OUT: creation digest
536 )
537 {
538     BYTE      creationBuffer[sizeof(TPMS_CREATION_DATA)];
539     BYTE      *buffer;
540     HASH_STATE hashState;
541     //
542     // Fill in TPMS_CREATION_DATA in outCreation
543
544     // Compute PCR digest
545     PCRComputeCurrentDigest(nameHashAlg, creationPCR,
546                            &outCreation->creationData.pcrDigest);
547
548     // Put back PCR selection list
549     outCreation->creationData.pcrSelect = *creationPCR;
550
551     // Get locality
552     outCreation->creationData.locality
553         = LocalityGetAttributes(_plat__LocalityGet());
554     outCreation->creationData.parentNameAlg = TPM_ALG_NULL;

```

```

555
556 // If the parent is either a primary seed or TPM_ALG_NULL, then the Name
557 // and QN of the parent are the parent's handle.
558 if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
559 {
560     buffer = &outCreation->creationData.parentName.t.name[0];
561     outCreation->creationData.parentName.t.size =
562         TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
563     // For a primary or temporary object, the parent name (a handle) and the
564     // parent's QN are the same
565     outCreation->creationData.parentQualifiedName
566         = outCreation->creationData.parentName;
567 }
568 else // Regular object
569 {
570     OBJECT *parentObject = HandleToObject(parentHandle);
571 //
572 // Set name algorithm
573 outCreation->creationData.parentNameAlg = parentObject->publicArea.nameAlg;
574
575 // Copy parent name
576 outCreation->creationData.parentName = parentObject->name;
577
578 // Copy parent qualified name
579 outCreation->creationData.parentQualifiedName = parentObject->qualifiedName;
580 }
581 // Copy outside information
582 outCreation->creationData.outsideInfo = *outsideData;
583
584 // Marshal creation data to canonical form
585 buffer = creationBuffer;
586 outCreation->size = TPMS_CREATION_DATA_Marshal(&outCreation->creationData,
587                                             &buffer, NULL);
588 // Compute hash for creation field in public template
589 creationDigest->t.size = CryptHashStart(&hashState, nameHashAlg);
590 CryptDigestUpdate(&hashState, outCreation->size, creationBuffer);
591 CryptHashEnd2B(&hashState, &creationDigest->b);
592
593 return;
594 }

```

7.6.3.7 GetSeedForKDF()

Get a seed for KDF. The KDF for encryption and HMAC key use the same seed.

```

595 const TPM2B *
596 GetSeedForKDF(
597     OBJECT *protector // IN: the protector handle
598 )
599 {
600     // Get seed for encryption key. Use input seed if provided.
601     // Otherwise, using protector object's seedValue. TPM_RH_NULL is the only
602     // exception that we may not have a loaded object as protector. In such a
603     // case, use nullProof as seed.
604     if(protector == NULL)
605         return &gr.nullProof.b;
606     else
607         return &protector->sensitive.seedValue.b;
608 }

```


7.6.3.8 ProduceOuterWrap()

This function produce outer wrap for a buffer containing the sensitive data. It requires the sensitive data being marshaled to the *outerBuffer*, with the leading bytes reserved for integrity hash. If iv is used, iv space should be reserved at the beginning of the buffer. It assumes the sensitive data starts at address (*outerBuffer* + integrity size [+ iv size]). This function performs:

- 1) Add IV before sensitive area if required
- 2) encrypt sensitive data, if iv is required, encrypt by iv. otherwise, encrypted by a NULL iv
- 3) add HMAC integrity at the beginning of the buffer It returns the total size of blob with outer wrap

```

609  UINT16
610  ProduceOuterWrap(
611      OBJECT      *protector,      // IN: The handle of the object that provides
612                                   // protection. For object, it is parent
613                                   // handle. For credential, it is the handle
614                                   // of encrypt object.
615      TPM2B       *name,           // IN: the name of the object
616      TPM_ALG_ID   hashAlg,        // IN: hash algorithm for outer wrap
617      TPM2B       *seed,           // IN: an external seed may be provided for
618                                   // duplication blob. For non duplication
619                                   // blob, this parameter should be NULL
620      BOOL         useIV,           // IN: indicate if an IV is used
621      UINT16       dataSize,        // IN: the size of sensitive data, excluding the
622                                   // leading integrity buffer size or the
623                                   // optional iv size
624      BYTE         *outerBuffer     // IN/OUT: outer buffer with sensitive data in
625                                   // it
626  )
627  {
628      TPM_ALG_ID   symAlg;
629      UINT16       keyBits;
630      TPM2B_SYM_KEY symKey;
631      TPM2B_IV     ivRNG;          // IV from RNG
632      TPM2B_IV     *iv = NULL;
633      UINT16       ivSize = 0;     // size of iv area, including the size field
634      BYTE         *sensitiveData; // pointer to the sensitive data
635      TPM2B_DIGEST integrity;
636      UINT16       integritySize;
637      BYTE         *buffer;        // Auxiliary buffer pointer
638  //
639  // Compute the beginning of sensitive data. The outer integrity should
640  // always exist if this function is called to make an outer wrap
641  integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
642  sensitiveData = outerBuffer + integritySize;
643
644  // If iv is used, adjust the pointer of sensitive data and add iv before it
645  if(useIV)
646  {
647      ivSize = GetIV2BSize(protector);
648
649      // Generate IV from RNG. The iv data size should be the total IV area
650      // size minus the size of size field
651      ivRNG.t.size = ivSize - sizeof(UINT16);
652      CryptRandomGenerate(ivRNG.t.size, ivRNG.t.buffer);
653
654      // Marshal IV to buffer
655      buffer = sensitiveData;
656      TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
657
658      // adjust sensitive data starting after IV area
659      sensitiveData += ivSize;
660  }

```



```

661         // Use iv for encryption
662         iv = &ivRNG;
663     }
664     // Compute symmetric key parameters for outer buffer encryption
665     ComputeProtectionKeyParms(protector, hashAlg, name, seed,
666                             &symAlg, &keyBits, &symKey);
667     // Encrypt inner buffer in place
668     CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
669                         symKey.t.buffer, iv, TPM_ALG_CFB, dataSize,
670                         sensitiveData);
671     // Compute outer integrity. Integrity computation includes the optional IV
672     // area
673     ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
674                         outerBuffer + integritySize, &integrity);
675     // Add integrity at the beginning of outer buffer
676     buffer = outerBuffer;
677     TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
678
679     // return the total size in outer wrap
680     return dataSize + integritySize + ivSize;
681 }

```

7.6.3.9 UnwrapOuter()

This function remove the outer wrap of a blob containing sensitive data This function performs:

- 1) check integrity of outer blob
- 2) decrypt outer blob

Error Return	Meaning
TPM_RCS_INSUFFICIENT	error during sensitive data unmarshaling
TPM_RCS_INTEGRITY	sensitive data integrity is broken
TPM_RCS_SIZE	error during sensitive data unmarshaling
TPM_RCS_VALUE	IV size for CFB does not match the encryption algorithm block size

```

682 TPM_RC
683 UnwrapOuter(
684     OBJECT          *protector,    // IN: The object that provides
685                                     // protection. For object, it is parent
686                                     // handle. For credential, it is the
687                                     // encrypt object.
688     TPM2B           *name,         // IN: the name of the object
689     TPM_ALG_ID      hashAlg,       // IN: hash algorithm for outer wrap
690     TPM2B           *seed,         // IN: an external seed may be provided for
691                                     // duplication blob. For non duplication
692                                     // blob, this parameter should be NULL.
693     BOOL            useIV,         // IN: indicates if an IV is used
694     UINT16          dataSize,      // IN: size of sensitive data in outerBuffer,
695                                     // including the leading integrity buffer
696                                     // size, and an optional iv area
697     BYTE            *outerBuffer   // IN/OUT: sensitive data
698 )
699 {
700     TPM_RC          result;
701     TPM_ALG_ID      symAlg = TPM_ALG_NULL;
702     TPM2B_SYM_KEY   symKey;
703     UINT16          keyBits = 0;
704     TPM2B_IV        ivIn;          // input IV retrieved from input buffer
705     TPM2B_IV        *iv = NULL;
706     BYTE            *sensitiveData; // pointer to the sensitive data

```

```

707     TPM2B_DIGEST    integrityToCompare;
708     TPM2B_DIGEST    integrity;
709     INT32            size;
710 //
711 // Unmarshal integrity
712 sensitiveData = outerBuffer;
713 size = (INT32)dataSize;
714 result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
715 if(result == TPM_RC_SUCCESS)
716 {
717     // Compute integrity to compare
718     ComputeOuterIntegrity(name, protector, hashAlg, seed,
719                           (UINT16)size, sensitiveData,
720                           &integrityToCompare);
721     // Compare outer blob integrity
722     if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
723         return TPM_RCS_INTEGRITY;
724     // Get the symmetric algorithm parameters used for encryption
725     ComputeProtectionKeyParms(protector, hashAlg, name, seed,
726                               &symAlg, &keyBits, &symKey);
727     // Retrieve IV if it is used
728     if(useIV)
729     {
730         result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
731         if(result == TPM_RC_SUCCESS)
732         {
733             // The input iv size for CFB must match the encryption algorithm
734             // block size
735             if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
736                 result = TPM_RC_VALUE;
737             else
738                 iv = &ivIn;
739         }
740     }
741 }
742 // If no errors, decrypt private in place. Since this function uses CFB,
743 // CryptSymmetricDecrypt() will not return any errors. It may fail but it will
744 // not return an error.
745 if(result == TPM_RC_SUCCESS)
746     CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
747                           symKey.t.buffer, iv, TPM_ALG_CFB,
748                           (UINT16)size, sensitiveData);
749 return result;
750 }

```

7.6.3.10 MarshalSensitive()

This function is used to marshal a sensitive area. Among other things, it adjusts the size of the *authValue* to be no smaller than the digest of *nameAlg*. Returns the size of the marshaled area.

```

751 static UINT16
752 MarshalSensitive(
753     OBJECT            *parent,           // IN: the object parent (optional)
754     BYTE              *buffer,           // OUT: receiving buffer
755     TPMT_SENSITIVE    *sensitive,        // IN: the sensitive area to marshal
756     TPMI_ALG_HASH     nameAlg            // IN:
757 )
758 {
759     BYTE              *sizeField = buffer; // saved so that size can be
760                                           // marshaled after it is known
761     UINT16            retVal;
762 //
763 // Pad the authValue if needed
764 MemoryPad2B(&sensitive->authValue.b, CryptHashGetDigestSize(nameAlg));

```

```

765     buffer += 2;
766
767     // Marshal the structure
768 #if ALG_RSA
769     // If the sensitive size is the special case for a prime in the type
770     if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) > 0)
771     {
772         UINT16          sizeSave = sensitive->sensitive.rsa.t.size;
773         //
774         // Turn off the flag that indicates that the sensitive->sensitive contains
775         // the CRT form of the exponent.
776         sensitive->sensitive.rsa.t.size &= ~(RSA_prime_flag);
777         // If the parent isn't fixedTPM, then truncate the sensitive data to be
778         // the size of the prime. Otherwise, leave it at the current size which
779         // is the full CRT size.
780         if(parent == NULL
781            || !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
782                           TPMA_OBJECT, fixedTPM))
783             sensitive->sensitive.rsa.t.size /= 5;
784         retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
785         // Restore the flag and the size.
786         sensitive->sensitive.rsa.t.size = sizeSave;
787     }
788     else
789 #endif
790     retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
791
792     // Marshal the size
793     retVal = (UINT16)(retVal + UINT16_Marshal(&retVal, &sizeField, NULL));
794
795     return retVal;
796 }

```

7.6.3.11 SensitiveToPrivate()

This function prepare the private blob for off the chip storage The operations in this function:

- 1) marshal TPM2B_SENSITIVE structure into the buffer of TPM2B_PRIVATE
- 2) apply encryption to the sensitive area.
- 3) apply outer integrity computation.

```

797 void
798 SensitiveToPrivate(
799     TPMT_SENSITIVE *sensitive,      // IN: sensitive structure
800     TPM2B_NAME *name,              // IN: the name of the object
801     OBJECT *parent,                // IN: The parent object
802     TPM_ALG_ID nameAlg,            // IN: hash algorithm in public area. This
803                                     // parameter is used when parentHandle is
804                                     // NULL, in which case the object is
805                                     // temporary.
806     TPM2B_PRIVATE *outPrivate      // OUT: output private structure
807 )
808 {
809     BYTE          *sensitiveData;    // pointer to the sensitive data
810     UINT16        dataSize;          // data blob size
811     TPMI_ALG_HASH hashAlg;           // hash algorithm for integrity
812     UINT16        integritySize;
813     UINT16        ivSize;
814     //
815     pAssert(name != NULL && name->t.size != 0);
816
817     // Find the hash algorithm for integrity computation
818     if(parent == NULL)

```

```

819     {
820         // For Temporary Object, using self name algorithm
821         hashAlg = nameAlg;
822     }
823     else
824     {
825         // Otherwise, using parent's name algorithm
826         hashAlg = parent->publicArea.nameAlg;
827     }
828     // Starting of sensitive data without wrappers
829     sensitiveData = outPrivate->t.buffer;
830
831     // Compute the integrity size
832     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
833
834     // Reserve space for integrity
835     sensitiveData += integritySize;
836
837     // Get iv size
838     ivSize = GetIV2BSize(parent);
839
840     // Reserve space for iv
841     sensitiveData += ivSize;
842
843     // Marshal the sensitive area including authValue size adjustments.
844     dataSize = MarshalSensitive(parent, sensitiveData, sensitive, nameAlg);
845
846     //Produce outer wrap, including encryption and HMAC
847     outPrivate->t.size = ProduceOuterWrap(parent, &name->b, hashAlg, NULL,
848                                         TRUE, dataSize, outPrivate->t.buffer);
849     return;
850 }

```

7.6.3.12 PrivateToSensitive()

Unwrap an input private area; check the integrity; decrypt and retrieve data to a sensitive structure. The operations in this function:

- 1) check the integrity HMAC of the input private area
- 2) decrypt the private buffer
- 3) unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

Error Return	Meaning
TPM_RCS_INTEGRITY	if the private area integrity is bad
TPM_RC_SENSITIVE	unmarshal errors while unmarshaling TPMS_ENCRYPT from input private
TPM_RCS_SIZE	error during sensitive data unmarshaling
TPM_RCS_VALUE	outer wrapper does not have an <i>iV</i> of the correct size

```

851 TPM_RC
852 PrivateToSensitive(
853     TPM2B          *inPrivate,      // IN: input private structure
854     TPM2B          *name,          // IN: the name of the object
855     OBJECT         *parent,        // IN: parent object
856     TPM_ALG_ID     nameAlg,        // IN: hash algorithm in public area. It is
857                                     // passed separately because we only pass
858                                     // name, rather than the whole public area
859                                     // of the object. This parameter is used in
860                                     // the following two cases: 1. primary
861                                     // objects. 2. duplication blob with inner

```

```

862                                     //      wrap. In other cases, this parameter
863                                     //      will be ignored
864     TPMT_SENSITIVE *sensitive        // OUT: sensitive structure
865 )
866 {
867     TPM_RC      result;
868     BYTE        *buffer;
869     INT32        size;
870     BYTE        *sensitiveData; // pointer to the sensitive data
871     UINT16       dataSize;
872     UINT16       dataSizeInput;
873     TPMI_ALG_HASH hashAlg;      // hash algorithm for integrity
874     UINT16       integritySize;
875     UINT16       ivSize;
876 //
877 // Make sure that name is provided
878 pAssert(name != NULL && name->size != 0);
879
880 // Find the hash algorithm for integrity computation
881 // For Temporary Object (parent == NULL) use self name algorithm;
882 // Otherwise, using parent's name algorithm
883 hashAlg = (parent == NULL) ? nameAlg : parent->publicArea.nameAlg;
884
885 // unwrap outer
886 result = UnwrapOuter(parent, name, hashAlg, NULL, TRUE,
887                     inPrivate->size, inPrivate->buffer);
888 if(result != TPM_RC_SUCCESS)
889     return result;
890 // Compute the inner integrity size.
891 integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
892
893 // Get iv size
894 ivSize = GetIV2BSize(parent);
895
896 // The starting of sensitive data and data size without outer wrapper
897 sensitiveData = inPrivate->buffer + integritySize + ivSize;
898 dataSize = inPrivate->size - integritySize - ivSize;
899
900 // Unmarshal input data size
901 buffer = sensitiveData;
902 size = (INT32)dataSize;
903 result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
904 if(result == TPM_RC_SUCCESS)
905 {
906     if((dataSizeInput + sizeof(UINT16)) != dataSize)
907         result = TPM_RC_SENSITIVE;
908     else
909     {
910         // Unmarshal sensitive buffer to sensitive structure
911         result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
912         if(result != TPM_RC_SUCCESS || size != 0)
913         {
914             result = TPM_RC_SENSITIVE;
915         }
916     }
917 }
918 return result;
919 }

```

7.6.3.13 SensitiveToDuplicate()

This function prepare the duplication blob from the sensitive area. The operations in this function:

- 1) marshal TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE

2) apply inner wrap to the sensitive area if required

3) apply outer wrap if required

```

920 void
921 SensitiveToDuplicate(
922     TPMT_SENSITIVE *sensitive,    // IN: sensitive structure
923     TPM2B *name,                 // IN: the name of the object
924     OBJECT *parent,              // IN: The new parent object
925     TPM_ALG_ID nameAlg,          // IN: hash algorithm in public area. It
926                                   // is passed separately because we
927                                   // only pass name, rather than the
928                                   // whole public area of the object.
929     TPM2B *seed,                 // IN: the external seed. If external
930                                   // seed is provided with size of 0,
931                                   // no outer wrap should be applied
932                                   // to duplication blob.
933     TPMT_SYM_DEF_OBJECT *symDef,  // IN: Symmetric key definition. If the
934                                   // symmetric key algorithm is NULL,
935                                   // no inner wrap should be applied.
936     TPM2B_DATA *innerSymKey,      // IN/OUT: a symmetric key may be
937                                   // provided to encrypt the inner
938                                   // wrap of a duplication blob. May
939                                   // be generated here if needed.
940     TPM2B_PRIVATE *outPrivate     // OUT: output private structure
941 )
942 {
943     BYTE *sensitiveData; // pointer to the sensitive data
944     TPMI_ALG_HASH outerHash = TPM_ALG_NULL; // The hash algorithm for outer wrap
945     TPMI_ALG_HASH innerHash = TPM_ALG_NULL; // The hash algorithm for inner wrap
946     UINT16 dataSize;      // data blob size
947     BOOL doInnerWrap = FALSE;
948     BOOL doOuterWrap = FALSE;
949     //
950     // Make sure that name is provided
951     pAssert(name != NULL && name->size != 0);
952
953     // Make sure symDef and innerSymKey are not NULL
954     pAssert(symDef != NULL && innerSymKey != NULL);
955
956     // Starting of sensitive data without wrappers
957     sensitiveData = outPrivate->t.buffer;
958
959     // Find out if inner wrap is required
960     if(symDef->algorithm != TPM_ALG_NULL)
961     {
962         doInnerWrap = TRUE;
963
964         // Use self nameAlg as inner hash algorithm
965         innerHash = nameAlg;
966
967         // Adjust sensitive data pointer
968         sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(innerHash);
969     }
970     // Find out if outer wrap is required
971     if(seed->size != 0)
972     {
973         doOuterWrap = TRUE;
974
975         // Use parent nameAlg as outer hash algorithm
976         outerHash = parent->publicArea.nameAlg;
977
978         // Adjust sensitive data pointer
979         sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
980     }
981     // Marshal sensitive area

```



```

982     dataSize = MarshalSensitive(NULL, sensitiveData, sensitive, nameAlg);
983
984     // Apply inner wrap for duplication blob. It includes both integrity and
985     // encryption
986     if(doInnerWrap)
987     {
988         BYTE          *innerBuffer = NULL;
989         BOOL          symKeyInput = TRUE;
990         innerBuffer = outPrivate->t.buffer;
991         // Skip outer integrity space
992         if(doOuterWrap)
993             innerBuffer += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
994         dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
995                                         innerBuffer);
996         // Generate inner encryption key if needed
997         if(innerSymKey->t.size == 0)
998         {
999             innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
1000             CryptRandomGenerate(innerSymKey->t.size, innerSymKey->t.buffer);
1001
1002             // TPM generates symmetric encryption. Set the flag to FALSE
1003             symKeyInput = FALSE;
1004         }
1005         else
1006         {
1007             // assume the input key size should matches the symmetric definition
1008             pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1009         }
1010         // Encrypt inner buffer in place
1011         CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
1012                               symDef->keyBits.sym, innerSymKey->t.buffer, NULL,
1013                               TPM_ALG_CFB, dataSize, innerBuffer);
1014
1015         // If the symmetric encryption key is imported, clear the buffer for
1016         // output
1017         if(symKeyInput)
1018             innerSymKey->t.size = 0;
1019     }
1020     // Apply outer wrap for duplication blob. It includes both integrity and
1021     // encryption
1022     if(doOuterWrap)
1023     {
1024         dataSize = ProduceOuterWrap(parent, name, outerHash, seed, FALSE,
1025                                     dataSize, outPrivate->t.buffer);
1026     }
1027     // Data size for output
1028     outPrivate->t.size = dataSize;
1029
1030     return;
1031 }

```

7.6.3.14 DuplicateToSensitive()

Unwrap a duplication blob. Check the integrity, decrypt and retrieve data to a sensitive structure. The operations in this function:

- 1) check the integrity HMAC of the input private area
- 2) decrypt the private buffer
- 3) unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

Error Return	Meaning
TPM_RC_INSUFFICIENT	unmarshaling sensitive data from <i>inPrivate</i> failed
TPM_RC_INTEGRITY	<i>inPrivate</i> data integrity is broken
TPM_RC_SIZE	unmarshaling sensitive data from <i>inPrivate</i> failed

```

1032 TPM_RC
1033 DuplicateToSensitive(
1034     TPM2B          *inPrivate,      // IN: input private structure
1035     TPM2B          *name,           // IN: the name of the object
1036     OBJECT         *parent,         // IN: the parent
1037     TPM_ALG_ID     nameAlg,         // IN: hash algorithm in public area.
1038     TPM2B          *seed,           // IN: an external seed may be provided.
1039                                     // If external seed is provided with
1040                                     // size of 0, no outer wrap is
1041                                     // applied
1042     TPMT_SYM_DEF_OBJECT *symDef,     // IN: Symmetric key definition. If the
1043                                     // symmetric key algorithm is NULL,
1044                                     // no inner wrap is applied
1045     TPM2B          *innerSymKey,     // IN: a symmetric key may be provided
1046                                     // to decrypt the inner wrap of a
1047                                     // duplication blob.
1048     TPMT_SENSITIVE *sensitive       // OUT: sensitive structure
1049 )
1050 {
1051     TPM_RC      result;
1052     BYTE        *buffer;
1053     INT32       size;
1054     BYTE        *sensitiveData; // pointer to the sensitive data
1055     UINT16      dataSize;
1056     UINT16      dataSizeInput;
1057     //
1058     // Make sure that name is provided
1059     pAssert(name != NULL && name->size != 0);
1060
1061     // Make sure symDef and innerSymKey are not NULL
1062     pAssert(symDef != NULL && innerSymKey != NULL);
1063
1064     // Starting of sensitive data
1065     sensitiveData = inPrivate->buffer;
1066     dataSize = inPrivate->size;
1067
1068     // Find out if outer wrap is applied
1069     if(seed->size != 0)
1070     {
1071         // Use parent nameAlg as outer hash algorithm
1072         TPMT_ALG_HASH outerHash = parent->publicArea.nameAlg;
1073
1074         result = UnwrapOuter(parent, name, outerHash, seed, FALSE,
1075                             dataSize, sensitiveData);
1076         if(result != TPM_RC_SUCCESS)
1077             return result;
1078         // Adjust sensitive data pointer and size
1079         sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1080         dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1081     }
1082     // Find out if inner wrap is applied
1083     if(symDef->algorithm != TPM_ALG_NULL)
1084     {
1085         // assume the input key size matches the symmetric definition
1086         pAssert(innerSymKey->size == (symDef->keyBits.sym + 7) / 8);
1087
1088         // Decrypt inner buffer in place
1089         CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,

```

```

1090         symDef->keyBits.sym, innerSymKey->buffer, NULL,
1091         TPM_ALG_CFB, dataSize, sensitiveData);
1092     // Check inner integrity
1093     result = CheckInnerIntegrity(name, nameAlg, dataSize, sensitiveData);
1094     if(result != TPM_RC_SUCCESS)
1095         return result;
1096     // Adjust sensitive data pointer and size
1097     sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1098     dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1099 }
1100 // Unmarshal input data size
1101 buffer = sensitiveData;
1102 size = (INT32)dataSize;
1103 result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1104 if(result == TPM_RC_SUCCESS)
1105 {
1106     if((dataSizeInput + sizeof(UINT16)) != dataSize)
1107         result = TPM_RC_SIZE;
1108     else
1109     {
1110         // Unmarshal sensitive buffer to sensitive structure
1111         result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1112
1113         // if the results is OK make sure that all the data was unmarshaled
1114         if(result == TPM_RC_SUCCESS && size != 0)
1115             result = TPM_RC_SIZE;
1116     }
1117 }
1118 return result;
1119 }

```

7.6.3.15 SecretToCredential()

This function prepare the credential blob from a secret (a TPM2B_DIGEST) The operations in this function:

- 1) marshal TPM2B_DIGEST structure into the buffer of TPM2B_ID_OBJECT
- 2) encrypt the private buffer, excluding the leading integrity HMAC area
- 3) compute integrity HMAC and append to the beginning of the buffer.
- 4) Set the total size of TPM2B_ID_OBJECT buffer

```

1120 void
1121 SecretToCredential(
1122     TPM2B_DIGEST      *secret,           // IN: secret information
1123     TPM2B              *name,            // IN: the name of the object
1124     TPM2B              *seed,            // IN: an external seed.
1125     OBJECT             *protector,       // IN: the protector
1126     TPM2B_ID_OBJECT    *outIDObject      // OUT: output credential
1127 )
1128 {
1129     BYTE               *buffer;          // Auxiliary buffer pointer
1130     BYTE               *sensitiveData;  // pointer to the sensitive data
1131     TPMI_ALG_HASH      outerHash;        // The hash algorithm for outer wrap
1132     UINT16              dataSize;        // data blob size
1133     //
1134     pAssert(secret != NULL && outIDObject != NULL);
1135
1136     // use protector's name algorithm as outer hash ???
1137     outerHash = protector->publicArea.nameAlg;
1138
1139     // Marshal secret area to credential buffer, leave space for integrity
1140     sensitiveData = outIDObject->t.credential

```

```

1141         + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1142 // Marshal secret area
1143     buffer = sensitiveData;
1144     dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1145
1146     // Apply outer wrap
1147     outIDObject->t.size = ProduceOuterWrap(protector, name, outerHash, seed, FALSE,
1148                                           dataSize, outIDObject->t.credential);
1149     return;
1150 }

```

7.6.3.16 CredentialToSecret()

Unwrap a credential. Check the integrity, decrypt and retrieve data to a TPM2B_DIGEST structure. The operations in this function:

- 1) check the integrity HMAC of the input credential area
- 2) decrypt the credential buffer
- 3) unmarshal TPM2B_DIGEST structure into the buffer of TPM2B_DIGEST

Error Return	Meaning
TPM_RC_INSUFFICIENT	error during credential unmarshaling
TPM_RC_INTEGRITY	credential integrity is broken
TPM_RC_SIZE	error during credential unmarshaling
TPM_RC_VALUE	IV size does not match the encryption algorithm block size

```

1151 TPM_RC
1152 CredentialToSecret(
1153     TPM2B             *inIDObject,    // IN: input credential blob
1154     TPM2B             *name,          // IN: the name of the object
1155     TPM2B             *seed,          // IN: an external seed.
1156     OBJECT            *protector,     // IN: the protector
1157     TPM2B_DIGEST       *secret        // OUT: secret information
1158 )
1159 {
1160     TPM_RC             result;
1161     BYTE               *buffer;
1162     INT32              size;
1163     TPMI_ALG_HASH      outerHash;     // The hash algorithm for outer wrap
1164     BYTE               *sensitiveData; // pointer to the sensitive data
1165     UINT16             dataSize;
1166 //
1167 // use protector's name algorithm as outer hash
1168     outerHash = protector->publicArea.nameAlg;
1169
1170 // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
1171     result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
1172                          inIDObject->size, inIDObject->buffer);
1173     if(result == TPM_RC_SUCCESS)
1174     {
1175         // Compute the beginning of sensitive data
1176         sensitiveData = inIDObject->buffer
1177             + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1178         dataSize = inIDObject->size
1179             - (sizeof(UINT16) + CryptHashGetDigestSize(outerHash));
1180         // Unmarshal secret buffer to TPM2B_DIGEST structure
1181         buffer = sensitiveData;
1182         size = (INT32)dataSize;
1183         result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
1184     }

```

```

1185         // If there were no other unmarshaling errors, make sure that the
1186         // expected amount of data was recovered
1187         if(result == TPM_RC_SUCCESS && size != 0)
1188             return TPM_RC_SIZE;
1189     }
1190     return result;
1191 }

```

7.6.3.17 MemoryRemoveTrailingZeros()

This function is used to adjust the length of an authorization value. It adjusts the size of the TPM2B so that it does not include octets at the end of the buffer that contain zero. The function returns the number of non-zero octets in the buffer.

```

1192 UINT16
1193 MemoryRemoveTrailingZeros(
1194     TPM2B_AUTH *auth           // IN/OUT: value to adjust
1195 )
1196 {
1197     while((auth->t.size > 0) && (auth->t.buffer[auth->t.size - 1] == 0))
1198         auth->t.size--;
1199     return auth->t.size;
1200 }

```

7.6.3.18 SetLabelAndContext()

This function sets the label and context for a derived key. It is possible that *label* or *context* can end up being an Empty Buffer.

```

1201 TPM_RC
1202 SetLabelAndContext(
1203     TPMS_DERIVE *labelContext, // IN/OUT: the recovered label and
1204                               // context
1205     TPM2B_SENSITIVE_DATA *sensitive // IN: the sensitive data
1206 )
1207 {
1208     TPMS_DERIVE sensitiveValue;
1209     TPM_RC result;
1210     INT32 size;
1211     BYTE *buff;
1212     //
1213     // Unmarshal a TPMS_DERIVE from the TPM2B_SENSITIVE_DATA buffer
1214     // If there is something to unmarshal...
1215     if(sensitive->t.size != 0)
1216     {
1217         size = sensitive->t.size;
1218         buff = sensitive->t.buffer;
1219         result = TPMS_DERIVE_Unmarshal(&sensitiveValue, &buff, &size);
1220         if(result != TPM_RC_SUCCESS)
1221             return result;
1222         // If there was a label in the public area leave it there, otherwise, copy
1223         // the new value
1224         if(labelContext->label.t.size == 0)
1225             MemoryCopy2B(&labelContext->label.b, &sensitiveValue.label.b,
1226                         sizeof(labelContext->label.t.buffer));
1227         // if there was a context string in publicArea, it overrides
1228         if(labelContext->context.t.size == 0)
1229             MemoryCopy2B(&labelContext->context.b, &sensitiveValue.context.b,
1230                         sizeof(labelContext->label.t.buffer));
1231     }
1232     return TPM_RC_SUCCESS;
1233 }

```

7.6.3.19 UnmarshalToPublic()

Support function to unmarshal the template. This is used because the Input may be a TPMT_TEMPLATE and that structure does not have the same size as a TPMT_PUBLIC because of the difference between the *unique* and *seed* fields. If *derive* is not NULL, then the *seed* field is assumed to contain a *label* and *context* that are unmarshaled into *derive*.

```

1234 TPM_RC
1235 UnmarshalToPublic(
1236     TPMT_PUBLIC      *tOut,          // OUT: output
1237     TPM2B_TEMPLATE    *tIn,          // IN:
1238     BOOL              derivation,     // IN: indicates if this is for a derivation
1239     TPMS_DERIVE       *labelContext,  // OUT: label and context if derivation
1240 )
1241 {
1242     BYTE              *buffer = tIn->t.buffer;
1243     INT32              size = tIn->t.size;
1244     TPM_RC             result;
1245     //
1246     // make sure that tOut is zeroed so that there are no remnants from previous
1247     // uses
1248     MemorySet(tOut, 0, sizeof(TPMT_PUBLIC));
1249     // Unmarshal the components of the TPMT_PUBLIC up to the unique field
1250     result = TPMI_ALG_PUBLIC_Unmarshal(&tOut->type, &buffer, &size);
1251     if(result != TPM_RC_SUCCESS)
1252         return result;
1253     result = TPMI_ALG_HASH_Unmarshal(&tOut->nameAlg, &buffer, &size, FALSE);
1254     if(result != TPM_RC_SUCCESS)
1255         return result;
1256     result = TPMA_OBJECT_Unmarshal(&tOut->objectAttributes, &buffer, &size);
1257     if(result != TPM_RC_SUCCESS)
1258         return result;
1259     result = TPM2B_DIGEST_Unmarshal(&tOut->authPolicy, &buffer, &size);
1260     if(result != TPM_RC_SUCCESS)
1261         return result;
1262     result = TPMU_PUBLIC_PARMS_Unmarshal(&tOut->parameters, &buffer, &size,
1263                                         tOut->type);
1264     if(result != TPM_RC_SUCCESS)
1265         return result;
1266     // Now unmarshal a TPMS_DERIVE if this is for derivation
1267     if(derivation)
1268         result = TPMS_DERIVE_Unmarshal(labelContext, &buffer, &size);
1269     else
1270         // otherwise, unmarshal a TPMU_PUBLIC_ID
1271         result = TPMU_PUBLIC_ID_Unmarshal(&tOut->unique, &buffer, &size,
1272                                         tOut->type);
1273     // Make sure the template was used up
1274     if((result == TPM_RC_SUCCESS) && (size != 0))
1275         result = TPM_RC_SIZE;
1276     return result;
1277 }

```

7.6.3.20 ObjectSetExternal()

Set the external attributes for an object.

```

1278 void
1279 ObjectSetExternal(
1280     OBJECT      *object
1281 )
1282 {
1283     object->attributes.external = SET;
1284 }

```

7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c)

```

1  #include "Tpm.h"
2  #include "EncryptDecrypt_fp.h"
3  #include "EncryptDecrypt_spt_fp.h"
4
5  #if CC_EncryptDecrypt2

```

Error Return	Meaning
TPM_RC_KEY	is not a symmetric decryption key with both public and private portions loaded
TPM_RC_SIZE	<i>ivIn</i> size is incompatible with the block cipher mode; or <i>inData</i> size is not an even multiple of the block size for CBC or ECB mode
TPM_RC_VALUE	<i>keyHandle</i> is restricted and the argument <i>mode</i> does not match the key's mode

```

6  TPM_RC
7  EncryptDecryptShared(
8      TPMI_DH_OBJECT      keyHandleIn,
9      TPMI_YES_NO         decryptIn,
10     TPMI_ALG_SYM_MODE    modeIn,
11     TPM2B_IV             *ivIn,
12     TPM2B_MAX_BUFFER     *inData,
13     EncryptDecrypt_Out    *out
14 )
15 {
16     OBJECT                *symKey;
17     UINT16                keySize;
18     UINT16                blockSize;
19     BYTE                  *key;
20     TPM_ALG_ID            alg;
21     TPM_ALG_ID            mode;
22     TPM_RC                result;
23     BOOL                  OK;
24     // Input Validation
25     symKey = HandleToObject(keyHandleIn);
26     mode = symKey->publicArea.parameters.symDetail.sym.mode.sym;
27
28     // The input key should be a symmetric key
29     if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
30         return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
31     // The key must be unrestricted and allow the selected operation
32     OK = !IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
33                       TPMA_OBJECT, restricted);
34     if(YES == decryptIn)
35         OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
36                                TPMA_OBJECT, decrypt);
37     else
38         OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
39                                TPMA_OBJECT, sign);
40     if(!OK)
41         return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;
42
43     // Make sure that key is an encrypt/decrypt key and not SMAC
44     if(!CryptSymModeIsValid(mode, TRUE))
45         return TPM_RCS_MODE + RC_EncryptDecrypt_keyHandle;
46
47     // If the key mode is not TPM_ALG_NULL...
48     // or TPM_ALG_NULL
49     if(mode != TPM_ALG_NULL)

```

```

50     {
51         // then the input mode has to be TPM_ALG_NULL or the same as the key
52         if((modeIn != TPM_ALG_NULL) && (modeIn != mode))
53             return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
54     }
55     else
56     {
57         // if the key mode is null, then the input can't be null
58         if(modeIn == TPM_ALG_NULL)
59             return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
60         mode = modeIn;
61     }
62     // The input iv for ECB mode should be an Empty Buffer. All the other modes
63     // should have an iv size same as encryption block size
64     keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
65     alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
66     blockSize = CryptGetSymmetricBlockSize(alg, keySize);
67
68     // reverify the algorithm. This is mainly to keep static analysis tools happy
69     if(blockSize == 0)
70         return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
71
72     if(((mode == TPM_ALG_ECB) && (ivIn->t.size != 0))
73         || ((mode != TPM_ALG_ECB) && (ivIn->t.size != blockSize)))
74         return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;
75
76     // The input data size of CBC mode or ECB mode must be an even multiple of
77     // the symmetric algorithm's block size
78     if(((mode == TPM_ALG_CBC) || (mode == TPM_ALG_ECB))
79         && ((inData->t.size % blockSize) != 0))
80         return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;
81
82     // Copy IV
83     // Note: This is copied here so that the calls to the encrypt/decrypt functions
84     // will modify the output buffer, not the input buffer
85     out->ivOut = *ivIn;
86
87     // Command Output
88     key = symKey->sensitive.sensitive.sym.t.buffer;
89     // For symmetric encryption, the cipher data size is the same as plain data
90     // size.
91     out->outData.t.size = inData->t.size;
92     if(decryptIn == YES)
93     {
94         // Decrypt data to output
95         result = CryptSymmetricDecrypt(out->outData.t.buffer, alg, keySize, key,
96                                     &(out->ivOut), mode, inData->t.size,
97                                     inData->t.buffer);
98     }
99     else
100     {
101         // Encrypt data to output
102         result = CryptSymmetricEncrypt(out->outData.t.buffer, alg, keySize, key,
103                                     &(out->ivOut), mode, inData->t.size,
104                                     inData->t.buffer);
105     }
106     return result;
107 }
108 #endif // CC_EncryptDecrypt

```


7.8 ACT Support (ACT_spt.c)

7.8.1 Introduction

This code implements the ACT update code. It does not use a mutex. This code uses a platform service (`_plat__ACT_UpdateCounter()`) that returns *false* if the update is not accepted. If this occurs, then `TPM_RC_RETRY` should be sent to the caller so that they can retry the operation later. The implementation of this is platform dependent but the reference uses a simple flag to indicate that an update is pending and the only process that can clear that flag is the process that does the actual update.

7.8.2 Includes

```
1  #include "Tpm.h"
2  #include "ACT_spt_fp.h"
3  #include "Platform_fp.h"
```

7.8.3 Functions

7.8.3.1 _ActResume()

This function does the resume processing for an ACT. It updates the saved count and turns signaling back on if necessary.

```
4  static void
5  _ActResume(
6      UINT32          act,           //IN: the act number
7      ACT_STATE      *actData      //IN: pointer to the saved ACT data
8  )
9  {
10     // If the act was non-zero, then restore the counter value.
11     if(actData->remaining > 0)
12         _plat__ACT_UpdateCounter(act, actData->remaining);
13     // if the counter was zero and the ACT signaling, enable the signaling.
14     else if(go.signaledACT & (1 << act))
15         _plat__ACT_SetSignaled(act, TRUE);
16 }
```

7.8.3.2 ActStartup()

This function is called by `TPM2_Startup()` to initialize the ACT counter values.

```
17  BOOL
18  ActStartup(
19      STARTUP_TYPE    type
20  )
21  {
22     // Reset all the ACT hardware
23     _plat__ACT_Initialize();
24
25     // If this not a cold start, copy all the current 'signaled' settings to
26     // 'preservedSignaled'.
27     if (g_powerWasLost)
28         go.preservedSignaled = 0;
29     else
30         go.preservedSignaled |= go.signaledACT;
31
32     // For TPM_RESET or TPM_RESTART, the ACTs will all be disabled and the output
33     // de-asserted.
```

```

34     if(type != SU_RESUME)
35     {
36         go.signaledACT = 0;
37     #define CLEAR_ACT_POLICY(N)
38         go.ACT_##N.hashAlg = TPM_ALG_NULL;
39         go.ACT_##N.authPolicy.b.size = 0;
40         FOR_EACH_ACT(CLEAR_ACT_POLICY)
41     }
42     else
43     {
44         // Resume each of the implemented ACT
45     #define RESUME_ACT(N) _ActResume(0x##N, &go.ACT_##N);
46
47         FOR_EACH_ACT(RESUME_ACT)
48     }
49     // set no ACT updated since last startup. This is to enable the halving of the
50     // timeout value
51     s_ActUpdated = 0;
52     _plat__ACT_EnableTicks(TRUE);
53     return TRUE;
54 }

```

7.8.3.3 _ActSaveState()

Get the counter state and the signaled state for an ACT. If the ACT has not been updated since the last time it was saved, then divide the count by 2.

```

55 static void
56 _ActSaveState(
57     UINT32          act,
58     P_ACT_STATE     actData
59 )
60 {
61     actData->remaining = _plat__ACT_GetRemaining(act);
62     // If the ACT hasn't been updated since the last startup, then it should be
63     // be halved.
64     if((s_ActUpdated & (1 << act)) == 0)
65     {
66         // Don't halve if the count is set to max or if halving would make it zero
67         if((actData->remaining != UINT32_MAX) && (actData->remaining > 1))
68             actData->remaining /= 2;
69     }
70     if(_plat__ACT_GetSignaled(act))
71         go.signaledACT |= (1 << act);
72 }

```

7.8.3.4 ActGetSignaled()

This function returns the state of the signaled flag associated with an ACT.

```

73 BOOL
74 ActGetSignaled(
75     TPM_RH          actHandle
76 )
77 {
78     UINT32          act = actHandle - TPM_RH_ACT_0;
79     //
80     return _plat__ACT_GetSignaled(act);
81 }

```

7.8.3.5 ActShutdown()

This function saves the current state of the counters

```

82  BOOL
83  ActShutdown(
84      TPM_SU          state          //IN: the type of the shutdown.
85  )
86  {
87      // if this is not shutdown state, then the only type of startup is TPM_RESTART
88      // so the timer values will be cleared. If this is shutdown state, get the current
89      // countdown and signaled values. Plus, if the counter has not been updated
90      // since the last restart, divide the time by 2 so that there is no attack on the
91      // countdown by saving the countdown state early and then not using the TPM.
92      if(state == TPM_SU_STATE)
93      {
94          // This will be populated as each of the ACT is queried
95          go.signaledACT = 0;
96          // Get the current count and the signaled state
97      #define SAVE_ACT_STATE(N) _ActSaveState(0x##N, &go.ACT_##N);
98
99          FOR_EACH_ACT(SAVE_ACT_STATE);
100     }
101     return TRUE;
102 }

```

7.8.3.6 ActIsImplemented()

This function determines if an ACT is implemented in both the TPM and the platform code.

```

103  BOOL
104  ActIsImplemented(
105      UINT32          act
106  )
107  {
108      // This switch accounts for the TPM implemented values.
109      switch(act)
110      {
111          FOR_EACH_ACT(CASE_ACT_NUMBER)
112          // This ensures that the platform implements the values implemented by
113          // the TPM
114          return _plat__ACT_GetImplemented(act);
115          default:
116              break;
117      }
118      return FALSE;
119 }

```

7.8.3.7 ActCounterUpdate()

This function updates the ACT counter. If the counter already has a pending update, it returns TPM_RC_RETRY so that the update can be tried again later.

```

120  TPM_RC
121  ActCounterUpdate(
122      TPM_RH          handle,          //IN: the handle of the act
123      UINT32          newValue         //IN: the value to set in the ACT
124  )
125  {
126      UINT32          act;
127      TPM_RC          result;
128      //

```

```

129     act = handle - TPM_RH_ACT_0;
130     // This should never fail, but...
131     if(!_plat__ACT_GetImplemented(act))
132         result = TPM_RC_VALUE;
133     else
134     {
135         // Will need to clear orderly so fail if we are orderly and NV is
136         // not available
137         if(NV_IS_ORDERLY)
138             RETURN_IF_NV_IS_NOT_AVAILABLE;
139         // if the attempt to update the counter fails, it means that there is an
140         // update pending so wait until it has occurred and then do an update.
141         if(!_plat__ACT_UpdateCounter(act, newValue))
142             result = TPM_RC_RETRY;
143         else
144         {
145             // Indicate that the ACT has been updated since last TPM2_Startup().
146             s_ActUpdated |= (UINT16)(1 << act);
147
148             // Clear the preservedSignaled attribute.
149             go.preservedSignaled &= ~((UINT16)(1 << act));
150
151             // Need to clear the orderly flag
152             g_clearOrderly = TRUE;
153
154             result = TPM_RC_SUCCESS;
155         }
156     }
157     return result;
158 }

```

7.8.3.8 ActGetCapabilityData()

This function returns the list of ACT data

Return Value	Meaning
YES	if more ACT data is available
NO	if no more ACT data to

```

159 TPMI_YES_NO
160 ActGetCapabilityData(
161     TPM_HANDLE    actHandle,        // IN: the handle for the starting ACT
162     UINT32        maxCount,         // IN: maximum allowed return values
163     TPML_ACT_DATA *actList          // OUT: ACT data list
164 )
165 {
166     // Initialize output property list
167     actList->count = 0;
168
169     // Make sure that the starting handle value is in range (again)
170     if((actHandle < TPM_RH_ACT_0) || (actHandle > TPM_RH_ACT_F))
171         return FALSE;
172     // The maximum count of curves we may return is MAX_ECC_CURVES
173     if(maxCount > MAX_ACT_DATA)
174         maxCount = MAX_ACT_DATA;
175     // Scan the ACT data from the starting ACT
176     for(; actHandle <= TPM_RH_ACT_F; actHandle++)
177     {
178         UINT32    act = actHandle - TPM_RH_ACT_0;
179         if(actList->count < maxCount)
180         {
181             if(ActIsImplemented(act))

```

```

182     {
183         TPMS_ACT_DATA    *actData = &actList->actData[actList->count];
184         //
185         memset(&actData->attributes, 0, sizeof(actData->attributes));
186         actData->handle = actHandle;
187         actData->timeout = _plat_ACT_GetRemaining(act);
188         if(_plat_ACT_GetSignaled(act))
189             SET_ATTRIBUTE(actData->attributes, TPMA_ACT, signaled);
190         else
191             CLEAR_ATTRIBUTE(actData->attributes, TPMA_ACT, signaled);
192         if (go.preservedSignaled & (1 << act))
193             SET_ATTRIBUTE(actData->attributes, TPMA_ACT, preserveSignaled);
194         actList->count++;
195     }
196 }
197 else
198 {
199     if(_plat_ACT_GetImplemented(act))
200         return YES;
201 }
202 }
203 // If we get here, either all of the ACT values were put in the list, or the list
204 // was filled and there are no more ACT values to return
205 return NO;
206 }

```

8 Subsystem

8.1 CommandAudit.c

8.1.1 Introduction

This file contains the functions that support command audit.

8.1.2 Includes

```
1 #include "Tpm.h"
```

8.1.3 Functions

8.1.3.1 CommandAuditPreInstall_Init()

This function initializes the command audit list. This function simulates the behavior of manufacturing. A function is used instead of a structure definition because this is easier than figuring out the initialization value for a bit array.

This function would not be implemented outside of a manufacturing or simulation environment.

```
2 void
3 CommandAuditPreInstall_Init(
4     void
5 )
6 {
7     // Clear all the audit commands
8     MemorySet(gp.auditCommands, 0x00, sizeof(gp.auditCommands));
9
10    // TPM_CC_SetCommandCodeAuditStatus always being audited
11    CommandAuditSet(TPM_CC_SetCommandCodeAuditStatus);
12
13    // Set initial command audit hash algorithm to be context integrity hash
14    // algorithm
15    gp.auditHashAlg = CONTEXT_INTEGRITY_HASH_ALG;
16
17    // Set up audit counter to be 0
18    gp.auditCounter = 0;
19
20    // Write command audit persistent data to NV
21    NV_SYNC_PERSISTENT(auditCommands);
22    NV_SYNC_PERSISTENT(auditHashAlg);
23    NV_SYNC_PERSISTENT(auditCounter);
24
25    return;
26 }
```

8.1.3.2 CommandAuditStartup()

This function clears the command audit digest on a TPM Reset.

```
27 BOOL
28 CommandAuditStartup(
29     STARTUP_TYPE    type           // IN: start up type
30 )
31 {
32     if((type != SU_RESTART) && (type != SU_RESUME))
```

```

33     {
34         // Reset the digest size to initialize the digest
35         gr.commandAuditDigest.t.size = 0;
36     }
37     return TRUE;
38 }

```

8.1.3.3 CommandAuditSet()

This function will SET the audit flag for a command. This function will not SET the audit flag for a command that is not implemented. This ensures that the audit status is not SET when TPM2_GetCapability() is used to read the list of audited commands.

This function is only used by TPM2_SetCommandCodeAuditStatus().

The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.

Return Value	Meaning
TRUE(1)	command code audit status was changed
FALSE(0)	command code audit status was not changed

```

39  BOOL
40  CommandAuditSet(
41      TPM_CC          commandCode    // IN: command code
42  )
43  {
44      COMMAND_INDEX    commandIndex = CommandCodeToCommandIndex(commandCode);
45
46      // Only SET a bit if the corresponding command is implemented
47      if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
48      {
49          // Can't audit shutdown
50          if(commandCode != TPM_CC_Shutdown)
51          {
52              if(!TEST_BIT(commandIndex, gp.auditCommands))
53              {
54                  // Set bit
55                  SET_BIT(commandIndex, gp.auditCommands);
56                  return TRUE;
57              }
58          }
59      }
60      // No change
61      return FALSE;
62  }

```

8.1.3.4 CommandAuditClear()

This function will CLEAR the audit flag for a command. It will not CLEAR the audit flag for TPM_CC_SetCommandCodeAuditStatus().

This function is only used by TPM2_SetCommandCodeAuditStatus().

The actions in TPM2_SetCommandCodeAuditStatus() are expected to cause the changes to be saved to NV after it is setting and clearing bits.

Return Value	Meaning
TRUE(1)	command code audit status was changed
FALSE(0)	command code audit status was not changed

```

63  BOOL
64  CommandAuditClear(
65      TPM_CC      commandCode    // IN: command code
66  )
67  {
68      COMMAND_INDEX      commandIndex = CommandCodeToCommandIndex(commandCode);
69
70      // Do nothing if the command is not implemented
71      if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
72      {
73          // The bit associated with TPM_CC_SetCommandCodeAuditStatus() cannot be
74          // cleared
75          if(commandCode != TPM_CC_SetCommandCodeAuditStatus)
76          {
77              if(TEST_BIT(commandIndex, gp.auditCommands))
78              {
79                  // Clear bit
80                  CLEAR_BIT(commandIndex, gp.auditCommands);
81                  return TRUE;
82              }
83          }
84      }
85      // No change
86      return FALSE;
87  }

```

8.1.3.5 CommandAuditIsRequired()

This function indicates if the audit flag is SET for a command.

Return Value	Meaning
TRUE(1)	command is audited
FALSE(0)	command is not audited

```

88  BOOL
89  CommandAuditIsRequired(
90      COMMAND_INDEX      commandIndex    // IN: command index
91  )
92  {
93      // Check the bit map. If the bit is SET, command audit is required
94      return(TEST_BIT(commandIndex, gp.auditCommands));
95  }

```

8.1.3.6 CommandAuditCapGetCCList()

This function returns a list of commands that have their audit bit SET.

The list starts at the input *commandCode*.

Return Value	Meaning
YES	if there are more command code available
NO	all the available command code has been returned

```

96  TPMI_YES_NO
97  CommandAuditCapGetCCList(
98      TPM_CC      commandCode,    // IN: start command code
99      UINT32      count,          // IN: count of returned TPM_CC
100     TPML_CC      *commandList    // OUT: list of TPM_CC
101 )
102 {
103     TPMI_YES_NO    more = NO;
104     COMMAND_INDEX  commandIndex;
105
106     // Initialize output handle list
107     commandList->count = 0;
108
109     // The maximum count of command we may return is MAX_CAP_CC
110     if(count > MAX_CAP_CC) count = MAX_CAP_CC;
111
112     // Find the implemented command that has a command code that is the same or
113     // higher than the input
114     // Collect audit commands
115     for(commandIndex = GetClosestCommandIndex(commandCode);
116         commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
117         commandIndex = GetNextCommandIndex(commandIndex))
118     {
119         if(CommandAuditIsRequired(commandIndex))
120         {
121             if(commandList->count < count)
122             {
123                 // If we have not filled up the return list, add this command
124                 // code to its
125                 TPM_CC    cc = GET_ATTRIBUTE(s_ccAttr[commandIndex],
126                                             TPMA_CC, commandIndex);
127                 if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
128                     cc += (1 << 29);
129                 commandList->commandCodes[commandList->count] = cc;
130                 commandList->count++;
131             }
132             else
133             {
134                 // If the return list is full but we still have command
135                 // available, report this and stop iterating
136                 more = YES;
137                 break;
138             }
139         }
140     }
141     return more;
142 }

```

8.1.3.7 CommandAuditGetDigest()

This command is used to create a digest of the commands being audited. The commands are processed in ascending numeric order with a list of TPM_CC being added to a hash. This operates as if all the audited command codes were concatenated and then hashed.

```

143  void
144  CommandAuditGetDigest(
145      TPM2B_DIGEST  *digest        // OUT: command digest

```

```
146     )
147 {
148     TPM_CC                commandCode;
149     COMMAND_INDEX         commandIndex;
150     HASH_STATE            hashState;
151
152     // Start hash
153     digest->t.size = CryptHashStart(&hashState, gp.auditHashAlg);
154
155     // Add command code
156     for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
157     {
158         if(CommandAuditIsRequired(commandIndex))
159         {
160             commandCode = GetCommandCode(commandIndex);
161             CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
162         }
163     }
164     // Complete hash
165     CryptHashEnd2B(&hashState, &digest->b);
166
167     return;
168 }
```

8.2 DA.c

8.2.1 Introduction

This file contains the functions and data definitions relating to the dictionary attack logic.

8.2.2 Includes and Data Definitions

```
1  #define DA_C
2  #include "Tpm.h"
```

8.2.3 Functions

8.2.3.1 DAPreInstall_Init()

This function initializes the DA parameters to their manufacturer-default values. The default values are determined by a platform-specific specification.

This function should not be called outside of a manufacturing or simulation environment.

The DA parameters will be restored to these initial values by TPM2_Clear().

```
3  void
4  DAPreInstall_Init(
5      void
6  )
7  {
8      gp.failedTries = 0;
9      gp.maxTries = 3;
10     gp.recoveryTime = 1000;           // in seconds (~16.67 minutes)
11     gp.lockoutRecovery = 1000;       // in seconds
12     gp.lockOutAuthEnabled = TRUE;    // Use of lockoutAuth is enabled
13
14     // Record persistent DA parameter changes to NV
15     NV_SYNC_PERSISTENT(failedTries);
16     NV_SYNC_PERSISTENT(maxTries);
17     NV_SYNC_PERSISTENT(recoveryTime);
18     NV_SYNC_PERSISTENT(lockoutRecovery);
19     NV_SYNC_PERSISTENT(lockOutAuthEnabled);
20
21     return;
22 }
```

8.2.3.2 DASTartup()

This function is called by TPM2_Startup() to initialize the DA parameters. In the case of Startup(CLEAR), use of *lockoutAuth* will be enabled if the lockout recovery time is 0. Otherwise, *lockoutAuth* will not be enabled until the TPM has been continuously powered for the *lockoutRecovery* time. This function requires that NV be available and not rate limiting.

```
23  BOOL
24  DASTartup(
25      STARTUP_TYPE    type           // IN: startup type
26  )
27  {
28      NOT_REFERENCED(type);
29      #if !ACCUMULATE_SELF_HEAL_TIMER
30          _plat_TimerWasReset();
31          s_selfHealTimer = 0;
32      #endif
33  }
```

```

32     s_lockoutTimer = 0;
33 #else
34     if(_plat__TimerWasReset())
35     {
36         if(!NV_IS_ORDERLY)
37         {
38             // If shutdown was not orderly, then don't really know if go.time has
39             // any useful value so reset the timer to 0. This is what the tick
40             // was reset to
41             s_selfHealTimer = 0;
42             s_lockoutTimer = 0;
43         }
44         else
45         {
46             // If we know how much time was accumulated at the last orderly shutdown
47             // subtract that from the saved timer values so that they effectively
48             // have the accumulated values
49             s_selfHealTimer -= go.time;
50             s_lockoutTimer -= go.time;
51         }
52     }
53 #endif
54
55     // For any Startup(), if lockoutRecovery is 0, enable use of lockoutAuth.
56     if(gp.lockoutRecovery == 0)
57     {
58         gp.lockOutAuthEnabled = TRUE;
59         // Record the changes to NV
60         NV_SYNC_PERSISTENT(lockOutAuthEnabled);
61     }
62     // If DA has not been disabled and the previous shutdown is not orderly
63     // failedTries is not already at its maximum then increment 'failedTries'
64     if(gp.recoveryTime != 0
65        && gp.failedTries < gp.maxTries
66        && !IS_ORDERLY(g_prevOrderlyState))
67     {
68 #if USE_DA_USED
69         gp.failedTries += g_daUsed;
70         g_daUsed = FALSE;
71 #else
72         gp.failedTries++;
73 #endif
74         // Record the change to NV
75         NV_SYNC_PERSISTENT(failedTries);
76     }
77     // Before Startup, the TPM will not do clock updates. At startup, need to
78     // do a time update which will do the DA update.
79     TimeUpdate();
80
81     return TRUE;
82 }

```

8.2.3.3 DARegisterFailure()

This function is called when an authorization failure occurs on an entity that is subject to dictionary-attack protection. When a DA failure is triggered, register the failure by resetting the relevant self-healing timer to the current time.

```

83 void
84 DARegisterFailure(
85     TPM_HANDLE    handle          // IN: handle for failure
86 )
87 {
88     // Reset the timer associated with lockout if the handle is the lockoutAuth.

```

```

89     if(handle == TPM_RH_LOCKOUT)
90         s_lockoutTimer = g_time;
91     else
92         s_selfHealTimer = g_time;
93     return;
94 }

```

8.2.3.4 DASelfHeal()

This function is called to check if sufficient time has passed to allow decrement of *failedTries* or to re-enable use of *lockoutAuth*.

This function should be called when the time interval is updated.

```

95 void
96 DASelfHeal(
97     void
98 )
99 {
100     // Regular authorization self healing logic
101     // If no failed authorization tries, do nothing. Otherwise, try to
102     // decrease failedTries
103     if(gp.failedTries != 0)
104     {
105         // if recovery time is 0, DA logic has been disabled. Clear failed tries
106         // immediately
107         if(gp.recoveryTime == 0)
108         {
109             gp.failedTries = 0;
110             // Update NV record
111             NV_SYNC_PERSISTENT(failedTries);
112         }
113         else
114         {
115             UINT64 decreaseCount;
116             #if 0 // Errata eliminates this code
117                 // In the unlikely event that failedTries should become larger than
118                 // maxTries
119                 if(gp.failedTries > gp.maxTries)
120                     gp.failedTries = gp.maxTries;
121             #endif
122             // How much can failedTries be decreased
123             // Cast s_selfHealTimer to an int in case it became negative at
124             // startup
125             decreaseCount = ((g_time - (INT64)s_selfHealTimer) / 1000)
126                 / gp.recoveryTime;
127             if(gp.failedTries <= (UINT32)decreaseCount)
128                 // should not set failedTries below zero
129                 gp.failedTries = 0;
130             else
131                 gp.failedTries -= (UINT32)decreaseCount;
132             // the cast prevents overflow of the product
133             s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
134             if(decreaseCount != 0)
135                 // If there was a change to the failedTries, record the changes
136                 // to NV
137                 NV_SYNC_PERSISTENT(failedTries);
138         }
139     }
140     // LockoutAuth self healing logic
141     // If lockoutAuth is enabled, do nothing. Otherwise, try to see if we

```

```
145     // may enable it
146     if(!gp.lockOutAuthEnabled)
147     {
148         // if lockout authorization recovery time is 0, a reboot is required to
149         // re-enable use of lockout authorization. Self-healing would not
150         // apply in this case.
151         if(gp.lockoutRecovery != 0)
152         {
153             if(((g_time - (INT64)s_lockoutTimer) / 1000) >= gp.lockoutRecovery)
154             {
155                 gp.lockOutAuthEnabled = TRUE;
156                 // Record the changes to NV
157                 NV_SYNC_PERSISTENT(lockOutAuthEnabled);
158             }
159         }
160     }
161     return;
162 }
```


8.3 Hierarchy.c

8.3.1 Introduction

This file contains the functions used for managing and accessing the hierarchy-related values.

8.3.2 Includes

```
1 #include "Tpm.h"
```

8.3.3 Functions

8.3.3.1 HierarchyPreInstall()

This function performs the initialization functions for the hierarchy when the TPM is simulated. This function should not be called if the TPM is not in a manufacturing mode at the manufacturer, or in a simulated environment.

```
2 void
3 HierarchyPreInstall_Init(
4     void
5 )
6 {
7     // Allow lockout clear command
8     gp.disableClear = FALSE;
9
10    // Initialize Primary Seeds
11    gp.EPSeed.t.size = sizeof(gp.EPSeed.t.buffer);
12    gp.SPSeed.t.size = sizeof(gp.SPSeed.t.buffer);
13    gp.PPSeed.t.size = sizeof(gp.PPSeed.t.buffer);
14    #if (defined USE_PLATFORM_EPS) && (USE_PLATFORM_EPS != NO)
15        _plat_GetEPS(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
16    #else
17        CryptRandomGenerate(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
18    #endif
19    CryptRandomGenerate(gp.SPSeed.t.size, gp.SPSeed.t.buffer);
20    CryptRandomGenerate(gp.PPSeed.t.size, gp.PPSeed.t.buffer);
21
22    // Initialize owner, endorsement and lockout authorization
23    gp.ownerAuth.t.size = 0;
24    gp.endorsementAuth.t.size = 0;
25    gp.lockoutAuth.t.size = 0;
26
27    // Initialize owner, endorsement, and lockout policy
28    gp.ownerAlg = TPM_ALG_NULL;
29    gp.ownerPolicy.t.size = 0;
30    gp.endorsementAlg = TPM_ALG_NULL;
31    gp.endorsementPolicy.t.size = 0;
32    gp.lockoutAlg = TPM_ALG_NULL;
33    gp.lockoutPolicy.t.size = 0;
34
35    // Initialize ehProof, shProof and phProof
36    gp.phProof.t.size = sizeof(gp.phProof.t.buffer);
37    gp.shProof.t.size = sizeof(gp.shProof.t.buffer);
38    gp.ehProof.t.size = sizeof(gp.ehProof.t.buffer);
39    CryptRandomGenerate(gp.phProof.t.size, gp.phProof.t.buffer);
40    CryptRandomGenerate(gp.shProof.t.size, gp.shProof.t.buffer);
41    CryptRandomGenerate(gp.ehProof.t.size, gp.ehProof.t.buffer);
42
43    // Write hierarchy data to NV
```

```

44     NV_SYNC_PERSISTENT(disableClear);
45     NV_SYNC_PERSISTENT(EPSeed);
46     NV_SYNC_PERSISTENT(SPSeed);
47     NV_SYNC_PERSISTENT(PPSeed);
48     NV_SYNC_PERSISTENT(ownerAuth);
49     NV_SYNC_PERSISTENT(endorsementAuth);
50     NV_SYNC_PERSISTENT(lockoutAuth);
51     NV_SYNC_PERSISTENT(ownerAlg);
52     NV_SYNC_PERSISTENT(ownerPolicy);
53     NV_SYNC_PERSISTENT(endorsementAlg);
54     NV_SYNC_PERSISTENT(endorsementPolicy);
55     NV_SYNC_PERSISTENT(lockoutAlg);
56     NV_SYNC_PERSISTENT(lockoutPolicy);
57     NV_SYNC_PERSISTENT(phProof);
58     NV_SYNC_PERSISTENT(shProof);
59     NV_SYNC_PERSISTENT(ehProof);
60
61     return;
62 }

```

8.3.3.2 HierarchyStartup()

This function is called at TPM2_Startup() to initialize the hierarchy related values.

```

63 BOOL
64 HierarchyStartup(
65     STARTUP_TYPE    type           // IN: start up type
66 )
67 {
68     // phEnable is SET on any startup
69     g_phEnable = TRUE;
70
71     // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
72     // TPM_RESTART
73     if(type != SU_RESUME)
74     {
75         gc.platformAuth.t.size = 0;
76         gc.platformPolicy.t.size = 0;
77         gc.platformAlg = TPM_ALG_NULL;
78
79         // enable the storage and endorsement hierarchies and the platformNV
80         gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
81     }
82     // nullProof and nullSeed are updated at every TPM_RESET
83     if((type != SU_RESTART) && (type != SU_RESUME))
84     {
85         gr.nullProof.t.size = sizeof(gr.nullProof.t.buffer);
86         CryptRandomGenerate(gr.nullProof.t.size, gr.nullProof.t.buffer);
87         gr.nullSeed.t.size = sizeof(gr.nullSeed.t.buffer);
88         CryptRandomGenerate(gr.nullSeed.t.size, gr.nullSeed.t.buffer);
89     }
90     return TRUE;
91 }

```

8.3.3.3 HierarchyGetProof()

This function finds the proof value associated with a hierarchy. It returns a pointer to the proof value.

```

92 TPM2B_PROOF *
93 HierarchyGetProof(
94     TPMI_RH_HIERARCHY    hierarchy    // IN: hierarchy constant
95 )
96 {

```

```

97     TPM2B_PROOF      *proof = NULL;
98
99     switch(hierarchy)
100     {
101     case TPM_RH_PLATFORM:
102         // phProof for TPM_RH_PLATFORM
103         proof = &gp.phProof;
104         break;
105     case TPM_RH_ENDORSEMENT:
106         // ehProof for TPM_RH_ENDORSEMENT
107         proof = &gp.ehProof;
108         break;
109     case TPM_RH_OWNER:
110         // shProof for TPM_RH_OWNER
111         proof = &gp.shProof;
112         break;
113     default:
114         // nullProof for TPM_RH_NULL or anything else
115         proof = &gr.nullProof;
116         break;
117     }
118     return proof;
119 }

```

8.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```

120 TPM2B_SEED *
121 HierarchyGetPrimarySeed(
122     TPMI_RH_HIERARCHY hierarchy // IN: hierarchy
123 )
124 {
125     TPM2B_SEED      *seed = NULL;
126     switch(hierarchy)
127     {
128     case TPM_RH_PLATFORM:
129         seed = &gp.PPSeed;
130         break;
131     case TPM_RH_OWNER:
132         seed = &gp.SPSeed;
133         break;
134     case TPM_RH_ENDORSEMENT:
135         seed = &gp.EPSeed;
136         break;
137     default:
138         seed = &gr.nullSeed;
139         break;
140     }
141     return seed;
142 }

```

8.3.3.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.

NOTE The TPM_RH_NULL hierarchy is always enabled.

Return Value	Meaning
TRUE(1)	hierarchy is enabled
FALSE(0)	hierarchy is disabled

```

143  BOOL
144  HierarchyIsEnabled(
145      TPMI_RH_HIERARCHY    hierarchy    // IN: hierarchy
146  )
147  {
148      BOOL                enabled = FALSE;
149
150      switch(hierarchy)
151      {
152          case TPM_RH_PLATFORM:
153              enabled = g_phEnable;
154              break;
155          case TPM_RH_OWNER:
156              enabled = gc.shEnable;
157              break;
158          case TPM_RH_ENDORSEMENT:
159              enabled = gc.ehEnable;
160              break;
161          case TPM_RH_NULL:
162              enabled = TRUE;
163              break;
164          default:
165              enabled = FALSE;
166              break;
167      }
168      return enabled;
169  }

```

8.4 NvDynamic.c

8.4.1 Introduction

The NV memory is divided into two areas: dynamic space for user defined NV indexes and evict objects, and reserved space for TPM persistent and state save data.

The entries in dynamic space are a linked list of entries. Each entry has, as its first field, a size. If the size field is zero, it marks the end of the list.

An Index allocation will contain an NV_INDEX structure. If the Index does not have the orderly attribute, the NV_INDEX is followed immediately by the NV data.

An evict object entry contains a handle followed by an OBJECT structure. This results in both the Index and Evict Object having an identifying handle as the first field following the size field.

When an Index has the orderly attribute, the data is kept in RAM. This RAM is saved to backing store in NV memory on any orderly shutdown. The entries in orderly memory are also a linked list using a size field as the first entry.

The attributes of an orderly index are maintained in RAM memory in order to reduce the number of NV writes needed for orderly data. When an orderly index is created, an entry is made in the dynamic NV memory space that holds the Index authorizations (*authPolicy* and *authValue*) and the size of the data. This entry is only modified if the *authValue* of the index is changed. The more volatile data of the index is kept in RAM. When an orderly Index is created or deleted, the RAM data is copied to NV backing store so that the image in the backing store matches the layout of RAM. In normal operation. The RAM data is also copied on any orderly shutdown. In normal operation, the only other reason for writing to the backing store for RAM is when a counter is first written (TPMA_NV_WRITTEN changes from CLEAR to SET) or when a counter "rolls over". Static space contains items that are individually modifiable. The values are in the *gp* PERSISTENT_DATA structure in RAM and mapped to locations in NV.

8.4.2 Includes, Defines and Data Definitions

```
1 #define NV_C
2 #include "Tpm.h"
```

8.4.3 Local Functions

8.4.3.1 NvNext()

This function provides a method to traverse every data entry in NV dynamic area.

To begin with, parameter *iter* should be initialized to NV_REF_INIT indicating the first element. Every time this function is called, the value in *iter* would be adjusted pointing to the next element in traversal. If there is no next element, *iter* value would be 0. This function returns the address of the *data entry* pointed by the *iter*. If there are no more elements in the set, a 0 value is returned indicating the end of traversal.

```
3 static NV_REF
4 NvNext(
5     NV_REF          *iter,           // IN/OUT: the list iterator
6     TPM_HANDLE      *handle        // OUT: the handle of the next item.
7 )
8 {
9     NV_REF          currentAddr;
10    NV_ENTRY_HEADER  header;
11    //
12    // If iterator is at the beginning of list
13    if(*iter == NV_REF_INIT)
14    {
```

```

15         // Initialize iterator
16         *iter = NV_USER_DYNAMIC;
17     }
18     // Step over the size field and point to the handle
19     currentAddr = *iter + sizeof(UINT32);
20
21     // read the header of the next entry
22     NvRead(&header, *iter, sizeof(NV_ENTRY_HEADER));
23
24     // if the size field is zero, then we have hit the end of the list
25     if(header.size == 0)
26         // leave the *iter pointing at the end of the list
27         return 0;
28     // advance the header by the size of the entry
29     *iter += header.size;
30
31     if(handle != NULL)
32         *handle = header.handle;
33     return currentAddr;
34 }

```

8.4.3.2 NvNextByType()

This function returns a reference to the next NV entry of the desired type

Return Value	Meaning
0	end of list
!= 0	the next entry of the indicated type

```

35 static NV_REF
36 NvNextByType(
37     TPM_HANDLE *handle,           // OUT: the handle of the found type or 0
38     NV_REF *iter,                // IN: the iterator
39     TPM_HT type                  // IN: the handle type to look for
40 )
41 {
42     NV_REF addr;
43     TPM_HANDLE nvHandle = 0;
44     //
45     while((addr = NvNext(iter, &nvHandle)) != 0)
46     {
47         // addr: the address of the location containing the handle of the value
48         // iter: the next location.
49         if(HandleGetType(nvHandle) == type)
50             break;
51     }
52     if(handle != NULL)
53         *handle = nvHandle;
54     return addr;
55 }

```

8.4.3.3 NvNextIndex()

This function returns the reference to the next NV Index entry. A value of 0 indicates the end of the list.

Return Value	Meaning
0	end of list
!= 0	the next reference

```

56 #define NvNextIndex(handle, iter) \
57     NvNextByType(handle, iter, TPM_HT_NV_INDEX)

```

8.4.3.4 NvNextEvict()

This function returns the offset in NV of the next evict object entry. A value of 0 indicates the end of the list.

```

58 #define NvNextEvict(handle, iter) \
59     NvNextByType(handle, iter, TPM_HT_PERSISTENT)

```

8.4.3.5 NvGetEnd()

Function to find the end of the NV dynamic data list

```

60 static NV_REF
61 NvGetEnd(
62     void
63 )
64 {
65     NV_REF      iter = NV_REF_INIT;
66     NV_REF      currentAddr;
67 //
68 // Scan until the next address is 0
69 while((currentAddr = NvNext(&iter, NULL)) != 0);
70 return iter;
71 }

```

8.4.3.6 NvGetFreeBytes()

This function returns the number of free octets in NV space.

```

72 static UINT32
73 NvGetFreeBytes(
74     void
75 )
76 {
77     // This does not have an overflow issue because NvGetEnd() cannot return a value
78     // that is larger than s_evictNvEnd. This is because there is always a 'stop'
79     // word in the NV memory that terminates the search for the end before the
80     // value can go past s_evictNvEnd.
81     return s_evictNvEnd - NvGetEnd();
82 }

```

8.4.3.7 NvTestSpace()

This function will test if there is enough space to add a new entity.

Return Value	Meaning
TRUE(1)	space available
FALSE(0)	no enough space

```

83  static BOOL
84  NvTestSpace(
85      UINT32      size,          // IN: size of the entity to be added
86      BOOL        isIndex,      // IN: TRUE if the entity is an index
87      BOOL        isCounter     // IN: TRUE if the index is a counter
88  )
89  {
90      UINT32      remainBytes = NvGetFreeBytes();
91      UINT32      reserved = sizeof(UINT32)          // size of the forward pointer
92      + sizeof(NV_LIST_TERMINATOR);
93  //
94  // Do a compile time sanity check on the setting for NV_MEMORY_SIZE
95  #if NV_MEMORY_SIZE < 1024
96  #error "NV_MEMORY_SIZE probably isn't large enough"
97  #endif
98
99  // For NV Index, need to make sure that we do not allocate an Index if this
100 // would mean that the TPM cannot allocate the minimum number of evict
101 // objects.
102 if(isIndex)
103 {
104     // Get the number of persistent objects allocated
105     UINT32      persistentNum = NvCapGetPersistentNumber();
106
107     // If we have not allocated the requisite number of evict objects, then we
108     // need to reserve space for them.
109     // NOTE: some of this is not written as simply as it might seem because
110     // the values are all unsigned and subtracting needs to be done carefully
111     // so that an underflow doesn't cause problems.
112     if(persistentNum < MIN_EVICT_OBJECTS)
113         reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
114 }
115 // If this is not an index or is not a counter, reserve space for the
116 // required number of counter indexes
117 if(!isIndex || !isCounter)
118 {
119     // Get the number of counters
120     UINT32      counterNum = NvCapGetCounterNumber();
121
122     // If the required number of counters have not been allocated, reserved
123     // space for the extra needed counters
124     if(counterNum < MIN_COUNTER_INDICES)
125         reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
126 }
127 // Check that the requested allocation will fit after making sure that there
128 // will be no chance of overflow
129 return ((reserved < remainBytes)
130     && (size <= remainBytes)
131     && (size + reserved <= remainBytes));
132 }

```

8.4.3.8 NvWriteNvListEnd()

Function to write the list terminator.

```

133  NV_REF
134  NvWriteNvListEnd(
135      NV_REF      end

```

```

136     )
137 {
138     // Marker is initialized with zeros
139     BYTE      listEndMarker[sizeof(NV_LIST_TERMINATOR)] = {0};
140     UINT64     maxCount = NvReadMaxCount();
141     //
142     // This is a constant check that can be resolved at compile time.
143     cAssert(sizeof(UINT64) <= sizeof(NV_LIST_TERMINATOR) - sizeof(UINT32));
144
145     // Copy the maxCount value to the marker buffer
146     MemoryCopy(&listEndMarker[sizeof(UINT32)], &maxCount, sizeof(UINT64));
147     pAssert(end + sizeof(NV_LIST_TERMINATOR) <= s_evictNvEnd);
148
149     // Write it to memory
150     NvWrite(end, sizeof(NV_LIST_TERMINATOR), &listEndMarker);
151     return end + sizeof(NV_LIST_TERMINATOR);
152 }

```

8.4.3.9 NvAdd()

This function adds a new entity to NV.

This function requires that there is enough space to add a new entity (i.e., that NvTestSpace() has been called and the available space is at least as large as the required space).

The *totalSize* will be the size of *entity*. If a handle is added, this function will increase the size accordingly.

```

153 static TPM_RC
154 NvAdd(
155     UINT32      totalSize,      // IN: total size needed for this entity For
156                                // evict object, totalSize is the same as
157                                // bufferSize. For NV Index, totalSize is
158                                // bufferSize plus index data size
159     UINT32      bufferSize,    // IN: size of initial buffer
160     TPM_HANDLE  handle,        // IN: optional handle
161     BYTE        *entity        // IN: initial buffer
162 )
163 {
164     NV_REF      newAddr;        // IN: where the new entity will start
165     NV_REF      nextAddr;
166     //
167     RETURN_IF_NV_IS_NOT_AVAILABLE;
168
169     // Get the end of data list
170     newAddr = NvGetEnd();
171
172     // Step over the forward pointer
173     nextAddr = newAddr + sizeof(UINT32);
174
175     // Optionally write the handle. For indexes, the handle is TPM_RH_UNASSIGNED
176     // so that the handle in the nvIndex is used instead of writing this value
177     if(handle != TPM_RH_UNASSIGNED)
178     {
179         NvWrite((UINT32)nextAddr, sizeof(TPM_HANDLE), &handle);
180         nextAddr += sizeof(TPM_HANDLE);
181     }
182     // Write entity data
183     NvWrite((UINT32)nextAddr, bufferSize, entity);
184
185     // Advance the pointer by the amount of the total
186     nextAddr += totalSize;
187
188     // Finish by writing the link value
189
190     // Write the next offset (relative addressing)

```

```

191     totalSize = nextAddr - newAddr;
192
193     // Write link value
194     NvWrite((UINT32)newAddr, sizeof(UINT32), &totalSize);
195
196     // Write the list terminator
197     NvWriteNvListEnd(nextAddr);
198
199     return TPM_RC_SUCCESS;
200 }

```

8.4.3.10 NvDelete()

This function is used to delete an NV Index or persistent object from NV memory.

```

201 static TPM_RC
202 NvDelete(
203     NV_REF          entityRef      // IN: reference to entity to be deleted
204 )
205 {
206     UINT32          entrySize;
207     // adjust entityAddr to back up and point to the forward pointer
208     NV_REF          entryRef = entityRef - sizeof(UINT32);
209     NV_REF          endRef = NvGetEnd();
210     NV_REF          nextAddr; // address of the next entry
211     //
212     RETURN_IF_NV_IS_NOT_AVAILABLE;
213
214     // Get the offset of the next entry. That is, back up and point to the size
215     // field of the entry
216     NvRead(&entrySize, entryRef, sizeof(UINT32));
217
218     // The next entry after the one being deleted is at a relative offset
219     // from the current entry
220     nextAddr = entryRef + entrySize;
221
222     // If this is not the last entry, move everything up
223     if(nextAddr < endRef)
224     {
225         pAssert(nextAddr > entryRef);
226         _plat__NvMemoryMove(nextAddr,
227                             entryRef,
228                             (endRef - nextAddr));
229     }
230     // The end of the used space is now moved up by the amount of space we just
231     // reclaimed
232     endRef -= entrySize;
233
234     // Write the end marker, and make the new end equal to the first byte after
235     // the just added end value. This will automatically update the NV value for
236     // maxCounter.
237     // NOTE: This is the call that sets flag to cause NV to be updated
238     endRef = NvWriteNvListEnd(endRef);
239
240     // Clear the reclaimed memory
241     _plat__NvMemoryClear(endRef, entrySize);
242
243     return TPM_RC_SUCCESS;
244 }

```

8.4.4 RAM-based NV Index Data Access Functions

8.4.4.1 Introduction

The data layout in ram buffer is {size of(NV_handle + attributes + data NV_handle, attributes, data} for each NV Index data stored in RAM.

NV storage associated with orderly data is updated when a NV Index is added but NOT when the data or attributes are changed. Orderly data is only updated to NV on an orderly shutdown (TPM2_Shutdown())

8.4.4.2 NvRamNext()

This function is used to iterate through the list of Ram Index values. *iter needs to be initialized by calling

```

245 static NV_RAM_REF
246 NvRamNext(
247     NV_RAM_REF      *iter,           // IN/OUT: the list iterator
248     TPM_HANDLE      *handle         // OUT: the handle of the next item.
249 )
250 {
251     NV_RAM_REF      currentAddr;
252     NV_RAM_HEADER   header;
253     //
254     // If iterator is at the beginning of list
255     if(*iter == NV_RAM_REF_INIT)
256     {
257         // Initialize iterator
258         *iter = &s_indexOrderlyRam[0];
259     }
260     // if we are going to return what the iter is currently pointing to...
261     currentAddr = *iter;
262
263     // If iterator reaches the end of NV space, then don't advance and return
264     // that we are at the end of the list. The end of the list occurs when
265     // we don't have space for a size and a handle
266     if(currentAddr + sizeof(NV_RAM_HEADER) > RAM_ORDERLY_END)
267         return NULL;
268     // read the header of the next entry
269     MemoryCopy(&header, currentAddr, sizeof(NV_RAM_HEADER));
270
271     // if the size field is zero, then we have hit the end of the list
272     if(header.size == 0)
273         // leave the *iter pointing at the end of the list
274         return NULL;
275     // advance the header by the size of the entry
276     *iter = currentAddr + header.size;
277
278     // pAssert(*iter <= RAM_ORDERLY_END);
279     if(handle != NULL)
280         *handle = header.handle;
281     return currentAddr;
282 }

```

8.4.4.3 NvRamGetEnd()

This routine performs the same function as NvGetEnd() but for the RAM data.

```

283 static NV_RAM_REF
284 NvRamGetEnd(
285     void
286 )

```

```

287 {
288     NV_RAM_REF        iter = NV_RAM_REF_INIT;
289     NV_RAM_REF        currentAddr;
290 //
291 // Scan until the next address is 0
292 while((currentAddr = NvRamNext(&iter, NULL)) != 0);
293 return iter;
294 }

```

8.4.4.4 NvRamTestSpaceIndex()

This function indicates if there is enough RAM space to add a data for a new NV Index.

Return Value	Meaning
TRUE(1)	space available
FALSE(0)	no enough space

```

295 static BOOL
296 NvRamTestSpaceIndex(
297     UINT32        size           // IN: size of the data to be added to RAM
298 )
299 {
300     UINT32        remaining = (UINT32) (RAM_ORDERLY_END - NvRamGetEnd());
301     UINT32        needed = sizeof(NV_RAM_HEADER) + size;
302 //
303 // NvRamGetEnd points to the next available byte.
304 return remaining >= needed;
305 }

```

8.4.4.5 NvRamGetIndex()

This function returns the offset of NV data in the RAM buffer

This function requires that NV Index is in RAM. That is, the index must be known to exist.

```

306 static NV_RAM_REF
307 NvRamGetIndex(
308     TPMI_RH_NV_INDEX    handle           // IN: NV handle
309 )
310 {
311     NV_RAM_REF        iter = NV_RAM_REF_INIT;
312     NV_RAM_REF        currentAddr;
313     TPM_HANDLE        foundHandle;
314 //
315 while((currentAddr = NvRamNext(&iter, &foundHandle)) != 0)
316 {
317     if(handle == foundHandle)
318         break;
319 }
320 return currentAddr;
321 }

```

8.4.4.6 NvUpdateIndexOrderlyData()

This function is used to cause an update of the orderly data to the NV backing store.

```

322 void
323 NvUpdateIndexOrderlyData(
324     void
325 )

```

```

326 {
327     // Write reserved RAM space to NV
328     NvWrite(NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam), s_indexOrderlyRam);
329 }

```

8.4.4.7 NvAddRAM()

This function adds a new data area to RAM.

This function requires that enough free RAM space is available to add the new data.

This function should be called after the NV Index space has been updated and the index removed. This insures that NV is available so that checking for NV availability is not required during this function.

```

330 static void
331 NvAddRAM(
332     TPMS_NV_PUBLIC *index          // IN: the index descriptor
333 )
334 {
335     NV_RAM_HEADER    header;
336     NV_RAM_REF       end = NvRamGetEnd();
337     //
338     header.size = sizeof(NV_RAM_HEADER) + index->dataSize;
339     header.handle = index->nvIndex;
340     MemoryCopy(&header.attributes, &index->attributes, sizeof(TPMA_NV));
341
342     pAssert(ORDERLY_RAM_ADDRESS_OK(end, header.size));
343
344     // Copy the header to the memory
345     MemoryCopy(end, &header, sizeof(NV_RAM_HEADER));
346
347     // Clear the data area (just in case)
348     MemorySet(end + sizeof(NV_RAM_HEADER), 0, index->dataSize);
349
350     // Step over this new entry
351     end += header.size;
352
353     // If the end marker will fit, add it
354     if(end + sizeof(UINT32) < RAM_ORDERLY_END)
355         MemorySet(end, 0, sizeof(UINT32));
356     // Write reserved RAM space to NV to reflect the newly added NV Index
357     SET_NV_UPDATE(UT_ORDERLY);
358
359     return;
360 }

```

8.4.4.8 NvDeleteRAM()

This function is used to delete a RAM-backed NV Index data area. The space used by the entry are overwritten by the contents of the Index data that comes after (the data is moved up to fill the hole left by removing this index. The reclaimed space is cleared to zeros. This function assumes the data of NV Index exists in RAM.

This function should be called after the NV Index space has been updated and the index removed. This insures that NV is available so that checking for NV availability is not required during this function.

```

361 static void
362 NvDeleteRAM(
363     TPMS_NV_PUBLIC *index          // IN: NV handle
364 )
365 {
366     NV_RAM_REF       nodeAddress;
367     NV_RAM_REF       nextNode;

```

```

368     UINT32                size;
369     NV_RAM_REF            lastUsed = NvRamGetEnd();
370 //
371     nodeAddress = NvRamGetIndex(handle);
372
373     pAssert(nodeAddress != 0);
374
375     // Get node size
376     MemoryCopy(&size, nodeAddress, sizeof(size));
377
378     // Get the offset of next node
379     nextNode = nodeAddress + size;
380
381     // Copy the data
382     MemoryCopy(nodeAddress, nextNode, (int)(lastUsed - nextNode));
383
384     // Clear out the reclaimed space
385     MemorySet(lastUsed - size, 0, size);
386
387     // Write reserved RAM space to NV to reflect the newly delete NV Index
388     SET_NV_UPDATE(UT_ORDERLY);
389
390     return;
391 }

```

8.4.4.9 NvReadIndex()

This function is used to read the NV Index NV_INDEX. This is used so that the index information can be compressed and only this function would be needed to decompress it. Mostly, compression would only be able to save the space needed by the policy.

```

392 void
393 NvReadNvIndexInfo(
394     NV_REF            ref,                // IN: points to NV where index is located
395     NV_INDEX          *nvIndex           // OUT: place to receive index data
396 )
397 {
398     pAssert(nvIndex != NULL);
399     NvRead(nvIndex, ref, sizeof(NV_INDEX));
400     return;
401 }

```

8.4.4.10 NvReadObject()

This function is used to read a persistent object. This is used so that the object information can be compressed and only this function would be needed to uncompress it.

```

402 void
403 NvReadObject(
404     NV_REF            ref,                // IN: points to NV where index is located
405     OBJECT            *object           // OUT: place to receive the object data
406 )
407 {
408     NvRead(object, (ref + sizeof(TPM_HANDLE)), sizeof(OBJECT));
409     return;
410 }

```

8.4.4.11 NvFindEvict()

This function will return the NV offset of an evict object

Return Value	Meaning
0	evict object not found
!= 0	offset of evict object

```

411 static NV_REF
412 NvFindEvict(
413     TPM_HANDLE    nvHandle,
414     OBJECT        *object
415 )
416 {
417     NV_REF        found = NvFindHandle(nvHandle);
418     //
419     // If we found the handle and the request included an object pointer, fill it in
420     if(found != 0 && object != NULL)
421         NvReadObject(found, object);
422     return found;
423 }

```

8.4.4.12 NvIndexIsDefined()

See if an index is already defined

```

424 BOOL
425 NvIndexIsDefined(
426     TPM_HANDLE    nvHandle    // IN: Index to look for
427 )
428 {
429     return (NvFindHandle(nvHandle) != 0);
430 }

```

8.4.4.13 NvConditionallyWrite()

Function to check if the data to be written has changed and write it if it has

Error Return	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```

431 static TPM_RC
432 NvConditionallyWrite(
433     NV_REF        entryAddr,    // IN: stating address
434     UINT32        size,        // IN: size of the data to write
435     void          *data        // IN: the data to write
436 )
437 {
438     // If the index data is actually changed, then a write to NV is required
439     if(_plat__NvIsDifferent(entryAddr, size, data))
440     {
441         // Write the data if NV is available
442         if(g_NvStatus == TPM_RC_SUCCESS)
443         {
444             NvWrite(entryAddr, size, data);
445         }
446         return g_NvStatus;
447     }
448     return TPM_RC_SUCCESS;
449 }

```

8.4.4.14 NvReadNvIndexAttributes()

This function returns the attributes of an NV Index.

```

450 static TPMA_NV
451 NvReadNvIndexAttributes(
452     NV_REF          locator          // IN: reference to an NV index
453 )
454 {
455     TPMA_NV          attributes;
456     //
457     NvRead(&attributes,
458            locator + offsetof(NV_INDEX, publicArea.attributes),
459            sizeof(TPMA_NV));
460     return attributes;
461 }

```

8.4.4.15 NvReadRamIndexAttributes()

This function returns the attributes from the RAM header structure. This function is used to deal with the fact that the header structure is only byte aligned.

```

462 static TPMA_NV
463 NvReadRamIndexAttributes(
464     NV_RAM_REF       ref             // IN: pointer to a NV_RAM_HEADER
465 )
466 {
467     TPMA_NV          attributes;
468     //
469     MemoryCopy(&attributes, ref + offsetof(NV_RAM_HEADER, attributes),
470               sizeof(TPMA_NV));
471     return attributes;
472 }

```

8.4.4.16 NvWriteNvIndexAttributes()

This function is used to write just the attributes of an index to NV.

Error Return	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```

473 static TPM_RC
474 NvWriteNvIndexAttributes(
475     NV_REF          locator,         // IN: location of the index
476     TPMA_NV          attributes      // IN: attributes to write
477 )
478 {
479     return NvConditionallyWrite(
480         locator + offsetof(NV_INDEX, publicArea.attributes),
481         sizeof(TPMA_NV),
482         &attributes);
483 }

```

8.4.4.17 NvWriteRamIndexAttributes()

This function is used to write the index attributes into an unaligned structure

```

484 static void
485 NvWriteRamIndexAttributes(
486     NV_RAM_REF      ref,           // IN: address of the header
487     TPMA_NV          attributes    // IN: the attributes to write
488 )
489 {
490     MemoryCopy(ref + offsetof(NV_RAM_HEADER, attributes), &attributes,
491               sizeof(TPMA_NV));
492     return;
493 }

```

8.4.5 Externally Accessible Functions

8.4.5.1 NvIsPlatformPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the platform.

Return Value	Meaning
TRUE(1)	handle references a platform persistent object
FALSE(0)	handle does not reference platform persistent object

```

494 BOOL
495 NvIsPlatformPersistentHandle(
496     TPM_HANDLE      handle        // IN: handle
497 )
498 {
499     return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
500 }

```

8.4.5.2 NvIsOwnerPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the owner.

Return Value	Meaning
TRUE(1)	handle is owner persistent handle
FALSE(0)	handle is not owner persistent handle and may not be a persistent handle at all

```

501 BOOL
502 NvIsOwnerPersistentHandle(
503     TPM_HANDLE      handle        // IN: handle
504 )
505 {
506     return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
507 }

```

8.4.5.3 NvIndexIsAccessible()

This function validates that a handle references a defined NV Index and that the Index is currently accessible.

Error Return	Meaning
TPM_RC_HANDLE	the handle points to an undefined NV Index If <i>shEnable</i> is CLEAR, this would include an index created using <i>ownerAuth</i> . If <i>phEnableNV</i> is CLEAR, this would include an index created using <i>platformAuth</i>
TPM_RC_NV_READLOCKED	Index is present but locked for reading and command does not write to the index
TPM_RC_NV_WRITELOCKED	Index is present but locked for writing and command writes to the index

```

508 TPM_RC
509 NvIndexIsAccessible(
510     TPMI_RH_NV_INDEX    handle        // IN: handle
511 )
512 {
513     NV_INDEX             *nvIndex = NvGetIndexInfo(handle, NULL);
514     //
515     if(nvIndex == NULL)
516         // If index is not found, return TPM_RC_HANDLE
517         return TPM_RC_HANDLE;
518     if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
519     {
520         // if shEnable is CLEAR, an ownerCreate NV Index should not be
521         // indicated as present
522         if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
523         {
524             if(gc.shEnable == FALSE)
525                 return TPM_RC_HANDLE;
526         }
527         // if phEnableNV is CLEAR, a platform created Index should not
528         // be visible
529         else if(gc.phEnableNV == FALSE)
530             return TPM_RC_HANDLE;
531     }
532     #if 0 // Writelock test for debug
533     // If the Index is write locked and this is an NV Write operation...
534     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITELOCKED)
535        && IsWriteOperation(commandIndex))
536     {
537         // then return a locked indication unless the command is TPM2_NV_WriteLock
538         if(GetCommandCode(commandIndex) != TPM_CC_NV_WriteLock)
539             return TPM_RC_NV_LOCKED;
540         return TPM_RC_SUCCESS;
541     }
542     #endif
543     #if 0 // Readlock Test for debug
544     // If the Index is read locked and this is an NV Read operation...
545     if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, READLOCKED)
546        && IsReadOperation(commandIndex))
547     {
548         // then return a locked indication unless the command is TPM2_NV_ReadLock
549         if(GetCommandCode(commandIndex) != TPM_CC_NV_ReadLock)
550             return TPM_RC_NV_LOCKED;
551     }
552     #endif
553     // NV Index is accessible
554     return TPM_RC_SUCCESS;
555 }

```

8.4.5.4 NvGetEvictObject()

This function is used to dereference an evict object handle and get a pointer to the object.

Error Return	Meaning
TPM_RC_HANDLE	the handle does not point to an existing persistent object

```

556 TPM_RC
557 NvGetEvictObject(
558     TPM_HANDLE    handle,        // IN: handle
559     OBJECT        *object        // OUT: object data
560 )
561 {
562     NV_REF         entityAddr;    // offset points to the entity
563     //
564     // Find the address of evict object and copy to object
565     entityAddr = NvFindEvict(handle, object);
566
567     // whether there is an error or not, make sure that the evict
568     // status of the object is set so that the slot will get freed on exit
569     // Must do this after NvFindEvict loads the object
570     object->attributes.evict = SET;
571
572     // If handle is not found, return an error
573     if(entityAddr == 0)
574         return TPM_RC_HANDLE;
575     return TPM_RC_SUCCESS;
576 }

```

8.4.5.5 NvIndexCacheInit()

Function to initialize the Index cache

```

577 void
578 NvIndexCacheInit(
579     void
580 )
581 {
582     s_cachedNvRef = NV_REF_INIT;
583     s_cachedNvRamRef = NV_RAM_REF_INIT;
584     s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
585     return;
586 }

```

8.4.5.6 NvGetIndexData()

This function is used to access the data in an NV Index. The data is returned as a byte sequence.

This function requires that the NV Index be defined, and that the required data is within the data range. It also requires that TPMA_NV_WRITTEN of the Index is SET.

```

587 void
588 NvGetIndexData(
589     NV_INDEX        *nvIndex,    // IN: the in RAM index descriptor
590     NV_REF          locator,     // IN: where the data is located
591     UINT32          offset,     // IN: offset of NV data
592     UINT16          size,       // IN: number of octets of NV data to read
593     void            *data       // OUT: data buffer
594 )
595 {
596     TPMA_NV         nvAttributes;

```

```

597 //
598 pAssert(nvIndex != NULL);
599
600 nvAttributes = nvIndex->publicArea.attributes;
601
602 pAssert(IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN));
603
604 if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, ORDERLY))
605 {
606     // Get data from RAM buffer
607     NV_RAM_REF ramAddr = NvRamGetIndex(nvIndex->publicArea.nvIndex);
608     pAssert(ramAddr != 0 && (size <=
609         ((NV_RAM_HEADER *)ramAddr)->size - sizeof(NV_RAM_HEADER) - offset));
610     MemoryCopy(data, ramAddr + sizeof(NV_RAM_HEADER) + offset, size);
611 }
612 else
613 {
614     // Validate that read falls within range of the index
615     pAssert(offset <= nvIndex->publicArea.dataSize
616         && size <= (nvIndex->publicArea.dataSize - offset));
617     NvRead(data, locator + sizeof(NV_INDEX) + offset, size);
618 }
619 return;
620 }

```

8.4.5.7 NvHashIndexData()

This function adds Index data to a hash. It does this in parts to avoid large stack buffers.

```

621 void
622 NvHashIndexData(
623     HASH_STATE *hashState, // IN: Initialized hash state
624     NV_INDEX *nvIndex, // IN: Index
625     NV_REF locator, // IN: where the data is located
626     UINT32 offset, // IN: starting offset
627     UINT16 size // IN: amount to hash
628 )
629 {
630     #define BUFFER_SIZE 64
631     BYTE buffer[BUFFER_SIZE];
632     if (offset > nvIndex->publicArea.dataSize)
633         return;
634     // Make sure that we don't try to read off the end.
635     if ((offset + size) > nvIndex->publicArea.dataSize)
636         size = nvIndex->publicArea.dataSize - (UINT16)offset;
637     #if BUFFER_SIZE >= MAX_NV_INDEX_SIZE
638         NvGetIndexData(nvIndex, locator, offset, size, buffer);
639         CryptDigestUpdate(hashState, size, buffer);
640     #else
641     {
642         INT16 i;
643         UINT16 readSize;
644         //
645         for (i = size; i > 0; offset += readSize, i -= readSize)
646         {
647             readSize = (i < BUFFER_SIZE) ? i : BUFFER_SIZE;
648             NvGetIndexData(nvIndex, locator, offset, readSize, buffer);
649             CryptDigestUpdate(hashState, readSize, buffer);
650         }
651     }
652     #endif // BUFFER_SIZE >= MAX_NV_INDEX_SIZE
653     #undef BUFFER_SIZE
654 }

```

8.4.5.8 NvGetUINT64Data()

Get data in integer format of a bit or counter NV Index.

This function requires that the NV Index is defined and that the NV Index previously has been written.

```

655  UINT64
656  NvGetUINT64Data(
657      NV_INDEX          *nvIndex,          // IN: the in RAM index descriptor
658      NV_REF            locator            // IN: where index exists in NV
659  )
660  {
661      UINT64            intVal;
662      //
663      // Read the value and convert it to internal format
664      NvGetIndexData(nvIndex, locator, 0, 8, &intVal);
665      return BYTE_ARRAY_TO_UINT64((BYTE *)&intVal);
666  }
```

8.4.5.9 NvWriteIndexAttributes()

This function is used to write just the attributes of an index.

Error Return	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```

667  TPM_RC
668  NvWriteIndexAttributes(
669      TPM_HANDLE        handle,
670      NV_REF            locator,          // IN: location of the index
671      TPMA_NV           attributes        // IN: attributes to write
672  )
673  {
674      TPM_RC            result;
675      //
676      if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
677      {
678          NV_RAM_REF     ram = NvRamGetIndex(handle);
679          NvWriteRamIndexAttributes(ram, attributes);
680          result = TPM_RC_SUCCESS;
681      }
682      else
683      {
684          result = NvWriteNvIndexAttributes(locator, attributes);
685      }
686      return result;
687  }
```

8.4.5.10 NvWriteIndexAuth()

This function is used to write the *authValue* of an index. It is used by TPM2_NV_ChangeAuth()

Error Return	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

688 **TPM_RC**


```

689 NvWriteIndexAuth(
690     NV_REF      locator,          // IN: location of the index
691     TPM2B_AUTH  *authValue        // IN: the authValue to write
692 )
693 {
694     TPM_RC      result;
695     //
696     // If the locator is pointing to the cached index value...
697     if(locator == s_cachedNvRef)
698     {
699         // copy the authValue to the cached index so it will be there if we
700         // look for it. This is a safety thing.
701         MemoryCopy2B(&s_cachedNvIndex.authValue.b, &authValue->b,
702                     sizeof(s_cachedNvIndex.authValue.t.buffer));
703     }
704     result = NvConditionallyWrite(
705         locator + offsetof(NV_INDEX, authValue),
706         sizeof(UINT16) + authValue->t.size,
707         authValue);
708     return result;
709 }

```

8.4.5.11 NvGetIndexInfo()

This function loads the *nvIndex* Info into the NV cache and returns a pointer to the NV_INDEX. If the returned value is zero, the index was not found. The *locator* parameter, if not NULL, will be set to the offset in NV of the Index (the location of the handle of the Index).

This function will set the index cache. If the index is orderly, the attributes from RAM are substituted for the attributes in the cached index

```

710 NV_INDEX *
711 NvGetIndexInfo(
712     TPM_HANDLE  nvHandle,          // IN: the index handle
713     NV_REF      *locator           // OUT: location of the index
714 )
715 {
716     if(s_cachedNvIndex.publicArea.nvIndex != nvHandle)
717     {
718         s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
719         s_cachedNvRamRef = 0;
720         s_cachedNvRef = NvFindHandle(nvHandle);
721         if(s_cachedNvRef == 0)
722             return NULL;
723         NvReadNvIndexInfo(s_cachedNvRef, &s_cachedNvIndex);
724         if(IS_ATTRIBUTE(s_cachedNvIndex.publicArea.attributes, TPMA_NV, ORDERLY))
725         {
726             s_cachedNvRamRef = NvRamGetIndex(nvHandle);
727             s_cachedNvIndex.publicArea.attributes =
728                 NvReadRamIndexAttributes(s_cachedNvRamRef);
729         }
730     }
731     if(locator != NULL)
732         *locator = s_cachedNvRef;
733     return &s_cachedNvIndex;
734 }

```

8.4.5.12 NvWriteIndexData()

This function is used to write NV index data. It is intended to be used to update the data associated with the default index.

This function requires that the NV Index is defined, and the data is within the defined data range for the index.

Index data is only written due to a command that modifies the data in a single index. There is no case where changes are made to multiple indexes data at the same time. Multiple attributes may be change but not multiple index data. This is important because we will normally be handling the index for which we have the cached pointer values.

Error Return	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```

735 TPM_RC
736 NvWriteIndexData(
737     NV_INDEX      *nvIndex,      // IN: the description of the index
738     UINT32         offset,        // IN: offset of NV data
739     UINT32         size,          // IN: size of NV data
740     void           *data          // IN: data buffer
741 )
742 {
743     TPM_RC         result = TPM_RC_SUCCESS;
744     //
745     pAssert(nvIndex != NULL);
746     // Make sure that this is dealing with the 'default' index.
747     // Note: it is tempting to change the calling sequence so that the 'default' is
748     // presumed.
749     pAssert(nvIndex->publicArea.nvIndex == s_cachedNvIndex.publicArea.nvIndex);
750
751     // Validate that write falls within range of the index
752     pAssert(offset <= nvIndex->publicArea.dataSize
753             && size <= (nvIndex->publicArea.dataSize - offset));
754
755     // Update TPMA_NV_WRITTEN bit if necessary
756     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
757     {
758         // Update the in memory version of the attributes
759         SET_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN);
760
761         // If this is not orderly, then update the NV version of
762         // the attributes
763         if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
764         {
765             result = NvWriteNvIndexAttributes(s_cachedNvRef,
766                                             nvIndex->publicArea.attributes);
767             if(result != TPM_RC_SUCCESS)
768                 return result;
769             // If this is a partial write of an ordinary index, clear the whole
770             // index.
771             if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes)
772                 && (nvIndex->publicArea.dataSize > size))
773                 _plat__NvMemoryClear(s_cachedNvRef + sizeof(NV_INDEX),
774                                     nvIndex->publicArea.dataSize);
775         }
776     }
777     else
778     {
779         // This is orderly so update the RAM version
780         MemoryCopy(s_cachedNvRamRef + offsetof(NV_RAM_HEADER, attributes),
781                 &nvIndex->publicArea.attributes, sizeof(TPMA_NV));
782         // If setting WRITTEN for an orderly counter, make sure that the
783         // state saved version of the counter is saved
784         if(IsNvCounterIndex(nvIndex->publicArea.attributes))
785             SET_NV_UPDATE(UT_ORDERLY);
786         // If setting the written attribute on an ordinary index, make sure that

```

```

786         // the data is all cleared out in case there is a partial write. This
787         // is only necessary for ordinary indexes because all of the other types
788         // are always written in total.
789         else if (IsNvOrdinaryIndex(nvIndex->publicArea.attributes))
790             MemorySet(s_cachedNvRamRef + sizeof(NV_RAM_HEADER),
791                     0, nvIndex->publicArea.dataSize);
792     }
793 }
794 // If this is orderly data, write it to RAM
795 if (IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
796 {
797     // Note: if this is the first write to a counter, the code above will queue
798     // the write to NV of the RAM data in order to update TPMA_NV_WRITTEN. In
799     // process of doing that write, it will also write the initial counter value
800
801     // Update RAM
802     MemoryCopy(s_cachedNvRamRef + sizeof(NV_RAM_HEADER) + offset, data, size);
803
804     // And indicate that the TPM is no longer orderly
805     g_clearOrderly = TRUE;
806 }
807 else
808 {
809     // Offset into the index to the first byte of the data to be written to NV
810     result = NvConditionallyWrite(s_cachedNvRef + sizeof(NV_INDEX) + offset,
811                                 size, data);
812 }
813 return result;
814 }

```

8.4.5.13 NvWriteUINT64Data()

This function to write back a UINT64 value. The various UINT64 values (bits, counters, and PINs) are kept in canonical format but manipulate in native format. This takes a native format value converts it and saves it back as in canonical format.

This function will return the value from NV or RAM depending on the type of the index (orderly or not)

```

815 TPM_RC
816 NvWriteUINT64Data(
817     NV_INDEX      *nvIndex,          // IN: the description of the index
818     UINT64         intValue           // IN: the value to write
819 )
820 {
821     BYTE          bytes[8];
822     UINT64_TO_BYTE_ARRAY(intValue, bytes);
823     //
824     return NvWriteIndexData(nvIndex, 0, 8, &bytes);
825 }

```

8.4.5.14 NvGetIndexName()

This function computes the Name of an index. The *name* buffer receives the bytes of the Name and the return value is the number of octets in the Name.

This function requires that the NV Index is defined.

```

826 TPM2B_NAME *
827 NvGetIndexName(
828     NV_INDEX      *nvIndex,          // IN: the index over which the name is to be
829                                     // computed
830     TPM2B_NAME    *name              // OUT: name of the index
831 )

```

```

832 {
833     UINT16      dataSize, digestSize;
834     BYTE        marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
835     BYTE        *buffer;
836     HASH_STATE  hashState;
837 //
838 // Marshal public area
839 buffer = marshalBuffer;
840 dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex->publicArea, &buffer, NULL);
841
842 // hash public area
843 digestSize = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
844 CryptDigestUpdate(&hashState, dataSize, marshalBuffer);
845
846 // Complete digest leaving room for the nameAlg
847 CryptHashEnd(&hashState, digestSize, &name->b.buffer[2]);
848
849 // Include the nameAlg
850 UINT16_TO_BYTE_ARRAY(nvIndex->publicArea.nameAlg, name->b.buffer);
851 name->t.size = digestSize + 2;
852 return name;
853 }

```

8.4.5.15 NvGetNameByIndexHandle()

This function is used to compute the Name of an NV Index referenced by handle.

The *name* buffer receives the bytes of the Name and the return value is the number of octets in the Name.

This function requires that the NV Index is defined.

```

854 TPM2B_NAME *
855 NvGetNameByIndexHandle(
856     TPMI_RH_NV_INDEX  handle,           // IN: handle of the index
857     TPM2B_NAME        *name            // OUT: name of the index
858 )
859 {
860     NV_INDEX          *nvIndex = NvGetIndexInfo(handle, NULL);
861 //
862 return NvGetIndexName(nvIndex, name);
863 }

```

8.4.5.16 NvDefineIndex()

This function is used to assign NV memory to an NV Index.

Error Return	Meaning
TPM_RC_NV_SPACE	insufficient NV space

```

864 TPM_RC
865 NvDefineIndex(
866     TPMS_NV_PUBLIC *publicArea, // IN: A template for an area to create.
867     TPM2B_AUTH     *authValue  // IN: The initial authorization value
868 )
869 {
870 // The buffer to be written to NV memory
871 NV_INDEX          nvIndex;        // the index data
872 UINT16            entrySize;      // size of entry
873 TPM_RC            result;
874 //
875 entrySize = sizeof(NV_INDEX);

```

```

876
877 // only allocate data space for indexes that are going to be written to NV.
878 // Orderly indexes don't need space.
879 if(!IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
880     entrySize += publicArea->dataSize;
881 // Check if we have enough space to create the NV Index
882 // In this implementation, the only resource limitation is the available NV
883 // space (and possibly RAM space.) Other implementation may have other
884 // limitation on counter or on NV slots
885 if(!NvTestSpace(entrySize, TRUE, IsNvCounterIndex(publicArea->attributes)))
886     return TPM_RC_NV_SPACE;
887
888 // if the index to be defined is RAM backed, check RAM space availability
889 // as well
890 if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY)
891     && !NvRamTestSpaceIndex(publicArea->dataSize))
892     return TPM_RC_NV_SPACE;
893 // Copy input value to nvBuffer
894 nvIndex.publicArea = *publicArea;
895
896 // Copy the authValue
897 nvIndex.authValue = *authValue;
898
899 // Add index to NV memory
900 result = NvAdd(entrySize, sizeof(NV_INDEX), TPM_RH_UNASSIGNED,
901               (BYTE *) &nvIndex);
902 if(result == TPM_RC_SUCCESS)
903 {
904     // If the data of NV Index is RAM backed, add the data area in RAM as well
905     if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
906         NvAddRAM(publicArea);
907 }
908 return result;
909 }

```

8.4.5.17 NvAddEvictObject()

This function is used to assign NV memory to a persistent object.

Error Return	Meaning
TPM_RC_NV_HANDLE	the requested handle is already in use
TPM_RC_NV_SPACE	insufficient NV space

```

910 TPM_RC
911 NvAddEvictObject(
912     TPMI_DH_OBJECT    evictHandle,    // IN: new evict handle
913     OBJECT             *object        // IN: object to be added
914 )
915 {
916     TPM_HANDLE    temp = object->evictHandle;
917     TPM_RC        result;
918     //
919     // Check if we have enough space to add the evict object
920     // An evict object needs 8 bytes in index table + sizeof OBJECT
921     // In this implementation, the only resource limitation is the available NV
922     // space. Other implementation may have other limitation on evict object
923     // handle space
924     if(!NvTestSpace(sizeof(OBJECT) + sizeof(TPM_HANDLE), FALSE, FALSE))
925         return TPM_RC_NV_SPACE;
926
927     // Set evict attribute and handle
928     object->attributes.evict = SET;

```

```

929     object->evictHandle = evictHandle;
930
931     // Now put this in NV
932     result = NvAdd(sizeof(OBJECT), sizeof(OBJECT), evictHandle, (BYTE *)object);
933
934     // Put things back the way they were
935     object->attributes.evict = CLEAR;
936     object->evictHandle = temp;
937
938     return result;
939 }

```

8.4.5.18 NvDeleteIndex()

This function is used to delete an NV Index.

Error Return	Meaning
TPM_RC_NV_UNAVAILABLE	NV is not accessible
TPM_RC_NV_RATE	NV is rate limiting

```

940 TPM_RC
941 NvDeleteIndex(
942     NV_INDEX      *nvIndex,          // IN: an in RAM index descriptor
943     NV_REF        entityAddr         // IN: location in NV
944 )
945 {
946     TPM_RC        result;
947 //
948     if(nvIndex != NULL)
949     {
950         // Whenever a counter is deleted, make sure that the MaxCounter value is
951         // updated to reflect the value
952         if(IsNvCounterIndex(nvIndex->publicArea.attributes)
953             && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
954             NvUpdateMaxCount(NvGetUINT64Data(nvIndex, entityAddr));
955         result = NvDelete(entityAddr);
956         if(result != TPM_RC_SUCCESS)
957             return result;
958         // If the NV Index is RAM backed, delete the RAM data as well
959         if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
960             NvDeleteRAM(nvIndex->publicArea.nvIndex);
961         NvIndexCacheInit();
962     }
963     return TPM_RC_SUCCESS;
964 }

```

8.4.5.19 NvDeleteEvict()

This function will delete a NV evict object. Will return success if object deleted or if it does not exist

```

965 TPM_RC
966 NvDeleteEvict(
967     TPM_HANDLE     handle             // IN: handle of entity to be deleted
968 )
969 {
970     NV_REF        entityAddr = NvFindEvict(handle, NULL); // pointer to entity
971     TPM_RC        result = TPM_RC_SUCCESS;
972 //
973     if(entityAddr != 0)
974         result = NvDelete(entityAddr);
975     return result;

```

976 }

8.4.5.20 NvFlushHierarchy()

This function will delete persistent objects belonging to the indicated hierarchy. If the storage hierarchy is selected, the function will also delete any NV Index defined using *ownerAuth*.

Error Return	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```

977 TPM_RC
978 NvFlushHierarchy(
979     TPMI_RH_HIERARCHY    hierarchy    // IN: hierarchy to be flushed.
980 )
981 {
982     NV_REF                iter = NV_REF_INIT;
983     NV_REF                currentAddr;
984     TPM_HANDLE            entityHandle;
985     TPM_RC                result = TPM_RC_SUCCESS;
986     //
987     while((currentAddr = NvNext(&iter, &entityHandle)) != 0)
988     {
989         if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
990         {
991             NV_INDEX        nvIndex;
992             //
993             // If flush endorsement or platform hierarchy, no NV Index would be
994             // flushed
995             if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
996                 continue;
997             // Get the index information
998             NvReadNvIndexInfo(currentAddr, &nvIndex);
999
1000             // For storage hierarchy, flush OwnerCreated index
1001             if(!IS_ATTRIBUTE(nvIndex.publicArea.attributes, TPMA_NV,
1002                             PLATFORMCREATE))
1003             {
1004                 // Delete the index (including RAM for orderly)
1005                 result = NvDeleteIndex(&nvIndex, currentAddr);
1006                 if(result != TPM_RC_SUCCESS)
1007                     break;
1008                 // Re-iterate from beginning after a delete
1009                 iter = NV_REF_INIT;
1010             }
1011         }
1012         else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
1013         {
1014             OBJECT_ATTRIBUTES    attributes;
1015             //
1016             NvRead(&attributes,
1017                   (UINT32) (currentAddr
1018                           + sizeof(TPM_HANDLE)
1019                           + offsetof(OBJECT, attributes)),
1020                   sizeof(OBJECT_ATTRIBUTES));
1021             // If the evict object belongs to the hierarchy to be flushed...
1022             if((hierarchy == TPM_RH_PLATFORM && attributes.ppsHierarchy == SET)
1023                || (hierarchy == TPM_RH_OWNER && attributes.spsHierarchy == SET)
1024                || (hierarchy == TPM_RH_ENDORSEMENT
1025                    && attributes.epsHierarchy == SET))
1026             {
1027                 // ...then delete the evict object

```



```

1028         result = NvDelete(currentAddr);
1029         if(result != TPM_RC_SUCCESS)
1030             break;
1031         // Re-iterate from beginning after a delete
1032         iter = NV_REF_INIT;
1033     }
1034 }
1035 else
1036 {
1037     FAIL(FATAL_ERROR_INTERNAL);
1038 }
1039 }
1040 return result;
1041 }

```

8.4.5.21 NvSetGlobalLock()

This function is used to SET the TPMA_NV_WRITELOCKED attribute for all NV indexes that have TPMA_NV_GLOBALLOCK SET. This function is use by TPM2_NV_GlobalWriteLock().

Error Return	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```

1042 TPM_RC
1043 NvSetGlobalLock(
1044     void
1045 )
1046 {
1047     NV_REF          iter = NV_REF_INIT;
1048     NV_RAM_REF      ramIter = NV_RAM_REF_INIT;
1049     NV_REF          currentAddr;
1050     NV_RAM_REF      currentRamAddr;
1051     TPM_RC          result = TPM_RC_SUCCESS;
1052 //
1053 // Check all normal indexes
1054 while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1055 {
1056     TPMA_NV          attributes = NvReadNvIndexAttributes(currentAddr);
1057 //
1058 // See if it should be locked
1059 if(!IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1060    && IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1061 {
1062     SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1063     result = NvWriteNvIndexAttributes(currentAddr, attributes);
1064     if(result != TPM_RC_SUCCESS)
1065         return result;
1066 }
1067 }
1068 // Now search all the orderly attributes
1069 while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1070 {
1071     // See if it should be locked
1072     TPMA_NV          attributes = NvReadRamIndexAttributes(currentRamAddr);
1073     if(IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1074     {
1075         SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1076         NvWriteRamIndexAttributes(currentRamAddr, attributes);
1077     }
1078 }
1079 return result;

```

1080 }

8.4.5.22 InsertSort()

Sort a handle into handle list in ascending order. The total handle number in the list should not exceed MAX_CAP_HANDLES

```

1081 static void
1082 InsertSort(
1083     TPML_HANDLE *handleList,    // IN/OUT: sorted handle list
1084     UINT32 count,              // IN: maximum count in the handle list
1085     TPM_HANDLE entityHandle    // IN: handle to be inserted
1086 )
1087 {
1088     UINT32 i, j;
1089     UINT32 originalCount;
1090     //
1091     // For a corner case that the maximum count is 0, do nothing
1092     if(count == 0)
1093         return;
1094     // For empty list, add the handle at the beginning and return
1095     if(handleList->count == 0)
1096     {
1097         handleList->handle[0] = entityHandle;
1098         handleList->count++;
1099         return;
1100     }
1101     // Check if the maximum of the list has been reached
1102     originalCount = handleList->count;
1103     if(originalCount < count)
1104         handleList->count++;
1105     // Insert the handle to the list
1106     for(i = 0; i < originalCount; i++)
1107     {
1108         if(handleList->handle[i] > entityHandle)
1109         {
1110             for(j = handleList->count - 1; j > i; j--)
1111             {
1112                 handleList->handle[j] = handleList->handle[j - 1];
1113             }
1114             break;
1115         }
1116     }
1117     // If a slot was found, insert the handle in this position
1118     if(i < originalCount || handleList->count > originalCount)
1119         handleList->handle[i] = entityHandle;
1120     return;
1121 }

```

8.4.5.23 NvCapGetPersistent()

This function is used to get a list of handles of the persistent objects, starting at *handle*.

Handle must be in valid persistent object handle range, but does not have to reference an existing persistent object.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

1122 TPMI_YES_NO

```

1123 NvCapGetPersistent(
1124     TPMI_DH_OBJECT    handle,           // IN: start handle
1125     UINT32             count,           // IN: maximum number of returned handles
1126     TPML_HANDLE        *handleList      // OUT: list of handle
1127 )
1128 {
1129     TPMI_YES_NO        more = NO;
1130     NV_REF             iter = NV_REF_INIT;
1131     NV_REF             currentAddr;
1132     TPM_HANDLE         entityHandle;
1133     //
1134     pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1135
1136     // Initialize output handle list
1137     handleList->count = 0;
1138
1139     // The maximum count of handles we may return is MAX_CAP_HANDLES
1140     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1141
1142     while((currentAddr = NvNextEvict(&entityHandle, &iter)) != 0)
1143     {
1144         // Ignore persistent handles that have values less than the input handle
1145         if(entityHandle < handle)
1146             continue;
1147         // if the handles in the list have reached the requested count, and there
1148         // are still handles need to be inserted, indicate that there are more.
1149         if(handleList->count == count)
1150             more = YES;
1151         // A handle with a value larger than start handle is a candidate
1152         // for return. Insert sort it to the return list. Insert sort algorithm
1153         // is chosen here for simplicity based on the assumption that the total
1154         // number of NV indexes is small. For an implementation that may allow
1155         // large number of NV indexes, a more efficient sorting algorithm may be
1156         // used here.
1157         InsertSort(handleList, count, entityHandle);
1158     }
1159     return more;
1160 }

```

8.4.5.24 NvCapGetIndex()

This function returns a list of handles of NV indexes, starting from *handle*. *Handle* must be in the range of NV indexes, but does not have to reference an existing NV Index.

Return Value	Meaning
YES	if there are more handles to report
NO	all the available handles has been reported

```

1161 TPMI_YES_NO
1162 NvCapGetIndex(
1163     TPMI_DH_OBJECT    handle,           // IN: start handle
1164     UINT32             count,           // IN: max number of returned handles
1165     TPML_HANDLE        *handleList      // OUT: list of handle
1166 )
1167 {
1168     TPMI_YES_NO        more = NO;
1169     NV_REF             iter = NV_REF_INIT;
1170     NV_REF             currentAddr;
1171     TPM_HANDLE         nvHandle;
1172     //
1173     pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1174 }

```

```

1175 // Initialize output handle list
1176 handleList->count = 0;
1177
1178 // The maximum count of handles we may return is MAX_CAP_HANDLES
1179 if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1180
1181 while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1182 {
1183     // Ignore index handles that have values less than the 'handle'
1184     if(nvHandle < handle)
1185         continue;
1186     // if the count of handles in the list has reached the requested count,
1187     // and there are still handles to report, set more.
1188     if(handleList->count == count)
1189         more = YES;
1190     // A handle with a value larger than start handle is a candidate
1191     // for return. Insert sort it to the return list. Insert sort algorithm
1192     // is chosen here for simplicity based on the assumption that the total
1193     // number of NV indexes is small. For an implementation that may allow
1194     // large number of NV indexes, a more efficient sorting algorithm may be
1195     // used here.
1196     InsertSort(handleList, count, nvHandle);
1197 }
1198 return more;
1199 }

```

8.4.5.25 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```

1200 UINT32
1201 NvCapGetIndexNumber (
1202     void
1203 )
1204 {
1205     UINT32      num = 0;
1206     NV_REF      iter = NV_REF_INIT;
1207 //
1208     while(NvNextIndex(NULL, &iter) != 0)
1209         num++;
1210     return num;
1211 }

```

8.4.5.26 NvCapGetPersistentNumber()

Function returns the count of persistent objects currently in NV memory.

```

1212 UINT32
1213 NvCapGetPersistentNumber (
1214     void
1215 )
1216 {
1217     UINT32      num = 0;
1218     NV_REF      iter = NV_REF_INIT;
1219     TPM_HANDLE   handle;
1220 //
1221     while(NvNextEvict(&handle, &iter) != 0)
1222         num++;
1223     return num;
1224 }

```

8.4.5.27 NvCapGetPersistentAvail()

This function returns an estimate of the number of additional persistent objects that could be loaded into NV memory.

```

1225  UINT32
1226  NvCapGetPersistentAvail(
1227      void
1228  )
1229  {
1230      UINT32      availNVSpace;
1231      UINT32      counterNum = NvCapGetCounterNumber();
1232      UINT32      reserved = sizeof(NV_LIST_TERMINATOR);
1233  //
1234      // Get the available space in NV storage
1235      availNVSpace = NvGetFreeBytes();
1236
1237      if(counterNum < MIN_COUNTER_INDICES)
1238      {
1239          // Some space has to be reserved for counter objects.
1240          reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
1241          if(reserved > availNVSpace)
1242              availNVSpace = 0;
1243          else
1244              availNVSpace -= reserved;
1245      }
1246      return availNVSpace / NV_EVICT_OBJECT_SIZE;
1247  }

```

8.4.5.28 NvCapGetCounterNumber()

Get the number of defined NV Indexes that are counter indexes.

```

1248  UINT32
1249  NvCapGetCounterNumber(
1250      void
1251  )
1252  {
1253      NV_REF      iter = NV_REF_INIT;
1254      NV_REF      currentAddr;
1255      UINT32      num = 0;
1256  //
1257      while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1258      {
1259          TPMA_NV      attributes = NvReadNvIndexAttributes(currentAddr);
1260          if(IsNvCounterIndex(attributes))
1261              num++;
1262      }
1263      return num;
1264  }

```

8.4.5.29 NvSetStartupAttributes()

Local function to set the attributes of an Index at TPM Reset and TPM Restart.

```

1265  static TPMA_NV
1266  NvSetStartupAttributes(
1267      TPMA_NV      attributes,          // IN: attributes to change
1268      STARTUP_TYPE  type               // IN: start up type
1269  )
1270  {
1271      // Clear read lock

```

```

1272     CLEAR_ATTRIBUTE(attributes, TPMA_NV, READLOCKED);
1273
1274     // Will change a non counter index to the unwritten state if:
1275     // a) TPMA_NV_CLEAR_STCLEAR is SET
1276     // b) orderly and TPM Reset
1277     if(!IsNvCounterIndex(attributes))
1278     {
1279         if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
1280            || (IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1281                && (type == SU_RESET)))
1282             CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITTEN);
1283     }
1284     // Unlock any index that is not written or that does not have
1285     // TPMA_NV_WRITEDEFINE SET.
1286     if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
1287        || !IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
1288         CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1289     return attributes;
1290 }

```

8.4.5.30 NvEntityStartup()

This function is called at TPM_Startup(). If the startup completes a TPM Resume cycle, no action is taken. If the startup is a TPM Reset or a TPM Restart, then this function will:

- a) clear read/write lock;
- b) reset NV Index data that has TPMA_NV_CLEAR_STCLEAR SET; and
- c) set the lower bits in orderly counters to 1 for a non-orderly startup

It is a prerequisite that NV be available for writing before this function is called.

```

1291 BOOL
1292 NvEntityStartup(
1293     STARTUP_TYPE    type           // IN: start up type
1294 )
1295 {
1296     NV_REF            iter = NV_REF_INIT;
1297     NV_RAM_REF        ramIter = NV_RAM_REF_INIT;
1298     NV_REF            currentAddr; // offset points to the current entity
1299     NV_RAM_REF        currentRamAddr;
1300     TPM_HANDLE        nvHandle;
1301     TPMA_NV           attributes;
1302 //
1303     // Restore RAM index data
1304     NvRead(s_indexOrderlyRam, NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam));
1305
1306     // Initialize the max NV counter value
1307     NvSetMaxCount(NvGetMaxCount());
1308
1309     // If recovering from state save, do nothing else
1310     if(type == SU_RESUME)
1311         return TRUE;
1312     // Iterate all the NV Index to clear the locks
1313     while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1314     {
1315         attributes = NvReadNvIndexAttributes(currentAddr);
1316
1317         // If this is an orderly index, defer processing until loop below
1318         if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
1319             continue;
1320         // Set the attributes appropriate for this startup type
1321         attributes = NvSetStartupAttributes(attributes, type);
1322         NvWriteNvIndexAttributes(currentAddr, attributes);

```

```

1323     }
1324     // Iterate all the orderly indexes to clear the locks and initialize counters
1325     while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1326     {
1327         attributes = NvReadRamIndexAttributes(currentRamAddr);
1328
1329         attributes = NvSetStartupAttributes(attributes, type);
1330
1331         // update attributes in RAM
1332         NvWriteRamIndexAttributes(currentRamAddr, attributes);
1333
1334         // Set the lower bits in an orderly counter to 1 for a non-orderly startup
1335         if(IsNvCounterIndex(attributes)
1336             && (g_prevOrderlyState == SU_NONE_VALUE))
1337         {
1338             UINT64        counter;
1339             //
1340             // Read the counter value last saved to NV.
1341             counter = BYTE_ARRAY_TO_UINT64(currentRamAddr + sizeof(NV_RAM_HEADER));
1342
1343             // Set the lower bits of counter to 1's
1344             counter |= MAX_ORDERLY_COUNT;
1345
1346             // Write back to RAM
1347             // NOTE: Do not want to force a write to NV here. The counter value will
1348             // stay in RAM until the next shutdown or rollover.
1349             UINT64_TO_BYTE_ARRAY(counter, currentRamAddr + sizeof(NV_RAM_HEADER));
1350         }
1351     }
1352     return TRUE;
1353 }

```

8.4.5.31 NvCapGetCounterAvail()

This function returns an estimate of the number of additional counter type NV indexes that can be defined.

```

1354     UINT32
1355     NvCapGetCounterAvail(
1356         void
1357     )
1358     {
1359         UINT32        availNVSpace;
1360         UINT32        availRAMSpace;
1361         UINT32        persistentNum = NvCapGetPersistentNumber();
1362         UINT32        reserved = sizeof(NV_LIST_TERMINATOR);
1363         //
1364         // Get the available space in NV storage
1365         availNVSpace = NvGetFreeBytes();
1366
1367         if(persistentNum < MIN_EVICT_OBJECTS)
1368         {
1369             // Some space has to be reserved for evict object. Adjust availNVSpace.
1370             reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
1371             if(reserved > availNVSpace)
1372                 availNVSpace = 0;
1373             else
1374                 availNVSpace -= reserved;
1375         }
1376         // Compute the available space in RAM
1377         availRAMSpace = (int)(RAM_ORDERLY_END - NvRamGetEnd());
1378
1379         // Return the min of counter number in NV and in RAM
1380         if(availNVSpace / NV_INDEX_COUNTER_SIZE

```



```

1381         > availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE)
1382     return availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE;
1383 else
1384     return availNVSpace / NV_INDEX_COUNTER_SIZE;
1385 }

```

8.4.5.32 NvFindHandle()

this function returns the offset in NV memory of the entity associated with the input handle. A value of zero indicates that handle does not exist reference an existing persistent object or defined NV Index.

```

1386 NV_REF
1387 NvFindHandle(
1388     TPM_HANDLE    handle
1389 )
1390 {
1391     NV_REF        addr;
1392     NV_REF        iter = NV_REF_INIT;
1393     TPM_HANDLE    nextHandle;
1394 //
1395 while((addr = NvNext(&iter, &nextHandle)) != 0)
1396 {
1397     if(nextHandle == handle)
1398         break;
1399 }
1400 return addr;
1401 }

```

8.4.6 NV Max Counter

8.4.6.1 Introduction

The TPM keeps track of the highest value of a deleted counter index. When an index is deleted, this value is updated if the deleted counter index is greater than the previous value. When a new index is created and first incremented, it will get a value that is at least one greater than any other index than any previously deleted index. This insures that it is not possible to roll back an index.

The highest counter value is kept in NV in a special end-of-list marker. This marker is only updated when an index is deleted. Otherwise it just moves.

When the TPM starts up, it searches NV for the end of list marker and initializes an in memory value (*s_maxCounter*).

8.4.6.2 NvReadMaxCount()

This function returns the max NV counter value.

```

1402 UINT64
1403 NvReadMaxCount(
1404     void
1405 )
1406 {
1407     return s_maxCounter;
1408 }

```

8.4.6.3 NvUpdateMaxCount()

This function updates the max counter value to NV memory. This is just staging for the actual write that will occur when the NV index memory is modified.

```

1409 void
1410 NvUpdateMaxCount(
1411     UINT64          count
1412 )
1413 {
1414     if(count > s_maxCounter)
1415         s_maxCounter = count;
1416 }

```

8.4.6.4 NvSetMaxCount()

This function is used at NV initialization time to set the initial value of the maximum counter.

```

1417 void
1418 NvSetMaxCount(
1419     UINT64          value
1420 )
1421 {
1422     s_maxCounter = value;
1423 }

```

8.4.6.5 NvGetMaxCount()

Function to get the NV max counter value from the end-of-list marker

```

1424 UINT64
1425 NvGetMaxCount(
1426     void
1427 )
1428 {
1429     NV_REF          iter = NV_REF_INIT;
1430     NV_REF          currentAddr;
1431     UINT64          maxCount;
1432     //
1433     // Find the end of list marker and initialize the NV Max Counter value.
1434     while((currentAddr = NvNext(&iter, NULL )) != 0);
1435     // 'iter' should be pointing at the end of list marker so read in the current
1436     // value of the s_maxCounter.
1437     NvRead(&maxCount, iter + sizeof(UINT32), sizeof(maxCount));
1438
1439     return maxCount;
1440 }

```

8.5 NvReserved.c

8.5.1 Introduction

The NV memory is divided into two areas: dynamic space for user defined NV Indices and evict objects, and reserved space for TPM persistent and state save data.

The entries in dynamic space are a linked list of entries. Each entry has, as its first field, a size. If the size field is zero, it marks the end of the list.

An allocation of an Index or evict object may use almost all of the remaining NV space such that the size field will not fit. The functions that search the list are aware of this and will terminate the search if they either find a zero size or recognize that there is insufficient space for the size field.

An Index allocation will contain an NV_INDEX structure. If the Index does not have the orderly attribute, the NV_INDEX is followed immediately by the NV data.

An evict object entry contains a handle followed by an OBJECT structure. This results in both the Index and Evict Object having an identifying handle as the first field following the size field.

When an Index has the orderly attribute, the data is kept in RAM. This RAM is saved to backing store in NV memory on any orderly shutdown. The entries in orderly memory are also a linked list using a size field as the first entry. As with the NV memory, the list is terminated by a zero size field or when the last entry leaves insufficient space for the terminating size field.

The attributes of an orderly index are maintained in RAM memory in order to reduce the number of NV writes needed for orderly data. When an orderly index is created, an entry is made in the dynamic NV memory space that holds the Index authorizations (*authPolicy* and *authValue*) and the size of the data. This entry is only modified if the *authValue* of the index is changed. The more volatile data of the index is kept in RAM. When an orderly Index is created or deleted, the RAM data is copied to NV backing store so that the image in the backing store matches the layout of RAM. In normal operation, the RAM data is also copied on any orderly shutdown. In normal operation, the only other reason for writing to the backing store for RAM is when a counter is first written (TPMA_NV_WRITTEN changes from CLEAR to SET) or when a counter **rolls over**. Static space contains items that are individually modifiable. The values are in the *gp* PERSISTENT_DATA structure in RAM and mapped to locations in NV.

8.5.2 Includes, Defines

```
1  #define NV_C
2  #include "Tpm.h"
```

8.5.3 Functions

8.5.3.1 NvInitStatic()

This function initializes the static variables used in the NV subsystem.

```
3  static void
4  NvInitStatic(
5      void
6  )
7  {
8      // In some implementations, the end of NV is variable and is set at boot time.
9      // This value will be the same for each boot, but is not necessarily known
10     // at compile time.
11     s_evictNvEnd = (NV_REF)NV_MEMORY_SIZE;
12     return;
13 }
```

8.5.3.2 NvCheckState()

Function to check the NV state by accessing the platform-specific function to get the NV state. The result state is registered in *s_NvIsAvailable* that will be reported by *NvIsAvailable()*. This function is called at the beginning of *ExecuteCommand()* before any potential check of *g_NvStatus*.

```

14 void
15 NvCheckState(
16     void
17 )
18 {
19     int    func_return;
20     //
21     func_return = _plat_IsNvAvailable();
22     if(func_return == 0)
23         g_NvStatus = TPM_RC_SUCCESS;
24     else if(func_return == 1)
25         g_NvStatus = TPM_RC_NV_UNAVAILABLE;
26     else
27         g_NvStatus = TPM_RC_NV_RATE;
28     return;
29 }

```

8.5.3.3 NvCommit()

This is a wrapper for the platform function to commit pending NV writes.

```

30 BOOL
31 NvCommit(
32     void
33 )
34 {
35     return (_plat_NvCommit() == 0);
36 }

```

8.5.3.4 NvPowerOn()

This function is called at *_TPM_Init* to initialize the NV environment.

Return Value	Meaning
TRUE(1)	all NV was initialized
FALSE(0)	the NV containing saved state had an error and TPM2_Startup(CLEAR) is required

```

37 BOOL
38 NvPowerOn(
39     void
40 )
41 {
42     int    nvError = 0;
43     // If power was lost, need to re-establish the RAM data that is loaded from
44     // NV and initialize the static variables
45     if(g_powerWasLost)
46     {
47         if((nvError = _plat_NVEnable(0)) < 0)
48             FAIL(FATAL_ERROR_NV_UNRECOVERABLE);
49         NvInitStatic();
50     }
51     return nvError == 0;
52 }

```

8.5.3.5 NvManufacture()

This function initializes the NV system at pre-install time.

This function should only be called in a manufacturing environment or in a simulation.

The layout of NV memory space is an implementation choice.

```

53 void
54 NvManufacture(
55     void
56 )
57 {
58 #if SIMULATION
59     // Simulate the NV memory being in the erased state.
60     _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
61 #endif
62     // Initialize static variables
63     NvInitStatic();
64     // Clear the RAM used for Orderly Index data
65     MemorySet(s_indexOrderlyRam, 0, RAM_INDEX_SPACE);
66     // Write that Orderly Index data to NV
67     NvUpdateIndexOrderlyData();
68     // Initialize the next offset of the first entry in evict/index list to 0 (the
69     // end of list marker) and the initial s_maxCounterValue;
70     NvSetMaxCount(0);
71     // Put the end of list marker at the end of memory. This contains the MaxCount
72     // value as well as the end marker.
73     NvWriteNvListEnd(NV_USER_DYNAMIC);
74     return;
75 }

```

8.5.3.6 NvRead()

This function is used to move reserved data from NV memory to RAM.

```

76 void
77 NvRead(
78     void          *outBuffer,    // OUT: buffer to receive data
79     UINT32        nvOffset,      // IN: offset in NV of value
80     UINT32        size,         // IN: size of the value to read
81 )
82 {
83     // Input type should be valid
84     pAssert(nvOffset + size < NV_MEMORY_SIZE);
85     _plat__NvMemoryRead(nvOffset, size, outBuffer);
86     return;
87 }

```

8.5.3.7 NvWrite()

This function is used to post reserved data for writing to NV memory. Before the TPM completes the operation, the value will be written.

```

88 BOOL
89 NvWrite(
90     UINT32        nvOffset,      // IN: location in NV to receive data
91     UINT32        size,         // IN: size of the data to move
92     void          *inBuffer     // IN: location containing data to write
93 )
94 {
95     // Input type should be valid

```

```

96     if(nvOffset + size <= NV_MEMORY_SIZE)
97     {
98         // Set the flag that a NV write happened
99         SET_NV_UPDATE(UT_NV);
100         return _plat__NvMemoryWrite(nvOffset, size, inBuffer);
101     }
102     return FALSE;
103 }

```

8.5.3.8 NvUpdatePersistent()

This function is used to update a value in the PERSISTENT_DATA structure and commits the value to NV.

```

104 void
105 NvUpdatePersistent(
106     UINT32      offset,          // IN: location in PERMANENT_DATA to be updated
107     UINT32      size,           // IN: size of the value
108     void        *buffer         // IN: the new data
109 )
110 {
111     pAssert(offset + size <= sizeof(gp));
112     MemoryCopy(&gp + offset, buffer, size);
113     NvWrite(offset, size, buffer);
114 }

```

8.5.3.9 NvClearPersistent()

This function is used to clear a persistent data entry and commit it to NV

```

115 void
116 NvClearPersistent(
117     UINT32      offset,          // IN: the offset in the PERMANENT_DATA
118                                     // structure to be cleared (zeroed)
119     UINT32      size             // IN: number of bytes to clear
120 )
121 {
122     pAssert(offset + size <= sizeof(gp));
123     MemorySet((&gp) + offset, 0, size);
124     NvWrite(offset, size, (&gp) + offset);
125 }

```

8.5.3.10 NvReadPersistent()

This function reads persistent data to the RAM copy of the *gp* structure.

```

126 void
127 NvReadPersistent(
128     void
129 )
130 {
131     NvRead(&gp, NV_PERSISTENT_DATA, sizeof(gp));
132     return;
133 }

```

8.6 Object.c

8.6.1 Introduction

This file contains the functions that manage the object store of the TPM.

8.6.2 Includes and Data Definitions

```
1  #define OBJECT_C
2
3  #include "Tpm.h"
```

8.6.3 Functions

8.6.3.1 ObjectFlush()

This function marks an object slot as available. Since there is no checking of the input parameters, it should be used judiciously.

NOTE This could be converted to a macro.

```
4  void
5  ObjectFlush(
6      OBJECT          *object
7  )
8  {
9      object->attributes.occupied = CLEAR;
10 }
```

8.6.3.2 ObjectSetInUse()

This access function sets the occupied attribute of an object slot.

```
11 void
12 ObjectSetInUse(
13     OBJECT          *object
14 )
15 {
16     object->attributes.occupied = SET;
17 }
```

8.6.3.3 ObjectStartup()

This function is called at TPM2_Startup() to initialize the object subsystem.

```
18 BOOL
19 ObjectStartup(
20     void
21 )
22 {
23     UINT32      i;
24     //
25     // object slots initialization
26     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
27     {
28         //Set the slot to not occupied
29         ObjectFlush(&s_objects[i]);
```



```

30     }
31     return TRUE;
32 }

```

8.6.3.4 ObjectCleanupEvict()

In this implementation, a persistent object is moved from NV into an object slot for processing. It is flushed after command execution. This function is called from ExecuteCommand().

```

33 void
34 ObjectCleanupEvict(
35     void
36 )
37 {
38     UINT32    i;
39     //
40     // This has to be iterated because a command may have two handles
41     // and they may both be persistent.
42     // This could be made to be more efficient so that a search is not needed.
43     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
44     {
45         // If an object is a temporary evict object, flush it from slot
46         OBJECT    *object = &s_objects[i];
47         if(object->attributes.evict == SET)
48             ObjectFlush(object);
49     }
50     return;
51 }

```

8.6.3.5 IsObjectPresent()

This function checks to see if a transient handle references a loaded object. This routine should not be called if the handle is not a transient handle. The function validates that the handle is in the implementation-dependent allowed in range for loaded transient objects.

Return Value	Meaning
TRUE(1)	handle references a loaded object
FALSE(0)	handle is not an object handle, or it does not reference to a loaded object

```

52 BOOL
53 IsObjectPresent(
54     TPMI_DH_OBJECT    handle        // IN: handle to be checked
55 )
56 {
57     UINT32    slotIndex = handle - TRANSIENT_FIRST;
58     // Since the handle is just an index into the array that is zero based, any
59     // handle value outside of the range of:
60     // TRANSIENT_FIRST -- (TRANSIENT_FIRST + MAX_LOADED_OBJECT - 1)
61     // will now be greater than or equal to MAX_LOADED_OBJECTS
62     if(slotIndex >= MAX_LOADED_OBJECTS)
63         return FALSE;
64     // Indicate if the slot is occupied
65     return (s_objects[slotIndex].attributes.occupied == TRUE);
66 }

```

8.6.3.6 ObjectIsSequence()

This function is used to check if the object is a sequence object. This function should not be called if the handle does not reference a loaded object.

Return Value	Meaning
TRUE(1)	object is an HMAC, hash, or event sequence object
FALSE(0)	object is not an HMAC, hash, or event sequence object

```

67  BOOL
68  ObjectIsSequence(
69      OBJECT          *object          // IN: handle to be checked
70  )
71  {
72      pAssert(object != NULL);
73      return (object->attributes.hmacSeq == SET
74          || object->attributes.hashSeq == SET
75          || object->attributes.eventSeq == SET);
76  }

```

8.6.3.7 HandleToObject()

This function is used to find the object structure associated with a handle.

This function requires that *handle* references a loaded object or a permanent handle.

```

77  OBJECT*
78  HandleToObject(
79      TPMI_DH_OBJECT    handle          // IN: handle of the object
80  )
81  {
82      UINT32              index;
83  //
84      // Return NULL if the handle references a permanent handle because there is no
85      // associated OBJECT.
86      if(HandleGetType(handle) == TPM_HT_PERMANENT)
87          return NULL;
88      // In this implementation, the handle is determined by the slot occupied by the
89      // object.
90      index = handle - TRANSIENT_FIRST;
91      pAssert(index < MAX_LOADED_OBJECTS);
92      pAssert(s_objects[index].attributes.occupied);
93      return &s_objects[index];
94  }

```

8.6.3.8 GetQualifiedName()

This function returns the Qualified Name of the object. In this implementation, the Qualified Name is computed when the object is loaded and is saved in the internal representation of the object. The alternative would be to retain the Name of the parent and compute the QN when needed. This would take the same amount of space so it is not recommended that the alternate be used.

This function requires that *handle* references a loaded object.

```

95  void
96  GetQualifiedName(
97      TPMI_DH_OBJECT    handle,          // IN: handle of the object
98      TPM2B_NAME        *qualifiedName  // OUT: qualified name of the object
99  )
100 {

```

```

101     OBJECT      *object;
102 //
103     switch(HandleGetType(handle))
104     {
105         case TPM_HT_PERMANENT:
106             qualifiedName->t.size = sizeof(TPM_HANDLE);
107             UINT32_TO_BYTE_ARRAY(handle, qualifiedName->t.name);
108             break;
109         case TPM_HT_TRANSIENT:
110             object = HandleToObject(handle);
111             if(object == NULL || object->publicArea.nameAlg == TPM_ALG_NULL)
112                 qualifiedName->t.size = 0;
113             else
114                 // Copy the name
115                 *qualifiedName = object->qualifiedName;
116             break;
117         default:
118             FAIL(FATAL_ERROR_INTERNAL);
119     }
120     return;
121 }

```

8.6.3.9 ObjectGetHierarchy()

This function returns the handle for the hierarchy of an object.

```

122 TPMI_RH_HIERARCHY
123 ObjectGetHierarchy(
124     OBJECT      *object          // IN :object
125 )
126 {
127     if(object->attributes.spsHierarchy)
128     {
129         return TPM_RH_OWNER;
130     }
131     else if(object->attributes.epsHierarchy)
132     {
133         return TPM_RH_ENDORSEMENT;
134     }
135     else if(object->attributes.ppsHierarchy)
136     {
137         return TPM_RH_PLATFORM;
138     }
139     else
140     {
141         return TPM_RH_NULL;
142     }
143 }

```

8.6.3.10 GetHierarchy()

This function returns the handle of the hierarchy to which a handle belongs. This function is similar to ObjectGetHierarchy() but this routine takes a handle but ObjectGetHierarchy() takes an pointer to an object.

This function requires that *handle* references a loaded object.

```

144 TPMI_RH_HIERARCHY
145 GetHierarchy(
146     TPMI_DH_OBJECT    handle          // IN :object handle
147 )
148 {
149     OBJECT      *object = HandleToObject(handle);

```

```

150 //
151 return ObjectGetHierarchy(object);
152 }

```

8.6.3.11 FindEmptyObjectSlot()

This function finds an open object slot, if any. It will clear the attributes but will not set the occupied attribute. This is so that a slot may be used and discarded if everything does not go as planned.

Return Value	Meaning*
NULL	no open slot found
!= NULL	pointer to available slot

```

153 OBJECT *
154 FindEmptyObjectSlot(
155     TPMI_DH_OBJECT *handle           // OUT: (optional)
156 )
157 {
158     UINT32 i;
159     OBJECT *object;
160     //
161     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
162     {
163         object = &s_objects[i];
164         if(object->attributes.occupied == CLEAR)
165         {
166             if(handle)
167                 *handle = i + TRANSIENT_FIRST;
168             // Initialize the object attributes
169             MemorySet(&object->attributes, 0, sizeof(OBJECT_ATTRIBUTES));
170             return object;
171         }
172     }
173     return NULL;
174 }

```

8.6.3.12 ObjectAllocateSlot()

This function is used to allocate a slot in internal object array.

```

175 OBJECT *
176 ObjectAllocateSlot(
177     TPMI_DH_OBJECT *handle           // OUT: handle of allocated object
178 )
179 {
180     OBJECT *object = FindEmptyObjectSlot(handle);
181     //
182     if(object != NULL)
183     {
184         // if found, mark as occupied
185         ObjectSetInUse(object);
186     }
187     return object;
188 }

```

8.6.3.13 ObjectSetLoadedAttributes()

This function sets the internal attributes for a loaded object. It is called to finalize the OBJECT attributes (not the TPMA_OBJECT attributes) for a loaded object.

```

189 void
190 ObjectSetLoadedAttributes(
191     OBJECT      *object,          // IN: object attributes to finalize
192     TPM_HANDLE  parentHandle     // IN: the parent handle
193 )
194 {
195     OBJECT      *parent = HandleToObject(parentHandle);
196     TPMA_OBJECT objectAttributes = object->publicArea.objectAttributes;
197     //
198     // Copy the stClear attribute from the public area. This could be overwritten
199     // if the parent has stClear SET
200     object->attributes.stClear =
201         IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear);
202     // If parent handle is a permanent handle, it is a primary (unless it is NULL
203     if(parent == NULL)
204     {
205         object->attributes.primary = SET;
206         switch(parentHandle)
207         {
208             case TPM_RH_ENDORSEMENT:
209                 object->attributes.epsHierarchy = SET;
210                 break;
211             case TPM_RH_OWNER:
212                 object->attributes.spsHierarchy = SET;
213                 break;
214             case TPM_RH_PLATFORM:
215                 object->attributes.ppsHierarchy = SET;
216                 break;
217             default:
218                 // Treat the temporary attribute as a hierarchy
219                 object->attributes.temporary = SET;
220                 object->attributes.primary = CLEAR;
221                 break;
222         }
223     }
224     else
225     {
226         // is this a stClear object
227         object->attributes.stClear =
228             (IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear)
229              || (parent->attributes.stClear == SET));
230         object->attributes.epsHierarchy = parent->attributes.epsHierarchy;
231         object->attributes.spsHierarchy = parent->attributes.spsHierarchy;
232         object->attributes.ppsHierarchy = parent->attributes.ppsHierarchy;
233         // An object is temporary if its parent is temporary or if the object
234         // is external
235         object->attributes.temporary = parent->attributes.temporary
236             || object->attributes.external;
237     }
238     // If this is an external object, set the QN == name but don't SET other
239     // key properties ('parent' or 'derived')
240     if(object->attributes.external)
241         object->qualifiedName = object->name;
242     else
243     {
244         // check attributes for different types of parents
245         if(IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, restricted)
246            && !object->attributes.publicOnly
247            && IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, decrypt)
248            && object->publicArea.nameAlg != TPM_ALG_NULL)
249         {
250             // This is a parent. If it is not a KEYEDHASH, it is an ordinary parent.
251             // Otherwise, it is a derivation parent.
252             if(object->publicArea.type == TPM_ALG_KEYEDHASH)
253                 object->attributes.derivation = SET;
254             else

```

```

255         object->attributes.isParent = SET;
256     }
257     ComputeQualifiedName(parentHandle, object->publicArea.nameAlg,
258                         &object->name, &object->qualifiedName);
259 }
260 // Set slot occupied
261 ObjectSetInUse(object);
262 return;
263 }

```

8.6.3.14 ObjectLoad()

Common function to load an object. A loaded object has its public area validated (unless its *nameAlg* is TPM_ALG_NULL). If a sensitive part is loaded, it is verified to be correct and if both public and sensitive parts are loaded, then the cryptographic binding between the objects is validated. This function does not cause the allocated slot to be marked as in use.

```

264 TPM_RC
265 ObjectLoad(
266     OBJECT      *object,           // IN: pointer to object slot
267                                     // object
268     OBJECT      *parent,           // IN: (optional) the parent object
269     TPMT_PUBLIC *publicArea,       // IN: public area to be installed in the object
270     TPMT_SENSITIVE *sensitive,     // IN: (optional) sensitive area to be
271                                     // installed in the object
272     TPM_RC      blamePublic,       // IN: parameter number to associate with the
273                                     // publicArea errors
274     TPM_RC      blameSensitive,    // IN: parameter number to associate with the
275                                     // sensitive area errors
276     TPM2B_NAME  *name              // IN: (optional)
277 )
278 {
279     TPM_RC      result = TPM_RC_SUCCESS;
280     //
281     // Do validations of public area object descriptions
282     pAssert(publicArea != NULL);
283
284     // Is this public only or a no-name object?
285     if(sensitive == NULL || publicArea->nameAlg == TPM_ALG_NULL)
286     {
287         // Need to have schemes checked so that we do the right thing with the
288         // public key.
289         result = SchemeChecks(NULL, publicArea);
290     }
291     else
292     {
293         // For any sensitive area, make sure that the seedSize is no larger than the
294         // digest size of nameAlg
295         if(sensitive->seedValue.t.size > CryptHashGetDigestSize(publicArea->nameAlg))
296             return TPM_RCS_KEY_SIZE + blameSensitive;
297         // Check attributes and schemes for consistency
298         result = PublicAttributesValidation(parent, publicArea);
299     }
300     if(result != TPM_RC_SUCCESS)
301         return RcSafeAddToResult(result, blamePublic);
302
303     // Sensitive area and binding checks
304
305     // On load, check nothing if the parent is fixedTPM. For all other cases, validate
306     // the keys.
307     if((parent == NULL)
308        || ((parent != NULL) && !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
309                                              TPMA_OBJECT, fixedTPM)))
310     {

```

```

311         // Do the cryptographic key validation
312         result = CryptValidateKeys(publicArea, sensitive, blamePublic,
313                                   blameSensitive);
314         if(result != TPM_RC_SUCCESS)
315             return result;
316     }
317 #if ALG_RSA
318     // If this is an RSA key, then expand the private exponent.
319     // Note: ObjectLoad() is only called by TPM2_Import() if the parent is fixedTPM.
320     // For any key that does not have a fixedTPM parent, the exponent is computed
321     // whenever it is loaded
322     if((publicArea->type == TPM_ALG_RSA) && (sensitive != NULL))
323     {
324         result = CryptRsaLoadPrivateExponent(publicArea, sensitive);
325         if(result != TPM_RC_SUCCESS)
326             return result;
327     }
328 #endif // ALG_RSA
329     // See if there is an object to populate
330     if((result == TPM_RC_SUCCESS) && (object != NULL))
331     {
332         // Initialize public
333         object->publicArea = *publicArea;
334         // Copy sensitive if there is one
335         if(sensitive == NULL)
336             object->attributes.publicOnly = SET;
337         else
338             object->sensitive = *sensitive;
339         // Set the name, if one was provided
340         if(name != NULL)
341             object->name = *name;
342         else
343             object->name.t.size = 0;
344     }
345     return result;
346 }

```

8.6.3.15 AllocateSequenceSlot()

This function allocates a sequence slot and initializes the parts that are used by the normal objects so that a sequence object is not inadvertently used for an operation that is not appropriate for a sequence.

```

347 static HASH_OBJECT *
348 AllocateSequenceSlot(
349     TPM_HANDLE      *newHandle,      // OUT: receives the allocated handle
350     TPM2B_AUTH      *auth           // IN: the authValue for the slot
351 )
352 {
353     HASH_OBJECT      *object = (HASH_OBJECT *)ObjectAllocateSlot(newHandle);
354     //
355     // Validate that the proper location of the hash state data relative to the
356     // object state data. It would be good if this could have been done at compile
357     // time but it can't so do it in something that can be removed after debug.
358     cAssert(sizeof(HASH_OBJECT, auth) == sizeof(OBJECT, publicArea.authPolicy));
359
360     if(object != NULL)
361     {
362         // Set the common values that a sequence object shares with an ordinary object
363         // First, clear all attributes
364         MemorySet(&object->objectAttributes, 0, sizeof(TPMA_OBJECT));
365
366         // The type is TPM_ALG_NULL
367         object->type = TPM_ALG_NULL;
368     }

```



```

369
370     // This has no name algorithm and the name is the Empty Buffer
371     object->nameAlg = TPM_ALG_NULL;
372
373     // A sequence object is considered to be in the NULL hierarchy so it should
374     // be marked as temporary so that it can't be persisted
375     object->attributes.temporary = SET;
376
377     // A sequence object is DA exempt.
378     SET_ATTRIBUTE(object->objectAttributes, TPMA_OBJECT, noDA);
379
380     // Copy the authorization value
381     if(auth != NULL)
382         object->auth = *auth;
383     else
384         object->auth.t.size = 0;
385 }
386 return object;
387 }
388 #if CC_HMAC_Start || CC_MAC_Start

```

8.6.3.16 ObjectCreateHMACSequence()

This function creates an internal HMAC sequence object.

Error Return	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```

389 TPM_RC
390 ObjectCreateHMACSequence(
391     TPMI_ALG_HASH    hashAlg,           // IN: hash algorithm
392     OBJECT           *keyObject,        // IN: the object containing the HMAC key
393     TPM2B_AUTH       *auth,            // IN: authValue
394     TPMI_DH_OBJECT   *newHandle        // OUT: HMAC sequence object handle
395 )
396 {
397     HASH_OBJECT       *hmacObject;
398     //
399     // Try to allocate a slot for new object
400     hmacObject = AllocateSequenceSlot(newHandle, auth);
401
402     if(hmacObject == NULL)
403         return TPM_RC_OBJECT_MEMORY;
404     // Set HMAC sequence bit
405     hmacObject->attributes.hmacSeq = SET;
406
407     #if !SMAC_IMPLEMENTED
408     if(CryptHmacStart(&hmacObject->state.hmacState, hashAlg,
409                     keyObject->sensitive.sensitive.bits.b.size,
410                     keyObject->sensitive.sensitive.bits.b.buffer) == 0)
411     #else
412     if(CryptMacStart(&hmacObject->state.hmacState,
413                    &keyObject->publicArea.parameters,
414                    hashAlg, &keyObject->sensitive.sensitive.any.b) == 0)
415     #endif // SMAC_IMPLEMENTED
416         return TPM_RC_FAILURE;
417     return TPM_RC_SUCCESS;
418 }
419 #endif

```

8.6.3.17 ObjectCreateHashSequence()

This function creates a hash sequence object.

Error Return	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```

420 TPM_RC
421 ObjectCreateHashSequence(
422     TPMI_ALG_HASH    hashAlg,          // IN: hash algorithm
423     TPM2B_AUTH        *auth,           // IN: authValue
424     TPMI_DH_OBJECT    *newHandle        // OUT: sequence object handle
425 )
426 {
427     HASH_OBJECT        *hashObject = AllocateSequenceSlot(newHandle, auth);
428     //
429     // See if slot allocated
430     if(hashObject == NULL)
431         return TPM_RC_OBJECT_MEMORY;
432     // Set hash sequence bit
433     hashObject->attributes.hashSeq = SET;
434
435     // Start hash for hash sequence
436     CryptHashStart(&hashObject->state.hashState[0], hashAlg);
437
438     return TPM_RC_SUCCESS;
439 }

```

8.6.3.18 ObjectCreateEventSequence()

This function creates an event sequence object.

Error Return	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```

440 TPM_RC
441 ObjectCreateEventSequence(
442     TPM2B_AUTH        *auth,           // IN: authValue
443     TPMI_DH_OBJECT    *newHandle        // OUT: sequence object handle
444 )
445 {
446     HASH_OBJECT        *hashObject = AllocateSequenceSlot(newHandle, auth);
447     UINT32              count;
448     TPM_ALG_ID          hash;
449     //
450     // See if slot allocated
451     if(hashObject == NULL)
452         return TPM_RC_OBJECT_MEMORY;
453     // Set the event sequence attribute
454     hashObject->attributes.eventSeq = SET;
455
456     // Initialize hash states for each implemented PCR algorithms
457     for(count = 0; (hash = CryptHashGetAlgByIndex(count)) != TPM_ALG_NULL; count++)
458         CryptHashStart(&hashObject->state.hashState[count], hash);
459     return TPM_RC_SUCCESS;
460 }

```

8.6.3.19 ObjectTerminateEvent()

This function is called to close out the event sequence and clean up the hash context states.

```

461 void
462 ObjectTerminateEvent(
463     void
464 )
465 {
466     HASH_OBJECT      *hashObject;
467     int               count;
468     BYTE              buffer[MAX_DIGEST_SIZE];
469     //
470     hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
471
472     // Don't assume that this is a proper sequence object
473     if(hashObject->attributes.eventSeq)
474     {
475         // If it is, close any open hash contexts. This is done in case
476         // the cryptographic implementation has some context values that need to be
477         // cleaned up (hygiene).
478         //
479         for(count = 0; CryptHashGetAlgByIndex(count) != TPM_ALG_NULL; count++)
480         {
481             CryptHashEnd(&hashObject->state.hashState[count], 0, buffer);
482         }
483         // Flush sequence object
484         FlushObject(g_DRTMHandle);
485     }
486     g_DRTMHandle = TPM_RH_UNASSIGNED;
487 }

```

8.6.3.20 ObjectContextLoad()

This function loads an object from a saved object context.

Return Value	Meaning*
NULL	if there is no free slot for an object
!= NULL	points to the loaded object

```

488 OBJECT *
489 ObjectContextLoad(
490     ANY_OBJECT_BUFFER *object,           // IN: pointer to object structure in saved
491                                           // context
492     TPMI_DH_OBJECT    *handle           // OUT: object handle
493 )
494 {
495     OBJECT      *newObject = ObjectAllocateSlot(handle);
496     //
497     // Try to allocate a slot for new object
498     if(newObject != NULL)
499     {
500         // Copy the first part of the object
501         MemoryCopy(newObject, object, offsetof(HASH_OBJECT, state));
502         // See if this is a sequence object
503         if(ObjectIsSequence(newObject))
504         {
505             // If this is a sequence object, import the data
506             SequenceDataImport((HASH_OBJECT *)newObject,
507                               (HASH_OBJECT_BUFFER *)object);
508         }
509         else
510         {
511             // Copy input object data to internal structure
512             MemoryCopy(newObject, object, sizeof(OBJECT));
513         }
514     }
515 }

```

```

514     }
515     return newObject;
516 }

```

8.6.3.21 FlushObject()

This function frees an object slot.

This function requires that the object is loaded.

```

517 void
518 FlushObject(
519     TPMI_DH_OBJECT    handle           // IN: handle to be freed
520 )
521 {
522     UINT32    index = handle - TRANSIENT_FIRST;
523     //
524     pAssert(index < MAX_LOADED_OBJECTS);
525     // Clear all the object attributes
526     MemorySet((BYTE*)&(s_objects[index].attributes),
527         0, sizeof(OBJECT_ATTRIBUTES));
528     return;
529 }

```

8.6.3.22 ObjectFlushHierarchy()

This function is called to flush all the loaded transient objects associated with a hierarchy when the hierarchy is disabled.

```

530 void
531 ObjectFlushHierarchy(
532     TPMI_RH_HIERARCHY    hierarchy     // IN: hierarchy to be flush
533 )
534 {
535     UINT16    i;
536     //
537     // iterate object slots
538     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
539     {
540         if(s_objects[i].attributes.occupied)           // If found an occupied slot
541         {
542             switch(hierarchy)
543             {
544                 case TPM_RH_PLATFORM:
545                     if(s_objects[i].attributes.ppsHierarchy == SET)
546                         s_objects[i].attributes.occupied = FALSE;
547                     break;
548                 case TPM_RH_OWNER:
549                     if(s_objects[i].attributes.spsHierarchy == SET)
550                         s_objects[i].attributes.occupied = FALSE;
551                     break;
552                 case TPM_RH_ENDORSEMENT:
553                     if(s_objects[i].attributes.epsHierarchy == SET)
554                         s_objects[i].attributes.occupied = FALSE;
555                     break;
556                 default:
557                     FAIL(FATAL_ERROR_INTERNAL);
558                     break;
559             }
560         }
561     }
562     return;
563 }

```

8.6.3.23 ObjectLoadEvict()

This function loads a persistent object into a transient object slot.

This function requires that *handle* is associated with a persistent object.

Error Return	Meaning
TPM_RC_HANDLE	the persistent object does not exist or the associated hierarchy is disabled.
TPM_RC_OBJECT_MEMORY	no object slot

```

564 TPM_RC
565 ObjectLoadEvict(
566     TPM_HANDLE      *handle,          // IN:OUT: evict object handle.  If success, it
567                                     // will be replace by the loaded object handle
568     COMMAND_INDEX   commandIndex     // IN: the command being processed
569 )
570 {
571     TPM_RC          result;
572     TPM_HANDLE      evictHandle = *handle; // Save the evict handle
573     OBJECT          *object;
574 //
575 // If this is an index that references a persistent object created by
576 // the platform, then return TPM_RH_HANDLE if the phEnable is FALSE
577 if(*handle >= PLATFORM_PERSISTENT)
578 {
579     // belongs to platform
580     if(g_phEnable == CLEAR)
581         return TPM_RC_HANDLE;
582 }
583 // belongs to owner
584 else if(gc.shEnable == CLEAR)
585     return TPM_RC_HANDLE;
586 // Try to allocate a slot for an object
587 object = ObjectAllocateSlot(handle);
588 if(object == NULL)
589     return TPM_RC_OBJECT_MEMORY;
590 // Copy persistent object to transient object slot. A TPM_RC_HANDLE
591 // may be returned at this point. This will mark the slot as containing
592 // a transient object so that it will be flushed at the end of the
593 // command
594 result = NvGetEvictObject(evictHandle, object);
595
596 // Bail out if this failed
597 if(result != TPM_RC_SUCCESS)
598     return result;
599 // check the object to see if it is in the endorsement hierarchy
600 // if it is and this is not a TPM2_EvictControl() command, indicate
601 // that the hierarchy is disabled.
602 // If the associated hierarchy is disabled, make it look like the
603 // handle is not defined
604 if(ObjectGetHierarchy(object) == TPM_RH_ENDORSEMENT
605    && gc.ehEnable == CLEAR
606    && GetCommandCode(commandIndex) != TPM_CC_EvictControl)
607     return TPM_RC_HANDLE;
608
609 return result;
610 }

```

8.6.3.24 ObjectComputeName()

This does the name computation from a public area (can be marshaled or not).

```

611 TPM2B_NAME *
612 ObjectComputeName(
613     UINT32          size,           // IN: the size of the area to digest
614     BYTE            *publicArea,    // IN: the public area to digest
615     TPM_ALG_ID       nameAlg,       // IN: the hash algorithm to use
616     TPM2B_NAME       *name         // OUT: Computed name
617 )
618 {
619     // Hash the publicArea into the name buffer leaving room for the nameAlg
620     name->t.size = CryptHashBlock(nameAlg, size, publicArea,
621                                 sizeof(name->t.name) - 2,
622                                 &name->t.name[2]);
623     // set the nameAlg
624     UINT16_TO_BYTE_ARRAY(nameAlg, name->t.name);
625     name->t.size += 2;
626     return name;
627 }

```

8.6.3.25 PublicMarshalAndComputeName()

This function computes the Name of an object from its public area.

```

628 TPM2B_NAME *
629 PublicMarshalAndComputeName(
630     TPMT_PUBLIC      *publicArea,    // IN: public area of an object
631     TPM2B_NAME       *name         // OUT: name of the object
632 )
633 {
634     // Will marshal a public area into a template. This is because the internal
635     // format for a TPMT_PUBLIC is a structure and not a simple BYTE buffer.
636     TPM2B_TEMPLATE    marshaled;     // this is big enough to hold a
637                                     // marshaled TPMT_PUBLIC
638     BYTE              *buffer = (BYTE *)&marshaled.t.buffer;
639     //
640     // if the nameAlg is NULL then there is no name.
641     if(publicArea->nameAlg == TPM_ALG_NULL)
642         name->t.size = 0;
643     else
644     {
645         // Marshal the public area into its canonical form
646         marshaled.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
647         // and compute the name
648         ObjectComputeName(marshaled.t.size, marshaled.t.buffer,
649                           publicArea->nameAlg, name);
650     }
651     return name;
652 }

```

8.6.3.26 ComputeQualifiedName()

This function computes the qualified name of an object.

```

653 void
654 ComputeQualifiedName(
655     TPM_HANDLE        parentHandle,  // IN: parent's handle
656     TPM_ALG_ID        nameAlg,       // IN: name hash
657     TPM2B_NAME        *name,         // IN: name of the object
658     TPM2B_NAME        *qualifiedName // OUT: qualified name of the object
659 )
660 {
661     HASH_STATE        hashState;     // hash state
662     TPM2B_NAME        parentName;
663     //

```

```

664     if(parentHandle == TPM_RH_UNASSIGNED)
665     {
666         MemoryCopy2B(&qualifiedName->b, &name->b, sizeof(qualifiedName->t.name));
667         *qualifiedName = *name;
668     }
669     else
670     {
671         GetQualifiedName(parentHandle, &parentName);
672
673         //      QN_A = hash_A (QN of parent || NAME_A)
674
675         // Start hash
676         qualifiedName->t.size = CryptHashStart(&hashState, nameAlg);
677
678         // Add parent's qualified name
679         CryptDigestUpdate2B(&hashState, &parentName.b);
680
681         // Add self name
682         CryptDigestUpdate2B(&hashState, &name->b);
683
684         // Complete hash leaving room for the name algorithm
685         CryptHashEnd(&hashState, qualifiedName->t.size,
686                     &qualifiedName->t.name[2]);
687         UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
688         qualifiedName->t.size += 2;
689     }
690     return;
691 }

```

8.6.3.27 ObjectIsStorage()

This function determines if an object has the attributes associated with a parent. A parent is an asymmetric or symmetric block cipher key that has its *restricted* and *decrypt* attributes SET, and *sign* CLEAR.

Return Value	Meaning
TRUE(1)	object is a storage key
FALSE(0)	object is not a storage key

```

692  BOOL
693  ObjectIsStorage(
694      TPMI_DH_OBJECT    handle           // IN: object handle
695  )
696  {
697      OBJECT             *object = HandleToObject(handle);
698      TPMT_PUBLIC         *publicArea = ((object != NULL) ? &object->publicArea : NULL);
699      //
700      return (publicArea != NULL
701              && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
702              && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
703              && !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
704              && (object->publicArea.type == TPM_ALG_RSA
705                  || object->publicArea.type == TPM_ALG_ECC));
706  }

```

8.6.3.28 ObjectCapGetLoaded()

This function returns a list of handles of loaded object, starting from *handle*. *Handle* must be in the range of valid transient object handles, but does not have to be the handle of a loaded transient object.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```

707 TPMI_YES_NO
708 ObjectCapGetLoaded(
709     TPMI_DH_OBJECT    handle,        // IN: start handle
710     UINT32             count,        // IN: count of returned handles
711     TPML_HANDLE       *handleList    // OUT: list of handle
712 )
713 {
714     TPMI_YES_NO        more = NO;
715     UINT32             i;
716     //
717     pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
718
719     // Initialize output handle list
720     handleList->count = 0;
721
722     // The maximum count of handles we may return is MAX_CAP_HANDLES
723     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
724
725     // Iterate object slots to get loaded object handles
726     for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
727     {
728         if(s_objects[i].attributes.occupied == TRUE)
729         {
730             // A valid transient object can not be the copy of a persistent object
731             pAssert(s_objects[i].attributes.evict == CLEAR);
732
733             if(handleList->count < count)
734             {
735                 // If we have not filled up the return list, add this object
736                 // handle to it
737                 handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
738                 handleList->count++;
739             }
740             else
741             {
742                 // If the return list is full but we still have loaded object
743                 // available, report this and stop iterating
744                 more = YES;
745                 break;
746             }
747         }
748     }
749     return more;
750 }

```

8.6.3.29 ObjectCapGetTransientAvail()

This function returns an estimate of the number of additional transient objects that could be loaded into the TPM.

```

751 UINT32
752 ObjectCapGetTransientAvail(
753     void
754 )
755 {
756     UINT32    i;
757     UINT32    num = 0;
758     //

```

```
759     // Iterate object slot to get the number of unoccupied slots
760     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
761     {
762         if(s_objects[i].attributes.occupied == FALSE) num++;
763     }
764     return num;
765 }
```

8.6.3.30 ObjectGetPublicAttributes()

Returns the attributes associated with an object handles.

```
766     TPMA_OBJECT
767     ObjectGetPublicAttributes(
768         TPM_HANDLE      handle
769     )
770     {
771         return HandleToObject(handle)->publicArea.objectAttributes;
772     }
773     OBJECT_ATTRIBUTES
774     ObjectGetProperties(
775         TPM_HANDLE      handle
776     )
777     {
778         return HandleToObject(handle)->attributes;
779     }
```

8.7 PCR.c

8.7.1 Introduction

This function contains the functions needed for PCR access and manipulation.

This implementation uses a static allocation for the PCR. The amount of memory is allocated based on the number of PCR in the implementation and the number of implemented hash algorithms. This is not the expected implementation. PCR SPACE DEFINITIONS.

In the definitions below, the *g_hashPcrMap* is a bit array that indicates which of the PCR are implemented. The *g_hashPcr* array is an array of digests. In this implementation, the space is allocated whether the PCR is implemented or not.

8.7.2 Includes, Defines, and Data Definitions

```
1  #define PCR_C
2  #include "Tpm.h"
```

The initial value of PCR attributes. The value of these fields should be consistent with PC Client specification. In this implementation, we assume the total number of implemented PCR is 24.

```
3  static const PCR_Attributes s_initAttributes[] =
4  {
5      // PCR 0 - 15, static RTM
6      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
7      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
8      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
9      {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
10
11     {0, 0x0F, 0x1F}, // PCR 16, Debug
12     {0, 0x10, 0x1C}, // PCR 17, Locality 4
13     {0, 0x10, 0x1C}, // PCR 18, Locality 3
14     {0, 0x10, 0x0C}, // PCR 19, Locality 2
15     {0, 0x14, 0x0E}, // PCR 20, Locality 1
16     {0, 0x14, 0x04}, // PCR 21, Dynamic OS
17     {0, 0x14, 0x04}, // PCR 22, Dynamic OS
18     {0, 0x0F, 0x1F}, // PCR 23, Application specific
19     {0, 0x0F, 0x1F}, // PCR 24, testing policy
20 };
```

8.7.3 Functions

8.7.3.1 PCRBelongsAuthGroup()

This function indicates if a PCR belongs to a group that requires an *authValue* in order to modify the PCR. If it does, *groupIndex* is set to value of the group index. This feature of PCR is decided by the platform specification.

Return Value	Meaning
TRUE(1)	PCR belongs an authorization group
FALSE(0)	PCR does not belong an authorization group

```
21  BOOL
22  PCRBelongsAuthGroup(
23      TPMI_DH_PCR    handle, // IN: handle of PCR
24      UINT32          *groupIndex // OUT: group index if PCR belongs a
```

```

25                                     //      group that allows authValue.  If PCR
26                                     //      does not belong to an authorization
27                                     //      group, the value in this parameter is
28                                     //      invalid
29      )
30  {
31  #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
32      // Platform specification determines to which authorization group a PCR belongs
33      // (if any). In this implementation, we assume there is only
34      // one authorization group which contains PCR[20-22]. If the platform
35      // specification requires differently, the implementation should be changed
36      // accordingly
37      if(handle >= 20 && handle <= 22)
38      {
39          *groupIndex = 0;
40          return TRUE;
41      }
42  #endif
43      return FALSE;
44  }

```

8.7.3.2 PCRBelongsPolicyGroup()

This function indicates if a PCR belongs to a group that requires a policy authorization in order to modify the PCR. If it does, *groupIndex* is set to value of the group index. This feature of PCR is decided by the platform specification.

Return Value	Meaning
TRUE(1)	PCR belongs to a policy group
FALSE(0)	PCR does not belong to a policy group

```

45  BOOL
46  PCRBelongsPolicyGroup(
47      TPMI_DH_PCR      handle,          // IN: handle of PCR
48      UINT32           *groupIndex      // OUT: group index if PCR belongs a group that
49                                          // allows policy. If PCR does not belong to
50                                          // a policy group, the value in this
51                                          // parameter is invalid
52  )
53  {
54  #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
55      // Platform specification decides if a PCR belongs to a policy group and
56      // belongs to which group. In this implementation, we assume there is only
57      // one policy group which contains PCR20-22. If the platform specification
58      // requires differently, the implementation should be changed accordingly
59      if(handle >= 20 && handle <= 22)
60      {
61          *groupIndex = 0;
62          return TRUE;
63      }
64  #endif
65      return FALSE;
66  }

```

8.7.3.3 PCRBelongsTCBGroup()

This function indicates if a PCR belongs to the TCB group.

Return Value	Meaning
TRUE(1)	PCR belongs to a TCB group
FALSE(0)	PCR does not belong to a TCB group

```

67 static BOOL
68 PCRBelongsTCBGroup(
69     TPMI_DH_PCR    handle           // IN: handle of PCR
70 )
71 {
72     #if ENABLE_PCR_NO_INCREMENT == YES
73         // Platform specification decides if a PCR belongs to a TCB group. In this
74         // implementation, we assume PCR[20-22] belong to TCB group. If the platform
75         // specification requires differently, the implementation should be
76         // changed accordingly
77         if(handle >= 20 && handle <= 22)
78             return TRUE;
79     #endif
80     return FALSE;
81 }
82

```

8.7.3.4 PCRPolicyIsAvailable()

This function indicates if a policy is available for a PCR.

Return Value	Meaning
TRUE(1)	the PCR may be authorized by policy
FALSE(0)	the PCR does not allow policy

```

83 BOOL
84 PCRPolicyIsAvailable(
85     TPMI_DH_PCR    handle           // IN: PCR handle
86 )
87 {
88     UINT32          groupIndex;
89     return PCRBelongsPolicyGroup(handle, &groupIndex);
90 }
91

```

8.7.3.5 PCRGetAuthValue()

This function is used to access the *authValue* of a PCR. If PCR does not belong to an *authValue* group, an EmptyAuth() will be returned.

```

92 TPM2B_AUTH *
93 PCRGetAuthValue(
94     TPMI_DH_PCR    handle           // IN: PCR handle
95 )
96 {
97     UINT32          groupIndex;
98
99     if(PCRBelongsAuthGroup(handle, &groupIndex))
100     {
101         return &gc.pcrAuthValues.auth[groupIndex];
102     }
103     else
104     {
105         return NULL;
106     }
107 }

```

```

106     }
107 }

```

8.7.3.6 PCRGetAuthPolicy()

This function is used to access the authorization policy of a PCR. It sets *policy* to the authorization policy and returns the hash algorithm for policy. If the PCR does not allow a policy, TPM_ALG_NULL is returned.

```

108 TPMI_ALG_HASH
109 PCRGetAuthPolicy(
110     TPMI_DH_PCR    handle,          // IN: PCR handle
111     TPM2B_DIGEST   *policy         // OUT: policy of PCR
112 )
113 {
114     UINT32          groupIndex;
115
116     if(PCRBelongsPolicyGroup(handle, &groupIndex))
117     {
118         *policy = gp.pcrPolicies.policy[groupIndex];
119         return gp.pcrPolicies.hashAlg[groupIndex];
120     }
121     else
122     {
123         policy->t.size = 0;
124         return TPM_ALG_NULL;
125     }
126 }

```

8.7.3.7 PCRSimStart()

This function is used to initialize the policies when a TPM is manufactured. This function would only be called in a manufacturing environment or in a TPM simulator.

```

127 void
128 PCRSimStart(
129     void
130 )
131 {
132     UINT32 i;
133     #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
134     for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
135     {
136         gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
137         gp.pcrPolicies.policy[i].t.size = 0;
138     }
139     #endif
140     #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
141     for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
142     {
143         gp.pcrAuthValues.auth[i].t.size = 0;
144     }
145     #endif
146     // We need to give an initial configuration on allocated PCR before
147     // receiving any TPM2_PCR_Allocate command to change this configuration
148     // When the simulation environment starts, we allocate all the PCRs
149     for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
150        gp.pcrAllocated.count++)
151     {
152         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
153             = CryptHashGetAlgByIndex(gp.pcrAllocated.count);
154
155         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect

```

```

156         = PCR_SELECT_MAX;
157     for(i = 0; i < PCR_SELECT_MAX; i++)
158         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
159         = 0xFF;
160     }
161     // Store the initial configuration to NV
162     NV_SYNC_PERSISTENT(pcrPolicies);
163     NV_SYNC_PERSISTENT(pcrAllocated);
164
165     return;
166 }

```

8.7.3.8 GetSavedPcrPointer()

This function returns the address of an array of state saved PCR based on the hash algorithm.

Return Value	Meaning*
NULL	no such algorithm
!= NULL	pointer to the 0th byte of the 0th PCR

```

167 static BYTE *
168 GetSavedPcrPointer(
169     TPM_ALG_ID      alg,           // IN: algorithm for bank
170     UINT32          pcrIndex       // IN: PCR index in PCR_SAVE
171 )
172 {
173     BYTE            *retVal;
174     switch(alg)
175     {
176 #define HASH_CASE(HASH, Hash) \
177     case TPM_ALG_##HASH: \
178         retVal = gc.pcrSave.Hash[pcrIndex]; \
179         break;
180
181     FOR_EACH_HASH(HASH_CASE)
182 #undef HASH_CASE
183
184     default:
185         FAIL(FATAL_ERROR_INTERNAL);
186     }
187     return retVal;
188 }

```

8.7.3.9 PcrIsAllocated()

This function indicates if a PCR number for the particular hash algorithm is allocated.

Return Value	Meaning
TRUE(1)	PCR is allocated
FALSE(0)	PCR is not allocated

```

189 BOOL
190 PcrIsAllocated(
191     UINT32          pcr,           // IN: The number of the PCR
192     TPMI_ALG_HASH   hashAlg       // IN: The PCR algorithm
193 )
194 {
195     UINT32          i;
196     BOOL            allocated = FALSE;

```



```

197
198     if(pcr < IMPLEMENTATION_PCR)
199     {
200         for(i = 0; i < gp.pcrAllocated.count; i++)
201         {
202             if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
203             {
204                 if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr / 8])
205                     & (1 << (pcr % 8))) != 0)
206                     allocated = TRUE;
207                 else
208                     allocated = FALSE;
209                 break;
210             }
211         }
212     }
213     return allocated;
214 }

```

8.7.3.10 GetPcrPointer()

This function returns the address of an array of PCR based on the hash algorithm.

Return Value	Meaning*
NULL	no such algorithm
!= NULL	pointer to the 0th byte of the 0th PCR

```

215 static BYTE *
216 GetPcrPointer(
217     TPM_ALG_ID    alg,           // IN: algorithm for bank
218     UINT32         pcrNumber     // IN: PCR number
219 )
220 {
221     static BYTE    *pcr = NULL;
222     //
223     if(!PcrIsAllocated(pcrNumber, alg))
224         return NULL;
225
226     switch(alg)
227     {
228 #define HASH_CASE(HASH, Hash)           \
229     case TPM_ALG_ ##HASH:               \
230         pcr = s_pcrs[pcrNumber].Hash##Pcr; \
231         break;
232
233     FOR_EACH_HASH(HASH_CASE)
234 #undef HASH_CASE
235
236     default:
237         FAIL(FATAL_ERROR_INTERNAL);
238         break;
239     }
240     return pcr;
241 }

```

8.7.3.11 IsPcrSelected()

This function indicates if an indicated PCR number is selected by the bit map in *selection*.

Return Value	Meaning
TRUE(1)	PCR is selected
FALSE(0)	PCR is not selected

```

242 static BOOL
243 IsPcrSelected(
244     UINT32          pcr,           // IN: The number of the PCR
245     TPMS_PCR_SELECTION *selection // IN: The selection structure
246 )
247 {
248     BOOL          selected;
249     selected = (pcr < IMPLEMENTATION_PCR
250         && ((selection->pcrSelect[pcr / 8]) & (1 << (pcr % 8))) != 0);
251     return selected;
252 }

```

8.7.3.12 FilterPcr()

This function modifies a PCR selection array based on the implemented PCR.

```

253 static void
254 FilterPcr(
255     TPMS_PCR_SELECTION *selection // IN: input PCR selection
256 )
257 {
258     UINT32 i;
259     TPMS_PCR_SELECTION *allocated = NULL;
260
261     // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
262     for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
263         selection->pcrSelect[i] = 0;
264
265     // Find the internal configuration for the bank
266     for(i = 0; i < gp.pcrAllocated.count; i++)
267     {
268         if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
269         {
270             allocated = &gp.pcrAllocated.pcrSelections[i];
271             break;
272         }
273     }
274     for(i = 0; i < selection->sizeofSelect; i++)
275     {
276         if(allocated == NULL)
277         {
278             // If the required bank does not exist, clear input selection
279             selection->pcrSelect[i] = 0;
280         }
281         else
282             selection->pcrSelect[i] &= allocated->pcrSelect[i];
283     }
284     return;
285 }

```

8.7.3.13 PcrDrtm()

This function does the DRTM and H-CRTM processing it is called from _TPM_Hash_End.

```

286 void
287 PcrDrtm(

```

```

288     const TPMI_DH_PCR      pcrHandle,      // IN: the index of the PCR to be
289                               // modified
290     const TPMI_ALG_HASH    hash,           // IN: the bank identifier
291     const TPM2B_DIGEST     *digest         // IN: the digest to modify the PCR
292 )
293 {
294     BYTE      *pcrData = GetPcrPointer(hash, pcrHandle);
295
296     if(pcrData != NULL)
297     {
298         // Rest the PCR to zeros
299         MemorySet(pcrData, 0, digest->t.size);
300
301         // if the TPM has not started, then set the PCR to 0...04 and then extend
302         if(!TPMIsStarted())
303         {
304             pcrData[digest->t.size - 1] = 4;
305         }
306         // Now, extend the value
307         PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
308     }
309 }

```

8.7.3.14 PCR_ClearAuth()

This function is used to reset the PCR authorization values. It is called on TPM2_Startup(CLEAR) and TPM2_Clear().

```

310 void
311 PCR_ClearAuth(
312     void
313 )
314 {
315     #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
316         int j;
317         for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
318         {
319             gc.pcrAuthValues.auth[j].t.size = 0;
320         }
321     #endif
322 }

```

8.7.3.15 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```

323 BOOL
324 PCRStartup(
325     STARTUP_TYPE    type,           // IN: startup type
326     BYTE            locality        // IN: startup locality
327 )
328 {
329     UINT32          pcr, j;
330     UINT32          saveIndex = 0;
331
332     g_pcrReConfig = FALSE;
333
334     // Don't test for SU_RESET because that should be the default when nothing
335     // else is selected
336     if(type != SU_RESUME && type != SU_RESTART)
337     {
338         // PCR generation counter is cleared at TPM_RESET
339         gr.pcrCounter = 0;

```

```

340     }
341     // Initialize/Restore PCR values
342     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
343     {
344         // On resume, need to know if this PCR had its state saved or not
345         UINT32      stateSaved;
346
347         if(type == SU_RESUME
348            && s_initAttributes[pcr].stateSave == SET)
349         {
350             stateSaved = 1;
351         }
352         else
353         {
354             stateSaved = 0;
355             PCRChanged(pcr);
356         }
357         // If this is the H-CRTM PCR and we are not doing a resume and we
358         // had an H-CRTM event, then we don't change this PCR
359         if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
360             continue;
361
362         // Iterate each hash algorithm bank
363         for(j = 0; j < gp.pcrAllocated.count; j++)
364         {
365             TPMI_ALG_HASH    hash = gp.pcrAllocated.pcrSelections[j].hash;
366             BYTE              *pcrData = GetPcrPointer(hash, pcr);
367             UINT16            pcrSize = CryptHashGetDigestSize(hash);
368
369             if(pcrData != NULL)
370             {
371                 // if state was saved
372                 if(stateSaved == 1)
373                 {
374                     // Restore saved PCR value
375                     BYTE      *pcrSavedData;
376                     pcrSavedData = GetSavedPcrPointer(
377                         gp.pcrAllocated.pcrSelections[j].hash,
378                         saveIndex);
379                     if(pcrSavedData == NULL)
380                         return FALSE;
381                     MemoryCopy(pcrData, pcrSavedData, pcrSize);
382                 }
383                 else
384                 {
385                     // PCR was not restored by state save
386
387                     // If the reset locality of the PCR is 4, then
388                     // the reset value is all one's, otherwise it is
389                     // all zero.
390                     if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
391                         MemorySet(pcrData, 0xFF, pcrSize);
392                     else
393                     {
394                         MemorySet(pcrData, 0, pcrSize);
395                         if(pcr == HCRTM_PCR)
396                             pcrData[pcrSize - 1] = locality;
397                     }
398                 }
399             }
400             saveIndex += stateSaved;
401         }
402         // Reset authValues on TPM2_Startup(CLEAR)
403         if(type != SU_RESUME)
404             PCR_ClearAuth();
405         return TRUE;

```

406 }

8.7.3.16 PCRStateSave()

This function is used to save the PCR values that will be restored on TPM Resume.

```

407 void
408 PCRStateSave(
409     TPM_SU          type          // IN: startup type
410 )
411 {
412     UINT32          pcr, j;
413     UINT32          saveIndex = 0;
414
415     // if state save CLEAR, nothing to be done. Return here
416     if(type == TPM_SU_CLEAR)
417         return;
418
419     // Copy PCR values to the structure that should be saved to NV
420     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
421     {
422         UINT32 stateSaved = (s_initAttributes[pcr].stateSave == SET) ? 1 : 0;
423
424         // Iterate each hash algorithm bank
425         for(j = 0; j < gp.pcrAllocated.count; j++)
426         {
427             BYTE    *pcrData;
428             UINT32   pcrSize;
429
430             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
431
432             if(pcrData != NULL)
433             {
434                 pcrSize
435                 = CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
436
437                 if(stateSaved == 1)
438                 {
439                     // Restore saved PCR value
440                     BYTE    *pcrSavedData;
441                     pcrSavedData
442                     = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
443                                         saveIndex);
444                     MemoryCopy(pcrSavedData, pcrData, pcrSize);
445                 }
446             }
447             saveIndex += stateSaved;
448         }
449     }
450     return;
451 }
```

8.7.3.17 PCRIsStateSaved()

This function indicates if the selected PCR is a PCR that is state saved on TPM2_Shutdown(STATE). The return value is based on PCR attributes.

Return Value	Meaning
TRUE(1)	PCR is state saved
FALSE(0)	PCR is not state saved

```

452  BOOL
453  PCRIsStateSaved(
454      TPMI_DH_PCR    handle          // IN: PCR handle to be extended
455  )
456  {
457      UINT32          pcr = handle - PCR_FIRST;
458
459      if(s_initAttributes[pcr].stateSave == SET)
460          return TRUE;
461      else
462          return FALSE;
463  }

```

8.7.3.18 PCRIsResetAllowed()

This function indicates if a PCR may be reset by the current command locality. The return value is based on PCR attributes, and not the PCR allocation.

Return Value	Meaning
TRUE(1)	TPM2_PCR_Reset is allowed
FALSE(0)	TPM2_PCR_Reset is not allowed

```

464  BOOL
465  PCRIsResetAllowed(
466      TPMI_DH_PCR    handle          // IN: PCR handle to be extended
467  )
468  {
469      UINT8          commandLocality;
470      UINT8          localityBits = 1;
471      UINT32          pcr = handle - PCR_FIRST;
472
473      // Check for the locality
474      commandLocality = _plat__LocalityGet();
475
476      #ifdef DRTM_PCR
477          // For a TPM that does DRTM, Reset is not allowed at locality 4
478          if(commandLocality == 4)
479              return FALSE;
480      #endif
481
482      localityBits = localityBits << commandLocality;
483      if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
484          return FALSE;
485      else
486          return TRUE;
487  }

```

8.7.3.19 PCRChanged()

This function checks a PCR handle to see if the attributes for the PCR are set so that any change to the PCR causes an increment of the *pcrCounter*. If it does, then the function increments the counter. Will also bump the counter if the handle is zero which means that PCR 0 can not be in the TCB group. Bump on zero is used by TPM2_Clear().

```

488 void
489 PCRChanged(
490     TPM_HANDLE      pcrHandle      // IN: the handle of the PCR that changed.
491 )
492 {
493     // For the reference implementation, the only change that does not cause
494     // increment is a change to a PCR in the TCB group.
495     if((pcrHandle == 0) || !PCRBelongsTCBGroup(pcrHandle))
496     {
497         gr.pcrCounter++;
498         if(gr.pcrCounter == 0)
499             FAIL(FATAL_ERROR_COUNTER_OVERFLOW);
500     }
501 }

```

8.7.3.20 PCRIsExtendAllowed()

This function indicates a PCR may be extended at the current command locality. The return value is based on PCR attributes, and not the PCR allocation.

Return Value	Meaning
TRUE(1)	extend is allowed
FALSE(0)	extend is not allowed

```

502 BOOL
503 PCRIsExtendAllowed(
504     TPMI_DH_PCR      handle      // IN: PCR handle to be extended
505 )
506 {
507     UINT8      commandLocality;
508     UINT8      localityBits = 1;
509     UINT32     pcr = handle - PCR_FIRST;
510
511     // Check for the locality
512     commandLocality = _plat_LocalityGet();
513     localityBits = localityBits << commandLocality;
514     if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
515         return FALSE;
516     else
517         return TRUE;
518 }

```

8.7.3.21 PCRExtend()

This function is used to extend a PCR in a specific bank.

```

519 void
520 PCRExtend(
521     TPMI_DH_PCR      handle,      // IN: PCR handle to be extended
522     TPMI_ALG_HASH     hash,      // IN: hash algorithm of PCR
523     UINT32            size,      // IN: size of data to be extended
524     BYTE              *data      // IN: data to be extended
525 )
526 {
527     BYTE      *pcrData;
528     HASH_STATE hashState;
529     UINT16    pcrSize;
530
531     pcrData = GetPcrPointer(hash, handle - PCR_FIRST);
532
533     // Extend PCR if it is allocated

```



```

534     if(pcrData != NULL)
535     {
536         pcrSize = CryptHashGetDigestSize(hash);
537         CryptHashStart(&hashState, hash);
538         CryptDigestUpdate(&hashState, pcrSize, pcrData);
539         CryptDigestUpdate(&hashState, size, data);
540         CryptHashEnd(&hashState, pcrSize, pcrData);
541
542         // PCR has changed so update the pcrCounter if necessary
543         PCRChanged(handle);
544     }
545     return;
546 }

```

8.7.3.22 PCRComputeCurrentDigest()

This function computes the digest of the selected PCR.

As a side-effect, *selection* is modified so that only the implemented PCR will have their bits still set.

```

547 void
548 PCRComputeCurrentDigest(
549     TPMI_ALG_HASH      hashAlg,           // IN: hash algorithm to compute digest
550     TPML_PCR_SELECTION *selection,        // IN/OUT: PCR selection (filtered on
551                                           // output)
552     TPM2B_DIGEST        *digest           // OUT: digest
553 )
554 {
555     HASH_STATE          hashState;
556     TPMS_PCR_SELECTION *select;
557     BYTE                *pcrData;         // will point to a digest
558     UINT32               pcrSize;
559     UINT32               pcr;
560     UINT32               i;
561
562     // Initialize the hash
563     digest->t.size = CryptHashStart(&hashState, hashAlg);
564     pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
565
566     // Iterate through the list of PCR selection structures
567     for(i = 0; i < selection->count; i++)
568     {
569         // Point to the current selection
570         select = &selection->pcrSelections[i]; // Point to the current selection
571         FilterPcr(select);                     // Clear out the bits for unimplemented PCR
572
573         // Need the size of each digest
574         pcrSize = CryptHashGetDigestSize(select->pcrSelections[i].hash);
575
576         // Iterate through the selection
577         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
578         {
579             if(IsPcrSelected(pcr, select))      // Is this PCR selected
580             {
581                 // Get pointer to the digest data for the bank
582                 pcrData = GetPcrPointer(select->pcrSelections[i].hash, pcr);
583                 pAssert(pcrData != NULL);
584                 CryptDigestUpdate(&hashState, pcrSize, pcrData); // add to digest
585             }
586         }
587     }
588     // Complete hash stack
589     CryptHashEnd2B(&hashState, &digest->b);
590
591     return;

```

592 }

8.7.3.23 PCRRead()

This function is used to read a list of selected PCR. If the requested PCR number exceeds the maximum number that can be output, the *selection* is adjusted to reflect the actual output PCR.

```

593 void
594 PCRRead(
595     TPML_PCR_SELECTION *selection,    // IN/OUT: PCR selection (filtered on
596                                     // output)
597     TPML_DIGEST         *digest,      // OUT: digest
598     UINT32              *pcrCounter   // OUT: the current value of PCR generation
599                                     // number
600 )
601 {
602     TPMS_PCR_SELECTION *select;
603     BYTE               *pcrData;      // will point to a digest
604     UINT32              pcr;
605     UINT32              i;
606
607     digest->count = 0;
608
609     // Iterate through the list of PCR selection structures
610     for(i = 0; i < selection->count; i++)
611     {
612         // Point to the current selection
613         select = &selection->pcrSelections[i]; // Point to the current selection
614         FilterPcr(select);                      // Clear out the bits for unimplemented PCR
615
616         // Iterate through the selection
617         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
618         {
619             if(IsPcrSelected(pcr, select))      // Is this PCR selected
620             {
621                 // Check if number of digest exceed upper bound
622                 if(digest->count > 7)
623                 {
624                     // Clear rest of the current select bitmap
625                     while(pcr < IMPLEMENTATION_PCR
626                         // do not round up!
627                         && (pcr / 8) < select->sizeofSelect)
628                     {
629                         // do not round up!
630                         select->pcrSelect[pcr / 8] &= (BYTE)~(1 << (pcr % 8));
631                         pcr++;
632                     }
633                     // Exit inner loop
634                     break;
635                 }
636                 // Need the size of each digest
637                 digest->digests[digest->count].t.size =
638                     CryptHashGetDigestSize(select->pcrSelections[i].hash);
639
640                 // Get pointer to the digest data for the bank
641                 pcrData = GetPcrPointer(select->pcrSelections[i].hash, pcr);
642                 pAssert(pcrData != NULL);
643                 // Add to the data to digest
644                 MemoryCopy(digest->digests[digest->count].t.buffer,
645                             pcrData,
646                             digest->digests[digest->count].t.size);
647                 digest->count++;
648             }
649         }
650     }

```

```

650         // If we exit inner loop because we have exceed the output upper bound
651         if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
652         {
653             // Clear rest of the selection
654             while(i < selection->count)
655             {
656                 MemorySet(selection->pcrSelections[i].pcrSelect, 0,
657                     selection->pcrSelections[i].sizeofSelect);
658                 i++;
659             }
660             // exit outer loop
661             break;
662         }
663     }
664     *pcrCounter = gr.pcrCounter;
665
666     return;
667 }

```

8.7.3.24 PCRAllocate()

This function is used to change the PCR allocation.

Error Return	Meaning
TPM_RC_NO_RESULT	allocate failed
TPM_RC_PCR	improper allocation

```

668 TPM_RC
669 PCRAllocate(
670     TPML_PCR_SELECTION *allocate,        // IN: required allocation
671     UINT32 *maxPCR,                      // OUT: Maximum number of PCR
672     UINT32 *sizeNeeded,                  // OUT: required space
673     UINT32 *sizeAvailable                // OUT: available space
674 )
675 {
676     UINT32 i, j, k;
677     TPML_PCR_SELECTION newAllocate;
678     // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
679     BOOL pcrHcrtm = FALSE;
680     BOOL pcrDrtn = FALSE;
681
682     // Create the expected new PCR allocation based on the existing allocation
683     // and the new input:
684     // 1. if a PCR bank does not appear in the new allocation, the existing
685     // allocation of this PCR bank will be preserved.
686     // 2. if a PCR bank appears multiple times in the new allocation, only the
687     // last one will be in effect.
688     newAllocate = gp.pcrAllocated;
689     for(i = 0; i < allocate->count; i++)
690     {
691         for(j = 0; j < newAllocate.count; j++)
692         {
693             // If hash matches, the new allocation covers the old allocation
694             // for this particular bank.
695             // The assumption is the initial PCR allocation (from manufacture)
696             // has all the supported hash algorithms with an assigned bank
697             // (possibly empty). So there must be a match for any new bank
698             // allocation from the input.
699             if(newAllocate.pcrSelections[j].hash ==
700                 allocate->pcrSelections[i].hash)
701             {
702                 newAllocate.pcrSelections[j] = allocate->pcrSelections[i];

```

```

703         break;
704     }
705 }
706 // The j loop must exit with a match.
707 pAssert(j < newAllocate.count);
708 }
709 // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
710 *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
711 if(*maxPCR > IMPLEMENTATION_PCR)
712     *maxPCR = IMPLEMENTATION_PCR;
713
714 // Compute required size for allocation
715 *sizeNeeded = 0;
716 for(i = 0; i < newAllocate.count; i++)
717 {
718     UINT32    digestSize
719         = CryptHashGetDigestSize(newAllocate.pcrSelections[i].hash);
720 #if defined(DRTM_PCR)
721     // Make sure that we end up with at least one DRTM PCR
722     pcrDrtm = pcrDrtm || TestBit(DRTM_PCR,
723                                 newAllocate.pcrSelections[i].pcrSelect,
724                                 newAllocate.pcrSelections[i].sizeofSelect);
725 #else
726     // if DRTM PCR is not required, indicate that the allocation is OK
727     pcrDrtm = TRUE;
728 #endif
729
730 #if defined(HCRTM_PCR)
731     // and one HCRTP PCR (since this is usually PCR 0...)
732     pcrHcrtm = pcrHcrtm || TestBit(HCRTM_PCR,
733                                   newAllocate.pcrSelections[i].pcrSelect,
734                                   newAllocate.pcrSelections[i].sizeofSelect);
735 #else
736     pcrHcrtm = TRUE;
737 #endif
738 for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
739 {
740     BYTE    mask = 1;
741     for(k = 0; k < 8; k++)
742     {
743         if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
744             *sizeNeeded += digestSize;
745         mask = mask << 1;
746     }
747 }
748 }
749 if(!pcrDrtm || !pcrHcrtm)
750     return TPM_RC_PCR;
751
752 // In this particular implementation, we always have enough space to
753 // allocate PCR. Different implementation may return a sizeAvailable less
754 // than the sizeNeed.
755 *sizeAvailable = sizeof(s_pcrs);
756
757 // Save the required allocation to NV. Note that after NV is written, the
758 // PCR allocation in NV is no longer consistent with the RAM data
759 // gp.pcrAllocated. The NV version reflect the allocate after next
760 // TPM_RESET, while the RAM version reflects the current allocation
761 NV_WRITE_PERSISTENT(pcrAllocated, newAllocate);
762
763 return TPM_RC_SUCCESS;
764 }

```

8.7.3.25 PCRSetValue()

This function is used to set the designated PCR in all banks to an initial value. The initial value is signed and will be sign extended into the entire PCR.

```

765 void
766 PCRSetValue(
767     TPM_HANDLE    handle,          // IN: the handle of the PCR to set
768     INT8          initialValue     // IN: the value to set
769 )
770 {
771     int            i;
772     UINT32         pcr = handle - PCR_FIRST;
773     TPMI_ALG_HASH  hash;
774     UINT16         digestSize;
775     BYTE           *pcrData;
776
777     // Iterate supported PCR bank algorithms to reset
778     for(i = 0; i < HASH_COUNT; i++)
779     {
780         hash = CryptHashGetAlgByIndex(i);
781         // Prevent runaway
782         if(hash == TPM_ALG_NULL)
783             break;
784
785         // Get a pointer to the data
786         pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
787
788         // If the PCR is allocated
789         if(pcrData != NULL)
790         {
791             // And the size of the digest
792             digestSize = CryptHashGetDigestSize(hash);
793
794             // Set the LSO to the input value
795             pcrData[digestSize - 1] = initialValue;
796
797             // Sign extend
798             if(initialValue >= 0)
799                 MemorySet(pcrData, 0, digestSize - 1);
800             else
801                 MemorySet(pcrData, -1, digestSize - 1);
802         }
803     }
804 }

```

8.7.3.26 PCRResetDynamics()

This function is used to reset a dynamic PCR to 0. This function is used in DRTM sequence.

```

805 void
806 PCRResetDynamics(
807     void
808 )
809 {
810     UINT32         pcr, i;
811
812     // Initialize PCR values
813     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
814     {
815         // Iterate each hash algorithm bank
816         for(i = 0; i < gp.pcrAllocated.count; i++)
817         {

```

```

818         BYTE    *pcrData;
819         UINT32   pcrSize;
820
821         pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
822
823         if(pcrData != NULL)
824         {
825             pcrSize =
826                 CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[i].hash);
827
828             // Reset PCR
829             // Any PCR can be reset by locality 4 should be reset to 0
830             if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
831                 MemorySet(pcrData, 0, pcrSize);
832         }
833     }
834 }
835 return;
836 }

```

8.7.3.27 PCRCapGetAllocation()

This function is used to get the current allocation of PCR banks.

Return Value	Meaning
YES	if the return count is 0
NO	if the return count is not 0

```

837 TPMI_YES_NO
838 PCRCapGetAllocation(
839     UINT32      count,           // IN: count of return
840     TPML_PCR_SELECTION *pcrSelection // OUT: PCR allocation list
841 )
842 {
843     if(count == 0)
844     {
845         pcrSelection->count = 0;
846         return YES;
847     }
848     else
849     {
850         *pcrSelection = gp.pcrAllocated;
851         return NO;
852     }
853 }

```

8.7.3.28 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```

854 static void
855 PCRSetSelectBit(
856     UINT32      pcr,           // IN: PCR number
857     BYTE        *bitmap       // OUT: bit map to be set
858 )
859 {
860     bitmap[pcr / 8] |= (1 << (pcr % 8));
861     return;
862 }

```

8.7.3.29 PCRGetProperty()

This function returns the selected PCR property.

Return Value	Meaning
TRUE(1)	the property type is implemented
FALSE(0)	the property type is not implemented

```

863 static BOOL
864 PCRGetProperty(
865     TPM_PT_PCR          property,
866     TPMS_TAGGED_PCR_SELECT *select
867 )
868 {
869     UINT32          pcr;
870     UINT32          groupIndex;
871
872     select->tag = property;
873     // Always set the bitmap to be the size of all PCR
874     select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
875
876     // Initialize bitmap
877     MemorySet(select->pcrSelect, 0, select->sizeofSelect);
878
879     // Collecting properties
880     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
881     {
882         switch(property)
883         {
884             case TPM_PT_PCR_SAVE:
885                 if(s_initAttributes[pcr].stateSave == SET)
886                     PCRSetSelectBit(pcr, select->pcrSelect);
887                 break;
888             case TPM_PT_PCR_EXTEND_L0:
889                 if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
890                     PCRSetSelectBit(pcr, select->pcrSelect);
891                 break;
892             case TPM_PT_PCR_RESET_L0:
893                 if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
894                     PCRSetSelectBit(pcr, select->pcrSelect);
895                 break;
896             case TPM_PT_PCR_EXTEND_L1:
897                 if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
898                     PCRSetSelectBit(pcr, select->pcrSelect);
899                 break;
900             case TPM_PT_PCR_RESET_L1:
901                 if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
902                     PCRSetSelectBit(pcr, select->pcrSelect);
903                 break;
904             case TPM_PT_PCR_EXTEND_L2:
905                 if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
906                     PCRSetSelectBit(pcr, select->pcrSelect);
907                 break;
908             case TPM_PT_PCR_RESET_L2:
909                 if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
910                     PCRSetSelectBit(pcr, select->pcrSelect);
911                 break;
912             case TPM_PT_PCR_EXTEND_L3:
913                 if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
914                     PCRSetSelectBit(pcr, select->pcrSelect);
915                 break;
916             case TPM_PT_PCR_RESET_L3:
917                 if((s_initAttributes[pcr].resetLocality & 0x08) != 0)

```



```

918         PCRSetSelectBit(pcr, select->pcrSelect);
919         break;
920     case TPM_PT_PCR_EXTEND_L4:
921         if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
922             PCRSetSelectBit(pcr, select->pcrSelect);
923         break;
924     case TPM_PT_PCR_RESET_L4:
925         if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
926             PCRSetSelectBit(pcr, select->pcrSelect);
927         break;
928     case TPM_PT_PCR_DRTM_RESET:
929         // DRTM reset PCRs are the PCR reset by locality 4
930         if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
931             PCRSetSelectBit(pcr, select->pcrSelect);
932         break;
933 #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
934     case TPM_PT_PCR_POLICY:
935         if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
936             PCRSetSelectBit(pcr, select->pcrSelect);
937         break;
938 #endif
939 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
940     case TPM_PT_PCR_AUTH:
941         if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
942             PCRSetSelectBit(pcr, select->pcrSelect);
943         break;
944 #endif
945 #if ENABLE_PCR_NO_INCREMENT == YES
946     case TPM_PT_PCR_NO_INCREMENT:
947         if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
948             PCRSetSelectBit(pcr, select->pcrSelect);
949         break;
950 #endif
951     default:
952         // If property is not supported, stop scanning PCR attributes
953         // and return.
954         return FALSE;
955         break;
956 }
957 }
958 return TRUE;
959 }

```

8.7.3.30 PCRCapGetProperties()

This function returns a list of PCR properties starting at *property*.

Return Value	Meaning
YES	if no more property is available
NO	if there are more properties not reported

```

960 TPMI_YES_NO
961 PCRCapGetProperties(
962     TPM_PT_PCR          property,    // IN: the starting PCR property
963     UINT32              count,      // IN: count of returned properties
964     TPML_TAGGED_PCR_PROPERTY *select // OUT: PCR select
965 )
966 {
967     TPMI_YES_NO    more = NO;
968     UINT32         i;
969
970     // Initialize output property list

```

```

971     select->count = 0;
972
973     // The maximum count of properties we may return is MAX_PCR_PROPERTIES
974     if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;
975
976     // TPM_PT_PCR_FIRST is defined as 0 in spec. It ensures that property
977     // value would never be less than TPM_PT_PCR_FIRST
978     cAssert(TPM_PT_PCR_FIRST == 0);
979
980     // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
981     // implemented on the TPM.
982     for(i = property; i <= TPM_PT_PCR_LAST; i++)
983     {
984         if(select->count < count)
985         {
986             // If we have not filled up the return list, add more properties to it
987             if(PCRGetProperty(i, &select->pcrProperty[select->count]))
988                 // only increment if the property is implemented
989                 select->count++;
990         }
991         else
992         {
993             // If the return list is full but we still have properties
994             // available, report this and stop iterating.
995             more = YES;
996             break;
997         }
998     }
999     return more;
1000 }

```

8.7.3.31 PCRCapGetHandles()

This function is used to get a list of handles of PCR, started from *handle*. If *handle* exceeds the maximum PCR handle range, an empty list will be returned and the return value will be NO.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```

1001 TPMI_YES_NO
1002 PCRCapGetHandles(
1003     TPMI_DH_PCR    handle,           // IN: start handle
1004     UINT32          count,           // IN: count of returned handles
1005     TPMI_HANDLE     *handleList      // OUT: list of handle
1006 )
1007 {
1008     TPMI_YES_NO     more = NO;
1009     UINT32          i;
1010
1011     pAssert(HandleGetType(handle) == TPM_HT_PCR);
1012
1013     // Initialize output handle list
1014     handleList->count = 0;
1015
1016     // The maximum count of handles we may return is MAX_CAP_HANDLES
1017     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1018
1019     // Iterate PCR handle range
1020     for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1021     {
1022         if(handleList->count < count)

```

```
1023     {
1024         // If we have not filled up the return list, add this PCR
1025         // handle to it
1026         handleList->handle[handleList->count] = i + PCR_FIRST;
1027         handleList->count++;
1028     }
1029     else
1030     {
1031         // If the return list is full but we still have PCR handle
1032         // available, report this and stop iterating
1033         more = YES;
1034         break;
1035     }
1036 }
1037 return more;
1038 }
```

8.8 PP.c

8.8.1 Introduction

This file contains the functions that support the physical presence operations of the TPM.

8.8.2 Includes

```
1 #include "Tpm.h"
```

8.8.3 Functions

8.8.3.1 PhysicalPresencePreInstall_Init()

This function is used to initialize the array of commands that always require confirmation with physical presence. The array is an array of bits that has a correspondence with the command code.

This command should only ever be executable in a manufacturing setting or in a simulation.

When set, these cannot be cleared.

```
2 void
3 PhysicalPresencePreInstall_Init(
4     void
5 )
6 {
7     COMMAND_INDEX    commandIndex;
8     // Clear all the PP commands
9     MemorySet(&gp.ppList, 0, sizeof(gp.ppList));
10
11     // Any command that is PP_REQUIRED should be SET
12     for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
13     {
14         if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED
15            && s_commandAttributes[commandIndex] & PP_REQUIRED)
16             SET_BIT(commandIndex, gp.ppList);
17     }
18     // Write PP list to NV
19     NV_SYNC_PERSISTENT(ppList);
20     return;
21 }
```

8.8.3.2 PhysicalPresenceCommandSet()

This function is used to set the indicator that a command requires PP confirmation.

```
22 void
23 PhysicalPresenceCommandSet(
24     TPM_CC    commandCode    // IN: command code
25 )
26 {
27     COMMAND_INDEX    commandIndex = CommandCodeToCommandIndex(commandCode);
28
29     // if the command isn't implemented, the do nothing
30     if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
31         return;
32
33     // only set the bit if this is a command for which PP is allowed
34     if(s_commandAttributes[commandIndex] & PP_COMMAND)
```

```

35     SET_BIT(commandIndex, gp.ppList);
36     return;
37 }

```

8.8.3.3 PhysicalPresenceCommandClear()

This function is used to clear the indicator that a command requires PP confirmation.

```

38 void
39 PhysicalPresenceCommandClear(
40     TPM_CC      commandCode    // IN: command code
41 )
42 {
43     COMMAND_INDEX  commandIndex = CommandCodeToCommandIndex(commandCode);
44
45     // If the command isn't implemented, then don't do anything
46     if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
47         return;
48
49     // Only clear the bit if the command does not require PP
50     if((s_commandAttributes[commandIndex] & PP_REQUIRED) == 0)
51         CLEAR_BIT(commandIndex, gp.ppList);
52
53     return;
54 }

```

8.8.3.4 PhysicalPresencelsRequired()

This function indicates if PP confirmation is required for a command.

Return Value	Meaning
TRUE(1)	physical presence is required
FALSE(0)	physical presence is not required

```

55 BOOL
56 PhysicalPresenceIsRequired(
57     COMMAND_INDEX  commandIndex    // IN: command index
58 )
59 {
60     // Check the bit map. If the bit is SET, PP authorization is required
61     return (TEST_BIT(commandIndex, gp.ppList));
62 }

```

8.8.3.5 PhysicalPresenceCapGetCCList()

This function returns a list of commands that require PP confirmation. The list starts from the first implemented command that has a command code that the same or greater than *commandCode*.

Return Value	Meaning
YES	if there are more command codes available
NO	all the available command codes have been returned

```

63 TPMI_YES_NO
64 PhysicalPresenceCapGetCCList(
65     TPM_CC      commandCode,    // IN: start command code
66     UINT32      count,          // IN: count of returned TPM_CC
67     TPML_CC     *commandList    // OUT: list of TPM_CC

```

```

68     )
69 {
70     TPMI_YES_NO    more = NO;
71     COMMAND_INDEX  commandIndex;
72
73     // Initialize output handle list
74     commandList->count = 0;
75
76     // The maximum count of command we may return is MAX_CAP_CC
77     if(count > MAX_CAP_CC) count = MAX_CAP_CC;
78
79     // Collect PP commands
80     for(commandIndex = GetClosestCommandIndex(commandCode);
81     commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
82     commandIndex = GetNextCommandIndex(commandIndex))
83     {
84         if(PhysicalPresenceIsRequired(commandIndex))
85         {
86             if(commandList->count < count)
87             {
88                 // If we have not filled up the return list, add this command
89                 // code to it
90                 commandList->commandCodes[commandList->count]
91                     = GetCommandCode(commandIndex);
92                 commandList->count++;
93             }
94             else
95             {
96                 // If the return list is full but we still have PP command
97                 // available, report this and stop iterating
98                 more = YES;
99                 break;
100             }
101         }
102     }
103     return more;
104 }

```

8.9 Session.c

8.9.1 Introduction

8.9.2 Includes, Defines, and Local Variables

```
1 #define SESSION_C
2 #include "Tpm.h"
```

8.9.3 File Scope Function -- ContextIdSetOldest()

```
3 static void
4 ContextIdSetOldest(
5     void
6 )
7 {
8     CONTEXT_SLOT    lowBits;
9     CONTEXT_SLOT    entry;
10    CONTEXT_SLOT    smallest = ((CONTEXT_SLOT)~0);
11    UINT32    i;
12
13    // Set oldestSaveContext to a value indicating none assigned
14    s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
15
16    lowBits = (CONTEXT_SLOT)gr.contextCounter;
17    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
18    {
19        entry = gr.contextArray[i];
20
21        // only look at entries that are saved contexts
22        if(entry > MAX_LOADED_SESSIONS)
23        {
24            // Use a less than or equal in case the oldest
25            // is brand new (= lowBits-1) and equal to our initial
26            // value for smallest.
27            if(((CONTEXT_SLOT)(entry - lowBits)) <= smallest)
28            {
29                smallest = (entry - lowBits);
30                s_oldestSavedSession = i;
31            }
32        }
33    }
34    // When we finish, either the s_oldestSavedSession still has its initial
35    // value, or it has the index of the oldest saved context.
36 }
```

8.9.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

```
37 BOOL
38 SessionStartup(
39     STARTUP_TYPE    type
40 )
41 {
42     UINT32    i;
43
44     // Initialize session slots. At startup, all the in-memory session slots
45     // are cleared and marked as not occupied
46     for(i = 0; i < MAX_LOADED_SESSIONS; i++)
```



```

47     s_sessions[i].occupied = FALSE;    // session slot is not occupied
48
49     // The free session slots the number of maximum allowed loaded sessions
50     s_freeSessionSlots = MAX_LOADED_SESSIONS;
51
52     // Initialize context ID data. On a ST_SAVE or hibernate sequence, it will
53     // scan the saved array of session context counts, and clear any entry that
54     // references a session that was in memory during the state save since that
55     // memory was not preserved over the ST_SAVE.
56     if(type == SU_RESUME || type == SU_RESTART)
57     {
58         // On ST_SAVE we preserve the contexts that were saved but not the ones
59         // in memory
60         for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
61         {
62             // If the array value is unused or references a loaded session then
63             // that loaded session context is lost and the array entry is
64             // reclaimed.
65             if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
66                 gr.contextArray[i] = 0;
67         }
68         // Find the oldest session in context ID data and set it in
69         // s_oldestSavedSession
70         ContextIdSetOldest();
71     }
72     else
73     {
74         // For STARTUP_CLEAR, clear out the contextArray
75         for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
76             gr.contextArray[i] = 0;
77
78         // reset the context counter
79         gr.contextCounter = MAX_LOADED_SESSIONS + 1;
80
81         // Initialize oldest saved session
82         s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
83     }
84     return TRUE;
85 }

```

8.9.5 Access Functions

8.9.5.1 SessionIsLoaded()

This function test a session handle references a loaded session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

NOTE A PWPAP authorization does not have a session.

Return Value	Meaning
TRUE(1)	session is loaded
FALSE(0)	session is not loaded

```

86  BOOL
87  SessionIsLoaded(
88      TPM_HANDLE    handle    // IN: session handle
89  )
90  {
91      pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
92              || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
93  }

```

```

94     handle = handle & HR_HANDLE_MASK;
95
96     // if out of range of possible active session, or not assigned to a loaded
97     // session return false
98     if(handle >= MAX_ACTIVE_SESSIONS
99         || gr.contextArray[handle] == 0
100        || gr.contextArray[handle] > MAX_LOADED_SESSIONS)
101         return FALSE;
102
103     return TRUE;
104 }

```

8.9.5.2 SessionIsSaved()

This function test a session handle references a saved session. The handle must have previously been checked to make sure that it is a valid handle for an authorization session.

NOTE An password authorization does not have a session. This function requires that the handle be a valid session handle.

Return Value	Meaning
TRUE(1)	session is saved
FALSE(0)	session is not saved

```

105 BOOL
106 SessionIsSaved(
107     TPM_HANDLE    handle           // IN: session handle
108 )
109 {
110     pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
111             || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
112
113     handle = handle & HR_HANDLE_MASK;
114     // if out of range of possible active session, or not assigned, or
115     // assigned to a loaded session, return false
116     if(handle >= MAX_ACTIVE_SESSIONS
117         || gr.contextArray[handle] == 0
118         || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
119        )
120         return FALSE;
121
122     return TRUE;
123 }

```

8.9.5.3 SequenceNumberForSavedContextIsValid()

This function validates that the sequence number and handle value within a saved context are valid.

```

124 BOOL
125 SequenceNumberForSavedContextIsValid(
126     TPMS_CONTEXT *context           // IN: pointer to a context structure to be
127                                     // validated
128 )
129 {
130     #define MAX_CONTEXT_GAP ((UINT64)((CONTEXT_SLOT) ~0) + 1)
131
132     TPM_HANDLE    handle = context->savedHandle & HR_HANDLE_MASK;
133
134     if(// Handle must be with the range of active sessions
135        handle >= MAX_ACTIVE_SESSIONS

```

```

136     // the array entry must be for a saved context
137     || gr.contextArray[handle] <= MAX_LOADED_SESSIONS
138     // the array entry must agree with the sequence number
139     || gr.contextArray[handle] != (CONTEXT_SLOT)context->sequence
140     // the provided sequence number has to be less than the current counter
141     || context->sequence > gr.contextCounter
142     // but not so much that it could not be a valid sequence number
143     || gr.contextCounter - context->sequence > MAX_CONTEXT_GAP)
144     return FALSE;
145
146     return TRUE;
147 }

```

8.9.5.4 SessionPCRValueIsCurrent()

This function is used to check if PCR values have been updated since the last time they were checked in a policy session.

This function requires the session is loaded.

Return Value	Meaning
TRUE(1)	PCR value is current
FALSE(0)	PCR value is not current

```

148 BOOL
149 SessionPCRValueIsCurrent(
150     SESSION      *session      // IN: session structure
151 )
152 {
153     if(session->pcrCounter != 0
154         && session->pcrCounter != gr.pcrCounter
155     )
156         return FALSE;
157     else
158         return TRUE;
159 }

```

8.9.5.5 SessionGet()

This function returns a pointer to the session object associated with a session handle.

The function requires that the session is loaded.

```

160 SESSION *
161 SessionGet(
162     TPM_HANDLE    handle      // IN: session handle
163 )
164 {
165     size_t        slotIndex;
166     CONTEXT_SLOT  sessionIndex;
167
168     pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
169         || HandleGetType(handle) == TPM_HT_HMAC_SESSION
170     );
171
172     slotIndex = handle & HR_HANDLE_MASK;
173
174     pAssert(slotIndex < MAX_ACTIVE_SESSIONS);
175
176     // get the contents of the session array. Because session is loaded, we
177     // should always get a valid sessionIndex

```

```

178     sessionIndex = gr.contextArray[slotIndex] - 1;
179
180     pAssert(sessionIndex < MAX_LOADED_SESSIONS);
181
182     return &s_sessions[sessionIndex].session;
183 }

```

8.9.6 Utility Functions

8.9.6.1 ContextIdSessionCreate()

This function is called when a session is created. It will check to see if the current gap would prevent a context from being saved. If so it will return TPM_RC_CONTEXT_GAP. Otherwise, it will try to find an open slot in *contextArray*, set *contextArray* to the slot. This routine requires that the caller has determined the session array index for the session.

Error Return	Meaning
TPM_RC_CONTEXT_GAP	can't assign a new <i>contextID</i> until the oldest saved session context is recycled
TPM_RC_SESSION_HANDLE	there is no slot available in the context array for tracking of this session context

```

184 static TPM_RC
185 ContextIdSessionCreate(
186     TPM_HANDLE      *handle,           // OUT: receives the assigned handle. This will
187                                         // be an index that must be adjusted by the
188                                         // caller according to the type of the
189                                         // session created
190     UINT32          sessionIndex       // IN: The session context array entry that will
191                                         // be occupied by the created session
192 )
193 {
194     pAssert(sessionIndex < MAX_LOADED_SESSIONS);
195
196     // check to see if creating the context is safe
197     // Is this going to be an assignment for the last session context
198     // array entry? If so, then there will be no room to recycle the
199     // oldest context if needed. If the gap is not at maximum, then
200     // it will be possible to save a context if it becomes necessary.
201     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
202        && s_freeSessionSlots == 1)
203     {
204         // See if the gap is at maximum
205         // The current value of the contextCounter will be assigned to the next
206         // saved context. If the value to be assigned would make the same as an
207         // existing context, then we can't use it because of the ambiguity it would
208         // create.
209         if((CONTEXT_SLOT)gr.contextCounter
210            == gr.contextArray[s_oldestSavedSession])
211             return TPM_RC_CONTEXT_GAP;
212     }
213     // Find an unoccupied entry in the contextArray
214     for(*handle = 0; *handle < MAX_ACTIVE_SESSIONS; (*handle)++)
215     {
216         if(gr.contextArray[*handle] == 0)
217         {
218             // indicate that the session associated with this handle
219             // references a loaded session
220             gr.contextArray[*handle] = (CONTEXT_SLOT)(sessionIndex + 1);
221             return TPM_RC_SUCCESS;
222         }

```

```

223     }
224     return TPM_RC_SESSION_HANDLES;
225 }

```

8.9.6.2 SessionCreate()

This function does the detailed work for starting an authorization session. This is done in a support routine rather than in the action code because the session management may differ in implementations. This implementation uses a fixed memory allocation to hold sessions and a fixed allocation to hold the *contextID* for the saved contexts.

Error Return	Meaning
TPM_RC_CONTEXT_GAP	need to recycle sessions
TPM_RC_SESSION_HANDLE	active session space is full
TPM_RC_SESSION_MEMORY	loaded session space is full

```

226 TPM_RC
227 SessionCreate(
228     TPM_SE      sessionType,    // IN: the session type
229     TPMI_ALG_HASH authHash,     // IN: the hash algorithm
230     TPM2B_NONCE *nonceCaller,   // IN: initial nonceCaller
231     TPMT_SYM_DEF *symmetric,    // IN: the symmetric algorithm
232     TPMI_DH_ENTITY bind,        // IN: the bind object
233     TPM2B_DATA *seed,           // IN: seed data
234     TPM_HANDLE *sessionHandle,  // OUT: the session handle
235     TPM2B_NONCE *nonceTpm      // OUT: the session nonce
236 )
237 {
238     TPM_RC      result = TPM_RC_SUCCESS;
239     CONTEXT_SLOT slotIndex;
240     SESSION     *session = NULL;
241
242     pAssert(sessionType == TPM_SE_HMAC
243             || sessionType == TPM_SE_POLICY
244             || sessionType == TPM_SE_TRIAL);
245
246     // If there are no open spots in the session array, then no point in searching
247     if(s_freeSessionSlots == 0)
248         return TPM_RC_SESSION_MEMORY;
249
250     // Find a space for loading a session
251     for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
252     {
253         // Is this available?
254         if(s_sessions[slotIndex].occupied == FALSE)
255         {
256             session = &s_sessions[slotIndex].session;
257             break;
258         }
259     }
260     // if no spot found, then this is an internal error
261     if(slotIndex >= MAX_LOADED_SESSIONS)
262         FAIL(FATAL_ERROR_INTERNAL);
263
264     // Call context ID function to get a handle.  TPM_RC_SESSION_HANDLE may be
265     // returned from ContextIdHandleAssign()
266     result = ContextIdSessionCreate(sessionHandle, slotIndex);
267     if(result != TPM_RC_SUCCESS)
268         return result;
269
270     /*** Only return from this point on is TPM_RC_SUCCESS

```

```

271
272 // Can now indicate that the session array entry is occupied.
273 s_freeSessionSlots--;
274 s_sessions[slotIndex].occupied = TRUE;
275
276 // Initialize the session data
277 MemorySet(session, 0, sizeof(SESSION));
278
279 // Initialize internal session data
280 session->authHashAlg = authHash;
281 // Initialize session type
282 if(sessionType == TPM_SE_HMAC)
283 {
284     *sessionHandle += HMAC_SESSION_FIRST;
285 }
286 else
287 {
288     *sessionHandle += POLICY_SESSION_FIRST;
289
290     // For TPM_SE_POLICY or TPM_SE_TRIAL
291     session->attributes.isPolicy = SET;
292     if(sessionType == TPM_SE_TRIAL)
293         session->attributes.isTrialPolicy = SET;
294
295     SessionSetStartTime(session);
296
297     // Initialize policyDigest. policyDigest is initialized with a string of 0
298     // of session algorithm digest size. Since the session is already clear.
299     // Just need to set the size
300     session->u2.policyDigest.t.size =
301         CryptHashGetDigestSize(session->authHashAlg);
302 }
303 // Create initial session nonce
304 session->nonceTPM.t.size = nonceCaller->t.size;
305 CryptRandomGenerate(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
306 MemoryCopy2B(&nonceTpm->b, &session->nonceTPM.b,
307             sizeof(nonceTpm->t.buffer));
308
309 // Set up session parameter encryption algorithm
310 session->symmetric = *symmetric;
311
312 // If there is a bind object or a session secret, then need to compute
313 // a sessionKey.
314 if(bind != TPM_RH_NULL || seed->t.size != 0)
315 {
316     // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
317     // nonceCaller, bits)
318     // The HMAC key for generating the sessionSecret can be the concatenation
319     // of an authorization value and a seed value
320     TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
321     TPM2B_KEY key;
322
323     // Get hash size, which is also the length of sessionKey
324     session->sessionKey.t.size = CryptHashGetDigestSize(session->authHashAlg);
325
326     // Get authValue of associated entity
327     EntityGetAuthValue(bind, (TPM2B_AUTH *)&key);
328     pAssert(key.t.size + seed->t.size <= sizeof(key.t.buffer));
329
330     // Concatenate authValue and seed
331     MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));
332
333     // Compute the session key
334     CryptKDFa(session->authHashAlg, &key.b, SESSION_KEY, &session->nonceTPM.b,
335             &nonceCaller->b,
336             session->sessionKey.t.size * 8, session->sessionKey.t.buffer,

```

```

337         NULL, FALSE);
338     }
339     // Copy the name of the entity that the HMAC session is bound to
340     // Policy session is not bound to an entity
341     if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
342     {
343         session->attributes.isBound = SET;
344         SessionComputeBoundEntity(bind, &session->ul.boundEntity);
345     }
346     // If there is a bind object and it is subject to DA, then use of this session
347     // is subject to DA regardless of how it is used.
348     session->attributes.isDaBound = (bind != TPM_RH_NULL)
349         && (IsDAExempted(bind) == FALSE);
350
351     // If the session is bound, then check to see if it is bound to lockoutAuth
352     session->attributes.isLockoutBound = (session->attributes.isDaBound == SET)
353         && (bind == TPM_RH_LOCKOUT);
354     return TPM_RC_SUCCESS;
355 }

```

8.9.6.3 SessionContextSave()

This function is called when a session context is to be saved. The *contextID* of the saved session is returned. If no *contextID* can be assigned, then the routine returns TPM_RC_CONTEXT_GAP. If the function completes normally, the session slot will be freed. This function requires that *handle* references a loaded session. Otherwise, it should not be called at the first place.

Error Return	Meaning
TPM_RC_CONTEXT_GAP	a <i>contextID</i> could not be assigned
TPM_RC_TOO_MANY_CONTEXTS	the counter maxed out

```

356 TPM_RC
357 SessionContextSave(
358     TPM_HANDLE          handle,          // IN: session handle
359     CONTEXT_COUNTER     *contextID      // OUT: assigned contextID
360 )
361 {
362     UINT32              contextIndex;
363     CONTEXT_SLOT        slotIndex;
364
365     pAssert(SessionIsLoaded(handle));
366
367     // check to see if the gap is already maxed out
368     // Need to have a saved session
369     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
370         // if the oldest saved session has the same value as the low bits
371         // of the contextCounter, then the GAP is maxed out.
372         && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
373         return TPM_RC_CONTEXT_GAP;
374
375     // if the caller wants the context counter, set it
376     if(contextID != NULL)
377         *contextID = gr.contextCounter;
378
379     contextIndex = handle & HR_HANDLE_MASK;
380     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
381
382     // Extract the session slot number referenced by the contextArray
383     // because we are going to overwrite this with the low order
384     // contextID value.
385     slotIndex = gr.contextArray[contextIndex] - 1;
386

```



```

387 // Set the contextID for the contextArray
388 gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
389
390 // Increment the counter
391 gr.contextCounter++;
392
393 // In the unlikely event that the 64-bit context counter rolls over...
394 if(gr.contextCounter == 0)
395 {
396     // back it up
397     gr.contextCounter--;
398     // return an error
399     return TPM_RC_TOO_MANY_CONTEXTS;
400 }
401 // if the low-order bits wrapped, need to advance the value to skip over
402 // the values used to indicate that a session is loaded
403 if(((CONTEXT_SLOT)gr.contextCounter) == 0)
404     gr.contextCounter += MAX_LOADED_SESSIONS + 1;
405
406 // If no other sessions are saved, this is now the oldest.
407 if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
408     s_oldestSavedSession = contextIndex;
409
410 // Mark the session slot as unoccupied
411 s_sessions[slotIndex].occupied = FALSE;
412
413 // and indicate that there is an additional open slot
414 s_freeSessionSlots++;
415
416 return TPM_RC_SUCCESS;
417 }

```

8.9.6.4 SessionContextLoad()

This function is used to load a session from saved context. The session handle must be for a saved context. If the gap is at a maximum, then the only session that can be loaded is the oldest session, otherwise TPM_RC_CONTEXT_GAP is returned. This function requires that *handle* references a valid saved session.

Error Return	Meaning
TPM_RC_SESSION_MEMORY	no free session slots
TPM_RC_CONTEXT_GAP	the gap count is maximum and this is not the oldest saved context

```

418 TPM_RC
419 SessionContextLoad(
420     SESSION_BUF    *session,           // IN: session structure from saved context
421     TPM_HANDLE     *handle             // IN/OUT: session handle
422 )
423 {
424     UINT32          contextIndex;
425     CONTEXT_SLOT    slotIndex;
426
427     pAssert(HandleGetType(*handle) == TPM_HT_POLICY_SESSION
428             || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);
429
430     // Don't bother looking if no openings
431     if(s_freeSessionSlots == 0)
432         return TPM_RC_SESSION_MEMORY;
433
434     // Find a free session slot to load the session
435     for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)

```

```

436         if(s_sessions[slotIndex].occupied == FALSE) break;
437
438     // if no spot found, then this is an internal error
439     pAssert(slotIndex < MAX_LOADED_SESSIONS);
440
441     contextIndex = *handle & HR_HANDLE_MASK;    // extract the index
442
443     // If there is only one slot left, and the gap is at maximum, the only session
444     // context that we can safely load is the oldest one.
445     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
446        && s_freeSessionSlots == 1
447        && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
448        && contextIndex != s_oldestSavedSession)
449         return TPM_RC_CONTEXT_GAP;
450
451     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
452
453     // set the contextArray value to point to the session slot where
454     // the context is loaded
455     gr.contextArray[contextIndex] = slotIndex + 1;
456
457     // if this was the oldest context, find the new oldest
458     if(contextIndex == s_oldestSavedSession)
459         ContextIdSetOldest();
460
461     // Copy session data to session slot
462     MemoryCopy(&s_sessions[slotIndex].session, session, sizeof(SESSION));
463
464     // Set session slot as occupied
465     s_sessions[slotIndex].occupied = TRUE;
466
467     // Reduce the number of open spots
468     s_freeSessionSlots--;
469
470     return TPM_RC_SUCCESS;
471 }

```

8.9.6.5 SessionFlush()

This function is used to flush a session referenced by its handle. If the session associated with *handle* is loaded, the session array entry is marked as available. This function requires that *handle* be a valid active session.

```

472 void
473 SessionFlush(
474     TPM_HANDLE    handle           // IN: loaded or saved session handle
475 )
476 {
477     CONTEXT_SLOT    slotIndex;
478     UINT32          contextIndex;  // Index into contextArray
479
480     pAssert((HandleGetType(handle) == TPM_HT_POLICY_SESSION
481             || HandleGetType(handle) == TPM_HT_HMAC_SESSION
482             )
483            && (SessionIsLoaded(handle) || SessionIsSaved(handle))
484            );
485
486     // Flush context ID of this session
487     // Convert handle to an index into the contextArray
488     contextIndex = handle & HR_HANDLE_MASK;
489
490     pAssert(contextIndex < sizeof(gr.contextArray) / sizeof(gr.contextArray[0]));
491
492     // Get the current contents of the array

```

```

493     slotIndex = gr.contextArray[contextIndex];
494
495     // Mark context array entry as available
496     gr.contextArray[contextIndex] = 0;
497
498     // Is this a saved session being flushed
499     if(slotIndex > MAX_LOADED_SESSIONS)
500     {
501         // Flushing the oldest session?
502         if(contextIndex == s_oldestSavedSession)
503             // If so, find a new value for oldest.
504             ContextIdSetOldest();
505     }
506     else
507     {
508         // Adjust slot index to point to session array index
509         slotIndex -= 1;
510
511         // Free session array index
512         s_sessions[slotIndex].occupied = FALSE;
513         s_freeSessionSlots++;
514     }
515     return;
516 }

```

8.9.6.6 SessionComputeBoundEntity()

This function computes the binding value for a session. The binding value for a reserved handle is the handle itself. For all the other entities, the *authValue* at the time of binding is included to prevent squatting. For those values, the Name and the *authValue* are concatenated into the bind buffer. If they will not both fit, they will be overlapped by XORing bytes. If XOR is required, the bind value will be full.

```

517 void
518 SessionComputeBoundEntity(
519     TPMI_DH_ENTITY    entityHandle, // IN: handle of entity
520     TPM2B_NAME         *bind        // OUT: binding value
521 )
522 {
523     TPM2B_AUTH         auth;
524     BYTE               *pAuth = auth.t.buffer;
525     UINT16              i;
526
527     // Get name
528     EntityGetName(entityHandle, bind);
529
530     // // The bound value of a reserved handle is the handle itself
531     // if(bind->t.size == sizeof(TPM_HANDLE)) return;
532
533     // For all the other entities, concatenate the authorization value to the name.
534     // Get a local copy of the authorization value because some overlapping
535     // may be necessary.
536     EntityGetAuthValue(entityHandle, &auth);
537
538     // Make sure that the extra space is zeroed
539     MemorySet(&bind->t.name[bind->t.size], 0, sizeof(bind->t.name) - bind->t.size);
540     // XOR the authValue at the end of the name
541     for(i = sizeof(bind->t.name) - auth.t.size; i < sizeof(bind->t.name); i++)
542         bind->t.name[i] ^= *pAuth++;
543
544     // Set the bind value to the maximum size
545     bind->t.size = sizeof(bind->t.name);
546
547     return;
548 }

```

8.9.6.7 SessionSetStartTime()

This function is used to initialize the session timing

```

549 void
550 SessionSetStartTime(
551     SESSION      *session      // IN: the session to update
552 )
553 {
554     session->startTime = g_time;
555     session->epoch = g_timeEpoch;
556     session->timeout = 0;
557 }

```

8.9.6.8 SessionResetPolicyData()

This function is used to reset the policy data without changing the nonce or the start time of the session.

```

558 void
559 SessionResetPolicyData(
560     SESSION      *session      // IN: the session to reset
561 )
562 {
563     SESSION_ATTRIBUTES oldAttributes;
564     pAssert(session != NULL);
565
566     // Will need later
567     oldAttributes = session->attributes;
568
569     // No command
570     session->commandCode = 0;
571
572     // No locality selected
573     MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
574
575     // The cpHash size to zero
576     session->u1.cpHash.b.size = 0;
577
578     // No timeout
579     session->timeout = 0;
580
581     // Reset the pcrCounter
582     session->pcrCounter = 0;
583
584     // Reset the policy hash
585     MemorySet(&session->u2.policyDigest.t.buffer, 0,
586             session->u2.policyDigest.t.size);
587
588     // Reset the session attributes
589     MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
590
591     // Restore the policy attributes
592     session->attributes.isPolicy = SET;
593     session->attributes.isTrialPolicy = oldAttributes.isTrialPolicy;
594
595     // Restore the bind attributes
596     session->attributes.isDaBound = oldAttributes.isDaBound;
597     session->attributes.isLockoutBound = oldAttributes.isLockoutBound;
598 }

```

8.9.6.9 SessionCapGetLoaded()

This function returns a list of handles of loaded session, started from input *handle*

Handle must be in valid loaded session handle range, but does not have to point to a loaded session.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```

599 TPMI_YES_NO
600 SessionCapGetLoaded(
601     TPMI_SH_POLICY    handle,          // IN: start handle
602     UINT32             count,          // IN: count of returned handles
603     TPML_HANDLE       *handleList     // OUT: list of handle
604 )
605 {
606     TPMI_YES_NO        more = NO;
607     UINT32             i;
608
609     pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
610
611     // Initialize output handle list
612     handleList->count = 0;
613
614     // The maximum count of handles we may return is MAX_CAP_HANDLES
615     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
616
617     // Iterate session context ID slots to get loaded session handles
618     for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
619     {
620         // If session is active
621         if(gr.contextArray[i] != 0)
622         {
623             // If session is loaded
624             if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
625             {
626                 if(handleList->count < count)
627                 {
628                     SESSION    *session;
629
630                     // If we have not filled up the return list, add this
631                     // session handle to it
632                     // assume that this is going to be an HMAC session
633                     handle = i + HMAC_SESSION_FIRST;
634                     session = SessionGet(handle);
635                     if(session->attributes.isPolicy)
636                         handle = i + POLICY_SESSION_FIRST;
637                     handleList->handle[handleList->count] = handle;
638                     handleList->count++;
639                 }
640                 else
641                 {
642                     // If the return list is full but we still have loaded object
643                     // available, report this and stop iterating
644                     more = YES;
645                     break;
646                 }
647             }
648         }
649     }
650     return more;
651 }

```

8.9.6.10 SessionCapGetSaved()

This function returns a list of handles for saved session, starting at *handle*.

Handle must be in a valid handle range, but does not have to point to a saved session

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```

652  TPMI_YES_NO
653  SessionCapGetSaved(
654      TPMI_SH_HMAC    handle,        // IN: start handle
655      UINT32          count,        // IN: count of returned handles
656      TPML_HANDLE     *handleList    // OUT: list of handle
657  )
658  {
659      TPMI_YES_NO     more = NO;
660      UINT32          i;
661
662      #ifdef TPM_HT_SAVED_SESSION
663          pAssert(HandleGetType(handle) == TPM_HT_SAVED_SESSION);
664      #else
665          pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);
666      #endif
667
668      // Initialize output handle list
669      handleList->count = 0;
670
671      // The maximum count of handles we may return is MAX_CAP_HANDLES
672      if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
673
674      // Iterate session context ID slots to get loaded session handles
675      for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
676      {
677          // If session is active
678          if(gr.contextArray[i] != 0)
679          {
680              // If session is saved
681              if(gr.contextArray[i] > MAX_LOADED_SESSIONS)
682              {
683                  if(handleList->count < count)
684                  {
685                      // If we have not filled up the return list, add this
686                      // session handle to it
687                      handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
688                      handleList->count++;
689                  }
690                  else
691                  {
692                      // If the return list is full but we still have loaded object
693                      // available, report this and stop iterating
694                      more = YES;
695                      break;
696                  }
697              }
698          }
699      }
700      return more;
701  }

```

8.9.6.11 SessionCapGetLoadedNumber()

This function return the number of authorization sessions currently loaded into TPM RAM.

```

702  UINT32
703  SessionCapGetLoadedNumber (
704      void
705  )
706  {
707      return MAX_LOADED_SESSIONS - s_freeSessionSlots;
708  }

```

8.9.6.12 SessionCapGetLoadedAvail()

This function returns the number of additional authorization sessions, of any type, that could be loaded into TPM RAM.

NOTE In other implementations, this number may just be an estimate. The only requirement for the estimate is, if it is one or more, then at least one session must be loadable.

```

709  UINT32
710  SessionCapGetLoadedAvail (
711      void
712  )
713  {
714      return s_freeSessionSlots;
715  }

```

8.9.6.13 SessionCapGetActiveNumber()

This function returns the number of active authorization sessions currently being tracked by the TPM.

```

716  UINT32
717  SessionCapGetActiveNumber (
718      void
719  )
720  {
721      UINT32 i;
722      UINT32 num = 0;
723
724      // Iterate the context array to find the number of non-zero slots
725      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
726      {
727          if(gr.contextArray[i] != 0) num++;
728      }
729      return num;
730  }

```

8.9.6.14 SessionCapGetActiveAvail()

This function returns the number of additional authorization sessions, of any type, that could be created. This not the number of slots for sessions, but the number of additional sessions that the TPM is capable of tracking.

```

731  UINT32
732  SessionCapGetActiveAvail (
733      void
734  )
735  {
736      UINT32 i;

```



```
737     UINT32          num = 0;
738
739     // Iterate the context array to find the number of zero slots
740     for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
741     {
742         if(gr.contextArray[i] == 0) num++;
743     }
744     return num;
745 }
```

DRAFT

8.10 Time.c

8.10.1 Introduction

This file contains the functions relating to the TPM's time functions including the interface to the implementation-specific time functions.

8.10.2 Includes

```
1  #include "Tpm.h"
2  #include "PlatformClock.h"
```

8.10.3 Functions

8.10.3.1 TimePowerOn()

This function initialize time info at `_TPM_Init()`.

This function is called at `_TPM_Init()` so that the TPM time can start counting as soon as the TPM comes out of reset and doesn't have to wait until `TPM2_Startup()` in order to begin the new time epoch. This could be significant for systems that could get powered up but not run any TPM commands for some period of time.

```
3  void
4  TimePowerOn(
5      void
6  )
7  {
8      g_time = _plat__TimerRead();
9  }
```

8.10.3.2 TimeNewEpoch()

This function does the processing to generate a new time epoch nonce and set NV for update. This function is only called when NV is known to be available and the clock is running. The epoch is updated to persistent data.

```
10 static void
11 TimeNewEpoch(
12     void
13 )
14 {
15 #if CLOCK_STOPS
16     CryptRandomGenerate(sizeof(CLOCK_NONCE), (BYTE *)&g_timeEpoch);
17 #else
18     // if the epoch is kept in NV, update it.
19     gp.timeEpoch++;
20     NV_SYNC_PERSISTENT(timeEpoch);
21 #endif
22     // Clean out any lingering state
23     _plat__TimerWasStopped();
24 }
```

8.10.3.3 TimeStartup()

This function updates the *resetCount* and *restartCount* components of `TPMS_CLOCK_INFO` structure at `TPM2_Startup()`.

This function will deal with the deferred creation of a new epoch. `TimeUpdateToCurrent()` will not start a new epoch even if one is due when `TPM_Startup()` has not been run. This is because the state of NV is not known until startup completes. When Startup is done, then it will create the epoch nonce to complete the initializations by calling this function.

```

25  BOOL
26  TimeStartup(
27      STARTUP_TYPE    type           // IN: start up type
28  )
29  {
30      NOT_REFERENCED(type);
31      // If the previous cycle is orderly shut down, the value of the safe bit
32      // the same as previously saved. Otherwise, it is not safe.
33      if(!NV_IS_ORDERLY)
34          go.clockSafe = NO;
35      return TRUE;
36  }

```

8.10.3.4 TimeClockUpdate()

This function updates `go.clock`. If *newTime* requires an update of NV, then NV is checked for availability. If it is not available or is rate limiting, then `go.clock` is not updated and the function returns an error. If *newTime* would not cause an NV write, then `go.clock` is updated. If an NV write occurs, then `go.safe` is SET.

```

37  void
38  TimeClockUpdate(
39      UINT64          newTime        // IN: New time value in mS.
40  )
41  {
42      #define CLOCK_UPDATE_MASK ((1ULL << NV_CLOCK_UPDATE_INTERVAL) - 1)
43
44      // Check to see if the update will cause a need for an nvClock update
45      if((newTime | CLOCK_UPDATE_MASK) > (go.clock | CLOCK_UPDATE_MASK))
46      {
47          pAssert(g_NvStatus == TPM_RC_SUCCESS);
48
49          // Going to update the NV time state so SET the safe flag
50          go.clockSafe = YES;
51
52          // update the time
53          go.clock = newTime;
54
55          NvWrite(NV_ORDERLY_DATA, sizeof(go), &go);
56      }
57      else
58          // No NV update needed so just update
59          go.clock = newTime;
60  }
61

```

8.10.3.5 TimeUpdate()

This function is used to update the time and clock values. If the TPM has run `TPM2_Startup()`, this function is called at the start of each command. If the TPM has not run `TPM2_Startup()`, this is called from `TPM2_Startup()` to get the clock values initialized. It is not called on command entry because, in this implementation, the `go` structure is not read from NV until `TPM2_Startup()`. The reason for this is that the initialization code (`_TPM_Init()`) may run before NV is accessible.

```

62  void
63  TimeUpdate(

```

```

64     void
65     )
66 {
67     UINT64          elapsed;
68     //
69     // Make sure that we consume the current _plat__TimerWasStopped() state.
70     if(_plat__TimerWasStopped())
71     {
72         TimeNewEpoch();
73     }
74     // Get the difference between this call and the last time we updated the tick
75     // timer.
76     elapsed = _plat__TimerRead() - g_time;
77     // Don't read +
78     g_time += elapsed;
79
80     // Don't need to check the result because it has to be success because have
81     // already checked that NV is available.
82     TimeClockUpdate(go.clock + elapsed);
83
84     // Call self healing logic for dictionary attack parameters
85     DASelfHeal();
86 }

```

8.10.3.6 TimeUpdateToCurrent()

This function updates the *Time* and *Clock* in the global TPMS_TIME_INFO structure.

In this implementation, *Time* and *Clock* are updated at the beginning of each command and the values are unchanged for the duration of the command.

Because *Clock* updates may require a write to NV memory, *Time* and *Clock* are not allowed to advance if NV is not available. When clock is not advancing, any function that uses *Clock* will fail and return TPM_RC_NV_UNAVAILABLE or TPM_RC_NV_RATE.

This implementation does not do rate limiting. If the implementation does do rate limiting, then the *Clock* update should not be inhibited even when doing rate limiting.

```

87 void
88 TimeUpdateToCurrent(
89     void
90 )
91 {
92     // Can't update time during the dark interval or when rate limiting so don't
93     // make any modifications to the internal clock value. Also, defer any clock
94     // processing until TPM has run TPM2_Startup()
95     if(!NV_IS_AVAILABLE || !TPMIsStarted())
96         return;
97
98     TimeUpdate();
99 }

```

8.10.3.7 TimeSetAdjustRate()

This function is used to perform rate adjustment on *Time* and *Clock*.

```

100 void
101 TimeSetAdjustRate(
102     TPM_CLOCK_ADJUST    adjust    // IN: adjust constant
103 )
104 {
105     switch(adjunct)
106     {

```

```

107     case TPM_CLOCK_COARSE_SLOWER:
108         _plat_ClockAdjustRate(CLOCK_ADJUST_COARSE);
109         break;
110     case TPM_CLOCK_COARSE_FASTER:
111         _plat_ClockAdjustRate(-CLOCK_ADJUST_COARSE);
112         break;
113     case TPM_CLOCK_MEDIUM_SLOWER:
114         _plat_ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
115         break;
116     case TPM_CLOCK_MEDIUM_FASTER:
117         _plat_ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
118         break;
119     case TPM_CLOCK_FINE_SLOWER:
120         _plat_ClockAdjustRate(CLOCK_ADJUST_FINE);
121         break;
122     case TPM_CLOCK_FINE_FASTER:
123         _plat_ClockAdjustRate(-CLOCK_ADJUST_FINE);
124         break;
125     case TPM_CLOCK_NO_CHANGE:
126         break;
127     default:
128         FAIL(FATAL_ERROR_INTERNAL);
129         break;
130 }
131 return;
132 }

```

8.10.3.8 TimeGetMarshaled()

This function is used to access TPMS_TIME_INFO in canonical form. The function collects the time information and marshals it into *dataBuffer* and returns the marshaled size

```

133 UUINT16
134 TimeGetMarshaled(
135     TIME_INFO      *dataBuffer    // OUT: result buffer
136 )
137 {
138     TPMS_TIME_INFO  timeInfo;
139
140     // Fill TPMS_TIME_INFO structure
141     timeInfo.time = g_time;
142     TimeFillInfo(&timeInfo.clockInfo);
143
144     // Marshal TPMS_TIME_INFO to canonical form
145     return TPMS_TIME_INFO_Marshal(&timeInfo, (BYTE **)&dataBuffer, NULL);
146 }

```

8.10.3.9 TimeFillInfo()

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```

147 void
148 TimeFillInfo(
149     TPMS_CLOCK_INFO *clockInfo
150 )
151 {
152     clockInfo->clock = go.clock;
153     clockInfo->resetCount = gp.resetCount;
154     clockInfo->restartCount = gr.restartCount;
155
156     // If NV is not available, clock stopped advancing and the value reported is
157     // not "safe".
158     if(NV_IS_AVAILABLE)

```

```
159         clockInfo->safe = go.clockSafe;  
160     else  
161         clockInfo->safe = NO;  
162  
163     return;  
164 }
```

DRAFT

9 Support

9.1 AlgorithmCap.c

9.1.1 Description

This file contains the algorithm property definitions for the algorithms and the code for the TPM2_GetCapability() to return the algorithm properties.

9.1.2 Includes and Defines

```

1  #include "Tpm.h"
2
3  typedef struct
4  {
5      TPM_ALG_ID      algID;
6      TPMA_ALGORITHM  attributes;
7  } ALGORITHM;
8
9  static const ALGORITHM  s_algorithms[] =
10 {
11     // The entries in this table need to be in ascending order but the table doesn't
12     // need to be full (gaps are allowed). One day, a tool might exist to fill in the
13     // table from the TPM_ALG description
14     #if ALG_RSA
15         {TPM_ALG_RSA,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
16     #endif
17     #if ALG_TDES
18         {TPM_ALG_TDES,        TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
19     #endif
20     #if ALG_SHA1
21         {TPM_ALG_SHA1,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
22     #endif
23
24         {TPM_ALG_HMAC,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 1, 0, 0, 0)},
25
26     #if ALG_AES
27         {TPM_ALG_AES,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
28     #endif
29     #if ALG_MGF1
30         {TPM_ALG_MGF1,          TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
31     #endif
32
33         {TPM_ALG_KEYEDHASH,      TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 1, 0, 1, 1, 0, 0)},
34
35     #if ALG_XOR
36         {TPM_ALG_XOR,          TPMA_ALGORITHM_INITIALIZER(0, 1, 1, 0, 0, 0, 0, 0, 0)},
37     #endif
38
39     #if ALG_SHA256
40         {TPM_ALG_SHA256,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
41     #endif
42     #if ALG_SHA384
43         {TPM_ALG_SHA384,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
44     #endif
45     #if ALG_SHA512
46         {TPM_ALG_SHA512,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
47     #endif
48     #if ALG_SM3_256
49         {TPM_ALG_SM3_256,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
50     #endif
51     #if ALG_SM4

```



```

52     {TPM_ALG_SM4,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
53 #endif
54 #if ALG_RSASSA
55     {TPM_ALG_RSASSA,      TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
56 #endif
57 #if ALG_RSAES
58     {TPM_ALG_RSAES,       TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
59 #endif
60 #if ALG_RSAPSS
61     {TPM_ALG_RSAPSS,      TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
62 #endif
63 #if ALG_OAEP
64     {TPM_ALG_OAEP,        TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
65 #endif
66 #if ALG_ECDSA
67     {TPM_ALG_ECDSA,       TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
68 #endif
69 #if ALG_ECDH
70     {TPM_ALG_ECDH,        TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 0, 1)},
71 #endif
72 #if ALG_ECDA
73     {TPM_ALG_ECDA,        TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
74 #endif
75 #if ALG_SM2
76     {TPM_ALG_SM2,         TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 1)},
77 #endif
78 #if ALG_ECSCHNORR
79     {TPM_ALG_ECSCHNORR,   TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
80 #endif
81 #if ALG_ECMQV
82     {TPM_ALG_ECMQV,       TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 0, 1)},
83 #endif
84 #if ALG_KDF1_SP800_56A
85     {TPM_ALG_KDF1_SP800_56A, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 1)},
86 #endif
87 #if ALG_KDF2
88     {TPM_ALG_KDF2,        TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 1)},
89 #endif
90 #if ALG_KDF1_SP800_108
91     {TPM_ALG_KDF1_SP800_108, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 1)},
92 #endif
93 #if ALG_ECC
94     {TPM_ALG_ECC,         TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
95 #endif
96
97     {TPM_ALG_SYMCIPHER,    TPMA_ALGORITHM_INITIALIZER(0, 0, 0, 1, 0, 0, 0, 0, 0)},
98
99 #if ALG_CAMELLIA
100    {TPM_ALG_CAMELLIA,     TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
101 #endif
102 #if ALG_CMAC
103    {TPM_ALG_CMAC,         TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
104 #endif
105 #if ALG_CTR
106    {TPM_ALG_CTR,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 1, 0)},
107 #endif
108 #if ALG_OFB
109    {TPM_ALG_OFB,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 1, 0)},
110 #endif
111 #if ALG_CBC
112    {TPM_ALG_CBC,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 1, 0)},
113 #endif
114 #if ALG_CFB
115    {TPM_ALG_CFB,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 1, 0)},
116 #endif
117 #if ALG_ECB

```

```

118     {TPM_ALG_ECB,                TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
119 #endif
120 };

```

9.1.3 AlgorithmCapGetImplemented()

This function is used by TPM2_GetCapability() to return a list of the implemented algorithms.

Return Value	Meaning
YES	more algorithms to report
NO	no more algorithms to report

```

121 TPMI_YES_NO
122 AlgorithmCapGetImplemented(
123     TPM_ALG_ID      algID,      // IN: the starting algorithm ID
124     UINT32           count,      // IN: count of returned algorithms
125     TPML_ALG_PROPERTY *algList  // OUT: algorithm list
126 )
127 {
128     TPMI_YES_NO    more = NO;
129     UINT32         i;
130     UINT32         algNum;
131
132     // initialize output algorithm list
133     algList->count = 0;
134
135     // The maximum count of algorithms we may return is MAX_CAP_ALGS.
136     if(count > MAX_CAP_ALGS)
137         count = MAX_CAP_ALGS;
138
139     // Compute how many algorithms are defined in s_algorithms array.
140     algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
141
142     // Scan the implemented algorithm list to see if there is a match to 'algID'.
143     for(i = 0; i < algNum; i++)
144     {
145         // If algID is less than the starting algorithm ID, skip it
146         if(s_algorithms[i].algID < algID)
147             continue;
148         if(algList->count < count)
149         {
150             // If we have not filled up the return list, add more algorithms
151             // to it
152             algList->algProperties[algList->count].alg = s_algorithms[i].algID;
153             algList->algProperties[algList->count].algProperties =
154                 s_algorithms[i].attributes;
155             algList->count++;
156         }
157         else
158         {
159             // If the return list is full but we still have algorithms
160             // available, report this and stop scanning.
161             more = YES;
162             break;
163         }
164     }
165     return more;
166 }

```

9.1.4 AlgorithmGetImplementedVector()

This function returns the bit vector of the implemented algorithms.

```
167 LIB_EXPORT
168 void(
169     ALGORITHM_VECTOR    *implemented    // OUT: the implemented bits are SET
170 )
171 {
172     int                index;
173
174     // Nothing implemented until we say it is
175     MemorySet(implemented, 0, sizeof(ALGORITHM_VECTOR));
176     // Go through the list of implemented algorithms and SET the corresponding bit in
177     // in the implemented vector
178     for(index = (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1;
179         index >= 0; index--)
180         SET_BIT(s_algorithms[index].algID, *implemented);
181     return;
182 }
```

9.2 Bits.c

9.2.1 Introduction

This file contains bit manipulation routines. They operate on bit arrays. The 0th bit in the array is the right-most bit in the 0th octet in the array.

NOTE If `pAssert()` is defined, the functions will assert if the indicated bit number is outside of the range of *bArray*. How the assert is handled is implementation dependent.

9.2.2 Includes

```
1 #include "Tpm.h"
```

9.2.3 Functions

9.2.3.1 TestBit()

This function is used to check the setting of a bit in an array of bits.

Return Value	Meaning
TRUE(1)	bit is set
FALSE(0)	bit is not set

```
2 BOOL
3 TestBit(
4     unsigned int    bitNum,           // IN: number of the bit in 'bArray'
5     BYTE            *bArray,         // IN: array containing the bits
6     unsigned int    bytesInArray    // IN: size in bytes of 'bArray'
7 )
8 {
9     pAssert(bytesInArray > (bitNum >> 3));
10    return((bArray[bitNum >> 3] & (1 << (bitNum & 7))) != 0);
11 }
```

9.2.3.2 SetBit()

This function will set the indicated bit in *bArray*.

```
12 void
13 SetBit(
14     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     pAssert(bytesInArray > (bitNum >> 3));
20     bArray[bitNum >> 3] |= (1 << (bitNum & 7));
21 }
```

9.2.3.3 ClearBit()

This function will clear the indicated bit in *bArray*.

```
22 void
```

```
23 ClearBit(  
24     unsigned int    bitNum,          // IN: number of the bit in 'bArray'.  
25     BYTE           *bArray,         // IN: array containing the bits  
26     unsigned int    bytesInArray    // IN: size in bytes of 'bArray'  
27 )  
28 {  
29     pAssert(bytesInArray > (bitNum >> 3));  
30     bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));  
31 }
```

DRAFT

9.3 CommandCodeAttributes.c

9.3.1 Introduction

This file contains the functions for testing various command properties.

9.3.2 Includes and Defines

```
1  #include "Tpm.h"
2  #include "CommandCodeAttributes_fp.h"
```

Set the default value for CC_VEND if not already set

```
3  #ifndef CC_VEND
4  #define CC_VEND (TPM_CC) (0x20000000)
5  #endif
6
7  typedef UINT16      ATTRIBUTE_TYPE;
```

The following file is produced from the command tables in part 3 of the specification. It defines the attributes for each of the commands.

NOTE This file is currently produced by an automated process. Files produced from Part 2 or Part 3 tables through automated processes are not included in the specification so that there is no ambiguity about the table containing the information being the normative definition.

```
8  #define _COMMAND_CODE_ATTRIBUTES_
9  #include "CommandAttributeData.h"
```

9.3.3 Command Attribute Functions

9.3.3.1 NextImplementedIndex()

This function is used when the lists are not compressed. In a compressed list, only the implemented commands are present. So, a search might find a value but that value may not be implemented. This function checks to see if the input *commandIndex* points to an implemented command and, if not, it searches upwards until it finds one. When the list is compressed, this function gets defined as a no-op.

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	command is not implemented
other	index of the command

```
10  #if !COMPRESSED_LISTS
11  static COMMAND_INDEX
12  NextImplementedIndex(
13      COMMAND_INDEX      commandIndex
14  )
15  {
16      for(; commandIndex < COMMAND_COUNT; commandIndex++)
17      {
18          if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
19              return commandIndex;
20      }
21      return UNIMPLEMENTED_COMMAND_INDEX;
22  }
23  #else
24  #define NextImplementedIndex(x) (x)
```

25 **#endif**

9.3.3.2 GetClosestCommandIndex()

This function returns the command index for the command with a value that is equal to or greater than the input value

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	command is not implemented
other	index of a command

```

26 COMMAND_INDEX
27 GetClosestCommandIndex(
28     TPM_CC      commandCode    // IN: the command code to start at
29 )
30 {
31     BOOL          vendor = (commandCode & CC_VEND) != 0;
32     COMMAND_INDEX searchIndex = (COMMAND_INDEX)commandCode;
33
34     // The commandCode is a UINT32 and the search index is UINT16. We are going to
35     // search for a match but need to make sure that the commandCode value is not
36     // out of range. To do this, need to clear the vendor bit of the commandCode
37     // (if set) and compare the result to the 16-bit searchIndex value. If it is
38     // out of range, indicate that the command is not implemented
39     if((commandCode & ~CC_VEND) != searchIndex)
40         return UNIMPLEMENTED_COMMAND_INDEX;
41
42     // if there is at least one vendor command, the last entry in the array will
43     // have the v bit set. If the input commandCode is larger than the last
44     // vendor-command, then it is out of range.
45     if(vendor)
46     {
47         #if VENDOR_COMMAND_ARRAY_SIZE > 0
48             COMMAND_INDEX    commandIndex;
49             COMMAND_INDEX    min;
50             COMMAND_INDEX    max;
51             int               diff;
52         #if LIBRARY_COMMAND_ARRAY_SIZE == COMMAND_COUNT
53             #error "Constants are not consistent."
54         #endif
55         // Check to see if the value is equal to or below the minimum
56         // entry.
57         // Note: Put this check first so that the typical case of only one vendor-
58         // specific command doesn't waste any more time.
59         if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE], TPMA_CC,
60             commandIndex) >= searchIndex)
61         {
62             // the vendor array is always assumed to be packed so there is
63             // no need to check to see if the command is implemented
64             return LIBRARY_COMMAND_ARRAY_SIZE;
65         }
66         // See if this is out of range on the top
67         if(GET_ATTRIBUTE(s_ccAttr[COMMAND_COUNT - 1], TPMA_CC, commandIndex)
68             < searchIndex)
69         {
70             return UNIMPLEMENTED_COMMAND_INDEX;
71         }
72         commandIndex = UNIMPLEMENTED_COMMAND_INDEX; // Needs initialization to keep
73                                                         // compiler happy
74         min = LIBRARY_COMMAND_ARRAY_SIZE;           // first vendor command
75         max = COMMAND_COUNT - 1;                     // last vendor command
76         diff = 1;                                     // needs initialization to keep

```



```

77                                     // compiler happy
78     while(min <= max)
79     {
80         commandIndex = (min + max + 1) / 2;
81         diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
82             - searchIndex;
83         if(diff == 0)
84             return commandIndex;
85         if(diff > 0)
86             max = commandIndex - 1;
87         else
88             min = commandIndex + 1;
89     }
90     // didn't find an exact match. commandIndex will be pointing at the last
91     // item tested. If 'diff' is positive, then the last item tested was
92     // larger index of the command code so it is the smallest value
93     // larger than the requested value.
94     if(diff > 0)
95         return commandIndex;
96     // if 'diff' is negative, then the value tested was smaller than
97     // the commandCode index and the next higher value is the correct one.
98     // Note: this will necessarily be in range because of the earlier check
99     // that the index was within range.
100    return commandIndex + 1;
101 #else
102     // If there are no vendor commands so anything with the vendor bit set is out
103     // of range
104     return UNIMPLEMENTED_COMMAND_INDEX;
105 #endif
106 }
107 // Get here if the V-Bit was not set in 'commandCode'
108
109 if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE - 1], TPMA_CC,
110     commandIndex) < searchIndex)
111 {
112     // requested index is out of the range to the top
113 #if VENDOR_COMMAND_ARRAY_SIZE > 0
114     // If there are vendor commands, then the first vendor command
115     // is the next value greater than the commandCode.
116     // NOTE: we got here if the starting index did not have the V bit but we
117     // reached the end of the array of library commands (non-vendor). Since
118     // there is at least one vendor command, and vendor commands are always
119     // in a compressed list that starts after the library list, the next
120     // index value contains a valid vendor command.
121     return LIBRARY_COMMAND_ARRAY_SIZE;
122 #else
123     // if there are no vendor commands, then this is out of range
124     return UNIMPLEMENTED_COMMAND_INDEX;
125 #endif
126 }
127 // If the request is lower than any value in the array, then return
128 // the lowest value (needs to be an index for an implemented command)
129 if(GET_ATTRIBUTE(s_ccAttr[0], TPMA_CC, commandIndex) >= searchIndex)
130 {
131     return NextImplementedIndex(0);
132 }
133 else
134 {
135 #if COMPRESSED_LISTS
136     COMMAND_INDEX      commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
137     COMMAND_INDEX      min = 0;
138     COMMAND_INDEX      max = LIBRARY_COMMAND_ARRAY_SIZE - 1;
139     int                diff = 1;
140 #if LIBRARY_COMMAND_ARRAY_SIZE == 0
141 #error "Something is terribly wrong"
142 #endif

```

```

143 // The s_ccAttr array contains an extra entry at the end (a zero value).
144 // Don't count this as an array entry. This means that max should start
145 // out pointing to the last valid entry in the array which is - 2
146 pAssert(max == (sizeof(s_ccAttr) / sizeof(TPMA_CC)
147             - VENDOR_COMMAND_ARRAY_SIZE - 2));
148 while(min <= max)
149 {
150     commandIndex = (min + max + 1) / 2;
151     diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC,
152                        commandIndex) - searchIndex;
153     if(diff == 0)
154         return commandIndex;
155     if(diff > 0)
156         max = commandIndex - 1;
157     else
158         min = commandIndex + 1;
159 }
160 // didn't find an exact match. commandIndex will be pointing at the
161 // last item tested. If diff is positive, then the last item tested was
162 // larger index of the command code so it is the smallest value
163 // larger than the requested value.
164 if(diff > 0)
165     return commandIndex;
166 // if diff is negative, then the value tested was smaller than
167 // the commandCode index and the next higher value is the correct one.
168 // Note: this will necessarily be in range because of the earlier check
169 // that the index was within range.
170 return commandIndex + 1;
171 #else
172 // The list is not compressed so offset into the array by the command
173 // code value of the first entry in the list. Then go find the first
174 // implemented command.
175 return NextImplementedIndex(searchIndex
176                             - (COMMAND_INDEX)s_ccAttr[0].commandIndex);
177 #endif
178 }
179 }

```

9.3.3.3 CommandCodeToCommandIndex()

This function returns the index in the various attributes arrays of the command.

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	command is not implemented
other	index of the command

```

180 COMMAND_INDEX
181 CommandCodeToCommandIndex(
182     TPM_CC      commandCode    // IN: the command code to look up
183 )
184 {
185     // Extract the low 16-bits of the command code to get the starting search index
186     COMMAND_INDEX searchIndex = (COMMAND_INDEX)commandCode;
187     BOOL          vendor = (commandCode & CC_VEND) != 0;
188     COMMAND_INDEX commandIndex;
189     #if !COMPRESSED_LISTS
190     if(!vendor)
191     {
192         commandIndex = searchIndex - (COMMAND_INDEX)s_ccAttr[0].commandIndex;
193         // Check for out of range or unimplemented.
194         // Note, since a COMMAND_INDEX is unsigned, if searchIndex is smaller than
195         // the lowest value of command, it will become a 'negative' number making

```

```

196     // it look like a large unsigned number, this will cause it to fail
197     // the unsigned check below.
198     if(commandIndex >= LIBRARY_COMMAND_ARRAY_SIZE
199        || (s_commandAttributes[commandIndex] & IS_IMPLEMENTED) == 0)
200         return UNIMPLEMENTED_COMMAND_INDEX;
201     return commandIndex;
202 }
203 #endif
204 // Need this code for any vendor code lookup or for compressed lists
205 commandIndex = GetClosestCommandIndex(commandCode);
206
207 // Look at the returned value from get closest. If it isn't the one that was
208 // requested, then the command is not implemented.
209 if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
210 {
211     if((GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
212        != searchIndex)
213        || (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V)) != vendor)
214         commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
215 }
216 return commandIndex;
217 }

```

9.3.3.4 GetNextCommandIndex()

This function returns the index of the next implemented command.

Return Value	Meaning
UNIMPLEMENTED_COMMAND_INDEX	no more implemented commands
other	the index of the next implemented command

```

218 COMMAND_INDEX
219 GetNextCommandIndex(
220     COMMAND_INDEX    commandIndex    // IN: the starting index
221 )
222 {
223     while(++commandIndex < COMMAND_COUNT)
224     {
225         #if !COMPRESSED_LISTS
226             if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
227                 #endif
228                 return commandIndex;
229     }
230     return UNIMPLEMENTED_COMMAND_INDEX;
231 }

```

9.3.3.5 GetCommandCode()

This function returns the *commandCode* associated with the command index

```

232 TPM_CC
233 GetCommandCode(
234     COMMAND_INDEX    commandIndex    // IN: the command index
235 )
236 {
237     TPM_CC    commandCode = GET_ATTRIBUTE(s_ccAttr[commandIndex],
238                                           TPMA_CC, commandIndex);
239     if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
240         commandCode += CC_VEND;
241     return commandCode;

```

242 }

9.3.3.6 CommandAuthRole()

This function returns the authorization role required of a handle.

Return Value	Meaning
AUTH_NONE	no authorization is required
AUTH_USER	user role authorization is required
AUTH_ADMIN	admin role authorization is required
AUTH_DUP	duplication role authorization is required

```

243 AUTH_ROLE
244 CommandAuthRole(
245     COMMAND_INDEX    commandIndex, // IN: command index
246     UINT32           handleIndex    // IN: handle index (zero based)
247 )
248 {
249     if(0 == handleIndex)
250     {
251         // Any authorization role set?
252         COMMAND_ATTRIBUTES properties = s_commandAttributes[commandIndex];
253
254         if(properties & HANDLE_1_USER)
255             return AUTH_USER;
256         if(properties & HANDLE_1_ADMIN)
257             return AUTH_ADMIN;
258         if(properties & HANDLE_1_DUP)
259             return AUTH_DUP;
260     }
261     else if(1 == handleIndex)
262     {
263         if(s_commandAttributes[commandIndex] & HANDLE_2_USER)
264             return AUTH_USER;
265     }
266     return AUTH_NONE;
267 }

```

9.3.3.7 EncryptSize()

This function returns the size of the decrypt size field. This function returns 0 if encryption is not allowed

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```

268 int
269 EncryptSize(
270     COMMAND_INDEX    commandIndex // IN: command index
271 )
272 {
273     return ((s_commandAttributes[commandIndex] & ENCRYPT_2) ? 2 :
274            (s_commandAttributes[commandIndex] & ENCRYPT_4) ? 4 : 0);
275 }

```

9.3.3.8 DecryptSize()

This function returns the size of the decrypt size field. This function returns 0 if decryption is not allowed

Return Value	Meaning
0	encryption not allowed
2	size field is two bytes
4	size field is four bytes

```

276  int
277  DecryptSize(
278      COMMAND_INDEX    commandIndex    // IN: command index
279  )
280  {
281      return ((s_commandAttributes[commandIndex] & DECRYPT_2) ? 2 :
282              (s_commandAttributes[commandIndex] & DECRYPT_4) ? 4 : 0);
283  }

```

9.3.3.9 IsSessionAllowed()

This function indicates if the command is allowed to have sessions.

This function must not be called if the command is not known to be implemented.

Return Value	Meaning
TRUE(1)	session is allowed with this command
FALSE(0)	session is not allowed with this command

```

284  BOOL
285  IsSessionAllowed(
286      COMMAND_INDEX    commandIndex    // IN: the command to be checked
287  )
288  {
289      return ((s_commandAttributes[commandIndex] & NO_SESSIONS) == 0);
290  }

```

9.3.3.10 IsHandleInResponse()

This function determines if a command has a handle in the response

```

291  BOOL
292  IsHandleInResponse(
293      COMMAND_INDEX    commandIndex
294  )
295  {
296      return ((s_commandAttributes[commandIndex] & R_HANDLE) != 0);
297  }

```

9.3.3.11 IsWriteOperation()

Checks to see if an operation will write to an NV Index and is subject to being blocked by read-lock

```

298  BOOL
299  IsWriteOperation(
300      COMMAND_INDEX    commandIndex    // IN: Command to check
301  )

```

```

302 {
303 #ifdef WRITE_LOCK
304     return ((s_commandAttributes[commandIndex] & WRITE_LOCK) != 0);
305 #else
306     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
307     {
308         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
309         {
310             case TPM_CC_NV_Write:
311 #if CC_NV_Increment
312                 case TPM_CC_NV_Increment:
313 #endif
314 #if CC_NV_SetBits
315                 case TPM_CC_NV_SetBits:
316 #endif
317 #if CC_NV_Extend
318                 case TPM_CC_NV_Extend:
319 #endif
320 #if CC_AC_Send
321                 case TPM_CC_AC_Send:
322 #endif
323                 // NV write lock counts as a write operation for authorization purposes.
324                 // We check to see if the NV is write locked before we do the
325                 // authorization. If it is locked, we fail the command early.
326                 case TPM_CC_NV_WriteLock:
327                     return TRUE;
328                 default:
329                     break;
330             }
331         }
332     return FALSE;
333 #endif
334 }

```

9.3.3.12 IsReadOperation()

Checks to see if an operation will write to an NV Index and is subject to being blocked by write-lock.

```

335 BOOL
336 IsReadOperation(
337     COMMAND_INDEX    commandIndex    // IN: Command to check
338 )
339 {
340 #ifdef READ_LOCK
341     return ((s_commandAttributes[commandIndex] & READ_LOCK) != 0);
342 #else
343     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
344     {
345         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
346         {
347             case TPM_CC_NV_Read:
348             case TPM_CC_PolicyNV:
349             case TPM_CC_NV_Certify:
350                 // NV read lock counts as a read operation for authorization purposes.
351                 // We check to see if the NV is read locked before we do the
352                 // authorization. If it is locked, we fail the command early.
353                 case TPM_CC_NV_ReadLock:
354                     return TRUE;
355                 default:
356                     break;
357             }
358         }
359     }
360     return FALSE;

```

```

361 #endif
362 }

```

9.3.3.13 CommandCapGetCCList()

This function returns a list of implemented commands and command attributes starting from the command in *commandCode*.

Return Value	Meaning
YES	more command attributes are available
NO	no more command attributes are available

```

363 TPMI_YES_NO
364 CommandCapGetCCList(
365     TPM_CC      commandCode,    // IN: start command code
366     UINT32      count,          // IN: maximum count for number of entries in
367                                // 'commandList'
368     TPML_CCA    *commandList    // OUT: list of TPMA_CC
369 )
370 {
371     TPMI_YES_NO    more = NO;
372     COMMAND_INDEX  commandIndex;
373
374     // initialize output handle list count
375     commandList->count = 0;
376
377     for(commandIndex = GetClosestCommandIndex(commandCode);
378         commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
379         commandIndex = GetNextCommandIndex(commandIndex))
380     {
381 #if !COMPRESSED_LISTS
382         // this check isn't needed for compressed lists.
383         if(!(s_commandAttributes[commandIndex] & IS_IMPLEMENTED))
384             continue;
385 #endif
386         if(commandList->count < count)
387         {
388             // If the list is not full, add the attributes for this command.
389             commandList->commandAttributes[commandList->count]
390                 = s_ccAttr[commandIndex];
391             commandList->count++;
392         }
393         else
394         {
395             // If the list is full but there are more commands to report,
396             // indicate this and return.
397             more = YES;
398             break;
399         }
400     }
401     return more;
402 }

```

9.3.3.14 IsVendorCommand()

Function indicates if a command index references a vendor command.

Return Value	Meaning
TRUE(1)	command is a vendor command
FALSE(0)	command is not a vendor command

```
403  BOOL  
404  IsVendorCommand(  
405      COMMAND_INDEX    commandIndex    // IN: command index to check  
406  )  
407  {  
408      return (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V));  
409  }
```

9.4 Entity.c

9.4.1 Description

The functions in this file are used for accessing properties for handles of various types. Functions in other files require handles of a specific type but the functions in this file allow use of any handle type.

9.4.2 Includes

```
1 #include "Tpm.h"
```

9.4.3 Functions

9.4.3.1 EntityGetLoadStatus()

This function will check that all the handles access loaded entities.

Error Return	Meaning
TPM_RC_HANDLE	handle type does not match
TPM_RC_REFERENCE_Hx	entity is not present
TPM_RC_HIERARCHY	entity belongs to a disabled hierarchy
TPM_RC_OBJECT_MEMORY	handle is an evict object but there is no space to load it to RAM

```
2 TPM_RC
3 EntityGetLoadStatus(
4     COMMAND      *command          // IN/OUT: command parsing structure
5 )
6 {
7     UINT32        i;
8     TPM_RC        result = TPM_RC_SUCCESS;
9     //
10    for(i = 0; i < command->handleNum; i++)
11    {
12        TPM_HANDLE  handle = command->handles[i];
13        switch(HandleGetType(handle))
14        {
15            // For handles associated with hierarchies, the entity is present
16            // only if the associated enable is SET.
17            case TPM_HT_PERMANENT:
18                switch(handle)
19                {
20                    case TPM_RH_OWNER:
21                        if(!gc.shEnable)
22                            result = TPM_RC_HIERARCHY;
23                        break;
24
25            #ifdef VENDOR_PERMANENT
26                case VENDOR_PERMANENT:
27            #endif
28                case TPM_RH_ENDORSEMENT:
29                    if(!gc.ehEnable)
30                        result = TPM_RC_HIERARCHY;
31                    break;
32                case TPM_RH_PLATFORM:
33                    if(!g_phEnable)
34                        result = TPM_RC_HIERARCHY;
35                    break;
```

```

36         // null handle, PW session handle and lockout
37         // handle are always available
38     case TPM_RH_NULL:
39     case TPM_RS_PW:
40         // Need to be careful for lockout. Lockout is always available
41         // for policy checks but not always available when authValue
42         // is being checked.
43     case TPM_RH_LOCKOUT:
44         // Rather than have #ifdefs all over the code,
45         // CASE_ACT_HANDLE is defined in ACT.h. It is 'case TPM_RH_ACT_x:'
46         // FOR_EACH_ACT(CASE_ACT_HANDLE) creates a simple
47         // case TPM_RH_ACT_x: // for each of the implemented ACT.
48     FOR_EACH_ACT(CASE_ACT_HANDLE)
49         break;
50     default:
51         // If the implementation has a manufacturer-specific value
52         // then test for it here. Since this implementation does
53         // not have any, this implementation returns the same failure
54         // that unmarshaling of a bad handle would produce.
55         if(((TPM_RH)handle >= TPM_RH_AUTH_00)
56            && ((TPM_RH)handle <= TPM_RH_AUTH_FF))
57             // if the implementation has a manufacturer-specific value
58             result = TPM_RC_VALUE;
59         else
60             // The handle is in the range of reserved handles but is
61             // not implemented in this TPM.
62             result = TPM_RC_VALUE;
63         break;
64     }
65     break;
66 case TPM_HT_TRANSIENT:
67     // For a transient object, check if the handle is associated
68     // with a loaded object.
69     if(!IsObjectPresent(handle))
70         result = TPM_RC_REFERENCE_H0;
71     break;
72 case TPM_HT_PERSISTENT:
73     // Persistent object
74     // Copy the persistent object to RAM and replace the handle with the
75     // handle of the assigned slot. A TPM_RC_OBJECT_MEMORY,
76     // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
77     // ObjectLoadEvict()
78     result = ObjectLoadEvict(&command->handles[i], command->index);
79     break;
80 case TPM_HT_HMAC_SESSION:
81     // For an HMAC session, see if the session is loaded
82     // and if the session in the session slot is actually
83     // an HMAC session.
84     if(SessionIsLoaded(handle))
85     {
86         SESSION *session;
87         session = SessionGet(handle);
88         // Check if the session is a HMAC session
89         if(session->attributes.isPolicy == SET)
90             result = TPM_RC_HANDLE;
91     }
92     else
93         result = TPM_RC_REFERENCE_H0;
94     break;
95 case TPM_HT_POLICY_SESSION:
96     // For a policy session, see if the session is loaded
97     // and if the session in the session slot is actually
98     // a policy session.
99     if(SessionIsLoaded(handle))
100     {
101         SESSION *session;

```

```

102         session = SessionGet(handle);
103         // Check if the session is a policy session
104         if(session->attributes.isPolicy == CLEAR)
105             result = TPM_RC_HANDLE;
106     }
107     else
108         result = TPM_RC_REFERENCE_H0;
109     break;
110 case TPM_HT_NV_INDEX:
111     // For an NV Index, use the TPM-specific routine
112     // to search the IN Index space.
113     result = NvIndexIsAccessible(handle);
114     break;
115 case TPM_HT_PCR:
116     // Any PCR handle that is unmarshaled successfully referenced
117     // a PCR that is defined.
118     break;
119 #if CC_AC_Send
120 case TPM_HT_AC:
121     // Use the TPM-specific routine to search for the AC
122     result = AcIsAccessible(handle);
123     break;
124 #endif
125 default:
126     // Any other handle type is a defect in the unmarshaling code.
127     FAIL(FATAL_ERROR_INTERNAL);
128     break;
129 }
130 if(result != TPM_RC_SUCCESS)
131 {
132     if(result == TPM_RC_REFERENCE_H0)
133         result = result + i;
134     else
135         result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
136     break;
137 }
138 }
139 return result;
140 }

```

9.4.3.2 EntityGetAuthValue()

This function is used to access the *authValue* associated with a handle. This function assumes that the handle references an entity that is accessible and the handle is not for a persistent objects. That is EntityGetLoadStatus() should have been called. Also, the accessibility of the *authValue* should have been verified by IsAuthValueAvailable().

This function copies the authorization value of the entity to *auth*.

Return Value	Meaning
count	number of bytes in the <i>authValue</i> with 0's stripped

```

141 UINT16
142 EntityGetAuthValue(
143     TPMI_DH_ENTITY    handle,           // IN: handle of entity
144     TPM2B_AUTH        *auth            // OUT: authValue of the entity
145 )
146 {
147     TPM2B_AUTH        *pAuth = NULL;
148
149     auth->t.size = 0;
150
151     switch(HandleGetType(handle))

```

```

152     {
153         case TPM_HT_PERMANENT:
154         {
155             switch (handle)
156             {
157                 case TPM_RH_OWNER:
158                     // ownerAuth for TPM_RH_OWNER
159                     pAuth = &gp.ownerAuth;
160                     break;
161                 case TPM_RH_ENDORSEMENT:
162                     // endorsementAuth for TPM_RH_ENDORSEMENT
163                     pAuth = &gp.endorsementAuth;
164                     break;
165                     // The ACT use platformAuth for auth
166                     FOR_EACH_ACT(CASE_ACT_HANDLE)
167                 case TPM_RH_PLATFORM:
168                     // platformAuth for TPM_RH_PLATFORM
169                     pAuth = &gc.platformAuth;
170                     break;
171                 case TPM_RH_LOCKOUT:
172                     // lockoutAuth for TPM_RH_LOCKOUT
173                     pAuth = &gp.lockoutAuth;
174                     break;
175                 case TPM_RH_NULL:
176                     // nullAuth for TPM_RH_NULL. Return 0 directly here
177                     return 0;
178                     break;
179 #ifndef VENDOR_PERMANENT
180                 case VENDOR_PERMANENT:
181                     // vendor authorization value
182                     pAuth = &g_platformUniqueDetails;
183 #endif
184                 default:
185                     // If any other permanent handle is present it is
186                     // a code defect.
187                     FAIL(FATAL_ERROR_INTERNAL);
188                     break;
189             }
190             break;
191         }
192         case TPM_HT_TRANSIENT:
193             // authValue for an object
194             // A persistent object would have been copied into RAM
195             // and would have an transient object handle here.
196             {
197                 OBJECT *object;
198
199                 object = HandleToObject(handle);
200                 // special handling if this is a sequence object
201                 if (ObjectIsSequence(object))
202                 {
203                     pAuth = &((HASH_OBJECT *)object)->auth;
204                 }
205                 else
206                 {
207                     // Authorization is available only when the private portion of
208                     // the object is loaded. The check should be made before
209                     // this function is called
210                     pAssert(object->attributes.publicOnly == CLEAR);
211                     pAuth = &object->sensitive.authValue;
212                 }
213             }
214             break;
215         case TPM_HT_NV_INDEX:
216             // authValue for an NV index
217         {

```

```

218         NV_INDEX      *nvIndex = NvGetIndexInfo(handle, NULL);
219         pAssert(nvIndex != NULL);
220         pAuth = &nvIndex->authValue;
221     }
222     break;
223     case TPM_HT_PCR:
224         // authValue for PCR
225         pAuth = PCRGetAuthValue(handle);
226         break;
227     default:
228         // If any other handle type is present here, then there is a defect
229         // in the unmarshaling code.
230         FAIL(FATAL_ERROR_INTERNAL);
231         break;
232 }
233 // Copy the authValue
234 MemoryCopy2B((TPM2B *)auth, (TPM2B *)pAuth, sizeof(auth->t.buffer));
235 MemoryRemoveTrailingZeros(auth);
236 return auth->t.size;
237 }

```

9.4.3.3 EntityGetAuthPolicy()

This function is used to access the *authPolicy* associated with a handle. This function assumes that the handle references an entity that is accessible and the handle is not for a persistent objects. That is EntityGetLoadStatus() should have been called. Also, the accessibility of the *authPolicy* should have been verified by IsAuthPolicyAvailable().

This function copies the authorization policy of the entity to *authPolicy*.

The return value is the hash algorithm for the policy.

```

238 TPMI_ALG_HASH
239 EntityGetAuthPolicy(
240     TPMI_DH_ENTITY    handle,           // IN: handle of entity
241     TPM2B_DIGEST      *authPolicy      // OUT: authPolicy of the entity
242 )
243 {
244     TPMI_ALG_HASH      hashAlg = TPM_ALG_NULL;
245     authPolicy->t.size = 0;
246
247     switch(HandleGetType(handle))
248     {
249     case TPM_HT_PERMANENT:
250         switch(handle)
251         {
252         case TPM_RH_OWNER:
253             // ownerPolicy for TPM_RH_OWNER
254             *authPolicy = gp.ownerPolicy;
255             hashAlg = gp.ownerAlg;
256             break;
257         case TPM_RH_ENDORSEMENT:
258             // endorsementPolicy for TPM_RH_ENDORSEMENT
259             *authPolicy = gp.endorsementPolicy;
260             hashAlg = gp.endorsementAlg;
261             break;
262         case TPM_RH_PLATFORM:
263             // platformPolicy for TPM_RH_PLATFORM
264             *authPolicy = gc.platformPolicy;
265             hashAlg = gc.platformAlg;
266             break;
267         case TPM_RH_LOCKOUT:
268             // lockoutPolicy for TPM_RH_LOCKOUT
269             *authPolicy = gp.lockoutPolicy;

```

```

270         hashAlg = gp.lockoutAlg;
271         break;
272 #define ACT_GET_POLICY(N)
273         case TPM_RH_ACT_##N:
274             *authPolicy = go.ACT_##N.authPolicy;
275             hashAlg = go.ACT_##N.hashAlg;
276             break;
277             // Get the policy for each implemented ACT
278             FOR_EACH_ACT(ACT_GET_POLICY)
279         default:
280             hashAlg = TPM_ALG_ERROR;
281             break;
282     }
283     break;
284 case TPM_HT_TRANSIENT:
285     // authPolicy for an object
286     {
287         OBJECT *object = HandleToObject(handle);
288         *authPolicy = object->publicArea.authPolicy;
289         hashAlg = object->publicArea.nameAlg;
290     }
291     break;
292 case TPM_HT_NV_INDEX:
293     // authPolicy for a NV index
294     {
295         NV_INDEX *nvIndex = NvGetIndexInfo(handle, NULL);
296         pAssert(nvIndex != 0);
297         *authPolicy = nvIndex->publicArea.authPolicy;
298         hashAlg = nvIndex->publicArea.nameAlg;
299     }
300     break;
301 case TPM_HT_PCR:
302     // authPolicy for a PCR
303     hashAlg = PCRGetAuthPolicy(handle, authPolicy);
304     break;
305 default:
306     // If any other handle type is present it is a code defect.
307     FAIL(FATAL_ERROR_INTERNAL);
308     break;
309 }
310 return hashAlg;
311 }

```

9.4.3.4 EntityGetName()

This function returns the Name associated with a handle.

```

312 TPM2B_NAME *
313 EntityGetName(
314     TPMI_DH_ENTITY handle,           // IN: handle of entity
315     TPM2B_NAME *name,               // OUT: name of entity
316 )
317 {
318     switch(HandleGetType(handle))
319     {
320     case TPM_HT_TRANSIENT:
321     {
322         // Name for an object
323         OBJECT *object = HandleToObject(handle);
324         // an object with no nameAlg has no name
325         if(object->publicArea.nameAlg == TPM_ALG_NULL)
326             name->b.size = 0;
327         else
328             *name = object->name;

```



```

329         break;
330     }
331     case TPM_HT_NV_INDEX:
332         // Name for a NV index
333         NvGetNameByIndexHandle(handle, name);
334         break;
335     default:
336         // For all other types, the handle is the Name
337         name->t.size = sizeof(TPM_HANDLE);
338         UINT32_TO_BYTE_ARRAY(handle, name->t.name);
339         break;
340 }
341 return name;
342 }

```

9.4.3.5 EntityGetHierarchy()

This function returns the hierarchy handle associated with an entity.

- a) A handle that is a hierarchy handle is associated with itself.
- b) An NV index belongs to TPM_RH_PLATFORM if TPMA_NV_PLATFORMCREATE, is SET, otherwise it belongs to TPM_RH_OWNER
- c) An object handle belongs to its hierarchy.

```

343 TPMI_RH_HIERARCHY
344 EntityGetHierarchy(
345     TPMI_DH_ENTITY    handle           // IN :handle of entity
346 )
347 {
348     TPMI_RH_HIERARCHY hierarchy = TPM_RH_NULL;
349
350     switch(HandleGetType(handle))
351     {
352     case TPM_HT_PERMANENT:
353         // hierarchy for a permanent handle
354         switch(handle)
355         {
356             case TPM_RH_PLATFORM:
357             case TPM_RH_ENDORSEMENT:
358             case TPM_RH_NULL:
359                 hierarchy = handle;
360                 break;
361             // all other permanent handles are associated with the owner
362             // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
363             default:
364                 hierarchy = TPM_RH_OWNER;
365                 break;
366         }
367         break;
368     case TPM_HT_NV_INDEX:
369         // hierarchy for NV index
370     {
371         NV_INDEX          *nvIndex = NvGetIndexInfo(handle, NULL);
372         pAssert(nvIndex != NULL);
373
374         // If only the platform can delete the index, then it is
375         // considered to be in the platform hierarchy, otherwise it
376         // is in the owner hierarchy.
377         if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV,
378             PLATFORMCREATE))
379             hierarchy = TPM_RH_PLATFORM;
380         else
381             hierarchy = TPM_RH_OWNER;
382     }
383     }
384 }

```

```
382     }
383     break;
384     case TPM_HT_TRANSIENT:
385         // hierarchy for an object
386         {
387             OBJECT *object;
388             object = HandleToObject(handle);
389             if(object->attributes.ppsHierarchy)
390             {
391                 hierarchy = TPM_RH_PLATFORM;
392             }
393             else if(object->attributes.epsHierarchy)
394             {
395                 hierarchy = TPM_RH_ENDORSEMENT;
396             }
397             else if(object->attributes.spsHierarchy)
398             {
399                 hierarchy = TPM_RH_OWNER;
400             }
401         }
402     break;
403     case TPM_HT_PCR:
404         hierarchy = TPM_RH_OWNER;
405         break;
406     default:
407         FAIL(FATAL_ERROR_INTERNAL);
408         break;
409 }
410 // this is unreachable but it provides a return value for the default
411 // case which makes the compiler happy
412 return hierarchy;
413 }
```

9.5 Global.c

9.5.1 Description

This file will instance the TPM variables that are not stack allocated.

Descriptions of global variables are in Global.h. There macro definitions that allows a variable to be instanced or simply defined as an external variable. When global.h is included from this .c file, GLOBAL_C is defined and values are instanced (and possibly initialized), but when global.h is included by any other file, they are simply defined as external values. DO NOT DEFINE GLOBAL_C IN ANY OTHER FILE.

NOTE This is a change from previous implementations where Global.h just contained the extern declaration and values were instanced in this file. This change keeps the definition and instance in one file making maintenance easier. The instanced data will still be in the global.obj file.

The OIDs.h file works in a way that is similar to the Global.h with the definition of the values in OIDs.h such that they are instanced in global.obj. The macros that are defined in Global.h are used in OIDs.h in the same way as they are in Global.h.

9.5.2 Defines and Includes

```
1  #define GLOBAL_C
2  #include "Tpm.h"
3  #include "OIDs.h"
4
5  #if CC_CertifyX509
6  #   include "X509.h"
7  #endif // CC_CertifyX509
```

9.6 Handle.c

9.6.1 Description

This file contains the functions that return the type of a handle.

9.6.2 Includes

```
1 #include "Tpm.h"
```

9.6.3 Functions

9.6.3.1 HandleGetType()

This function returns the type of a handle which is the MSO of the handle.

```
2 TPM_HT
3 HandleGetType(
4     TPM_HANDLE      handle          // IN: a handle to be checked
5 )
6 {
7     // return the upper bytes of input data
8     return (TPM_HT)((handle & HR_RANGE_MASK) >> HR_SHIFT);
9 }
```

9.6.3.2 NextPermanentHandle()

This function returns the permanent handle that is equal to the input value or is the next higher value. If there is no handle with the input value and there is no next higher value, it returns 0:

```
10 TPM_HANDLE
11 NextPermanentHandle(
12     TPM_HANDLE      inHandle        // IN: the handle to check
13 )
14 {
15     // If inHandle is below the start of the range of permanent handles
16     // set it to the start and scan from there
17     if(inHandle < TPM_RH_FIRST)
18         inHandle = TPM_RH_FIRST;
19     // scan from input value until we find an implemented permanent handle
20     // or go out of range
21     for(; inHandle <= TPM_RH_LAST; inHandle++)
22     {
23         switch(inHandle)
24         {
25             case TPM_RH_OWNER:
26             case TPM_RH_NULL:
27             case TPM_RS_PW:
28             case TPM_RH_LOCKOUT:
29             case TPM_RH_ENDORSEMENT:
30             case TPM_RH_PLATFORM:
31             case TPM_RH_PLATFORM_NV:
32 #ifdef VENDOR_PERMANENT
33             case VENDOR_PERMANENT:
34 #endif
35             // Each of the implemented ACT
36 #define ACT_IMPLEMENTED_CASE(N)
37             case TPM_RH_ACT_##N:
38 }
```

```

39         FOR_EACH_ACT (ACT_IMPLEMENTED_CASE)
40
41         return inHandle;
42         break;
43     default:
44         break;
45     }
46 }
47 // Out of range on the top
48 return 0;
49 }

```

9.6.3.3 PermanentCapGetHandles()

This function returns a list of the permanent handles of PCR, started from *handle*. If *handle* is larger than the largest permanent handle, an empty list will be returned with *more* set to NO.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```

50 TPMI_YES_NO
51 PermanentCapGetHandles (
52     TPM_HANDLE    handle,        // IN: start handle
53     UINT32        count,        // IN: count of returned handles
54     TPML_HANDLE   *handleList    // OUT: list of handle
55 )
56 {
57     TPMI_YES_NO    more = NO;
58     UINT32        i;
59
60     pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
61
62     // Initialize output handle list
63     handleList->count = 0;
64
65     // The maximum count of handles we may return is MAX_CAP_HANDLES
66     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
67
68     // Iterate permanent handle range
69     for(i = NextPermanentHandle(handle);
70         i != 0; i = NextPermanentHandle(i + 1))
71     {
72         if(handleList->count < count)
73         {
74             // If we have not filled up the return list, add this permanent
75             // handle to it
76             handleList->handle[handleList->count] = i;
77             handleList->count++;
78         }
79         else
80         {
81             // If the return list is full but we still have permanent handle
82             // available, report this and stop iterating
83             more = YES;
84             break;
85         }
86     }
87     return more;
88 }

```

9.6.3.4 PermanentHandleGetPolicy()

This function returns a list of the permanent handles of PCR, started from *handle*. If *handle* is larger than the largest permanent handle, an empty list will be returned with *more* set to NO.

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```

89  TPMI_YES_NO
90  PermanentHandleGetPolicy(
91      TPM_HANDLE      handle,          // IN: start handle
92      UINT32          count,          // IN: max count of returned handles
93      TPML_TAGGED_POLICY *policyList  // OUT: list of handle
94  )
95  {
96      TPMI_YES_NO      more = NO;
97
98      pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
99
100     // Initialize output handle list
101     policyList->count = 0;
102
103     // The maximum count of policies we may return is MAX_TAGGED_POLICIES
104     if(count > MAX_TAGGED_POLICIES)
105         count = MAX_TAGGED_POLICIES;
106
107     // Iterate permanent handle range
108     for(handle = NextPermanentHandle(handle);
109         handle != 0;
110         handle = NextPermanentHandle(handle + 1))
111     {
112         TPM2B_DIGEST    policyDigest;
113         TPM_ALG_ID      policyAlg;
114         // Check to see if this permanent handle has a policy
115         policyAlg = EntityGetAuthPolicy(handle, &policyDigest);
116         if(policyAlg == TPM_ALG_ERROR)
117             continue;
118         if(policyList->count < count)
119         {
120             // If we have not filled up the return list, add this
121             // policy to the list;
122             policyList->policies[policyList->count].handle = handle;
123             policyList->policies[policyList->count].policyHash.hashAlg = policyAlg;
124             MemoryCopy(&policyList->policies[policyList->count].policyHash.digest,
125                     policyDigest.t.buffer, policyDigest.t.size);
126             policyList->count++;
127         }
128         else
129         {
130             // If the return list is full but we still have permanent handle
131             // available, report this and stop iterating
132             more = YES;
133             break;
134         }
135     }
136     return more;
137 }

```

9.7 IoBuffers.c

9.7.1 Includes and Data Definitions

This definition allows this module to **see** the values that are private to this module but kept in Global.c for ease of state migration.

```
1  #define IO_BUFFER_C
2  #include "Tpm.h"
3  #include "IoBuffers_fp.h"
```

9.7.2 Buffers and Functions

These buffers are set aside to hold command and response values. In this implementation, it is not guaranteed that the code will stop accessing the `s_actionInputBuffer` before starting to put values in the `s_actionOutputBuffer` so different buffers are required.

9.7.2.1 MemoryIoBufferAllocationReset()

This function is used to reset the allocation of buffers.

```
4  void
5  MemoryIoBufferAllocationReset(
6      void
7  )
8  {
9      s_actionIoAllocation = 0;
10 }
```

9.7.2.2 MemoryIoBufferZero()

Function zeros the action I/O buffer at the end of a command. Calling this is not mandatory for proper functionality.

```
11 void
12 MemoryIoBufferZero(
13     void
14 )
15 {
16     memset(s_actionIoBuffer, 0, s_actionIoAllocation);
17 }
```

9.7.2.3 MemoryGetInBuffer()

This function returns the address of the buffer into which the command parameters will be unmarshaled in preparation for calling the command actions.

```
18 BYTE *
19 MemoryGetInBuffer(
20     UINT32      size           // Size, in bytes, required for the input
21                                     // unmarshaling
22 )
23 {
24     pAssert(size <= sizeof(s_actionIoBuffer));
25     // In this implementation, a static buffer is set aside for the command action
26     // buffers. The buffer is shared between input and output. This is because
27     // there is no need to allocate for the worst case input and worst case output
```



```

28     // at the same time.
29     // Round size up
30     #define UoM (sizeof(s_actionIoBuffer[0]))
31     size = (size + (UoM - 1)) & (UINT32_MAX - (UoM - 1));
32     memset(s_actionIoBuffer, 0, size);
33     s_actionIoAllocation = size;
34     return (BYTE *)&s_actionIoBuffer[0];
35 }

```

9.7.2.4 MemoryGetOutBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```

36 BYTE *
37 MemoryGetOutBuffer(
38     UINT32      size           // required size of the buffer
39 )
40 {
41     BYTE      *retVal = (BYTE *)&s_actionIoBuffer[s_actionIoAllocation / UoM];
42     pAssert((size + s_actionIoAllocation) < (sizeof(s_actionIoBuffer)));
43     // In this implementation, a static buffer is set aside for the command action
44     // output buffer.
45     memset(retVal, 0, size);
46     s_actionIoAllocation += size;
47     return retVal;
48 }

```

9.7.2.5 IsLabelProperlyFormatted()

This function checks that a label is a null-terminated string.

NOTE this function is here because there was no better place for it.

Return Value	Meaning
TRUE(1)	string is null terminated
FALSE(0)	string is not null terminated

```

49 BOOL
50 IsLabelProperlyFormatted(
51     TPM2B      *x
52 )
53 {
54     return ((x->size == 0) || ((x->buffer[x->size - 1] == 0));
55 }

```

9.8 Locality.c

9.8.1 Includes

```
1 #include "Tpm.h"
```

9.8.2 LocalityGetAttributes()

This function will convert a locality expressed as an integer into TPMA_LOCALITY form.

The function returns the locality attribute.

```
2 TPMA_LOCALITY
3 LocalityGetAttributes(
4     UINT8          locality          // IN: locality value
5 )
6 {
7     TPMA_LOCALITY    locality_attributes;
8     BYTE             *localityAsByte = (BYTE *)&locality_attributes;
9
10    MemorySet(&locality_attributes, 0, sizeof(TPMA_LOCALITY));
11    switch(locality)
12    {
13        case 0:
14            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ZERO);
15            break;
16        case 1:
17            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ONE);
18            break;
19        case 2:
20            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_TWO);
21            break;
22        case 3:
23            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_THREE);
24            break;
25        case 4:
26            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_FOUR);
27            break;
28        default:
29            pAssert(locality > 31);
30            *localityAsByte = locality;
31            break;
32    }
33    return locality_attributes;
34 }
```

9.9 Manufacture.c

9.9.1 Description

This file contains the function that performs the **manufacturing** of the TPM in a simulated environment. These functions should not be used outside of a manufacturing or simulation environment.

9.9.2 Includes and Data Definitions

```
1  #define MANUFACTURE_C
2  #include "Tpm.h"
3  #include "TpmSizeChecks_fp.h"
```

9.9.3 Functions

9.9.3.1 TPM_Manufacture()

This function initializes the TPM values in preparation for the TPM's first use. This function will fail if previously called. The TPM can be re-manufactured by calling TPM_Teardown() first and then calling this function again.

Return Value	Meaning
-1	failure
0	success
1	manufacturing process previously performed

```
4  LIB_EXPORT int
5  TPM_Manufacture(
6      int      firstTime      // IN: indicates if this is the first call from
7                               //      main()
8  )
9  {
10     TPM_SU      orderlyShutdown;
11
12     #if RUNTIME_SIZE_CHECKS
13         // Call the function to verify the sizes of values that result from different
14         // compile options.
15         if(!TpmSizeChecks())
16             return -1;
17     #endif
18     #if LIBRARY_COMPATIBILITY_CHECK
19         // Make sure that the attached library performs as expected.
20         if(!MathLibraryCompatibilityCheck())
21             return -1;
22     #endif
23
24     // If TPM has been manufactured, return indication.
25     if(!firstTime && g_manufactured)
26         return 1;
27
28     // Do power on initializations of the cryptographic libraries.
29     CryptInit();
30
31     s_DAPendingOnNV = FALSE;
32
33     // initialize NV
34     NvManufacture();
```

```

35
36 // Clear the magic value in the DRBG state
37 go.drbgState.magic = 0;
38
39 CryptStartup(SU_RESET);
40
41 // default configuration for PCR
42 PCRSimStart();
43
44 // initialize pre-installed hierarchy data
45 // This should happen after NV is initialized because hierarchy data is
46 // stored in NV.
47 HierarchyPreInstall_Init();
48
49 // initialize dictionary attack parameters
50 DAPreInstall_Init();
51
52 // initialize PP list
53 PhysicalPresencePreInstall_Init();
54
55 // initialize command audit list
56 CommandAuditPreInstall_Init();
57
58 // first start up is required to be Startup(CLEAR)
59 orderlyShutdown = TPM_SU_CLEAR;
60 NV_WRITE_PERSISTENT(orderlyState, orderlyShutdown);
61
62 // initialize the firmware version
63 gp.firmwareV1 = FIRMWARE_V1;
64 #ifdef FIRMWARE_V2
65 gp.firmwareV2 = FIRMWARE_V2;
66 #else
67 gp.firmwareV2 = 0;
68 #endif
69 NV_SYNC_PERSISTENT(firmwareV1);
70 NV_SYNC_PERSISTENT(firmwareV2);
71
72 // initialize the total reset counter to 0
73 gp.totalResetCount = 0;
74 NV_SYNC_PERSISTENT(totalResetCount);
75
76 // initialize the clock stuff
77 go.clock = 0;
78 go.clockSafe = YES;
79
80 NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
81
82 // Commit NV writes. Manufacture process is an artificial process existing
83 // only in simulator environment and it is not defined in the specification
84 // that what should be the expected behavior if the NV write fails at this
85 // point. Therefore, it is assumed the NV write here is always success and
86 // no return code of this function is checked.
87 NvCommit();
88
89 g_manufactured = TRUE;
90
91 return 0;
92 }

```

9.9.3.2 TPM_TearDown()

This function prepares the TPM for re-manufacture. It should not be implemented in anything other than a simulated TPM.

In this implementation, all that is needed is to stop the cryptographic units and set a flag to indicate that the TPM can be re-manufactured. This should be all that is necessary to start the manufacturing process again.

Return Value	Meaning
0	success
1	TPM not previously manufactured

```

93  LIB_EXPORT int
94  TPM_TearDown(
95      void
96  )
97  {
98      g_manufactured = FALSE;
99      return 0;
100 }

```

9.9.3.3 TpmEndSimulation()

This function is called at the end of the simulation run. It is used to provoke printing of any statistics that might be needed.

```

101 LIB_EXPORT void
102 TpmEndSimulation(
103     void
104 )
105 {
106     #if SIMULATION
107         HashLibSimulationEnd();
108         SymLibSimulationEnd();
109         MathLibSimulationEnd();
110     #if ALG_RSA
111         RsaSimulationEnd();
112     #endif
113     #if ALG_ECC
114         EccSimulationEnd();
115     #endif
116     #endif // SIMULATION
117 }

```

9.10 Marshal.c

9.10.1 Introduction

This file contains the marshaling and unmarshaling code.

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).

9.10.2 Unmarshal and Marshal a Value

In TPM 2.0 Part 2, a TPMI_DI_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:

```

1  TPM_RC
2  TPMI_DH_OBJECT_Unmarshal(TPMI_DH_OBJECT *target, BYTE **buffer, INT32 *size,
3                           BOOL flag)
4  {
5      TPM_RC    result;
6      result = TPM_HANDLE_Unmarshal((TPM_HANDLE *)target, buffer, size);
7      if(result != TPM_RC_SUCCESS)
8          return result;
9      if(*target == TPM_RH_NULL)
10     {
11         if(flag)
12             return TPM_RC_SUCCESS;
13         else
14             return TPM_RC_VALUE;
15     }
16     if((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
17         &&((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST))
18             return TPM_RC_VALUE;
19     return TPM_RC_SUCCESS;
20 }
```

and the following marshaling code:

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

```

1  UINT16
2  TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
```

```

3 {
4     return UINT32_Marshal((UINT32 *)source, buffer, size);
5 }

```

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`.

```

6 #define TPMI_DH_OBJECT_Marshal(source, buffer, size) \
7     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`.

9.10.3 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.

```

1  TPM_RC
2  TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size,
3                               UINT32 selector)
4  {
5      switch(selector) {
6          #if ALG_KEYEDHASH
7              case TPM_ALG_KEYEDHASH:
8                  return TPMS_KEYEDHASH_PARMS_Unmarshal(
9                      (TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
10             #endif
11             #if ALG_SYMCIPHER
12                 case TPM_ALG_SYMCIPHER:
13                     return TPMT_SYM_DEF_OBJECT_Unmarshal(
14                         (TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
15             #endif
16             #if ALG_RSA
17                 case TPM_ALG_RSA:
18                     return TPMS_RSA_PARMS_Unmarshal(
19                         (TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
20             #endif
21             #if ALG_ECC
22                 case TPM_ALG_ECC:
23                     return TPMS_ECC_PARMS_Unmarshal(
24                         (TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
25             #endif
26         }

```



```

27     return TPM_RC_SELECTOR;
28 }

```

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:

```

1  UINT16
2  TPMU_PUBLIC_PARMS_Marshal(TPMU_PUBLIC_PARMS *source, BYTE **buffer, INT32 *size,
3                             UINT32 selector)
4  {
5      switch(selector) {
6          #if ALG_KEYEDHASH
7              case TPM_ALG_KEYEDHASH:
8                  return TPMS_KEYEDHASH_PARMS_Marshal(
9                      (TPMS_KEYEDHASH_PARMS *)&(source->keyedHash), buffer, size);
10             #endif
11             #if ALG_SYMCIPHER
12                 case TPM_ALG_SYMCIPHER:
13                     return TPMT_SYM_DEF_OBJECT_Marshal(
14                         (TPMT_SYM_DEF_OBJECT *)&(source->symDetail), buffer, size);
15                 #endif
16             #if ALG_RSA
17                 case TPM_ALG_RSA:
18                     return TPMS_RSA_PARMS_Marshal(
19                         (TPMS_RSA_PARMS *)&(source->rsaDetail), buffer, size);
20                 #endif
21             #if ALG_ECC
22                 case TPM_ALG_ECC:
23                     return TPMS_ECC_PARMS_Marshal(
24                         (TPMS_ECC_PARMS *)&(source->eccDetail), buffer, size);
25                 #endif
26             }
27             assert(1);
28             return 0;
29 }

```

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 clause illustrates this.

9.10.4 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 nameAlg of the object. 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:

```

1  TPM_RC
2  TPMT_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
3  {
4      TPM_RC    result;
5      result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)&(target->type),
6                                          buffer, size);
7      if(result != TPM_RC_SUCCESS)
8          return result;
9      result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)&(target->nameAlg),
10                                      buffer, size, flag);
11     if(result != TPM_RC_SUCCESS)
12         return result;
13     result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)&(target->objectAttributes),
14                                    buffer, size);
15     if(result != TPM_RC_SUCCESS)
16         return result;
17     result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)&(target->authPolicy),
18                                    buffer, size);
19     if(result != TPM_RC_SUCCESS)
20         return result;
21
22     result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)&(target->parameters),
23                                          buffer, size, (UINT32)target->type);
24     if(result != TPM_RC_SUCCESS)
25         return result;
26
27     result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)&(target->unique),
28                                      buffer, size, (UINT32)target->type);
29     if(result != TPM_RC_SUCCESS)
30         return result;
31
32     return TPM_RC_SUCCESS;
33 }
```

The marshaling code for the TPMT_PUBLIC structure is:

```

1  UINT16
2  TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
3  {
4      UINT16    result = 0;
5      result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
6          (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
7      result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
8          (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size))
9      ;
10     result = (UINT16)(result + TPMA_OBJECT_Marshal(
11         (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
12
13     result = (UINT16)(result + TPM2B_DIGEST_Marshal(
14         (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));
15
16     result = (UINT16)(result + TPMU_PUBLIC_PARMS_Marshal(
17         (TPMU_PUBLIC_PARMS *)&(source->parameters), buffer, size,
18         (UINT32)source->type));
19
20     result = (UINT16)(result + TPMU_PUBLIC_ID_Marshal(
21         (TPMU_PUBLIC_ID *)&(source->unique), buffer, size,
22         (UINT32)source->type));
23
24     return result;
25 }

```

9.10.5 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:

```

1  TPM_RC
2  TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
3  {
4      TPM_RC    result;
5      result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
6      if(result != TPM_RC_SUCCESS)
7          return result;
8
9      if( (target->count < 2))           // This check is triggered by the {2:} notation
10                                         // on 'count'
11          return TPM_RC_SIZE;
12
13      if((target->count) > 8)           // This check is triggered by the {:8} notation

```

```

14                                     // on 'digests'.
15     return TPM_RC_SIZE;
16
17     result = TPM2B_DIGEST_Array_Unmarshal((TPM2B_DIGEST *) (target->digests),
18                                           buffer, size, (INT32) (target->count));
19     if(result != TPM_RC_SUCCESS)
20         return result;
21
22     return TPM_RC_SUCCESS;
23 }

```

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:

```

1  TPM_RC
2  TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                               INT32 count)
4  {
5      TPM_RC    result;
6      INT32 i;
7      for(i = 0; i < count; i++) {
8          result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9          if(result != TPM_RC_SUCCESS)
10             return result;
11     }
12     return TPM_RC_SUCCESS;
13 }
14

```

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.

```

1  UINT16
2  TPML_DIGEST_Marshal(TPML_DIGEST *source, BYTE **buffer, INT32 *size)
3  {
4      UINT16    result = 0;
5      result = (UINT16) (result + UINT32_Marshal((UINT32 *) (&(source->count)), buffer,
6                                                  size));
7      result = (UINT16) (result + TPM2B_DIGEST_Array_Marshal(
8          (TPM2B_DIGEST *) (source->digests), buffer, size,
9          (INT32) (source->count)));
10
11     return result;
12 }

```

The marshaling code for the array is:

```

1  TPM_RC
2  TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
3                               INT32 count)
4  {
5      TPM_RC    result;
6      INT32 i;
7      for(i = 0; i < count; i++) {
8          result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
9          if(result != TPM_RC_SUCCESS)
10             return result;
11     }
12     return TPM_RC_SUCCESS;
13 }

```

9.10.6 TPM2B Handling

A TPM2B structure is handled as a special case. The unmarshaling code is similar to what is shown in 9.10.5 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

```

1  TPM_RC
2  TPM2B_EVENT_Unmarshal(TPM2B_EVENT *target, BYTE **buffer, INT32 *size)
3  {
4      TPM_RC    result;
5      result = UINT16_Unmarshal((UINT16 *)&(target->t.size), buffer, size);
6      if(result != TPM_RC_SUCCESS)
7          return result;
8      // if size equal to 0, the rest of the structure is a zero buffer
9      // so stop processing
10     if(target->t.size == 0)
11         return TPM_RC_SUCCESS;
12     if((target->t.size) > 1024)    // This check is triggered by the {:1024}
13                                 // notation on 'buffer'
14         return TPM_RC_SIZE;
15     result = BYTE_Array_Unmarshal((BYTE *) (target->t.buffer), buffer, size,
16                                   (INT32) (target->t.size));
17     if(result != TPM_RC_SUCCESS)
18         return result;
19     return TPM_RC_SUCCESS;
20 }
```

using these structure definitions:

```

1  typedef union {
2      struct {
3          UINT16    size;
4          BYTE       buffer[1024];
5      }            t;
6      TPM2B         b;
7  } TPM2B_EVENT;
```

9.10.7 Table Marshal Headers

9.10.7.1 TableMarshal.h

```

1  #ifndef _TABLE_MARSHAL_H_
2  #define _TABLE_MARSHAL_H_
```

These are the basic unmarshaling types. This is in the first byte of each structure descriptor that is passed to Marshal()/Unmarshal() for processing.

```
3  #define UINT_MTYPE          0
4  #define VALUES_MTYPE      (UINT_MTYPE + 1)
5  #define TABLE_MTYPE       (VALUES_MTYPE + 1)
6  #define MIN_MAX_MTYPE      (TABLE_MTYPE + 1)
7  #define ATTRIBUTES_MTYPE   (MIN_MAX_MTYPE + 1)
8  #define STRUCTURE_MTYPE    (ATTRIBUTES_MTYPE + 1)
9  #define TPM2B_MTYPE        (STRUCTURE_MTYPE + 1)
10 #define TPM2BS_MTYPE       (TPM2B_MTYPE + 1)
11 #define LIST_MTYPE          (TPM2BS_MTYPE + 1) // TPML
12 #define ERROR_MTYPE         (LIST_MTYPE + 1)
13 #define NULL_MTYPE          (ERROR_MTYPE + 1)
14 #define COMPOSITE_MTYPE     (NULL_MTYPE + 1)
```

9.10.7.1.1.1 The Marshal Index

A structure is used to hold the values that guide the marshaling/unmarshaling of each of the types. Each structure has a name and an address. For a structure to define a TPMS_name, the structure is a TPMS_name_MARSHAL_STRUCT and its index is TPMS_name_MARSHAL_INDEX. So, to get the proper structure, use the associated marshal index. The marshal index is passed to Marshal() or Unmarshal() and those functions look up the proper structure.

To handle structures that allow a null value, the upper bit of each marshal index indicates if the null value is allowed. This is the NULL_FLAG. It is defined in TableMarshalIndex.h because it is needed by code outside of the marshaling code.

A structure will have a list of marshal indexes to indicate what to unmarshal. When that index appears in a structure/union, the value will contain a flag to indicate that the NULL_FLAG should be SET on the call to Unmarshal() to unmarshal the type. The caller simply takes the entry and passes it to Unmarshal() to indicate that the NULL_FLAG is SET. There is also the opportunity to SET the NULL_FLAG in the called structure if the NULL_FLAG was set in the call to the calling structure. This is indicated by:

```
15  #define NULL_MASK      ~(NULL_FLAG)
```

When looking up the value to marshal, the upper bit of the marshal index is masked to yield the actual index. The MSb is the flag bit that indicates if a null flag is set. Code does not verify that the bit is clear when the called object does not take a flag as this is a benign error.

the modifier byte as used by each MTYPE shown as a structure. They are expressed as a bit maps below. However, the code uses masking and not bit fields. The types show below are just to help in understanding.

NOTE LSb0 bit numbering is assumed in these typedefs.

When used in an UINT_MTYPE

```
16  typedef struct integerModifier {
17      unsigned      size : 2;
18      unsigned      sign : 1;
19      unsigned      unused : 7;
20  } integerModifier;
```

When used in a VALUES_MTYPE

```
21  typedef struct valuesModifier {
22      unsigned      size : 2;
23      unsigned      sign : 1;
24      unsigned      unused : 5;
25      unsigned      takesNull : 1;
26  } valuesModifier;
```

When used in a TABLE_MTYPE

```
27  typedef struct tableModifier {
28      unsigned      size : 2;
29      unsigned      sign : 1;
30      unsigned      unused : 3;
31      unsigned      hasBits : 1;
32      unsigned      takesNull : 1;
33  } tableModifier;
```

the modifier byte for MIN_MAX_MTYPE

```
34  typedef struct minMaxModifier {
```



```
35     unsigned          size : 2;
36     unsigned          sign : 1;
37     unsigned          unused : 3;
38     unsigned          hasBits : 1;
39     unsigned          takesNull : 1;
40 } minMaxModifier;
```

the modifier byte for ATTRIBUTES_MTYPE

```
41 typedef struct attributesModifier {
42     unsigned          size : 2;
43     unsigned          sign : 1;
44     unsigned          unused : 5;
45 } attributesModifier;
```

the modifier byte is not present in a STRUCTURE_MTYPE or an TPM2B_MTYPE

the modifier byte for a TPM2BS_MTYPE

```
46 typedef struct tpm2bsModifier {
47     unsigned          offset : 4;
48     unsigned          unused : 2;
49     unsigned          sizeEqual : 1;
50     unsigned          propagateNull : 1;
51 } tpm2bsModifier;
```

the modifier byte for a LIST_MTYPE

```
52 typedef struct listModifier {
53     unsigned          offset : 4;
54     unsigned          unused : 2;
55     unsigned          sizeEqual : 1;
56     unsigned          propagateNull : 1;
57 } listModifier;
```

9.10.7.1.1.2 Modifier Octet Values

These are in used in anything that is an integer value. Theses would not be in structure modifier bytes (they would be used in values in structures but not the STRUCTURE_MTYPE header.

```

58 #define ONE_BYTES          (0)
59 #define TWO_BYTES         (1)
60 #define FOUR_BYTES        (2)
61 #define EIGHT_BYTES       (3)
62 #define SIZE_MASK          (0x3)
63 #define IS_SIGNED          (1 << 2)    // when the unmarshaled type is a signed value
64 #define SIGNED_MASK        (SIZE_MASK | IS_SIGNED)

```

This may be used for any type except a UINT_MTYPE

```

65 #define TAKES_NULL          (1 << 7)    // when the type takes a null

```

When referencing a structure, this flag indicates if a null is to be propagated to the referenced structure or type.

```

66 #define PROPAGATE_NULL      (TAKES_NULL)

```

Can be used in min-max or table structures.

```

67 #define HAS_BITS            (1 << 6)    // when bit mask is present

```

In a union, we need to know if this is a union of constant arrays.

```

68 #define IS_ARRAY_UNION      (1 << 6)

```

In a TPM2BS_MTYPE

```

69 #define SIZE_EQUAL          (1 << 6)
70 #define OFFSET_MASK         (0xF)

```

Right now, there are three spare bits in the modifiers field.

Within the descriptor word of each entry in a StructMarsh_mst(), there is a selector field to determine which of the sub-types the entry represents and a field that is used to reference another structure entry. This is a 6-bit field allowing a structure to have 64 entries. This should be more than enough as the structures are not that long. As of now, only 10-bits of the descriptor word leaving room for expansion.

These are the values used in a STRUCTURE_MTYPE to identify the sub-type of the thing being processed

```

71 #define SIMPLE_STYPE        0
72 #define UNION_STYPE         1
73 #define ARRAY_STYPE         2

```

The code used GET_ to get the element type and the compiler uses SET_ to initialize the value. The element type is the three bits (2:0).

```

74 #define GET_ELEMENT_TYPE(val) (val & 7)
75 #define SET_ELEMENT_TYPE(val) (val & 7)

```

When an entry is an array or union, this references the structure entry that contains the dimension or selector value. The code then uses this number to look up the structure entry for that element to find out what it and where is it in memory. When this is not a reference, it is a simple type and it could be used as

an array value or a union selector. When a simple value, this field contains the size of the associated value (ONE_BYTES, TWO_BYTES ...)

The entry size/number is 6 bits (13:8).

```

76 #define GET_ELEMENT_NUMBER(val)      (((val) >> 8) & 0x3F)
77 #define SET_ELEMENT_NUMBER(val)      (((val) & 0x3F) << 8)
78 #define GET_ELEMENT_SIZE(val)        GET_ELEMENT_NUMBER(val)
79 #define SET_ELEMENT_SIZE(val)        SET_ELEMENT_NUMBER(val)

```

This determines if the null flag is propagated to this type. If generate, the NULL_FLAG is SET in the index value. This flag is one bit (7)

```

80 #define ELEMENT_PROPAGATE            (PROPAGATE_NULL)
81
82 #define INDEX_MASK                    ((UINT16) NULL_MASK)

```

This is used in all bit-field checks. These are used when a value that is checked is conditional (dependent on the compilation). For example, if AES_128 is (NO), then the bit associated with AES_128 will be 0. In some cases, the bit value is found by checking that the input is within the range of the table, and then using the (val - min) value to index the bit. This would be used when verifying that a particular algorithm is implemented. In other cases, there is a bit for each value in a table. For example, if checking the key sizes, there is a list of possible key sizes allowed by the algorithm registry and a bit field to indicate if that key size is allowed in the implementation. The smallest bit field has 32-bits because it is implemented as part of the *values* array in structures that allow bit fields.

```

83 #define IS_BIT_SET32(bit, bits)      \
84 (((UINT32 *)bits)[bit >> 5] & (1 << (bit & 0x1F))) != 0

```

For a COMPOSITE_MTYPE, the qualifiers byte has an element size and count.

```

85 #define SET_ELEMENT_COUNT(count)     ((count & 0x1F) << 3)
86 #define GET_ELEMENT_COUNT(val)       ((val >> 3) & 0x1F)
87
88 #endif // _TABLE_MARSHAL_H_

```

9.10.7.2 TableMarshalDefines.h

```

1  #ifndef _TABLE_MARSHAL_DEFINES_H_
2  #define _TABLE_MARSHAL_DEFINES_H_
3
4  #define NULL_SHIFT 15
5  #define NULL_FLAG (1 << NULL_SHIFT)

```

The range macro processes a min, max value and produces a values that is used in the computation to see if something is within a range. The max value is (max-min). This lets the check for something (*val*) within a range become: if((val - min) <= max) // passes if in range if((val - min) > max) // passes if not in range This works because all values are converted to UINT32 values before the compare. For (val - min), all values greater than or equal to val will become positive values with a value equal to *min* being zero. This means that in an unsigned compare against *max*, any value that is outside the range will appear to be a number greater than max. The benefit of this operation is that this will work even if the input value is a signed number as long as the input is sign extended.

```

6  #define RANGE(_min_, _max_, _base_) \
7      (UINT32)_min_, (UINT32)((_base_)(_max_ - _min_))

```

This macro is like the offsetof macro but, instead of computing the offset of a structure element, it computes the stride between elements that are in a structure array. This is used instead of sizeof() because the sizeof() operator on a structure can return an implementation dependent value.

```

8  #define STRIDE(s)      ((UINT16)(size_t)&(((s *)0)[1]))
9
10 #define MARSHAL_REF(TYPE)      ((UINT16)(offsetof(MARSHAL_DATA, TYPE)))

```

This macro creates the entry in the array lookup table

```

11 #define ARRAY_MARSHAL_ENTRY(TYPE)
12     { (marshalIndex_t)TYPE##_MARSHAL_REF, (UINT16)STRIDE(TYPE) }

```

Defines for array lookup

```

13 #define UINT8_ARRAY_MARSHAL_INDEX      0    // 0x00
14 #define TPM_CC_ARRAY_MARSHAL_INDEX     1    // 0x01
15 #define TPMA_CC_ARRAY_MARSHAL_INDEX     2    // 0x02
16 #define TPM_ALG_ID_ARRAY_MARSHAL_INDEX 3    // 0x03
17 #define TPM_HANDLE_ARRAY_MARSHAL_INDEX 4    // 0x04
18 #define TPM2B_DIGEST_ARRAY_MARSHAL_INDEX 5    // 0x05
19 #define TPMT_HA_ARRAY_MARSHAL_INDEX     6    // 0x06
20 #define TPMS_PCR_SELECTION_ARRAY_MARSHAL_INDEX 7    // 0x07
21 #define TPMS_ALG_PROPERTY_ARRAY_MARSHAL_INDEX 8    // 0x08
22 #define TPMS_TAGGED_PROPERTY_ARRAY_MARSHAL_INDEX 9    // 0x09
23 #define TPMS_TAGGED_PCR_SELECT_ARRAY_MARSHAL_INDEX 10    // 0x0A
24 #define TPM_ECC_CURVE_ARRAY_MARSHAL_INDEX 11    // 0x0B
25 #define TPMS_TAGGED_POLICY_ARRAY_MARSHAL_INDEX 12    // 0x0C
26 #define TPMS_ACT_DATA_ARRAY_MARSHAL_INDEX 13    // 0x0D
27 #define TPMS_AC_OUTPUT_ARRAY_MARSHAL_INDEX 14    // 0x0E

```

Defines for referencing a type by offset

```

28 #define UINT8_MARSHAL_REF      \
29     ((UINT16)(offsetof(MarshalData_st, UINT8_DATA)))
30 #define BYTE_MARSHAL_REF      UINT8_MARSHAL_REF
31 #define TPM_HT_MARSHAL_REF     UINT8_MARSHAL_REF
32 #define TPMA_LOCALITY_MARSHAL_REF  UINT8_MARSHAL_REF
33 #define UINT16_MARSHAL_REF     \
34     ((UINT16)(offsetof(MarshalData_st, UINT16_DATA)))
35 #define TPM_KEY_SIZE_MARSHAL_REF  UINT16_MARSHAL_REF
36 #define TPM_KEY_BITS_MARSHAL_REF  UINT16_MARSHAL_REF
37 #define TPM_ALG_ID_MARSHAL_REF    UINT16_MARSHAL_REF
38 #define TPM_ST_MARSHAL_REF        UINT16_MARSHAL_REF
39 #define UINT32_MARSHAL_REF     \
40     ((UINT16)(offsetof(MarshalData_st, UINT32_DATA)))
41 #define TPM_ALGORITHM_ID_MARSHAL_REF  UINT32_MARSHAL_REF
42 #define TPM_MODIFIER_INDICATOR_MARSHAL_REF  UINT32_MARSHAL_REF
43 #define TPM_AUTHORIZATION_SIZE_MARSHAL_REF  UINT32_MARSHAL_REF
44 #define TPM_PARAMETER_SIZE_MARSHAL_REF  UINT32_MARSHAL_REF
45 #define TPM_SPEC_MARSHAL_REF        UINT32_MARSHAL_REF
46 #define TPM_CONSTANTS32_MARSHAL_REF  UINT32_MARSHAL_REF
47 #define TPM_CC_MARSHAL_REF          UINT32_MARSHAL_REF
48 #define TPM_RC_MARSHAL_REF          UINT32_MARSHAL_REF
49 #define TPM_PT_MARSHAL_REF          UINT32_MARSHAL_REF
50 #define TPM_PT_PCR_MARSHAL_REF      UINT32_MARSHAL_REF
51 #define TPM_PS_MARSHAL_REF          UINT32_MARSHAL_REF
52 #define TPM_HANDLE_MARSHAL_REF      UINT32_MARSHAL_REF
53 #define TPM_RH_MARSHAL_REF          UINT32_MARSHAL_REF
54 #define TPM_HC_MARSHAL_REF          UINT32_MARSHAL_REF
55 #define TPMA_PERMANENT_MARSHAL_REF  UINT32_MARSHAL_REF
56 #define TPMA_STARTUP_CLEAR_MARSHAL_REF  UINT32_MARSHAL_REF
57 #define TPMA_MEMORY_MARSHAL_REF     UINT32_MARSHAL_REF
58 #define TPMA_CC_MARSHAL_REF         UINT32_MARSHAL_REF
59 #define TPMA_MODES_MARSHAL_REF      UINT32_MARSHAL_REF
60 #define TPMA_X509_KEY_USAGE_MARSHAL_REF  UINT32_MARSHAL_REF
61 #define TPM_NV_INDEX_MARSHAL_REF    UINT32_MARSHAL_REF
62 #define TPM_AE_MARSHAL_REF          UINT32_MARSHAL_REF

```

```

63 #define UINT64_MARSHAL_REF \
64     ((UINT16) (offsetof(MarshalData_st, UINT64_DATA)))
65 #define INT8_MARSHAL_REF \
66     ((UINT16) (offsetof(MarshalData_st, INT8_DATA)))
67 #define INT16_MARSHAL_REF \
68     ((UINT16) (offsetof(MarshalData_st, INT16_DATA)))
69 #define INT32_MARSHAL_REF \
70     ((UINT16) (offsetof(MarshalData_st, INT32_DATA)))
71 #define INT64_MARSHAL_REF \
72     ((UINT16) (offsetof(MarshalData_st, INT64_DATA)))
73 #define UINT0_MARSHAL_REF \
74     ((UINT16) (offsetof(MarshalData_st, UINT0_DATA)))
75 #define TPM_ECC_CURVE_MARSHAL_REF \
76     ((UINT16) (offsetof(MarshalData_st, TPM_ECC_CURVE_DATA)))
77 #define TPM_CLOCK_ADJUST_MARSHAL_REF \
78     ((UINT16) (offsetof(MarshalData_st, TPM_CLOCK_ADJUST_DATA)))
79 #define TPM_EO_MARSHAL_REF \
80     ((UINT16) (offsetof(MarshalData_st, TPM_EO_DATA)))
81 #define TPM_SU_MARSHAL_REF \
82     ((UINT16) (offsetof(MarshalData_st, TPM_SU_DATA)))
83 #define TPM_SE_MARSHAL_REF \
84     ((UINT16) (offsetof(MarshalData_st, TPM_SE_DATA)))
85 #define TPM_CAP_MARSHAL_REF \
86     ((UINT16) (offsetof(MarshalData_st, TPM_CAP_DATA)))
87 #define TPMA_ALGORITHM_MARSHAL_REF \
88     ((UINT16) (offsetof(MarshalData_st, TPMA_ALGORITHM_DATA)))
89 #define TPMA_OBJECT_MARSHAL_REF \
90     ((UINT16) (offsetof(MarshalData_st, TPMA_OBJECT_DATA)))
91 #define TPMA_SESSION_MARSHAL_REF \
92     ((UINT16) (offsetof(MarshalData_st, TPMA_SESSION_DATA)))
93 #define TPMA_ACT_MARSHAL_REF \
94     ((UINT16) (offsetof(MarshalData_st, TPMA_ACT_DATA)))
95 #define TPMI_YES_NO_MARSHAL_REF \
96     ((UINT16) (offsetof(MarshalData_st, TPMI_YES_NO_DATA)))
97 #define TPMI_DH_OBJECT_MARSHAL_REF \
98     ((UINT16) (offsetof(MarshalData_st, TPMI_DH_OBJECT_DATA)))
99 #define TPMI_DH_PARENT_MARSHAL_REF \
100     ((UINT16) (offsetof(MarshalData_st, TPMI_DH_PARENT_DATA)))
101 #define TPMI_DH_PERSISTENT_MARSHAL_REF \
102     ((UINT16) (offsetof(MarshalData_st, TPMI_DH_PERSISTENT_DATA)))
103 #define TPMI_DH_ENTITY_MARSHAL_REF \
104     ((UINT16) (offsetof(MarshalData_st, TPMI_DH_ENTITY_DATA)))
105 #define TPMI_DH_PCR_MARSHAL_REF \
106     ((UINT16) (offsetof(MarshalData_st, TPMI_DH_PCR_DATA)))
107 #define TPMI_SH_AUTH_SESSION_MARSHAL_REF \
108     ((UINT16) (offsetof(MarshalData_st, TPMI_SH_AUTH_SESSION_DATA)))
109 #define TPMI_SH_HMAC_MARSHAL_REF \
110     ((UINT16) (offsetof(MarshalData_st, TPMI_SH_HMAC_DATA)))
111 #define TPMI_SH_POLICY_MARSHAL_REF \
112     ((UINT16) (offsetof(MarshalData_st, TPMI_SH_POLICY_DATA)))
113 #define TPMI_DH_CONTEXT_MARSHAL_REF \
114     ((UINT16) (offsetof(MarshalData_st, TPMI_DH_CONTEXT_DATA)))
115 #define TPMI_DH_SAVED_MARSHAL_REF \
116     ((UINT16) (offsetof(MarshalData_st, TPMI_DH_SAVED_DATA)))
117 #define TPMI_RH_HIERARCHY_MARSHAL_REF \
118     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_HIERARCHY_DATA)))
119 #define TPMI_RH_ENABLES_MARSHAL_REF \
120     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_ENABLES_DATA)))
121 #define TPMI_RH_HIERARCHY_AUTH_MARSHAL_REF \
122     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_HIERARCHY_AUTH_DATA)))
123 #define TPMI_RH_HIERARCHY_POLICY_MARSHAL_REF \
124     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_HIERARCHY_POLICY_DATA)))
125 #define TPMI_RH_PLATFORM_MARSHAL_REF \
126     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_PLATFORM_DATA)))
127 #define TPMI_RH_OWNER_MARSHAL_REF \
128     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_OWNER_DATA)))

```



```

129 #define TPMI_RH_ENDORSEMENT_MARSHAL_REF \
130     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_ENDORSEMENT_DATA)))
131 #define TPMI_RH_PROVISION_MARSHAL_REF \
132     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_PROVISION_DATA)))
133 #define TPMI_RH_CLEAR_MARSHAL_REF \
134     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_CLEAR_DATA)))
135 #define TPMI_RH_NV_AUTH_MARSHAL_REF \
136     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_NV_AUTH_DATA)))
137 #define TPMI_RH_LOCKOUT_MARSHAL_REF \
138     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_LOCKOUT_DATA)))
139 #define TPMI_RH_NV_INDEX_MARSHAL_REF \
140     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_NV_INDEX_DATA)))
141 #define TPMI_RH_AC_MARSHAL_REF \
142     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_AC_DATA)))
143 #define TPMI_RH_ACT_MARSHAL_REF \
144     ((UINT16) (offsetof(MarshalData_st, TPMI_RH_ACT_DATA)))
145 #define TPMI_ALG_HASH_MARSHAL_REF \
146     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_HASH_DATA)))
147 #define TPMI_ALG_ASYM_MARSHAL_REF \
148     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_ASYM_DATA)))
149 #define TPMI_ALG_SYM_MARSHAL_REF \
150     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_SYM_DATA)))
151 #define TPMI_ALG_SYM_OBJECT_MARSHAL_REF \
152     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_SYM_OBJECT_DATA)))
153 #define TPMI_ALG_SYM_MODE_MARSHAL_REF \
154     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_SYM_MODE_DATA)))
155 #define TPMI_ALG_KDF_MARSHAL_REF \
156     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_KDF_DATA)))
157 #define TPMI_ALG_SIG_SCHEME_MARSHAL_REF \
158     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_SIG_SCHEME_DATA)))
159 #define TPMI_ECC_KEY_EXCHANGE_MARSHAL_REF \
160     ((UINT16) (offsetof(MarshalData_st, TPMI_ECC_KEY_EXCHANGE_DATA)))
161 #define TPMI_ST_COMMAND_TAG_MARSHAL_REF \
162     ((UINT16) (offsetof(MarshalData_st, TPMI_ST_COMMAND_TAG_DATA)))
163 #define TPMI_ALG_MAC_SCHEME_MARSHAL_REF \
164     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_MAC_SCHEME_DATA)))
165 #define TPMI_ALG_CIPHER_MODE_MARSHAL_REF \
166     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_CIPHER_MODE_DATA)))
167 #define TPMS_EMPTY_MARSHAL_REF \
168     ((UINT16) (offsetof(MarshalData_st, TPMS_EMPTY_DATA)))
169 #define TPMS_ENC_SCHEME_RSAES_MARSHAL_REF \
170     TPMS_EMPTY_MARSHAL_REF
171 #define TPMS_ALGORITHM_DESCRIPTION_MARSHAL_REF \
172     ((UINT16) (offsetof(MarshalData_st, TPMS_ALGORITHM_DESCRIPTION_DATA)))
173 #define TPMU_HA_MARSHAL_REF \
174     ((UINT16) (offsetof(MarshalData_st, TPMU_HA_DATA)))
175 #define TPMT_HA_MARSHAL_REF \
176     ((UINT16) (offsetof(MarshalData_st, TPMT_HA_DATA)))
177 #define TPM2B_DIGEST_MARSHAL_REF \
178     ((UINT16) (offsetof(MarshalData_st, TPM2B_DIGEST_DATA)))
179 #define TPM2B_NONCE_MARSHAL_REF \
180     TPM2B_DIGEST_MARSHAL_REF
181 #define TPM2B_AUTH_MARSHAL_REF \
182     TPM2B_DIGEST_MARSHAL_REF
183 #define TPM2B_OPERAND_MARSHAL_REF \
184     TPM2B_DIGEST_MARSHAL_REF
185 #define TPM2B_DATA_MARSHAL_REF \
186     ((UINT16) (offsetof(MarshalData_st, TPM2B_DATA_DATA)))
187 #define TPM2B_EVENT_MARSHAL_REF \
188     ((UINT16) (offsetof(MarshalData_st, TPM2B_EVENT_DATA)))
189 #define TPM2B_MAX_BUFFER_MARSHAL_REF \
190     ((UINT16) (offsetof(MarshalData_st, TPM2B_MAX_BUFFER_DATA)))
191 #define TPM2B_MAX_NV_BUFFER_MARSHAL_REF \
192     ((UINT16) (offsetof(MarshalData_st, TPM2B_MAX_NV_BUFFER_DATA)))
193 #define TPM2B_TIMEOUT_MARSHAL_REF \
194     ((UINT16) (offsetof(MarshalData_st, TPM2B_TIMEOUT_DATA)))
195 #define TPM2B_IV_MARSHAL_REF \
196     ((UINT16) (offsetof(MarshalData_st, TPM2B_IV_DATA)))
197 #define NULL_UNION_MARSHAL_REF \
198     ((UINT16) (offsetof(MarshalData_st, NULL_UNION_DATA)))

```

```

195 #define TPMU_NAME_MARSHAL_REF NULL_UNION_MARSHAL_REF
196 #define TPMU_SENSITIVE_CREATE_MARSHAL_REF NULL_UNION_MARSHAL_REF
197 #define TPM2B_NAME_MARSHAL_REF \
198     ((UINT16) (offsetof(MarshalData_st, TPM2B_NAME_DATA)))
199 #define TPMS_PCR_SELECT_MARSHAL_REF \
200     ((UINT16) (offsetof(MarshalData_st, TPMS_PCR_SELECT_DATA)))
201 #define TPMS_PCR_SELECTION_MARSHAL_REF \
202     ((UINT16) (offsetof(MarshalData_st, TPMS_PCR_SELECTION_DATA)))
203 #define TPMT_TK_CREATION_MARSHAL_REF \
204     ((UINT16) (offsetof(MarshalData_st, TPMT_TK_CREATION_DATA)))
205 #define TPMT_TK_VERIFIED_MARSHAL_REF \
206     ((UINT16) (offsetof(MarshalData_st, TPMT_TK_VERIFIED_DATA)))
207 #define TPMT_TK_AUTH_MARSHAL_REF \
208     ((UINT16) (offsetof(MarshalData_st, TPMT_TK_AUTH_DATA)))
209 #define TPMT_TK_HASHCHECK_MARSHAL_REF \
210     ((UINT16) (offsetof(MarshalData_st, TPMT_TK_HASHCHECK_DATA)))
211 #define TPMS_ALG_PROPERTY_MARSHAL_REF \
212     ((UINT16) (offsetof(MarshalData_st, TPMS_ALG_PROPERTY_DATA)))
213 #define TPMS_TAGGED_PROPERTY_MARSHAL_REF \
214     ((UINT16) (offsetof(MarshalData_st, TPMS_TAGGED_PROPERTY_DATA)))
215 #define TPMS_TAGGED_PCR_SELECT_MARSHAL_REF \
216     ((UINT16) (offsetof(MarshalData_st, TPMS_TAGGED_PCR_SELECT_DATA)))
217 #define TPMS_TAGGED_POLICY_MARSHAL_REF \
218     ((UINT16) (offsetof(MarshalData_st, TPMS_TAGGED_POLICY_DATA)))
219 #define TPMS_ACT_DATA_MARSHAL_REF \
220     ((UINT16) (offsetof(MarshalData_st, TPMS_ACT_DATA_DATA)))
221 #define TPML_CC_MARSHAL_REF \
222     ((UINT16) (offsetof(MarshalData_st, TPML_CC_DATA)))
223 #define TPML_CCA_MARSHAL_REF \
224     ((UINT16) (offsetof(MarshalData_st, TPML_CCA_DATA)))
225 #define TPML_ALG_MARSHAL_REF \
226     ((UINT16) (offsetof(MarshalData_st, TPML_ALG_DATA)))
227 #define TPML_HANDLE_MARSHAL_REF \
228     ((UINT16) (offsetof(MarshalData_st, TPML_HANDLE_DATA)))
229 #define TPML_DIGEST_MARSHAL_REF \
230     ((UINT16) (offsetof(MarshalData_st, TPML_DIGEST_DATA)))
231 #define TPML_DIGEST_VALUES_MARSHAL_REF \
232     ((UINT16) (offsetof(MarshalData_st, TPML_DIGEST_VALUES_DATA)))
233 #define TPML_PCR_SELECTION_MARSHAL_REF \
234     ((UINT16) (offsetof(MarshalData_st, TPML_PCR_SELECTION_DATA)))
235 #define TPML_ALG_PROPERTY_MARSHAL_REF \
236     ((UINT16) (offsetof(MarshalData_st, TPML_ALG_PROPERTY_DATA)))
237 #define TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF \
238     ((UINT16) (offsetof(MarshalData_st, TPML_TAGGED_TPM_PROPERTY_DATA)))
239 #define TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF \
240     ((UINT16) (offsetof(MarshalData_st, TPML_TAGGED_PCR_PROPERTY_DATA)))
241 #define TPML_ECC_CURVE_MARSHAL_REF \
242     ((UINT16) (offsetof(MarshalData_st, TPML_ECC_CURVE_DATA)))
243 #define TPML_TAGGED_POLICY_MARSHAL_REF \
244     ((UINT16) (offsetof(MarshalData_st, TPML_TAGGED_POLICY_DATA)))
245 #define TPML_ACT_DATA_MARSHAL_REF \
246     ((UINT16) (offsetof(MarshalData_st, TPML_ACT_DATA_DATA)))
247 #define TPMU_CAPABILITIES_MARSHAL_REF \
248     ((UINT16) (offsetof(MarshalData_st, TPMU_CAPABILITIES_DATA)))
249 #define TPMS_CAPABILITY_DATA_MARSHAL_REF \
250     ((UINT16) (offsetof(MarshalData_st, TPMS_CAPABILITY_DATA_DATA)))
251 #define TPMS_CLOCK_INFO_MARSHAL_REF \
252     ((UINT16) (offsetof(MarshalData_st, TPMS_CLOCK_INFO_DATA)))
253 #define TPMS_TIME_INFO_MARSHAL_REF \
254     ((UINT16) (offsetof(MarshalData_st, TPMS_TIME_INFO_DATA)))
255 #define TPMS_TIME_ATTEST_INFO_MARSHAL_REF \
256     ((UINT16) (offsetof(MarshalData_st, TPMS_TIME_ATTEST_INFO_DATA)))
257 #define TPMS_CERTIFY_INFO_MARSHAL_REF \
258     ((UINT16) (offsetof(MarshalData_st, TPMS_CERTIFY_INFO_DATA)))
259 #define TPMS_QUOTE_INFO_MARSHAL_REF \
260     ((UINT16) (offsetof(MarshalData_st, TPMS_QUOTE_INFO_DATA)))

```



```

261 #define TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF \
262 ((UINT16) (offsetof(MarshalData_st, TPMS_COMMAND_AUDIT_INFO_DATA)))
263 #define TPMS_SESSION_AUDIT_INFO_MARSHAL_REF \
264 ((UINT16) (offsetof(MarshalData_st, TPMS_SESSION_AUDIT_INFO_DATA)))
265 #define TPMS_CREATION_INFO_MARSHAL_REF \
266 ((UINT16) (offsetof(MarshalData_st, TPMS_CREATION_INFO_DATA)))
267 #define TPMS_NV_CERTIFY_INFO_MARSHAL_REF \
268 ((UINT16) (offsetof(MarshalData_st, TPMS_NV_CERTIFY_INFO_DATA)))
269 #define TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF \
270 ((UINT16) (offsetof(MarshalData_st, TPMS_NV_DIGEST_CERTIFY_INFO_DATA)))
271 #define TPMI_ST_ATTEST_MARSHAL_REF \
272 ((UINT16) (offsetof(MarshalData_st, TPMI_ST_ATTEST_DATA)))
273 #define TPMU_ATTEST_MARSHAL_REF \
274 ((UINT16) (offsetof(MarshalData_st, TPMU_ATTEST_DATA)))
275 #define TPMS_ATTEST_MARSHAL_REF \
276 ((UINT16) (offsetof(MarshalData_st, TPMS_ATTEST_DATA)))
277 #define TPM2B_ATTEST_MARSHAL_REF \
278 ((UINT16) (offsetof(MarshalData_st, TPM2B_ATTEST_DATA)))
279 #define TPMS_AUTH_COMMAND_MARSHAL_REF \
280 ((UINT16) (offsetof(MarshalData_st, TPMS_AUTH_COMMAND_DATA)))
281 #define TPMS_AUTH_RESPONSE_MARSHAL_REF \
282 ((UINT16) (offsetof(MarshalData_st, TPMS_AUTH_RESPONSE_DATA)))
283 #define TPMI_TDES_KEY_BITS_MARSHAL_REF \
284 ((UINT16) (offsetof(MarshalData_st, TPMI_TDES_KEY_BITS_DATA)))
285 #define TPMI_AES_KEY_BITS_MARSHAL_REF \
286 ((UINT16) (offsetof(MarshalData_st, TPMI_AES_KEY_BITS_DATA)))
287 #define TPMI_SM4_KEY_BITS_MARSHAL_REF \
288 ((UINT16) (offsetof(MarshalData_st, TPMI_SM4_KEY_BITS_DATA)))
289 #define TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF \
290 ((UINT16) (offsetof(MarshalData_st, TPMI_CAMELLIA_KEY_BITS_DATA)))
291 #define TPMU_SYM_KEY_BITS_MARSHAL_REF \
292 ((UINT16) (offsetof(MarshalData_st, TPMU_SYM_KEY_BITS_DATA)))
293 #define TPMU_SYM_MODE_MARSHAL_REF \
294 ((UINT16) (offsetof(MarshalData_st, TPMU_SYM_MODE_DATA)))
295 #define TPMT_SYM_DEF_MARSHAL_REF \
296 ((UINT16) (offsetof(MarshalData_st, TPMT_SYM_DEF_DATA)))
297 #define TPMT_SYM_DEF_OBJECT_MARSHAL_REF \
298 ((UINT16) (offsetof(MarshalData_st, TPMT_SYM_DEF_OBJECT_DATA)))
299 #define TPM2B_SYM_KEY_MARSHAL_REF \
300 ((UINT16) (offsetof(MarshalData_st, TPM2B_SYM_KEY_DATA)))
301 #define TPMS_SYMCIPHER_PARMS_MARSHAL_REF \
302 ((UINT16) (offsetof(MarshalData_st, TPMS_SYMCIPHER_PARMS_DATA)))
303 #define TPM2B_LABEL_MARSHAL_REF \
304 ((UINT16) (offsetof(MarshalData_st, TPM2B_LABEL_DATA)))
305 #define TPMS_DERIVE_MARSHAL_REF \
306 ((UINT16) (offsetof(MarshalData_st, TPMS_DERIVE_DATA)))
307 #define TPM2B_DERIVE_MARSHAL_REF \
308 ((UINT16) (offsetof(MarshalData_st, TPM2B_DERIVE_DATA)))
309 #define TPM2B_SENSITIVE_DATA_MARSHAL_REF \
310 ((UINT16) (offsetof(MarshalData_st, TPM2B_SENSITIVE_DATA_DATA)))
311 #define TPMS_SENSITIVE_CREATE_MARSHAL_REF \
312 ((UINT16) (offsetof(MarshalData_st, TPMS_SENSITIVE_CREATE_DATA)))
313 #define TPM2B_SENSITIVE_CREATE_MARSHAL_REF \
314 ((UINT16) (offsetof(MarshalData_st, TPM2B_SENSITIVE_CREATE_DATA)))
315 #define TPMS_SCHEME_HASH_MARSHAL_REF \
316 ((UINT16) (offsetof(MarshalData_st, TPMS_SCHEME_HASH_DATA)))
317 #define TPMS_SCHEME_HMAC_MARSHAL_REF \
318 TPMS_SCHEME_HASH_MARSHAL_REF
319 #define TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF \
320 TPMS_SCHEME_HASH_MARSHAL_REF
321 #define TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF \
322 TPMS_SCHEME_HASH_MARSHAL_REF
323 #define TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF \
324 TPMS_SCHEME_HASH_MARSHAL_REF
325 #define TPMS_SIG_SCHEME_SM2_MARSHAL_REF \
326 TPMS_SCHEME_HASH_MARSHAL_REF
327 #define TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF \
328 TPMS_SCHEME_HASH_MARSHAL_REF
329 #define TPMS_ENC_SCHEME_OAEP_MARSHAL_REF \
330 TPMS_SCHEME_HASH_MARSHAL_REF
331 #define TPMS_KEY_SCHEME_ECDH_MARSHAL_REF \
332 TPMS_SCHEME_HASH_MARSHAL_REF
333 #define TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF \
334 TPMS_SCHEME_HASH_MARSHAL_REF
335 #define TPMS_KDF_SCHEME_MGF1_MARSHAL_REF \
336 TPMS_SCHEME_HASH_MARSHAL_REF

```

```

327 #define TPMS_KDF_SCHEME_KDF1_SP800_56A_MARSHAL_REF TPMS_SCHEME_HASH_MARSHAL_REF
328 #define TPMS_KDF_SCHEME_KDF2_MARSHAL_REF TPMS_SCHEME_HASH_MARSHAL_REF
329 #define TPMS_KDF_SCHEME_KDF1_SP800_108_MARSHAL_REF TPMS_SCHEME_HASH_MARSHAL_REF
330 #define TPMS_SCHEME_ECDSA_MARSHAL_REF \
331     ((UINT16) (offsetof(MarshalData_st, TPMS_SCHEME_ECDSA_DATA)))
332 #define TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF TPMS_SCHEME_ECDSA_MARSHAL_REF
333 #define TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF \
334     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_KEYEDHASH_SCHEME_DATA)))
335 #define TPMS_SCHEME_XOR_MARSHAL_REF \
336     ((UINT16) (offsetof(MarshalData_st, TPMS_SCHEME_XOR_DATA)))
337 #define TPMU_SCHEME_KEYEDHASH_MARSHAL_REF \
338     ((UINT16) (offsetof(MarshalData_st, TPMU_SCHEME_KEYEDHASH_DATA)))
339 #define TPMT_KEYEDHASH_SCHEME_MARSHAL_REF \
340     ((UINT16) (offsetof(MarshalData_st, TPMT_KEYEDHASH_SCHEME_DATA)))
341 #define TPMU_SIG_SCHEME_MARSHAL_REF \
342     ((UINT16) (offsetof(MarshalData_st, TPMU_SIG_SCHEME_DATA)))
343 #define TPMT_SIG_SCHEME_MARSHAL_REF \
344     ((UINT16) (offsetof(MarshalData_st, TPMT_SIG_SCHEME_DATA)))
345 #define TPMU_KDF_SCHEME_MARSHAL_REF \
346     ((UINT16) (offsetof(MarshalData_st, TPMU_KDF_SCHEME_DATA)))
347 #define TPMT_KDF_SCHEME_MARSHAL_REF \
348     ((UINT16) (offsetof(MarshalData_st, TPMT_KDF_SCHEME_DATA)))
349 #define TPMI_ALG_ASYM_SCHEME_MARSHAL_REF \
350     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_ASYM_SCHEME_DATA)))
351 #define TPMU_ASYM_SCHEME_MARSHAL_REF \
352     ((UINT16) (offsetof(MarshalData_st, TPMU_ASYM_SCHEME_DATA)))
353 #define TPMI_ALG_RSA_SCHEME_MARSHAL_REF \
354     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_RSA_SCHEME_DATA)))
355 #define TPMT_RSA_SCHEME_MARSHAL_REF \
356     ((UINT16) (offsetof(MarshalData_st, TPMT_RSA_SCHEME_DATA)))
357 #define TPMI_ALG_RSA_DECRYPT_MARSHAL_REF \
358     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_RSA_DECRYPT_DATA)))
359 #define TPMT_RSA_DECRYPT_MARSHAL_REF \
360     ((UINT16) (offsetof(MarshalData_st, TPMT_RSA_DECRYPT_DATA)))
361 #define TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF \
362     ((UINT16) (offsetof(MarshalData_st, TPM2B_PUBLIC_KEY_RSA_DATA)))
363 #define TPMI_RSA_KEY_BITS_MARSHAL_REF \
364     ((UINT16) (offsetof(MarshalData_st, TPMI_RSA_KEY_BITS_DATA)))
365 #define TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF \
366     ((UINT16) (offsetof(MarshalData_st, TPM2B_PRIVATE_KEY_RSA_DATA)))
367 #define TPM2B_ECC_PARAMETER_MARSHAL_REF \
368     ((UINT16) (offsetof(MarshalData_st, TPM2B_ECC_PARAMETER_DATA)))
369 #define TPMS_ECC_POINT_MARSHAL_REF \
370     ((UINT16) (offsetof(MarshalData_st, TPMS_ECC_POINT_DATA)))
371 #define TPM2B_ECC_POINT_MARSHAL_REF \
372     ((UINT16) (offsetof(MarshalData_st, TPM2B_ECC_POINT_DATA)))
373 #define TPMI_ALG_ECC_SCHEME_MARSHAL_REF \
374     ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_ECC_SCHEME_DATA)))
375 #define TPMI_ECC_CURVE_MARSHAL_REF \
376     ((UINT16) (offsetof(MarshalData_st, TPMI_ECC_CURVE_DATA)))
377 #define TPMT_ECC_SCHEME_MARSHAL_REF \
378     ((UINT16) (offsetof(MarshalData_st, TPMT_ECC_SCHEME_DATA)))
379 #define TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_REF \
380     ((UINT16) (offsetof(MarshalData_st, TPMS_ALGORITHM_DETAIL_ECC_DATA)))
381 #define TPMS_SIGNATURE_RSA_MARSHAL_REF \
382     ((UINT16) (offsetof(MarshalData_st, TPMS_SIGNATURE_RSA_DATA)))
383 #define TPMS_SIGNATURE_RSASSA_MARSHAL_REF TPMS_SIGNATURE_RSA_MARSHAL_REF
384 #define TPMS_SIGNATURE_RSAPSS_MARSHAL_REF TPMS_SIGNATURE_RSA_MARSHAL_REF
385 #define TPMS_SIGNATURE_ECC_MARSHAL_REF \
386     ((UINT16) (offsetof(MarshalData_st, TPMS_SIGNATURE_ECC_DATA)))
387 #define TPMS_SIGNATURE_ECDSA_MARSHAL_REF TPMS_SIGNATURE_ECC_MARSHAL_REF
388 #define TPMS_SIGNATURE_ECDSA_MARSHAL_REF TPMS_SIGNATURE_ECC_MARSHAL_REF
389 #define TPMS_SIGNATURE_SM2_MARSHAL_REF TPMS_SIGNATURE_ECC_MARSHAL_REF
390 #define TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF TPMS_SIGNATURE_ECC_MARSHAL_REF
391 #define TPMU_SIGNATURE_MARSHAL_REF \
392     ((UINT16) (offsetof(MarshalData_st, TPMU_SIGNATURE_DATA)))

```

```

393 #define TPMT_SIGNATURE_MARSHAL_REF \
394 ((UINT16) (offsetof(MarshalData_st, TPMT_SIGNATURE_DATA)))
395 #define TPMU_ENCRYPTED_SECRET_MARSHAL_REF \
396 ((UINT16) (offsetof(MarshalData_st, TPMU_ENCRYPTED_SECRET_DATA)))
397 #define TPM2B_ENCRYPTED_SECRET_MARSHAL_REF \
398 ((UINT16) (offsetof(MarshalData_st, TPM2B_ENCRYPTED_SECRET_DATA)))
399 #define TPMI_ALG_PUBLIC_MARSHAL_REF \
400 ((UINT16) (offsetof(MarshalData_st, TPMI_ALG_PUBLIC_DATA)))
401 #define TPMU_PUBLIC_ID_MARSHAL_REF \
402 ((UINT16) (offsetof(MarshalData_st, TPMU_PUBLIC_ID_DATA)))
403 #define TPMS_KEYEDHASH_PARMS_MARSHAL_REF \
404 ((UINT16) (offsetof(MarshalData_st, TPMS_KEYEDHASH_PARMS_DATA)))
405 #define TPMS_RSA_PARMS_MARSHAL_REF \
406 ((UINT16) (offsetof(MarshalData_st, TPMS_RSA_PARMS_DATA)))
407 #define TPMS_ECC_PARMS_MARSHAL_REF \
408 ((UINT16) (offsetof(MarshalData_st, TPMS_ECC_PARMS_DATA)))
409 #define TPMU_PUBLIC_PARMS_MARSHAL_REF \
410 ((UINT16) (offsetof(MarshalData_st, TPMU_PUBLIC_PARMS_DATA)))
411 #define TPMT_PUBLIC_PARMS_MARSHAL_REF \
412 ((UINT16) (offsetof(MarshalData_st, TPMT_PUBLIC_PARMS_DATA)))
413 #define TPMT_PUBLIC_MARSHAL_REF \
414 ((UINT16) (offsetof(MarshalData_st, TPMT_PUBLIC_DATA)))
415 #define TPM2B_PUBLIC_MARSHAL_REF \
416 ((UINT16) (offsetof(MarshalData_st, TPM2B_PUBLIC_DATA)))
417 #define TPM2B_TEMPLATE_MARSHAL_REF \
418 ((UINT16) (offsetof(MarshalData_st, TPM2B_TEMPLATE_DATA)))
419 #define TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_REF \
420 ((UINT16) (offsetof(MarshalData_st, TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA)))
421 #define TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF \
422 ((UINT16) (offsetof(MarshalData_st, TPMU_SENSITIVE_COMPOSITE_DATA)))
423 #define TPMT_SENSITIVE_MARSHAL_REF \
424 ((UINT16) (offsetof(MarshalData_st, TPMT_SENSITIVE_DATA)))
425 #define TPM2B_SENSITIVE_MARSHAL_REF \
426 ((UINT16) (offsetof(MarshalData_st, TPM2B_SENSITIVE_DATA)))
427 #define TPM2B_PRIVATE_MARSHAL_REF \
428 ((UINT16) (offsetof(MarshalData_st, TPM2B_PRIVATE_DATA)))
429 #define TPM2B_ID_OBJECT_MARSHAL_REF \
430 ((UINT16) (offsetof(MarshalData_st, TPM2B_ID_OBJECT_DATA)))
431 #define TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_REF \
432 ((UINT16) (offsetof(MarshalData_st, TPMS_NV_PIN_COUNTER_PARAMETERS_DATA)))
433 #define TPMA_NV_MARSHAL_REF \
434 ((UINT16) (offsetof(MarshalData_st, TPMA_NV_DATA)))
435 #define TPMS_NV_PUBLIC_MARSHAL_REF \
436 ((UINT16) (offsetof(MarshalData_st, TPMS_NV_PUBLIC_DATA)))
437 #define TPM2B_NV_PUBLIC_MARSHAL_REF \
438 ((UINT16) (offsetof(MarshalData_st, TPM2B_NV_PUBLIC_DATA)))
439 #define TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF \
440 ((UINT16) (offsetof(MarshalData_st, TPM2B_CONTEXT_SENSITIVE_DATA)))
441 #define TPMS_CONTEXT_DATA_MARSHAL_REF \
442 ((UINT16) (offsetof(MarshalData_st, TPMS_CONTEXT_DATA_DATA)))
443 #define TPM2B_CONTEXT_DATA_MARSHAL_REF \
444 ((UINT16) (offsetof(MarshalData_st, TPM2B_CONTEXT_DATA_DATA)))
445 #define TPMS_CONTEXT_MARSHAL_REF \
446 ((UINT16) (offsetof(MarshalData_st, TPMS_CONTEXT_DATA)))
447 #define TPMS_CREATION_DATA_MARSHAL_REF \
448 ((UINT16) (offsetof(MarshalData_st, TPMS_CREATION_DATA_DATA)))
449 #define TPM2B_CREATION_DATA_MARSHAL_REF \
450 ((UINT16) (offsetof(MarshalData_st, TPM2B_CREATION_DATA_DATA)))
451 #define TPM_AT_MARSHAL_REF \
452 ((UINT16) (offsetof(MarshalData_st, TPM_AT_DATA)))
453 #define TPMS_AC_OUTPUT_MARSHAL_REF \
454 ((UINT16) (offsetof(MarshalData_st, TPMS_AC_OUTPUT_DATA)))
455 #define TPML_AC_CAPABILITIES_MARSHAL_REF \
456 ((UINT16) (offsetof(MarshalData_st, TPML_AC_CAPABILITIES_DATA)))
457 #define Type00_MARSHAL_REF \
458 ((UINT16) (offsetof(MarshalData_st, Type00_DATA)))

```

```

459 #define Type01_MARSHAL_REF \
460     ((UINT16) (offsetof(MarshalData_st, Type01_DATA)))
461 #define Type02_MARSHAL_REF \
462     ((UINT16) (offsetof(MarshalData_st, Type02_DATA)))
463 #define Type03_MARSHAL_REF \
464     ((UINT16) (offsetof(MarshalData_st, Type03_DATA)))
465 #define Type04_MARSHAL_REF \
466     ((UINT16) (offsetof(MarshalData_st, Type04_DATA)))
467 #define Type05_MARSHAL_REF \
468     ((UINT16) (offsetof(MarshalData_st, Type05_DATA)))
469 #define Type06_MARSHAL_REF \
470     ((UINT16) (offsetof(MarshalData_st, Type06_DATA)))
471 #define Type07_MARSHAL_REF \
472     ((UINT16) (offsetof(MarshalData_st, Type07_DATA)))
473 #define Type08_MARSHAL_REF \
474     ((UINT16) (offsetof(MarshalData_st, Type08_DATA)))
475 #define Type09_MARSHAL_REF Type08_MARSHAL_REF
476 #define Type14_MARSHAL_REF Type08_MARSHAL_REF
477 #define Type10_MARSHAL_REF \
478     ((UINT16) (offsetof(MarshalData_st, Type10_DATA)))
479 #define Type11_MARSHAL_REF \
480     ((UINT16) (offsetof(MarshalData_st, Type11_DATA)))
481 #define Type12_MARSHAL_REF \
482     ((UINT16) (offsetof(MarshalData_st, Type12_DATA)))
483 #define Type13_MARSHAL_REF \
484     ((UINT16) (offsetof(MarshalData_st, Type13_DATA)))
485 #define Type15_MARSHAL_REF \
486     ((UINT16) (offsetof(MarshalData_st, Type15_DATA)))
487 #define Type16_MARSHAL_REF Type15_MARSHAL_REF
488 #define Type17_MARSHAL_REF \
489     ((UINT16) (offsetof(MarshalData_st, Type17_DATA)))
490 #define Type18_MARSHAL_REF \
491     ((UINT16) (offsetof(MarshalData_st, Type18_DATA)))
492 #define Type19_MARSHAL_REF \
493     ((UINT16) (offsetof(MarshalData_st, Type19_DATA)))
494 #define Type20_MARSHAL_REF \
495     ((UINT16) (offsetof(MarshalData_st, Type20_DATA)))
496 #define Type21_MARSHAL_REF Type20_MARSHAL_REF
497 #define Type22_MARSHAL_REF \
498     ((UINT16) (offsetof(MarshalData_st, Type22_DATA)))
499 #define Type23_MARSHAL_REF \
500     ((UINT16) (offsetof(MarshalData_st, Type23_DATA)))
501 #define Type24_MARSHAL_REF \
502     ((UINT16) (offsetof(MarshalData_st, Type24_DATA)))
503 #define Type25_MARSHAL_REF \
504     ((UINT16) (offsetof(MarshalData_st, Type25_DATA)))
505 #define Type26_MARSHAL_REF \
506     ((UINT16) (offsetof(MarshalData_st, Type26_DATA)))
507 #define Type27_MARSHAL_REF \
508     ((UINT16) (offsetof(MarshalData_st, Type27_DATA)))
509 #define Type28_MARSHAL_REF \
510     ((UINT16) (offsetof(MarshalData_st, Type28_DATA)))
511 #define Type29_MARSHAL_REF \
512     ((UINT16) (offsetof(MarshalData_st, Type29_DATA)))
513 #define Type30_MARSHAL_REF \
514     ((UINT16) (offsetof(MarshalData_st, Type30_DATA)))
515 #define Type31_MARSHAL_REF \
516     ((UINT16) (offsetof(MarshalData_st, Type31_DATA)))
517 #define Type32_MARSHAL_REF \
518     ((UINT16) (offsetof(MarshalData_st, Type32_DATA)))
519 #define Type33_MARSHAL_REF \
520     ((UINT16) (offsetof(MarshalData_st, Type33_DATA)))
521 #define Type34_MARSHAL_REF \
522     ((UINT16) (offsetof(MarshalData_st, Type34_DATA)))
523 #define Type35_MARSHAL_REF \
524     ((UINT16) (offsetof(MarshalData_st, Type35_DATA)))

```



```

525 #define Type36_MARSHAL_REF \
526     ((UINT16) (offsetof(MarshalData_st, Type36_DATA)))
527 #define Type37_MARSHAL_REF \
528     ((UINT16) (offsetof(MarshalData_st, Type37_DATA)))
529 #define Type38_MARSHAL_REF \
530     ((UINT16) (offsetof(MarshalData_st, Type38_DATA)))
531 #define Type39_MARSHAL_REF \
532     ((UINT16) (offsetof(MarshalData_st, Type39_DATA)))
533 #define Type40_MARSHAL_REF \
534     ((UINT16) (offsetof(MarshalData_st, Type40_DATA)))
535 #define Type41_MARSHAL_REF \
536     ((UINT16) (offsetof(MarshalData_st, Type41_DATA)))
537 #define Type42_MARSHAL_REF \
538     ((UINT16) (offsetof(MarshalData_st, Type42_DATA)))
539 #define Type43_MARSHAL_REF \
540     ((UINT16) (offsetof(MarshalData_st, Type43_DATA)))
541 #define Type44_MARSHAL_REF \
542     ((UINT16) (offsetof(MarshalData_st, Type44_DATA)))
543
544 // #defines to change calling sequence for code using marshaling
545 #define UINT8_Unmarshal(target, buffer, size) \
546     Unmarshal(UINT8_MARSHAL_REF, (target), (buffer), (size))
547 #define UINT8_Marshal(source, buffer, size) \
548     Marshal(UINT8_MARSHAL_REF, (source), (buffer), (size))
549 #define BYTE_Unmarshal(target, buffer, size) \
550     Unmarshal(UINT8_MARSHAL_REF, (target), (buffer), (size))
551 #define BYTE_Marshal(source, buffer, size) \
552     Marshal(UINT8_MARSHAL_REF, (source), (buffer), (size))
553 #define INT8_Unmarshal(target, buffer, size) \
554     Unmarshal(INT8_MARSHAL_REF, (target), (buffer), (size))
555 #define INT8_Marshal(source, buffer, size) \
556     Marshal(INT8_MARSHAL_REF, (source), (buffer), (size))
557 #define UINT16_Unmarshal(target, buffer, size) \
558     Unmarshal(UINT16_MARSHAL_REF, (target), (buffer), (size))
559 #define UINT16_Marshal(source, buffer, size) \
560     Marshal(UINT16_MARSHAL_REF, (source), (buffer), (size))
561 #define INT16_Unmarshal(target, buffer, size) \
562     Unmarshal(INT16_MARSHAL_REF, (target), (buffer), (size))
563 #define INT16_Marshal(source, buffer, size) \
564     Marshal(INT16_MARSHAL_REF, (source), (buffer), (size))
565 #define UINT32_Unmarshal(target, buffer, size) \
566     Unmarshal(UINT32_MARSHAL_REF, (target), (buffer), (size))
567 #define UINT32_Marshal(source, buffer, size) \
568     Marshal(UINT32_MARSHAL_REF, (source), (buffer), (size))
569 #define INT32_Unmarshal(target, buffer, size) \
570     Unmarshal(INT32_MARSHAL_REF, (target), (buffer), (size))
571 #define INT32_Marshal(source, buffer, size) \
572     Marshal(INT32_MARSHAL_REF, (source), (buffer), (size))
573 #define UINT64_Unmarshal(target, buffer, size) \
574     Unmarshal(UINT64_MARSHAL_REF, (target), (buffer), (size))
575 #define UINT64_Marshal(source, buffer, size) \
576     Marshal(UINT64_MARSHAL_REF, (source), (buffer), (size))
577 #define INT64_Unmarshal(target, buffer, size) \
578     Unmarshal(INT64_MARSHAL_REF, (target), (buffer), (size))
579 #define INT64_Marshal(source, buffer, size) \
580     Marshal(INT64_MARSHAL_REF, (source), (buffer), (size))
581 #define TPM_ALGORITHM_ID_Unmarshal(target, buffer, size) \
582     Unmarshal(TPM_ALGORITHM_ID_MARSHAL_REF, (target), (buffer), (size))
583 #define TPM_ALGORITHM_ID_Marshal(source, buffer, size) \
584     Marshal(TPM_ALGORITHM_ID_MARSHAL_REF, (source), (buffer), (size))
585 #define TPM_MODIFIER_INDICATOR_Unmarshal(target, buffer, size) \
586     Unmarshal(TPM_MODIFIER_INDICATOR_MARSHAL_REF, (target), (buffer), (size))
587 #define TPM_MODIFIER_INDICATOR_Marshal(source, buffer, size) \
588     Marshal(TPM_MODIFIER_INDICATOR_MARSHAL_REF, (source), (buffer), (size))
589 #define TPM_AUTHORIZATION_SIZE_Unmarshal(target, buffer, size) \
590     Unmarshal(TPM_AUTHORIZATION_SIZE_MARSHAL_REF, (target), (buffer), (size))

```

```
591 #define TPM_AUTHORIZATION_SIZE_Marshal(source, buffer, size) \
592     Marshal(TPM_AUTHORIZATION_SIZE_MARSHAL_REF, (source), (buffer), (size))
593 #define TPM_PARAMETER_SIZE_Unmarshal(target, buffer, size) \
594     Unmarshal(TPM_PARAMETER_SIZE_MARSHAL_REF, (target), (buffer), (size))
595 #define TPM_PARAMETER_SIZE_Marshal(source, buffer, size) \
596     Marshal(TPM_PARAMETER_SIZE_MARSHAL_REF, (source), (buffer), (size))
597 #define TPM_KEY_SIZE_Unmarshal(target, buffer, size) \
598     Unmarshal(TPM_KEY_SIZE_MARSHAL_REF, (target), (buffer), (size))
599 #define TPM_KEY_SIZE_Marshal(source, buffer, size) \
600     Marshal(TPM_KEY_SIZE_MARSHAL_REF, (source), (buffer), (size))
601 #define TPM_KEY_BITS_Unmarshal(target, buffer, size) \
602     Unmarshal(TPM_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
603 #define TPM_KEY_BITS_Marshal(source, buffer, size) \
604     Marshal(TPM_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
605 #define TPM_CONSTANTS32_Marshal(source, buffer, size) \
606     Marshal(TPM_CONSTANTS32_MARSHAL_REF, (source), (buffer), (size))
607 #define TPM_ALG_ID_Unmarshal(target, buffer, size) \
608     Unmarshal(TPM_ALG_ID_MARSHAL_REF, (target), (buffer), (size))
609 #define TPM_ALG_ID_Marshal(source, buffer, size) \
610     Marshal(TPM_ALG_ID_MARSHAL_REF, (source), (buffer), (size))
611 #define TPM_ECC_CURVE_Unmarshal(target, buffer, size) \
612     Unmarshal(TPM_ECC_CURVE_MARSHAL_REF, (target), (buffer), (size))
613 #define TPM_ECC_CURVE_Marshal(source, buffer, size) \
614     Marshal(TPM_ECC_CURVE_MARSHAL_REF, (source), (buffer), (size))
615 #define TPM_CC_Unmarshal(target, buffer, size) \
616     Unmarshal(TPM_CC_MARSHAL_REF, (target), (buffer), (size))
617 #define TPM_CC_Marshal(source, buffer, size) \
618     Marshal(TPM_CC_MARSHAL_REF, (source), (buffer), (size))
619 #define TPM_RC_Marshal(source, buffer, size) \
620     Marshal(TPM_RC_MARSHAL_REF, (source), (buffer), (size))
621 #define TPM_CLOCK_ADJUST_Unmarshal(target, buffer, size) \
622     Unmarshal(TPM_CLOCK_ADJUST_MARSHAL_REF, (target), (buffer), (size))
623 #define TPM_EO_Unmarshal(target, buffer, size) \
624     Unmarshal(TPM_EO_MARSHAL_REF, (target), (buffer), (size))
625 #define TPM_EO_Marshal(source, buffer, size) \
626     Marshal(TPM_EO_MARSHAL_REF, (source), (buffer), (size))
627 #define TPM_ST_Unmarshal(target, buffer, size) \
628     Unmarshal(TPM_ST_MARSHAL_REF, (target), (buffer), (size))
629 #define TPM_ST_Marshal(source, buffer, size) \
630     Marshal(TPM_ST_MARSHAL_REF, (source), (buffer), (size))
631 #define TPM_SU_Unmarshal(target, buffer, size) \
632     Unmarshal(TPM_SU_MARSHAL_REF, (target), (buffer), (size))
633 #define TPM_SE_Unmarshal(target, buffer, size) \
634     Unmarshal(TPM_SE_MARSHAL_REF, (target), (buffer), (size))
635 #define TPM_CAP_Unmarshal(target, buffer, size) \
636     Unmarshal(TPM_CAP_MARSHAL_REF, (target), (buffer), (size))
637 #define TPM_CAP_Marshal(source, buffer, size) \
638     Marshal(TPM_CAP_MARSHAL_REF, (source), (buffer), (size))
639 #define TPM_PT_Unmarshal(target, buffer, size) \
640     Unmarshal(TPM_PT_MARSHAL_REF, (target), (buffer), (size))
641 #define TPM_PT_Marshal(source, buffer, size) \
642     Marshal(TPM_PT_MARSHAL_REF, (source), (buffer), (size))
643 #define TPM_PT_PCR_Unmarshal(target, buffer, size) \
644     Unmarshal(TPM_PT_PCR_MARSHAL_REF, (target), (buffer), (size))
645 #define TPM_PT_PCR_Marshal(source, buffer, size) \
646     Marshal(TPM_PT_PCR_MARSHAL_REF, (source), (buffer), (size))
647 #define TPM_PS_Marshal(source, buffer, size) \
648     Marshal(TPM_PS_MARSHAL_REF, (source), (buffer), (size))
649 #define TPM_HANDLE_Unmarshal(target, buffer, size) \
650     Unmarshal(TPM_HANDLE_MARSHAL_REF, (target), (buffer), (size))
651 #define TPM_HANDLE_Marshal(source, buffer, size) \
652     Marshal(TPM_HANDLE_MARSHAL_REF, (source), (buffer), (size))
653 #define TPM_HT_Unmarshal(target, buffer, size) \
654     Unmarshal(TPM_HT_MARSHAL_REF, (target), (buffer), (size))
655 #define TPM_HT_Marshal(source, buffer, size) \
656     Marshal(TPM_HT_MARSHAL_REF, (source), (buffer), (size))
```

```

657 #define TPM_RH_Unmarshal(target, buffer, size) \
658     Unmarshal(TPM_RH_MARSHAL_REF, (target), (buffer), (size))
659 #define TPM_RH_Marshal(source, buffer, size) \
660     Marshal(TPM_RH_MARSHAL_REF, (source), (buffer), (size))
661 #define TPM_HC_Unmarshal(target, buffer, size) \
662     Unmarshal(TPM_HC_MARSHAL_REF, (target), (buffer), (size))
663 #define TPM_HC_Marshal(source, buffer, size) \
664     Marshal(TPM_HC_MARSHAL_REF, (source), (buffer), (size))
665 #define TPMA_ALGORITHM_Unmarshal(target, buffer, size) \
666     Unmarshal(TPMA_ALGORITHM_MARSHAL_REF, (target), (buffer), (size))
667 #define TPMA_ALGORITHM_Marshal(source, buffer, size) \
668     Marshal(TPMA_ALGORITHM_MARSHAL_REF, (source), (buffer), (size))
669 #define TPMA_OBJECT_Unmarshal(target, buffer, size) \
670     Unmarshal(TPMA_OBJECT_MARSHAL_REF, (target), (buffer), (size))
671 #define TPMA_OBJECT_Marshal(source, buffer, size) \
672     Marshal(TPMA_OBJECT_MARSHAL_REF, (source), (buffer), (size))
673 #define TPMA_SESSION_Unmarshal(target, buffer, size) \
674     Unmarshal(TPMA_SESSION_MARSHAL_REF, (target), (buffer), (size))
675 #define TPMA_SESSION_Marshal(source, buffer, size) \
676     Marshal(TPMA_SESSION_MARSHAL_REF, (source), (buffer), (size))
677 #define TPMA_LOCALITY_Unmarshal(target, buffer, size) \
678     Unmarshal(TPMA_LOCALITY_MARSHAL_REF, (target), (buffer), (size))
679 #define TPMA_LOCALITY_Marshal(source, buffer, size) \
680     Marshal(TPMA_LOCALITY_MARSHAL_REF, (source), (buffer), (size))
681 #define TPMA_PERMANENT_Marshal(source, buffer, size) \
682     Marshal(TPMA_PERMANENT_MARSHAL_REF, (source), (buffer), (size))
683 #define TPMA_STARTUP_CLEAR_Marshal(source, buffer, size) \
684     Marshal(TPMA_STARTUP_CLEAR_MARSHAL_REF, (source), (buffer), (size))
685 #define TPMA_MEMORY_Marshal(source, buffer, size) \
686     Marshal(TPMA_MEMORY_MARSHAL_REF, (source), (buffer), (size))
687 #define TPMA_CC_Marshal(source, buffer, size) \
688     Marshal(TPMA_CC_MARSHAL_REF, (source), (buffer), (size))
689 #define TPMA_MODES_Marshal(source, buffer, size) \
690     Marshal(TPMA_MODES_MARSHAL_REF, (source), (buffer), (size))
691 #define TPMA_X509_KEY_USAGE_Marshal(source, buffer, size) \
692     Marshal(TPMA_X509_KEY_USAGE_MARSHAL_REF, (source), (buffer), (size))
693 #define TPMA_ACT_Unmarshal(target, buffer, size) \
694     Unmarshal(TPMA_ACT_MARSHAL_REF, (target), (buffer), (size))
695 #define TPMA_ACT_Marshal(source, buffer, size) \
696     Marshal(TPMA_ACT_MARSHAL_REF, (source), (buffer), (size))
697 #define TPMI_YES_NO_Unmarshal(target, buffer, size) \
698     Unmarshal(TPMI_YES_NO_MARSHAL_REF, (target), (buffer), (size))
699 #define TPMI_YES_NO_Marshal(source, buffer, size) \
700     Marshal(TPMI_YES_NO_MARSHAL_REF, (source), (buffer), (size))
701 #define TPMI_DH_OBJECT_Unmarshal(target, buffer, size, flag) \
702     Unmarshal(TPMI_DH_OBJECT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
703     (size))
704 #define TPMI_DH_OBJECT_Marshal(source, buffer, size) \
705     Marshal(TPMI_DH_OBJECT_MARSHAL_REF, (source), (buffer), (size))
706 #define TPMI_DH_PARENT_Unmarshal(target, buffer, size, flag) \
707     Unmarshal(TPMI_DH_PARENT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
708     (size))
709 #define TPMI_DH_PARENT_Marshal(source, buffer, size) \
710     Marshal(TPMI_DH_PARENT_MARSHAL_REF, (source), (buffer), (size))
711 #define TPMI_DH_PERSISTENT_Unmarshal(target, buffer, size) \
712     Unmarshal(TPMI_DH_PERSISTENT_MARSHAL_REF, (target), (buffer), (size))
713 #define TPMI_DH_PERSISTENT_Marshal(source, buffer, size) \
714     Marshal(TPMI_DH_PERSISTENT_MARSHAL_REF, (source), (buffer), (size))
715 #define TPMI_DH_ENTITY_Unmarshal(target, buffer, size, flag) \
716     Unmarshal(TPMI_DH_ENTITY_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
717     (size))
718 #define TPMI_DH_PCR_Unmarshal(target, buffer, size, flag) \
719     Unmarshal(TPMI_DH_PCR_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
720     (size))
721 #define TPMI_SH_AUTH_SESSION_Unmarshal(target, buffer, size, flag) \
722     Unmarshal(TPMI_SH_AUTH_SESSION_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \

```



```

723     (buffer), (size))
724 #define TPMI_SH_AUTH_SESSION_Marshal(source, buffer, size) \
725     Marshal(TPMI_SH_AUTH_SESSION_MARSHAL_REF, (source), (buffer), (size))
726 #define TPMI_SH_HMAC_Unmarshal(target, buffer, size) \
727     Unmarshal(TPMI_SH_HMAC_MARSHAL_REF, (target), (buffer), (size))
728 #define TPMI_SH_HMAC_Marshal(source, buffer, size) \
729     Marshal(TPMI_SH_HMAC_MARSHAL_REF, (source), (buffer), (size))
730 #define TPMI_SH_POLICY_Unmarshal(target, buffer, size) \
731     Unmarshal(TPMI_SH_POLICY_MARSHAL_REF, (target), (buffer), (size))
732 #define TPMI_SH_POLICY_Marshal(source, buffer, size) \
733     Marshal(TPMI_SH_POLICY_MARSHAL_REF, (source), (buffer), (size))
734 #define TPMI_DH_CONTEXT_Unmarshal(target, buffer, size) \
735     Unmarshal(TPMI_DH_CONTEXT_MARSHAL_REF, (target), (buffer), (size))
736 #define TPMI_DH_CONTEXT_Marshal(source, buffer, size) \
737     Marshal(TPMI_DH_CONTEXT_MARSHAL_REF, (source), (buffer), (size))
738 #define TPMI_DH_SAVED_Unmarshal(target, buffer, size) \
739     Unmarshal(TPMI_DH_SAVED_MARSHAL_REF, (target), (buffer), (size))
740 #define TPMI_DH_SAVED_Marshal(source, buffer, size) \
741     Marshal(TPMI_DH_SAVED_MARSHAL_REF, (source), (buffer), (size))
742 #define TPMI_RH_HIERARCHY_Unmarshal(target, buffer, size, flag) \
743     Unmarshal(TPMI_RH_HIERARCHY_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
744     (buffer), (size))
745 #define TPMI_RH_HIERARCHY_Marshal(source, buffer, size) \
746     Marshal(TPMI_RH_HIERARCHY_MARSHAL_REF, (source), (buffer), (size))
747 #define TPMI_RH_ENABLES_Unmarshal(target, buffer, size, flag) \
748     Unmarshal(TPMI_RH_ENABLES_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
749     (buffer), (size))
750 #define TPMI_RH_ENABLES_Marshal(source, buffer, size) \
751     Marshal(TPMI_RH_ENABLES_MARSHAL_REF, (source), (buffer), (size))
752 #define TPMI_RH_HIERARCHY_AUTH_Unmarshal(target, buffer, size) \
753     Unmarshal(TPMI_RH_HIERARCHY_AUTH_MARSHAL_REF, (target), (buffer), (size))
754 #define TPMI_RH_HIERARCHY_POLICY_Unmarshal(target, buffer, size) \
755     Unmarshal(TPMI_RH_HIERARCHY_POLICY_MARSHAL_REF, (target), (buffer), (size))
756 #define TPMI_RH_PLATFORM_Unmarshal(target, buffer, size) \
757     Unmarshal(TPMI_RH_PLATFORM_MARSHAL_REF, (target), (buffer), (size))
758 #define TPMI_RH_OWNER_Unmarshal(target, buffer, size, flag) \
759     Unmarshal(TPMI_RH_OWNER_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
760     (size))
761 #define TPMI_RH_ENDORSEMENT_Unmarshal(target, buffer, size, flag) \
762     Unmarshal(TPMI_RH_ENDORSEMENT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
763     (buffer), (size))
764 #define TPMI_RH_PROVISION_Unmarshal(target, buffer, size) \
765     Unmarshal(TPMI_RH_PROVISION_MARSHAL_REF, (target), (buffer), (size))
766 #define TPMI_RH_CLEAR_Unmarshal(target, buffer, size) \
767     Unmarshal(TPMI_RH_CLEAR_MARSHAL_REF, (target), (buffer), (size))
768 #define TPMI_RH_NV_AUTH_Unmarshal(target, buffer, size) \
769     Unmarshal(TPMI_RH_NV_AUTH_MARSHAL_REF, (target), (buffer), (size))
770 #define TPMI_RH_LOCKOUT_Unmarshal(target, buffer, size) \
771     Unmarshal(TPMI_RH_LOCKOUT_MARSHAL_REF, (target), (buffer), (size))
772 #define TPMI_RH_NV_INDEX_Unmarshal(target, buffer, size) \
773     Unmarshal(TPMI_RH_NV_INDEX_MARSHAL_REF, (target), (buffer), (size))
774 #define TPMI_RH_NV_INDEX_Marshal(source, buffer, size) \
775     Marshal(TPMI_RH_NV_INDEX_MARSHAL_REF, (source), (buffer), (size))
776 #define TPMI_RH_AC_Unmarshal(target, buffer, size) \
777     Unmarshal(TPMI_RH_AC_MARSHAL_REF, (target), (buffer), (size))
778 #define TPMI_RH_ACT_Unmarshal(target, buffer, size) \
779     Unmarshal(TPMI_RH_ACT_MARSHAL_REF, (target), (buffer), (size))
780 #define TPMI_RH_ACT_Marshal(source, buffer, size) \
781     Marshal(TPMI_RH_ACT_MARSHAL_REF, (source), (buffer), (size))
782 #define TPMI_ALG_HASH_Unmarshal(target, buffer, size, flag) \
783     Unmarshal(TPMI_ALG_HASH_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
784     (size))
785 #define TPMI_ALG_HASH_Marshal(source, buffer, size) \
786     Marshal(TPMI_ALG_HASH_MARSHAL_REF, (source), (buffer), (size))
787 #define TPMI_ALG_ASYM_Unmarshal(target, buffer, size, flag) \
788     Unmarshal(TPMI_ALG_ASYM_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \

```

```

789     (size))
790 #define TPMI_ALG_ASYM_Marshal(source, buffer, size) \
791     Marshal(TPMI_ALG_ASYM_MARSHAL_REF, (source), (buffer), (size))
792 #define TPMI_ALG_SYM_Unmarshal(target, buffer, size, flag) \
793     Unmarshal(TPMI_ALG_SYM_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
794     (size))
795 #define TPMI_ALG_SYM_Marshal(source, buffer, size) \
796     Marshal(TPMI_ALG_SYM_MARSHAL_REF, (source), (buffer), (size))
797 #define TPMI_ALG_SYM_OBJECT_Unmarshal(target, buffer, size, flag) \
798     Unmarshal(TPMI_ALG_SYM_OBJECT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
799     (buffer), (size))
800 #define TPMI_ALG_SYM_OBJECT_Marshal(source, buffer, size) \
801     Marshal(TPMI_ALG_SYM_OBJECT_MARSHAL_REF, (source), (buffer), (size))
802 #define TPMI_ALG_SYM_MODE_Unmarshal(target, buffer, size, flag) \
803     Unmarshal(TPMI_ALG_SYM_MODE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
804     (buffer), (size))
805 #define TPMI_ALG_SYM_MODE_Marshal(source, buffer, size) \
806     Marshal(TPMI_ALG_SYM_MODE_MARSHAL_REF, (source), (buffer), (size))
807 #define TPMI_ALG_KDF_Unmarshal(target, buffer, size, flag) \
808     Unmarshal(TPMI_ALG_KDF_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
809     (size))
810 #define TPMI_ALG_KDF_Marshal(source, buffer, size) \
811     Marshal(TPMI_ALG_KDF_MARSHAL_REF, (source), (buffer), (size))
812 #define TPMI_ALG_SIG_SCHEME_Unmarshal(target, buffer, size, flag) \
813     Unmarshal(TPMI_ALG_SIG_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
814     (buffer), (size))
815 #define TPMI_ALG_SIG_SCHEME_Marshal(source, buffer, size) \
816     Marshal(TPMI_ALG_SIG_SCHEME_MARSHAL_REF, (source), (buffer), (size))
817 #define TPMI_ECC_KEY_EXCHANGE_Unmarshal(target, buffer, size, flag) \
818     Unmarshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
819     (buffer), (size))
820 #define TPMI_ECC_KEY_EXCHANGE_Marshal(source, buffer, size) \
821     Marshal(TPMI_ECC_KEY_EXCHANGE_MARSHAL_REF, (source), (buffer), (size))
822 #define TPMI_ST_COMMAND_TAG_Unmarshal(target, buffer, size) \
823     Unmarshal(TPMI_ST_COMMAND_TAG_MARSHAL_REF, (target), (buffer), (size))
824 #define TPMI_ST_COMMAND_TAG_Marshal(source, buffer, size) \
825     Marshal(TPMI_ST_COMMAND_TAG_MARSHAL_REF, (source), (buffer), (size))
826 #define TPMI_ALG_MAC_SCHEME_Unmarshal(target, buffer, size, flag) \
827     Unmarshal(TPMI_ALG_MAC_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
828     (buffer), (size))
829 #define TPMI_ALG_MAC_SCHEME_Marshal(source, buffer, size) \
830     Marshal(TPMI_ALG_MAC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
831 #define TPMI_ALG_CIPHER_MODE_Unmarshal(target, buffer, size, flag) \
832     Unmarshal(TPMI_ALG_CIPHER_MODE_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
833     (buffer), (size))
834 #define TPMI_ALG_CIPHER_MODE_Marshal(source, buffer, size) \
835     Marshal(TPMI_ALG_CIPHER_MODE_MARSHAL_REF, (source), (buffer), (size))
836 #define TPMS_EMPTY_Unmarshal(target, buffer, size) \
837     Unmarshal(TPMS_EMPTY_MARSHAL_REF, (target), (buffer), (size))
838 #define TPMS_EMPTY_Marshal(source, buffer, size) \
839     Marshal(TPMS_EMPTY_MARSHAL_REF, (source), (buffer), (size))
840 #define TPMS_ALGORITHM_DESCRIPTION_Marshal(source, buffer, size) \
841     Marshal(TPMS_ALGORITHM_DESCRIPTION_MARSHAL_REF, (source), (buffer), (size))
842 #define TPMU_HA_Unmarshal(target, buffer, size, selector) \
843     UnmarshalUnion(TPMU_HA_MARSHAL_REF, (target), (buffer), (size), (selector))
844 #define TPMU_HA_Marshal(source, buffer, size, selector) \
845     MarshalUnion(TPMU_HA_MARSHAL_REF, (source), (buffer), (size), (selector))
846 #define TPMT_HA_Unmarshal(target, buffer, size, flag) \
847     Unmarshal(TPMT_HA_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), (buffer), \
848     (size))
849 #define TPMT_HA_Marshal(source, buffer, size) \
850     Marshal(TPMT_HA_MARSHAL_REF, (source), (buffer), (size))
851 #define TPM2B_DIGEST_Unmarshal(target, buffer, size) \
852     Unmarshal(TPM2B_DIGEST_MARSHAL_REF, (target), (buffer), (size))
853 #define TPM2B_DIGEST_Marshal(source, buffer, size) \
854     Marshal(TPM2B_DIGEST_MARSHAL_REF, (source), (buffer), (size))

```

```

855 #define TPM2B_DATA_Unmarshal(target, buffer, size) \
856     Unmarshal(TPM2B_DATA_MARSHAL_REF, (target), (buffer), (size)) \
857 #define TPM2B_DATA_Marshal(source, buffer, size) \
858     Marshal(TPM2B_DATA_MARSHAL_REF, (source), (buffer), (size)) \
859 #define TPM2B_NONCE_Unmarshal(target, buffer, size) \
860     Unmarshal(TPM2B_NONCE_MARSHAL_REF, (target), (buffer), (size)) \
861 #define TPM2B_NONCE_Marshal(source, buffer, size) \
862     Marshal(TPM2B_NONCE_MARSHAL_REF, (source), (buffer), (size)) \
863 #define TPM2B_AUTH_Unmarshal(target, buffer, size) \
864     Unmarshal(TPM2B_AUTH_MARSHAL_REF, (target), (buffer), (size)) \
865 #define TPM2B_AUTH_Marshal(source, buffer, size) \
866     Marshal(TPM2B_AUTH_MARSHAL_REF, (source), (buffer), (size)) \
867 #define TPM2B_OPERAND_Unmarshal(target, buffer, size) \
868     Unmarshal(TPM2B_OPERAND_MARSHAL_REF, (target), (buffer), (size)) \
869 #define TPM2B_OPERAND_Marshal(source, buffer, size) \
870     Marshal(TPM2B_OPERAND_MARSHAL_REF, (source), (buffer), (size)) \
871 #define TPM2B_EVENT_Unmarshal(target, buffer, size) \
872     Unmarshal(TPM2B_EVENT_MARSHAL_REF, (target), (buffer), (size)) \
873 #define TPM2B_EVENT_Marshal(source, buffer, size) \
874     Marshal(TPM2B_EVENT_MARSHAL_REF, (source), (buffer), (size)) \
875 #define TPM2B_MAX_BUFFER_Unmarshal(target, buffer, size) \
876     Unmarshal(TPM2B_MAX_BUFFER_MARSHAL_REF, (target), (buffer), (size)) \
877 #define TPM2B_MAX_BUFFER_Marshal(source, buffer, size) \
878     Marshal(TPM2B_MAX_BUFFER_MARSHAL_REF, (source), (buffer), (size)) \
879 #define TPM2B_MAX_NV_BUFFER_Unmarshal(target, buffer, size) \
880     Unmarshal(TPM2B_MAX_NV_BUFFER_MARSHAL_REF, (target), (buffer), (size)) \
881 #define TPM2B_MAX_NV_BUFFER_Marshal(source, buffer, size) \
882     Marshal(TPM2B_MAX_NV_BUFFER_MARSHAL_REF, (source), (buffer), (size)) \
883 #define TPM2B_TIMEOUT_Unmarshal(target, buffer, size) \
884     Unmarshal(TPM2B_TIMEOUT_MARSHAL_REF, (target), (buffer), (size)) \
885 #define TPM2B_TIMEOUT_Marshal(source, buffer, size) \
886     Marshal(TPM2B_TIMEOUT_MARSHAL_REF, (source), (buffer), (size)) \
887 #define TPM2B_IV_Unmarshal(target, buffer, size) \
888     Unmarshal(TPM2B_IV_MARSHAL_REF, (target), (buffer), (size)) \
889 #define TPM2B_IV_Marshal(source, buffer, size) \
890     Marshal(TPM2B_IV_MARSHAL_REF, (source), (buffer), (size)) \
891 #define TPM2B_NAME_Unmarshal(target, buffer, size) \
892     Unmarshal(TPM2B_NAME_MARSHAL_REF, (target), (buffer), (size)) \
893 #define TPM2B_NAME_Marshal(source, buffer, size) \
894     Marshal(TPM2B_NAME_MARSHAL_REF, (source), (buffer), (size)) \
895 #define TPMS_PCR_SELECT_Unmarshal(target, buffer, size) \
896     Unmarshal(TPMS_PCR_SELECT_MARSHAL_REF, (target), (buffer), (size)) \
897 #define TPMS_PCR_SELECT_Marshal(source, buffer, size) \
898     Marshal(TPMS_PCR_SELECT_MARSHAL_REF, (source), (buffer), (size)) \
899 #define TPMS_PCR_SELECTION_Unmarshal(target, buffer, size) \
900     Unmarshal(TPMS_PCR_SELECTION_MARSHAL_REF, (target), (buffer), (size)) \
901 #define TPMS_PCR_SELECTION_Marshal(source, buffer, size) \
902     Marshal(TPMS_PCR_SELECTION_MARSHAL_REF, (source), (buffer), (size)) \
903 #define TPMT_TK_CREATION_Unmarshal(target, buffer, size) \
904     Unmarshal(TPMT_TK_CREATION_MARSHAL_REF, (target), (buffer), (size)) \
905 #define TPMT_TK_CREATION_Marshal(source, buffer, size) \
906     Marshal(TPMT_TK_CREATION_MARSHAL_REF, (source), (buffer), (size)) \
907 #define TPMT_TK_VERIFIED_Unmarshal(target, buffer, size) \
908     Unmarshal(TPMT_TK_VERIFIED_MARSHAL_REF, (target), (buffer), (size)) \
909 #define TPMT_TK_VERIFIED_Marshal(source, buffer, size) \
910     Marshal(TPMT_TK_VERIFIED_MARSHAL_REF, (source), (buffer), (size)) \
911 #define TPMT_TK_AUTH_Unmarshal(target, buffer, size) \
912     Unmarshal(TPMT_TK_AUTH_MARSHAL_REF, (target), (buffer), (size)) \
913 #define TPMT_TK_AUTH_Marshal(source, buffer, size) \
914     Marshal(TPMT_TK_AUTH_MARSHAL_REF, (source), (buffer), (size)) \
915 #define TPMT_TK_HASHCHECK_Unmarshal(target, buffer, size) \
916     Unmarshal(TPMT_TK_HASHCHECK_MARSHAL_REF, (target), (buffer), (size)) \
917 #define TPMT_TK_HASHCHECK_Marshal(source, buffer, size) \
918     Marshal(TPMT_TK_HASHCHECK_MARSHAL_REF, (source), (buffer), (size)) \
919 #define TPMS_ALG_PROPERTY_Marshal(source, buffer, size) \
920     Marshal(TPMS_ALG_PROPERTY_MARSHAL_REF, (source), (buffer), (size))

```



```

921 #define TPMS_TAGGED_PROPERTY_Marshal(source, buffer, size) \
922     Marshal(TPMS_TAGGED_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
923 #define TPMS_TAGGED_PCR_SELECT_Marshal(source, buffer, size) \
924     Marshal(TPMS_TAGGED_PCR_SELECT_MARSHAL_REF, (source), (buffer), (size))
925 #define TPMS_TAGGED_POLICY_Marshal(source, buffer, size) \
926     Marshal(TPMS_TAGGED_POLICY_MARSHAL_REF, (source), (buffer), (size))
927 #define TPMS_ACT_DATA_Marshal(source, buffer, size) \
928     Marshal(TPMS_ACT_DATA_MARSHAL_REF, (source), (buffer), (size))
929 #define TPML_CC_Unmarshal(target, buffer, size) \
930     Unmarshal(TPML_CC_MARSHAL_REF, (target), (buffer), (size))
931 #define TPML_CC_Marshal(source, buffer, size) \
932     Marshal(TPML_CC_MARSHAL_REF, (source), (buffer), (size))
933 #define TPML_CCA_Marshal(source, buffer, size) \
934     Marshal(TPML_CCA_MARSHAL_REF, (source), (buffer), (size))
935 #define TPML_ALG_Unmarshal(target, buffer, size) \
936     Unmarshal(TPML_ALG_MARSHAL_REF, (target), (buffer), (size))
937 #define TPML_ALG_Marshal(source, buffer, size) \
938     Marshal(TPML_ALG_MARSHAL_REF, (source), (buffer), (size))
939 #define TPML_HANDLE_Marshal(source, buffer, size) \
940     Marshal(TPML_HANDLE_MARSHAL_REF, (source), (buffer), (size))
941 #define TPML_DIGEST_Unmarshal(target, buffer, size) \
942     Unmarshal(TPML_DIGEST_MARSHAL_REF, (target), (buffer), (size))
943 #define TPML_DIGEST_Marshal(source, buffer, size) \
944     Marshal(TPML_DIGEST_MARSHAL_REF, (source), (buffer), (size))
945 #define TPML_DIGEST_VALUES_Unmarshal(target, buffer, size) \
946     Unmarshal(TPML_DIGEST_VALUES_MARSHAL_REF, (target), (buffer), (size))
947 #define TPML_DIGEST_VALUES_Marshal(source, buffer, size) \
948     Marshal(TPML_DIGEST_VALUES_MARSHAL_REF, (source), (buffer), (size))
949 #define TPML_PCR_SELECTION_Unmarshal(target, buffer, size) \
950     Unmarshal(TPML_PCR_SELECTION_MARSHAL_REF, (target), (buffer), (size))
951 #define TPML_PCR_SELECTION_Marshal(source, buffer, size) \
952     Marshal(TPML_PCR_SELECTION_MARSHAL_REF, (source), (buffer), (size))
953 #define TPML_ALG_PROPERTY_Marshal(source, buffer, size) \
954     Marshal(TPML_ALG_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
955 #define TPML_TAGGED_TPM_PROPERTY_Marshal(source, buffer, size) \
956     Marshal(TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
957 #define TPML_TAGGED_PCR_PROPERTY_Marshal(source, buffer, size) \
958     Marshal(TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF, (source), (buffer), (size))
959 #define TPML_ECC_CURVE_Marshal(source, buffer, size) \
960     Marshal(TPML_ECC_CURVE_MARSHAL_REF, (source), (buffer), (size))
961 #define TPML_TAGGED_POLICY_Marshal(source, buffer, size) \
962     Marshal(TPML_TAGGED_POLICY_MARSHAL_REF, (source), (buffer), (size))
963 #define TPML_ACT_DATA_Marshal(source, buffer, size) \
964     Marshal(TPML_ACT_DATA_MARSHAL_REF, (source), (buffer), (size))
965 #define TPMU_CAPABILITIES_Marshal(source, buffer, size, selector) \
966     MarshalUnion(TPMU_CAPABILITIES_MARSHAL_REF, (source), (buffer), (size), \
967         (selector))
968 #define TPMS_CAPABILITY_DATA_Marshal(source, buffer, size) \
969     Marshal(TPMS_CAPABILITY_DATA_MARSHAL_REF, (source), (buffer), (size))
970 #define TPMS_CLOCK_INFO_Unmarshal(target, buffer, size) \
971     Unmarshal(TPMS_CLOCK_INFO_MARSHAL_REF, (target), (buffer), (size))
972 #define TPMS_CLOCK_INFO_Marshal(source, buffer, size) \
973     Marshal(TPMS_CLOCK_INFO_MARSHAL_REF, (source), (buffer), (size))
974 #define TPMS_TIME_INFO_Unmarshal(target, buffer, size) \
975     Unmarshal(TPMS_TIME_INFO_MARSHAL_REF, (target), (buffer), (size))
976 #define TPMS_TIME_INFO_Marshal(source, buffer, size) \
977     Marshal(TPMS_TIME_INFO_MARSHAL_REF, (source), (buffer), (size))
978 #define TPMS_TIME_ATTEST_INFO_Marshal(source, buffer, size) \
979     Marshal(TPMS_TIME_ATTEST_INFO_MARSHAL_REF, (source), (buffer), (size))
980 #define TPMS_CERTIFY_INFO_Marshal(source, buffer, size) \
981     Marshal(TPMS_CERTIFY_INFO_MARSHAL_REF, (source), (buffer), (size))
982 #define TPMS_QUOTE_INFO_Marshal(source, buffer, size) \
983     Marshal(TPMS_QUOTE_INFO_MARSHAL_REF, (source), (buffer), (size))
984 #define TPMS_COMMAND_AUDIT_INFO_Marshal(source, buffer, size) \
985     Marshal(TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF, (source), (buffer), (size))
986 #define TPMS_SESSION_AUDIT_INFO_Marshal(source, buffer, size) \

```

```

987     Marshal(TPMS_SESSION_AUDIT_INFO_MARSHAL_REF, (source), (buffer), (size))
988 #define TPMS_CREATION_INFO_Marshal(source, buffer, size) \
989     Marshal(TPMS_CREATION_INFO_MARSHAL_REF, (source), (buffer), (size))
990 #define TPMS_NV_CERTIFY_INFO_Marshal(source, buffer, size) \
991     Marshal(TPMS_NV_CERTIFY_INFO_MARSHAL_REF, (source), (buffer), (size))
992 #define TPMS_NV_DIGEST_CERTIFY_INFO_Marshal(source, buffer, size) \
993     Marshal(TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF, (source), (buffer), (size))
994 #define TPMI_ST_ATTEST_Marshal(source, buffer, size) \
995     Marshal(TPMI_ST_ATTEST_MARSHAL_REF, (source), (buffer), (size))
996 #define TPMU_ATTEST_Marshal(source, buffer, size, selector) \
997     MarshalUnion(TPMU_ATTEST_MARSHAL_REF, (source), (buffer), (size), (selector))
998 #define TPMS_ATTEST_Marshal(source, buffer, size) \
999     Marshal(TPMS_ATTEST_MARSHAL_REF, (source), (buffer), (size))
1000 #define TPM2B_ATTEST_Marshal(source, buffer, size) \
1001     Marshal(TPM2B_ATTEST_MARSHAL_REF, (source), (buffer), (size))
1002 #define TPMS_AUTH_COMMAND_Unmarshal(target, buffer, size) \
1003     Unmarshal(TPMS_AUTH_COMMAND_MARSHAL_REF, (target), (buffer), (size))
1004 #define TPMS_AUTH_RESPONSE_Marshal(source, buffer, size) \
1005     Marshal(TPMS_AUTH_RESPONSE_MARSHAL_REF, (source), (buffer), (size))
1006 #define TPMI_TDES_KEY_BITS_Unmarshal(target, buffer, size) \
1007     Unmarshal(TPMI_TDES_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1008 #define TPMI_TDES_KEY_BITS_Marshal(source, buffer, size) \
1009     Marshal(TPMI_TDES_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1010 #define TPMI_AES_KEY_BITS_Unmarshal(target, buffer, size) \
1011     Unmarshal(TPMI_AES_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1012 #define TPMI_AES_KEY_BITS_Marshal(source, buffer, size) \
1013     Marshal(TPMI_AES_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1014 #define TPMI_SM4_KEY_BITS_Unmarshal(target, buffer, size) \
1015     Unmarshal(TPMI_SM4_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1016 #define TPMI_SM4_KEY_BITS_Marshal(source, buffer, size) \
1017     Marshal(TPMI_SM4_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1018 #define TPMI_CAMELLIA_KEY_BITS_Unmarshal(target, buffer, size) \
1019     Unmarshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1020 #define TPMI_CAMELLIA_KEY_BITS_Marshal(source, buffer, size) \
1021     Marshal(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1022 #define TPMU_SYM_KEY_BITS_Unmarshal(target, buffer, size, selector) \
1023     UnmarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_REF, (target), (buffer), (size), \
1024     (selector))
1025 #define TPMU_SYM_KEY_BITS_Marshal(source, buffer, size, selector) \
1026     MarshalUnion(TPMU_SYM_KEY_BITS_MARSHAL_REF, (source), (buffer), (size), \
1027     (selector))
1028 #define TPMU_SYM_MODE_Unmarshal(target, buffer, size, selector) \
1029     UnmarshalUnion(TPMU_SYM_MODE_MARSHAL_REF, (target), (buffer), (size), \
1030     (selector))
1031 #define TPMU_SYM_MODE_Marshal(source, buffer, size, selector) \
1032     MarshalUnion(TPMU_SYM_MODE_MARSHAL_REF, (source), (buffer), (size), (selector))
1033 #define TPMT_SYM_DEF_Unmarshal(target, buffer, size, flag) \
1034     Unmarshal(TPMT_SYM_DEF_MARSHAL_REF| (flag ? NULL_FLAG : 0), (target), (buffer), \
1035     (size))
1036 #define TPMT_SYM_DEF_Marshal(source, buffer, size) \
1037     Marshal(TPMT_SYM_DEF_MARSHAL_REF, (source), (buffer), (size))
1038 #define TPMT_SYM_DEF_OBJECT_Unmarshal(target, buffer, size, flag) \
1039     Unmarshal(TPMT_SYM_DEF_OBJECT_MARSHAL_REF| (flag ? NULL_FLAG : 0), (target), \
1040     (buffer), (size))
1041 #define TPMT_SYM_DEF_OBJECT_Marshal(source, buffer, size) \
1042     Marshal(TPMT_SYM_DEF_OBJECT_MARSHAL_REF, (source), (buffer), (size))
1043 #define TPM2B_SYM_KEY_Unmarshal(target, buffer, size) \
1044     Unmarshal(TPM2B_SYM_KEY_MARSHAL_REF, (target), (buffer), (size))
1045 #define TPM2B_SYM_KEY_Marshal(source, buffer, size) \
1046     Marshal(TPM2B_SYM_KEY_MARSHAL_REF, (source), (buffer), (size))
1047 #define TPMS_SYMCIPHER_PARMS_Unmarshal(target, buffer, size) \
1048     Unmarshal(TPMS_SYMCIPHER_PARMS_MARSHAL_REF, (target), (buffer), (size))
1049 #define TPMS_SYMCIPHER_PARMS_Marshal(source, buffer, size) \
1050     Marshal(TPMS_SYMCIPHER_PARMS_MARSHAL_REF, (source), (buffer), (size))
1051 #define TPM2B_LABEL_Unmarshal(target, buffer, size) \
1052     Unmarshal(TPM2B_LABEL_MARSHAL_REF, (target), (buffer), (size))

```

```
1053 #define TPM2B_LABEL_Marshal(source, buffer, size) \
1054     Marshal(TPM2B_LABEL_MARSHAL_REF, (source), (buffer), (size))
1055 #define TPMS_DERIVE_Unmarshal(target, buffer, size) \
1056     Unmarshal(TPMS_DERIVE_MARSHAL_REF, (target), (buffer), (size))
1057 #define TPMS_DERIVE_Marshal(source, buffer, size) \
1058     Marshal(TPMS_DERIVE_MARSHAL_REF, (source), (buffer), (size))
1059 #define TPM2B_DERIVE_Unmarshal(target, buffer, size) \
1060     Unmarshal(TPM2B_DERIVE_MARSHAL_REF, (target), (buffer), (size))
1061 #define TPM2B_DERIVE_Marshal(source, buffer, size) \
1062     Marshal(TPM2B_DERIVE_MARSHAL_REF, (source), (buffer), (size))
1063 #define TPM2B_SENSITIVE_DATA_Unmarshal(target, buffer, size) \
1064     Unmarshal(TPM2B_SENSITIVE_DATA_MARSHAL_REF, (target), (buffer), (size))
1065 #define TPM2B_SENSITIVE_DATA_Marshal(source, buffer, size) \
1066     Marshal(TPM2B_SENSITIVE_DATA_MARSHAL_REF, (source), (buffer), (size))
1067 #define TPMS_SENSITIVE_CREATE_Unmarshal(target, buffer, size) \
1068     Unmarshal(TPMS_SENSITIVE_CREATE_MARSHAL_REF, (target), (buffer), (size))
1069 #define TPM2B_SENSITIVE_CREATE_Unmarshal(target, buffer, size) \
1070     Unmarshal(TPM2B_SENSITIVE_CREATE_MARSHAL_REF, (target), (buffer), (size))
1071 #define TPMS_SCHEME_HASH_Unmarshal(target, buffer, size) \
1072     Unmarshal(TPMS_SCHEME_HASH_MARSHAL_REF, (target), (buffer), (size))
1073 #define TPMS_SCHEME_HASH_Marshal(source, buffer, size) \
1074     Marshal(TPMS_SCHEME_HASH_MARSHAL_REF, (source), (buffer), (size))
1075 #define TPMS_SCHEME_ECDSA_Unmarshal(target, buffer, size) \
1076     Unmarshal(TPMS_SCHEME_ECDSA_MARSHAL_REF, (target), (buffer), (size))
1077 #define TPMS_SCHEME_ECDSA_Marshal(source, buffer, size) \
1078     Marshal(TPMS_SCHEME_ECDSA_MARSHAL_REF, (source), (buffer), (size))
1079 #define TPMI_ALG_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag) \
1080     Unmarshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF| (flag ? NULL_FLAG : 0), \
1081     (target), (buffer), (size))
1082 #define TPMI_ALG_KEYEDHASH_SCHEME_Marshal(source, buffer, size) \
1083     Marshal(TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1084 #define TPMS_SCHEME_HMAC_Unmarshal(target, buffer, size) \
1085     Unmarshal(TPMS_SCHEME_HMAC_MARSHAL_REF, (target), (buffer), (size))
1086 #define TPMS_SCHEME_HMAC_Marshal(source, buffer, size) \
1087     Marshal(TPMS_SCHEME_HMAC_MARSHAL_REF, (source), (buffer), (size))
1088 #define TPMS_SCHEME_XOR_Unmarshal(target, buffer, size) \
1089     Unmarshal(TPMS_SCHEME_XOR_MARSHAL_REF, (target), (buffer), (size))
1090 #define TPMS_SCHEME_XOR_Marshal(source, buffer, size) \
1091     Marshal(TPMS_SCHEME_XOR_MARSHAL_REF, (source), (buffer), (size))
1092 #define TPMU_SCHEME_KEYEDHASH_Unmarshal(target, buffer, size, selector) \
1093     UnmarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_REF, (target), (buffer), (size), \
1094     (selector))
1095 #define TPMU_SCHEME_KEYEDHASH_Marshal(source, buffer, size, selector) \
1096     MarshalUnion(TPMU_SCHEME_KEYEDHASH_MARSHAL_REF, (source), (buffer), (size), \
1097     (selector))
1098 #define TPMT_KEYEDHASH_SCHEME_Unmarshal(target, buffer, size, flag) \
1099     Unmarshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_REF| (flag ? NULL_FLAG : 0), (target), \
1100     (buffer), (size))
1101 #define TPMT_KEYEDHASH_SCHEME_Marshal(source, buffer, size) \
1102     Marshal(TPMT_KEYEDHASH_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1103 #define TPMS_SIG_SCHEME_RSASSA_Unmarshal(target, buffer, size) \
1104     Unmarshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF, (target), (buffer), (size))
1105 #define TPMS_SIG_SCHEME_RSASSA_Marshal(source, buffer, size) \
1106     Marshal(TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF, (source), (buffer), (size))
1107 #define TPMS_SIG_SCHEME_RSAPSS_Unmarshal(target, buffer, size) \
1108     Unmarshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF, (target), (buffer), (size))
1109 #define TPMS_SIG_SCHEME_RSAPSS_Marshal(source, buffer, size) \
1110     Marshal(TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF, (source), (buffer), (size))
1111 #define TPMS_SIG_SCHEME_ECDSA_Unmarshal(target, buffer, size) \
1112     Unmarshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF, (target), (buffer), (size))
1113 #define TPMS_SIG_SCHEME_ECDSA_Marshal(source, buffer, size) \
1114     Marshal(TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF, (source), (buffer), (size))
1115 #define TPMS_SIG_SCHEME_SM2_Unmarshal(target, buffer, size) \
1116     Unmarshal(TPMS_SIG_SCHEME_SM2_MARSHAL_REF, (target), (buffer), (size))
1117 #define TPMS_SIG_SCHEME_SM2_Marshal(source, buffer, size) \
1118     Marshal(TPMS_SIG_SCHEME_SM2_MARSHAL_REF, (source), (buffer), (size))
```



```

1119 #define TPMS_SIG_SCHEME_EC Schnorr_Unmarshal(target, buffer, size) \
1120     Unmarshal(TPMS_SIG_SCHEME_EC Schnorr_Marshal_REF, (target), (buffer), (size)) \
1121 #define TPMS_SIG_SCHEME_EC Schnorr_Marshal(source, buffer, size) \
1122     Marshal(TPMS_SIG_SCHEME_EC Schnorr_Marshal_REF, (source), (buffer), (size)) \
1123 #define TPMS_SIG_SCHEME_ECDSA_Unmarshal(target, buffer, size) \
1124     Unmarshal(TPMS_SIG_SCHEME_ECDSA_Marshal_REF, (target), (buffer), (size)) \
1125 #define TPMS_SIG_SCHEME_ECDSA_Marshal(source, buffer, size) \
1126     Marshal(TPMS_SIG_SCHEME_ECDSA_Marshal_REF, (source), (buffer), (size)) \
1127 #define TPMU_SIG_SCHEME_Unmarshal(target, buffer, size, selector) \
1128     UnmarshalUnion(TPMU_SIG_SCHEME_Marshal_REF, (target), (buffer), (size), \
1129         (selector)) \
1130 #define TPMU_SIG_SCHEME_Marshal(source, buffer, size, selector) \
1131     MarshalUnion(TPMU_SIG_SCHEME_Marshal_REF, (source), (buffer), (size), \
1132         (selector)) \
1133 #define TPMT_SIG_SCHEME_Unmarshal(target, buffer, size, flag) \
1134     Unmarshal(TPMT_SIG_SCHEME_Marshal_REF | (flag ? NULL_FLAG : 0), (target), \
1135         (buffer), (size)) \
1136 #define TPMT_SIG_SCHEME_Marshal(source, buffer, size) \
1137     Marshal(TPMT_SIG_SCHEME_Marshal_REF, (source), (buffer), (size)) \
1138 #define TPMS_ENC_SCHEME_OAEP_Unmarshal(target, buffer, size) \
1139     Unmarshal(TPMS_ENC_SCHEME_OAEP_Marshal_REF, (target), (buffer), (size)) \
1140 #define TPMS_ENC_SCHEME_OAEP_Marshal(source, buffer, size) \
1141     Marshal(TPMS_ENC_SCHEME_OAEP_Marshal_REF, (source), (buffer), (size)) \
1142 #define TPMS_ENC_SCHEME_RSAES_Unmarshal(target, buffer, size) \
1143     Unmarshal(TPMS_ENC_SCHEME_RSAES_Marshal_REF, (target), (buffer), (size)) \
1144 #define TPMS_ENC_SCHEME_RSAES_Marshal(source, buffer, size) \
1145     Marshal(TPMS_ENC_SCHEME_RSAES_Marshal_REF, (source), (buffer), (size)) \
1146 #define TPMS_KEY_SCHEME_ECDH_Unmarshal(target, buffer, size) \
1147     Unmarshal(TPMS_KEY_SCHEME_ECDH_Marshal_REF, (target), (buffer), (size)) \
1148 #define TPMS_KEY_SCHEME_ECDH_Marshal(source, buffer, size) \
1149     Marshal(TPMS_KEY_SCHEME_ECDH_Marshal_REF, (source), (buffer), (size)) \
1150 #define TPMS_KEY_SCHEME_ECMQV_Unmarshal(target, buffer, size) \
1151     Unmarshal(TPMS_KEY_SCHEME_ECMQV_Marshal_REF, (target), (buffer), (size)) \
1152 #define TPMS_KEY_SCHEME_ECMQV_Marshal(source, buffer, size) \
1153     Marshal(TPMS_KEY_SCHEME_ECMQV_Marshal_REF, (source), (buffer), (size)) \
1154 #define TPMS_KDF_SCHEME_MGF1_Unmarshal(target, buffer, size) \
1155     Unmarshal(TPMS_KDF_SCHEME_MGF1_Marshal_REF, (target), (buffer), (size)) \
1156 #define TPMS_KDF_SCHEME_MGF1_Marshal(source, buffer, size) \
1157     Marshal(TPMS_KDF_SCHEME_MGF1_Marshal_REF, (source), (buffer), (size)) \
1158 #define TPMS_KDF_SCHEME_KDF1_SP800_56A_Unmarshal(target, buffer, size) \
1159     Unmarshal(TPMS_KDF_SCHEME_KDF1_SP800_56A_Marshal_REF, (target), (buffer), \
1160         (size)) \
1161 #define TPMS_KDF_SCHEME_KDF1_SP800_56A_Marshal(source, buffer, size) \
1162     Marshal(TPMS_KDF_SCHEME_KDF1_SP800_56A_Marshal_REF, (source), (buffer), (size)) \
1163 #define TPMS_KDF_SCHEME_KDF2_Unmarshal(target, buffer, size) \
1164     Unmarshal(TPMS_KDF_SCHEME_KDF2_Marshal_REF, (target), (buffer), (size)) \
1165 #define TPMS_KDF_SCHEME_KDF2_Marshal(source, buffer, size) \
1166     Marshal(TPMS_KDF_SCHEME_KDF2_Marshal_REF, (source), (buffer), (size)) \
1167 #define TPMS_KDF_SCHEME_KDF1_SP800_108_Unmarshal(target, buffer, size) \
1168     Unmarshal(TPMS_KDF_SCHEME_KDF1_SP800_108_Marshal_REF, (target), (buffer), \
1169         (size)) \
1170 #define TPMS_KDF_SCHEME_KDF1_SP800_108_Marshal(source, buffer, size) \
1171     Marshal(TPMS_KDF_SCHEME_KDF1_SP800_108_Marshal_REF, (source), (buffer), (size)) \
1172 #define TPMU_KDF_SCHEME_Unmarshal(target, buffer, size, selector) \
1173     UnmarshalUnion(TPMU_KDF_SCHEME_Marshal_REF, (target), (buffer), (size), \
1174         (selector)) \
1175 #define TPMU_KDF_SCHEME_Marshal(source, buffer, size, selector) \
1176     MarshalUnion(TPMU_KDF_SCHEME_Marshal_REF, (source), (buffer), (size), \
1177         (selector)) \
1178 #define TPMT_KDF_SCHEME_Unmarshal(target, buffer, size, flag) \
1179     Unmarshal(TPMT_KDF_SCHEME_Marshal_REF | (flag ? NULL_FLAG : 0), (target), \
1180         (buffer), (size)) \
1181 #define TPMT_KDF_SCHEME_Marshal(source, buffer, size) \
1182     Marshal(TPMT_KDF_SCHEME_Marshal_REF, (source), (buffer), (size)) \
1183 #define TPMI_ALG_ASYNC_SCHEME_Unmarshal(target, buffer, size, flag) \
1184     Unmarshal(TPMI_ALG_ASYNC_SCHEME_Marshal_REF | (flag ? NULL_FLAG : 0), (target), \

```



```

1185     (buffer), (size))
1186 #define TPMI_ALG_ASYNC_SCHEME_Marshal(source, buffer, size) \
1187     Marshal(TPMI_ALG_ASYNC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1188 #define TPMU_ASYNC_SCHEME_Unmarshal(target, buffer, size, selector) \
1189     UnmarshalUnion(TPMU_ASYNC_SCHEME_MARSHAL_REF, (target), (buffer), (size), \
1190     (selector))
1191 #define TPMU_ASYNC_SCHEME_Marshal(source, buffer, size, selector) \
1192     MarshalUnion(TPMU_ASYNC_SCHEME_MARSHAL_REF, (source), (buffer), (size), \
1193     (selector))
1194 #define TPMI_ALG_RSA_SCHEME_Unmarshal(target, buffer, size, flag) \
1195     Unmarshal(TPMI_ALG_RSA_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
1196     (buffer), (size))
1197 #define TPMI_ALG_RSA_SCHEME_Marshal(source, buffer, size) \
1198     Marshal(TPMI_ALG_RSA_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1199 #define TPMT_RSA_SCHEME_Unmarshal(target, buffer, size, flag) \
1200     Unmarshal(TPMT_RSA_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
1201     (buffer), (size))
1202 #define TPMT_RSA_SCHEME_Marshal(source, buffer, size) \
1203     Marshal(TPMT_RSA_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1204 #define TPMI_ALG_RSA_DECRYPT_Unmarshal(target, buffer, size, flag) \
1205     Unmarshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
1206     (buffer), (size))
1207 #define TPMI_ALG_RSA_DECRYPT_Marshal(source, buffer, size) \
1208     Marshal(TPMI_ALG_RSA_DECRYPT_MARSHAL_REF, (source), (buffer), (size))
1209 #define TPMT_RSA_DECRYPT_Unmarshal(target, buffer, size, flag) \
1210     Unmarshal(TPMT_RSA_DECRYPT_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
1211     (buffer), (size))
1212 #define TPMT_RSA_DECRYPT_Marshal(source, buffer, size) \
1213     Marshal(TPMT_RSA_DECRYPT_MARSHAL_REF, (source), (buffer), (size))
1214 #define TPM2B_PUBLIC_KEY_RSA_Unmarshal(target, buffer, size) \
1215     Unmarshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF, (target), (buffer), (size))
1216 #define TPM2B_PUBLIC_KEY_RSA_Marshal(source, buffer, size) \
1217     Marshal(TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF, (source), (buffer), (size))
1218 #define TPMI_RSA_KEY_BITS_Unmarshal(target, buffer, size) \
1219     Unmarshal(TPMI_RSA_KEY_BITS_MARSHAL_REF, (target), (buffer), (size))
1220 #define TPMI_RSA_KEY_BITS_Marshal(source, buffer, size) \
1221     Marshal(TPMI_RSA_KEY_BITS_MARSHAL_REF, (source), (buffer), (size))
1222 #define TPM2B_PRIVATE_KEY_RSA_Unmarshal(target, buffer, size) \
1223     Unmarshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF, (target), (buffer), (size))
1224 #define TPM2B_PRIVATE_KEY_RSA_Marshal(source, buffer, size) \
1225     Marshal(TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF, (source), (buffer), (size))
1226 #define TPM2B_ECC_PARAMETER_Unmarshal(target, buffer, size) \
1227     Unmarshal(TPM2B_ECC_PARAMETER_MARSHAL_REF, (target), (buffer), (size))
1228 #define TPM2B_ECC_PARAMETER_Marshal(source, buffer, size) \
1229     Marshal(TPM2B_ECC_PARAMETER_MARSHAL_REF, (source), (buffer), (size))
1230 #define TPMS_ECC_POINT_Unmarshal(target, buffer, size) \
1231     Unmarshal(TPMS_ECC_POINT_MARSHAL_REF, (target), (buffer), (size))
1232 #define TPMS_ECC_POINT_Marshal(source, buffer, size) \
1233     Marshal(TPMS_ECC_POINT_MARSHAL_REF, (source), (buffer), (size))
1234 #define TPM2B_ECC_POINT_Unmarshal(target, buffer, size) \
1235     Unmarshal(TPM2B_ECC_POINT_MARSHAL_REF, (target), (buffer), (size))
1236 #define TPM2B_ECC_POINT_Marshal(source, buffer, size) \
1237     Marshal(TPM2B_ECC_POINT_MARSHAL_REF, (source), (buffer), (size))
1238 #define TPMI_ALG_ECC_SCHEME_Unmarshal(target, buffer, size, flag) \
1239     Unmarshal(TPMI_ALG_ECC_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
1240     (buffer), (size))
1241 #define TPMI_ALG_ECC_SCHEME_Marshal(source, buffer, size) \
1242     Marshal(TPMI_ALG_ECC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1243 #define TPMI_ECC_CURVE_Unmarshal(target, buffer, size) \
1244     Unmarshal(TPMI_ECC_CURVE_MARSHAL_REF, (target), (buffer), (size))
1245 #define TPMI_ECC_CURVE_Marshal(source, buffer, size) \
1246     Marshal(TPMI_ECC_CURVE_MARSHAL_REF, (source), (buffer), (size))
1247 #define TPMT_ECC_SCHEME_Unmarshal(target, buffer, size, flag) \
1248     Unmarshal(TPMT_ECC_SCHEME_MARSHAL_REF|(flag ? NULL_FLAG : 0), (target), \
1249     (buffer), (size))
1250 #define TPMT_ECC_SCHEME_Marshal(source, buffer, size) \

```

```

1251     Marshal(TPMT_ECC_SCHEME_MARSHAL_REF, (source), (buffer), (size))
1252 #define TPMS_ALGORITHM_DETAIL_ECC_Marshal(source, buffer, size) \
1253     Marshal(TPMS_ALGORITHM_DETAIL_ECC_MARSHAL_REF, (source), (buffer), (size))
1254 #define TPMS_SIGNATURE_RSA_Unmarshal(target, buffer, size) \
1255     Unmarshal(TPMS_SIGNATURE_RSA_MARSHAL_REF, (target), (buffer), (size))
1256 #define TPMS_SIGNATURE_RSA_Marshal(source, buffer, size) \
1257     Marshal(TPMS_SIGNATURE_RSA_MARSHAL_REF, (source), (buffer), (size))
1258 #define TPMS_SIGNATURE_RSASSA_Unmarshal(target, buffer, size) \
1259     Unmarshal(TPMS_SIGNATURE_RSASSA_MARSHAL_REF, (target), (buffer), (size))
1260 #define TPMS_SIGNATURE_RSASSA_Marshal(source, buffer, size) \
1261     Marshal(TPMS_SIGNATURE_RSASSA_MARSHAL_REF, (source), (buffer), (size))
1262 #define TPMS_SIGNATURE_RSAPSS_Unmarshal(target, buffer, size) \
1263     Unmarshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_REF, (target), (buffer), (size))
1264 #define TPMS_SIGNATURE_RSAPSS_Marshal(source, buffer, size) \
1265     Marshal(TPMS_SIGNATURE_RSAPSS_MARSHAL_REF, (source), (buffer), (size))
1266 #define TPMS_SIGNATURE_ECC_Unmarshal(target, buffer, size) \
1267     Unmarshal(TPMS_SIGNATURE_ECC_MARSHAL_REF, (target), (buffer), (size))
1268 #define TPMS_SIGNATURE_ECC_Marshal(source, buffer, size) \
1269     Marshal(TPMS_SIGNATURE_ECC_MARSHAL_REF, (source), (buffer), (size))
1270 #define TPMS_SIGNATURE_ECDSA_Unmarshal(target, buffer, size) \
1271     Unmarshal(TPMS_SIGNATURE_ECDSA_MARSHAL_REF, (target), (buffer), (size))
1272 #define TPMS_SIGNATURE_ECDSA_Marshal(source, buffer, size) \
1273     Marshal(TPMS_SIGNATURE_ECDSA_MARSHAL_REF, (source), (buffer), (size))
1274 #define TPMS_SIGNATURE_SM2_Unmarshal(target, buffer, size) \
1275     Unmarshal(TPMS_SIGNATURE_SM2_MARSHAL_REF, (target), (buffer), (size))
1276 #define TPMS_SIGNATURE_SM2_Marshal(source, buffer, size) \
1277     Marshal(TPMS_SIGNATURE_SM2_MARSHAL_REF, (source), (buffer), (size))
1278 #define TPMS_SIGNATURE_EC Schnorr_Unmarshal(target, buffer, size) \
1279     Unmarshal(TPMS_SIGNATURE_EC Schnorr_MARSHAL_REF, (target), (buffer), (size))
1280 #define TPMS_SIGNATURE_EC Schnorr_Marshal(source, buffer, size) \
1281     Marshal(TPMS_SIGNATURE_EC Schnorr_MARSHAL_REF, (source), (buffer), (size))
1282 #define TPMS_SIGNATURE_EC Schnorr_Unmarshal(target, buffer, size) \
1283     Unmarshal(TPMS_SIGNATURE_EC Schnorr_MARSHAL_REF, (target), (buffer), (size))
1284 #define TPMS_SIGNATURE_EC Schnorr_Marshal(source, buffer, size) \
1285     Marshal(TPMS_SIGNATURE_EC Schnorr_MARSHAL_REF, (source), (buffer), (size))
1286 #define TPMU_SIGNATURE_Unmarshal(target, buffer, size, selector) \
1287     UnmarshalUnion(TPMU_SIGNATURE_MARSHAL_REF, (target), (buffer), (size), \
1288         (selector))
1289 #define TPMU_SIGNATURE_Marshal(source, buffer, size, selector) \
1290     MarshalUnion(TPMU_SIGNATURE_MARSHAL_REF, (source), (buffer), (size), (selector))
1291 #define TPMT_SIGNATURE_Unmarshal(target, buffer, size, flag) \
1292     Unmarshal(TPMT_SIGNATURE_MARSHAL_REF | (flag ? NULL_FLAG : 0), (target), (buffer), \
1293         (size))
1294 #define TPMT_SIGNATURE_Marshal(source, buffer, size) \
1295     Marshal(TPMT_SIGNATURE_MARSHAL_REF, (source), (buffer), (size))
1296 #define TPMU_ENCRYPTED_SECRET_Unmarshal(target, buffer, size, selector) \
1297     UnmarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_REF, (target), (buffer), (size), \
1298         (selector))
1299 #define TPMU_ENCRYPTED_SECRET_Marshal(source, buffer, size, selector) \
1300     MarshalUnion(TPMU_ENCRYPTED_SECRET_MARSHAL_REF, (source), (buffer), (size), \
1301         (selector))
1302 #define TPM2B_ENCRYPTED_SECRET_Unmarshal(target, buffer, size) \
1303     Unmarshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_REF, (target), (buffer), (size))
1304 #define TPM2B_ENCRYPTED_SECRET_Marshal(source, buffer, size) \
1305     Marshal(TPM2B_ENCRYPTED_SECRET_MARSHAL_REF, (source), (buffer), (size))
1306 #define TPMI_ALG_PUBLIC_Unmarshal(target, buffer, size) \
1307     Unmarshal(TPMI_ALG_PUBLIC_MARSHAL_REF, (target), (buffer), (size))
1308 #define TPMI_ALG_PUBLIC_Marshal(source, buffer, size) \
1309     Marshal(TPMI_ALG_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1310 #define TPMU_PUBLIC_ID_Unmarshal(target, buffer, size, selector) \
1311     UnmarshalUnion(TPMU_PUBLIC_ID_MARSHAL_REF, (target), (buffer), (size), \
1312         (selector))
1313 #define TPMU_PUBLIC_ID_Marshal(source, buffer, size, selector) \
1314     MarshalUnion(TPMU_PUBLIC_ID_MARSHAL_REF, (source), (buffer), (size), (selector))
1315 #define TPMS_KEYEDHASH_PARMS_Unmarshal(target, buffer, size) \
1316     Unmarshal(TPMS_KEYEDHASH_PARMS_MARSHAL_REF, (target), (buffer), (size))

```

```

1317 #define TPMS_KEYEDHASH_PARMS_Marshal(source, buffer, size) \
1318     Marshal(TPMS_KEYEDHASH_PARMS_MARSHAL_REF, (source), (buffer), (size))
1319 #define TPMS_RSA_PARMS_Unmarshal(target, buffer, size) \
1320     Unmarshal(TPMS_RSA_PARMS_MARSHAL_REF, (target), (buffer), (size))
1321 #define TPMS_RSA_PARMS_Marshal(source, buffer, size) \
1322     Marshal(TPMS_RSA_PARMS_MARSHAL_REF, (source), (buffer), (size))
1323 #define TPMS_ECC_PARMS_Unmarshal(target, buffer, size) \
1324     Unmarshal(TPMS_ECC_PARMS_MARSHAL_REF, (target), (buffer), (size))
1325 #define TPMS_ECC_PARMS_Marshal(source, buffer, size) \
1326     Marshal(TPMS_ECC_PARMS_MARSHAL_REF, (source), (buffer), (size))
1327 #define TPMU_PUBLIC_PARMS_Unmarshal(target, buffer, size, selector) \
1328     UnmarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_REF, (target), (buffer), (size), \
1329         (selector))
1330 #define TPMU_PUBLIC_PARMS_Marshal(source, buffer, size, selector) \
1331     MarshalUnion(TPMU_PUBLIC_PARMS_MARSHAL_REF, (source), (buffer), (size), \
1332         (selector))
1333 #define TPMT_PUBLIC_PARMS_Unmarshal(target, buffer, size) \
1334     Unmarshal(TPMT_PUBLIC_PARMS_MARSHAL_REF, (target), (buffer), (size))
1335 #define TPMT_PUBLIC_PARMS_Marshal(source, buffer, size) \
1336     Marshal(TPMT_PUBLIC_PARMS_MARSHAL_REF, (source), (buffer), (size))
1337 #define TPMT_PUBLIC_Unmarshal(target, buffer, size, flag) \
1338     Unmarshal(TPMT_PUBLIC_MARSHAL_REF | (flag ? NULL_FLAG : 0), (target), (buffer), \
1339         (size))
1340 #define TPMT_PUBLIC_Marshal(source, buffer, size) \
1341     Marshal(TPMT_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1342 #define TPM2B_PUBLIC_Unmarshal(target, buffer, size, flag) \
1343     Unmarshal(TPM2B_PUBLIC_MARSHAL_REF | (flag ? NULL_FLAG : 0), (target), (buffer), \
1344         (size))
1345 #define TPM2B_PUBLIC_Marshal(source, buffer, size) \
1346     Marshal(TPM2B_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1347 #define TPM2B_TEMPLATE_Unmarshal(target, buffer, size) \
1348     Unmarshal(TPM2B_TEMPLATE_MARSHAL_REF, (target), (buffer), (size))
1349 #define TPM2B_TEMPLATE_Marshal(source, buffer, size) \
1350     Marshal(TPM2B_TEMPLATE_MARSHAL_REF, (source), (buffer), (size))
1351 #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Unmarshal(target, buffer, size) \
1352     Unmarshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_REF, (target), (buffer), (size))
1353 #define TPM2B_PRIVATE_VENDOR_SPECIFIC_Marshal(source, buffer, size) \
1354     Marshal(TPM2B_PRIVATE_VENDOR_SPECIFIC_MARSHAL_REF, (source), (buffer), (size))
1355 #define TPMU_SENSITIVE_COMPOSITE_Unmarshal(target, buffer, size, selector) \
1356     UnmarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF, (target), (buffer), (size), \
1357         (selector))
1358 #define TPMU_SENSITIVE_COMPOSITE_Marshal(source, buffer, size, selector) \
1359     MarshalUnion(TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF, (source), (buffer), (size), \
1360         (selector))
1361 #define TPMT_SENSITIVE_Unmarshal(target, buffer, size) \
1362     Unmarshal(TPMT_SENSITIVE_MARSHAL_REF, (target), (buffer), (size))
1363 #define TPMT_SENSITIVE_Marshal(source, buffer, size) \
1364     Marshal(TPMT_SENSITIVE_MARSHAL_REF, (source), (buffer), (size))
1365 #define TPM2B_SENSITIVE_Unmarshal(target, buffer, size) \
1366     Unmarshal(TPM2B_SENSITIVE_MARSHAL_REF, (target), (buffer), (size))
1367 #define TPM2B_SENSITIVE_Marshal(source, buffer, size) \
1368     Marshal(TPM2B_SENSITIVE_MARSHAL_REF, (source), (buffer), (size))
1369 #define TPM2B_PRIVATE_Unmarshal(target, buffer, size) \
1370     Unmarshal(TPM2B_PRIVATE_MARSHAL_REF, (target), (buffer), (size))
1371 #define TPM2B_PRIVATE_Marshal(source, buffer, size) \
1372     Marshal(TPM2B_PRIVATE_MARSHAL_REF, (source), (buffer), (size))
1373 #define TPM2B_ID_OBJECT_Unmarshal(target, buffer, size) \
1374     Unmarshal(TPM2B_ID_OBJECT_MARSHAL_REF, (target), (buffer), (size))
1375 #define TPM2B_ID_OBJECT_Marshal(source, buffer, size) \
1376     Marshal(TPM2B_ID_OBJECT_MARSHAL_REF, (source), (buffer), (size))
1377 #define TPM_NV_INDEX_Marshal(source, buffer, size) \
1378     Marshal(TPM_NV_INDEX_MARSHAL_REF, (source), (buffer), (size))
1379 #define TPMS_NV_PIN_COUNTER_PARAMETERS_Unmarshal(target, buffer, size) \
1380     Unmarshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_REF, (target), (buffer), \
1381         (size))
1382 #define TPMS_NV_PIN_COUNTER_PARAMETERS_Marshal(source, buffer, size) \

```

```

1383     Marshal(TPMS_NV_PIN_COUNTER_PARAMETERS_MARSHAL_REF, (source), (buffer), (size))
1384 #define TPMA_NV_Unmarshal(target, buffer, size) \
1385     Unmarshal(TPMA_NV_MARSHAL_REF, (target), (buffer), (size))
1386 #define TPMA_NV_Marshal(source, buffer, size) \
1387     Marshal(TPMA_NV_MARSHAL_REF, (source), (buffer), (size))
1388 #define TPMS_NV_PUBLIC_Unmarshal(target, buffer, size) \
1389     Unmarshal(TPMS_NV_PUBLIC_MARSHAL_REF, (target), (buffer), (size))
1390 #define TPMS_NV_PUBLIC_Marshal(source, buffer, size) \
1391     Marshal(TPMS_NV_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1392 #define TPM2B_NV_PUBLIC_Unmarshal(target, buffer, size) \
1393     Unmarshal(TPM2B_NV_PUBLIC_MARSHAL_REF, (target), (buffer), (size))
1394 #define TPM2B_NV_PUBLIC_Marshal(source, buffer, size) \
1395     Marshal(TPM2B_NV_PUBLIC_MARSHAL_REF, (source), (buffer), (size))
1396 #define TPM2B_CONTEXT_SENSITIVE_Unmarshal(target, buffer, size) \
1397     Unmarshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF, (target), (buffer), (size))
1398 #define TPM2B_CONTEXT_SENSITIVE_Marshal(source, buffer, size) \
1399     Marshal(TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF, (source), (buffer), (size))
1400 #define TPMS_CONTEXT_DATA_Unmarshal(target, buffer, size) \
1401     Unmarshal(TPMS_CONTEXT_DATA_MARSHAL_REF, (target), (buffer), (size))
1402 #define TPMS_CONTEXT_DATA_Marshal(source, buffer, size) \
1403     Marshal(TPMS_CONTEXT_DATA_MARSHAL_REF, (source), (buffer), (size))
1404 #define TPM2B_CONTEXT_DATA_Unmarshal(target, buffer, size) \
1405     Unmarshal(TPM2B_CONTEXT_DATA_MARSHAL_REF, (target), (buffer), (size))
1406 #define TPM2B_CONTEXT_DATA_Marshal(source, buffer, size) \
1407     Marshal(TPM2B_CONTEXT_DATA_MARSHAL_REF, (source), (buffer), (size))
1408 #define TPMS_CONTEXT_Unmarshal(target, buffer, size) \
1409     Unmarshal(TPMS_CONTEXT_MARSHAL_REF, (target), (buffer), (size))
1410 #define TPMS_CONTEXT_Marshal(source, buffer, size) \
1411     Marshal(TPMS_CONTEXT_MARSHAL_REF, (source), (buffer), (size))
1412 #define TPMS_CREATION_DATA_Marshal(source, buffer, size) \
1413     Marshal(TPMS_CREATION_DATA_MARSHAL_REF, (source), (buffer), (size))
1414 #define TPM2B_CREATION_DATA_Marshal(source, buffer, size) \
1415     Marshal(TPM2B_CREATION_DATA_MARSHAL_REF, (source), (buffer), (size))
1416 #define TPM_AT_Unmarshal(target, buffer, size) \
1417     Unmarshal(TPM_AT_MARSHAL_REF, (target), (buffer), (size))
1418 #define TPM_AT_Marshal(source, buffer, size) \
1419     Marshal(TPM_AT_MARSHAL_REF, (source), (buffer), (size))
1420 #define TPM_AE_Marshal(source, buffer, size) \
1421     Marshal(TPM_AE_MARSHAL_REF, (source), (buffer), (size))
1422 #define TPMS_AC_OUTPUT_Marshal(source, buffer, size) \
1423     Marshal(TPMS_AC_OUTPUT_MARSHAL_REF, (source), (buffer), (size))
1424 #define TPML_AC_CAPABILITIES_Marshal(source, buffer, size) \
1425     Marshal(TPML_AC_CAPABILITIES_MARSHAL_REF, (source), (buffer), (size))
1426
1427 #endif // _TABLE_MARSHAL_DEFINES_H_

```

9.10.7.3 TableMarshalTypes.h

```

1 #ifndef _TABLE_MARSHAL_TYPES_H_
2 #define _TABLE_MARSHAL_TYPES_H_
3
4 typedef UINT16     marshalIndex_t;

```


9.10.7.3.1.1 Structure Entries

A structure contains a list of elements to unmarshal. Each of the entries is a UINT16. The structure descriptor is:

The *values* array contains indicators for the things to marshal. The *elements* parameter indicates how many different entities are unmarshaled. This number nominally corresponds to the number of rows in the Part 2 table that describes the structure (the number of rows minus the title row and any error code rows).

A schematic of a simple structure entry is shown here but the values are not actually in a structure. As shown, the third value is the offset in the structure where the value is placed when unmarshaled, or fetched from when marshaling. This is sufficient when the element type indicated by *index* is always a simple type and never a union or array. This is just shown for illustrative purposes.

```

5  typedef struct simpleStructureEntry_t {
6      UINT16          qualifiers;          // indicates the type of entry (array, union
7                                          // etc.)
8      marshalIndex_t  index;              // the index into the appropriate array of
9                                          // the descriptor of this type
10     UINT16          offset;              // where this comes from or is placed
11 } simpleStructureEntry_t;
12
13 typedef const struct UintMarshal_mst
14 {
15     UINT8          marshalType;          // UINT_MTYPE
16     UINT8          modifiers;            // size and signed indicator.
17 } UintMarshal_mst;
18
19 typedef struct UnionMarshal_mst
20 {
21     UINT8          countOfselectors;
22     UINT8          modifiers;            // NULL_SELECTOR
23     UINT16         offsetOfUnmarshalTypes;
24     UINT32         selectors[1];
25     //  UINT16      marshalingTypes[1]; // This is not part of the prototypical
26                                          // entry. It is here to show where the
27                                          // marshaling types will be in a union
28 } UnionMarshal_mst;
29
30 typedef struct NullUnionMarshal_mst
31 {
32     UINT8          count;
33 } NullUnionMarshal_mst;
34
35 typedef struct MarshalHeader_mst
36 {
37     UINT8          marshalType;          // VALUES_MTYPE
38     UINT8          modifiers;
39     UINT8          errorCode;
40 } MarshalHeader_mst;
41
42 typedef const struct ArrayMarshal_mst // used in a structure
43 {
44     marshalIndex_t  type;
45     UINT16          stride;
46 } ArrayMarshal_mst;
47
48 typedef const struct StructMarshal_mst
49 {
50     UINT8          marshalType;          // STRUCTURE_MTYPE
51     UINT8          elements;
52     UINT16         values[1];           // three times elements
53 } StructMarshal_mst;
54

```

```

55 typedef const struct ValuesMarshal_mst
56 {
57     UINT8        marshalType;           // VALUES_MTYPE
58     UINT8        modifiers;
59     UINT8        errorCode;
60     UINT8        ranges;
61     UINT8        singles;
62     UINT32       values[1];
63 } ValuesMarshal_mst;
64
65 typedef const struct TableMarshal_mst
66 {
67     UINT8        marshalType;           // TABLE_MTYPE
68     UINT8        modifiers;
69     UINT8        errorCode;
70     UINT8        singles;
71     UINT32       values[1];
72 } TableMarshal_mst;
73
74 typedef const struct MinMaxMarshal_mst
75 {
76     UINT8        marshalType;           // MIN_MAX_MTYPE
77     UINT8        modifiers;
78     UINT8        errorCode;
79     UINT32       values[2];
80 } MinMaxMarshal_mst;
81
82 typedef const struct Tpm2bMarshal_mst
83 {
84     UINT8        unmarshalType;         // TPM2B_MTYPE
85     UINT16       sizeIndex;             // reference to type for this size value
86 } Tpm2bMarshal_mst;
87
88 typedef const struct Tpm2bsMarshal_mst
89 {
90     UINT8        unmarshalType;         // TPM2BS_MTYPE
91     UINT8        modifiers;             // size= and offset (2 - 7)
92     UINT16       sizeIndex;             // index of the size value;
93     UINT16       dataIndex;            // the structure
94 } Tpm2bsMarshal_mst;
95
96 typedef const struct ListMarshal_mst
97 {
98     UINT8        unmarshalType;         // LIST_MTYPE (for TPML)
99     UINT8        modifiers;             // size offset 2-7
100    UINT16       sizeIndex;             // reference to the minmax structure that
101                                         // unmarshals the size parameter
102    UINT16       arrayRef;              // reference to an array definition (type
103                                         // and stride)
104 } ListMarshal_mst;
105
106 typedef const struct AttributesMarshal_mst
107 {
108     UINT8        unmarshashType;        // ATTRIBUTE_MTYPE
109     UINT8        modifiers;             // size (ONE_BYTES, TWO_BYTES, or FOUR_BYTES
110     UINT32       attributeMask;         // the values that must be zero.
111 } AttributesMarshal_mst;
112
113 typedef const struct CompositeMarshal_mst
114 {
115     UINT8        unmarshashType;        // COMPOSITE_MTYPE
116     UINT8        modifiers;             // number of entries and size
117     marshalIndex_t types[1];           // array of unmarshaling types
118 } CompositeMarshal_mst;
119
120 typedef const struct TPM_ECC_CURVE_mst {

```

```
121     UINT8      marshalType;
122     UINT8      modifiers;
123     UINT8      errorCode;
124     UINT32     values[4];
125 } TPM_ECC_CURVE_mst;
126
127 typedef const struct TPM_CLOCK_ADJUST_mst {
128     UINT8      marshalType;
129     UINT8      modifiers;
130     UINT8      errorCode;
131     UINT32     values[2];
132 } TPM_CLOCK_ADJUST_mst;
133
134 typedef const struct TPM_EO_mst {
135     UINT8      marshalType;
136     UINT8      modifiers;
137     UINT8      errorCode;
138     UINT32     values[2];
139 } TPM_EO_mst;
140
141 typedef const struct TPM_SU_mst {
142     UINT8      marshalType;
143     UINT8      modifiers;
144     UINT8      errorCode;
145     UINT8      entries;
146     UINT32     values[2];
147 } TPM_SU_mst;
148
149 typedef const struct TPM_SE_mst {
150     UINT8      marshalType;
151     UINT8      modifiers;
152     UINT8      errorCode;
153     UINT8      entries;
154     UINT32     values[3];
155 } TPM_SE_mst;
156
157 typedef const struct TPM_CAP_mst {
158     UINT8      marshalType;
159     UINT8      modifiers;
160     UINT8      errorCode;
161     UINT8      ranges;
162     UINT8      singles;
163     UINT32     values[3];
164 } TPM_CAP_mst;
165
166 typedef const struct TPMT_YES_NO_mst {
167     UINT8      marshalType;
168     UINT8      modifiers;
169     UINT8      errorCode;
170     UINT8      entries;
171     UINT32     values[2];
172 } TPMT_YES_NO_mst;
173
174 typedef const struct TPMT_DH_OBJECT_mst {
175     UINT8      marshalType;
176     UINT8      modifiers;
177     UINT8      errorCode;
178     UINT8      ranges;
179     UINT8      singles;
180     UINT32     values[5];
181 } TPMT_DH_OBJECT_mst;
182
183 typedef const struct TPMT_DH_PARENT_mst {
184     UINT8      marshalType;
185     UINT8      modifiers;
186     UINT8      errorCode;
```



```

187     UINT8         ranges;
188     UINT8         singles;
189     UINT32        values[8];
190 } TPMI_DH_PARENT_mst;
191
192 typedef const struct TPMI_DH_PERSISTENT_mst {
193     UINT8         marshalType;
194     UINT8         modifiers;
195     UINT8         errorCode;
196     UINT32        values[2];
197 } TPMI_DH_PERSISTENT_mst;
198
199 typedef const struct TPMI_DH_ENTITY_mst {
200     UINT8         marshalType;
201     UINT8         modifiers;
202     UINT8         errorCode;
203     UINT8         ranges;
204     UINT8         singles;
205     UINT32        values[15];
206 } TPMI_DH_ENTITY_mst;
207
208 typedef const struct TPMI_DH_PCR_mst {
209     UINT8         marshalType;
210     UINT8         modifiers;
211     UINT8         errorCode;
212     UINT32        values[3];
213 } TPMI_DH_PCR_mst;
214
215 typedef const struct TPMI_SH_AUTH_SESSION_mst {
216     UINT8         marshalType;
217     UINT8         modifiers;
218     UINT8         errorCode;
219     UINT8         ranges;
220     UINT8         singles;
221     UINT32        values[5];
222 } TPMI_SH_AUTH_SESSION_mst;
223
224 typedef const struct TPMI_SH_HMAC_mst {
225     UINT8         marshalType;
226     UINT8         modifiers;
227     UINT8         errorCode;
228     UINT32        values[2];
229 } TPMI_SH_HMAC_mst;
230
231 typedef const struct TPMI_SH_POLICY_mst {
232     UINT8         marshalType;
233     UINT8         modifiers;
234     UINT8         errorCode;
235     UINT32        values[2];
236 } TPMI_SH_POLICY_mst;
237
238 typedef const struct TPMI_DH_CONTEXT_mst {
239     UINT8         marshalType;
240     UINT8         modifiers;
241     UINT8         errorCode;
242     UINT8         ranges;
243     UINT8         singles;
244     UINT32        values[6];
245 } TPMI_DH_CONTEXT_mst;
246
247 typedef const struct TPMI_DH_SAVED_mst {
248     UINT8         marshalType;
249     UINT8         modifiers;
250     UINT8         errorCode;
251     UINT8         ranges;
252     UINT8         singles;

```

```
253     UINT32         values[7];
254 } TPMI_DH_SAVED_mst;
255
256 typedef const struct TPMI_RH_HIERARCHY_mst {
257     UINT8           marshalType;
258     UINT8           modifiers;
259     UINT8           errorCode;
260     UINT8           entries;
261     UINT32          values[4];
262 } TPMI_RH_HIERARCHY_mst;
263
264 typedef const struct TPMI_RH_ENABLES_mst {
265     UINT8           marshalType;
266     UINT8           modifiers;
267     UINT8           errorCode;
268     UINT8           entries;
269     UINT32          values[5];
270 } TPMI_RH_ENABLES_mst;
271
272 typedef const struct TPMI_RH_HIERARCHY_AUTH_mst {
273     UINT8           marshalType;
274     UINT8           modifiers;
275     UINT8           errorCode;
276     UINT8           entries;
277     UINT32          values[4];
278 } TPMI_RH_HIERARCHY_AUTH_mst;
279
280 typedef const struct TPMI_RH_HIERARCHY_POLICY_mst {
281     UINT8           marshalType;
282     UINT8           modifiers;
283     UINT8           errorCode;
284     UINT8           ranges;
285     UINT8           singles;
286     UINT32          values[6];
287 } TPMI_RH_HIERARCHY_POLICY_mst;
288
289 typedef const struct TPMI_RH_PLATFORM_mst {
290     UINT8           marshalType;
291     UINT8           modifiers;
292     UINT8           errorCode;
293     UINT8           entries;
294     UINT32          values[1];
295 } TPMI_RH_PLATFORM_mst;
296
297 typedef const struct TPMI_RH_OWNER_mst {
298     UINT8           marshalType;
299     UINT8           modifiers;
300     UINT8           errorCode;
301     UINT8           entries;
302     UINT32          values[2];
303 } TPMI_RH_OWNER_mst;
304
305 typedef const struct TPMI_RH_ENDORSEMENT_mst {
306     UINT8           marshalType;
307     UINT8           modifiers;
308     UINT8           errorCode;
309     UINT8           entries;
310     UINT32          values[2];
311 } TPMI_RH_ENDORSEMENT_mst;
312
313 typedef const struct TPMI_RH_PROVISION_mst {
314     UINT8           marshalType;
315     UINT8           modifiers;
316     UINT8           errorCode;
317     UINT8           entries;
318     UINT32          values[2];
```

```
319 } TPMI_RH_PROVISION_mst;
320
321 typedef const struct TPMI_RH_CLEAR_mst {
322     UINT8      marshalType;
323     UINT8      modifiers;
324     UINT8      errorCode;
325     UINT8      entries;
326     UINT32     values[2];
327 } TPMI_RH_CLEAR_mst;
328
329 typedef const struct TPMI_RH_NV_AUTH_mst {
330     UINT8      marshalType;
331     UINT8      modifiers;
332     UINT8      errorCode;
333     UINT8      ranges;
334     UINT8      singles;
335     UINT32     values[4];
336 } TPMI_RH_NV_AUTH_mst;
337
338 typedef const struct TPMI_RH_LOCKOUT_mst {
339     UINT8      marshalType;
340     UINT8      modifiers;
341     UINT8      errorCode;
342     UINT8      entries;
343     UINT32     values[1];
344 } TPMI_RH_LOCKOUT_mst;
345
346 typedef const struct TPMI_RH_NV_INDEX_mst {
347     UINT8      marshalType;
348     UINT8      modifiers;
349     UINT8      errorCode;
350     UINT32     values[2];
351 } TPMI_RH_NV_INDEX_mst;
352
353 typedef const struct TPMI_RH_AC_mst {
354     UINT8      marshalType;
355     UINT8      modifiers;
356     UINT8      errorCode;
357     UINT32     values[2];
358 } TPMI_RH_AC_mst;
359
360 typedef const struct TPMI_RH_ACT_mst {
361     UINT8      marshalType;
362     UINT8      modifiers;
363     UINT8      errorCode;
364     UINT32     values[2];
365 } TPMI_RH_ACT_mst;
366
367 typedef const struct TPMI_ALG_HASH_mst {
368     UINT8      marshalType;
369     UINT8      modifiers;
370     UINT8      errorCode;
371     UINT32     values[5];
372 } TPMI_ALG_HASH_mst;
373
374 typedef const struct TPMI_ALG_ASYM_mst {
375     UINT8      marshalType;
376     UINT8      modifiers;
377     UINT8      errorCode;
378     UINT32     values[5];
379 } TPMI_ALG_ASYM_mst;
380
381 typedef const struct TPMI_ALG_SYM_mst {
382     UINT8      marshalType;
383     UINT8      modifiers;
384     UINT8      errorCode;
```

```

385     UINT32     values[5];
386 } TPMI_ALG_SYM_mst;
387
388 typedef const struct TPMI_ALG_SYM_OBJECT_mst {
389     UINT8     marshalType;
390     UINT8     modifiers;
391     UINT8     errorCode;
392     UINT32     values[5];
393 } TPMI_ALG_SYM_OBJECT_mst;
394
395 typedef const struct TPMI_ALG_SYM_MODE_mst {
396     UINT8     marshalType;
397     UINT8     modifiers;
398     UINT8     errorCode;
399     UINT32     values[4];
400 } TPMI_ALG_SYM_MODE_mst;
401
402 typedef const struct TPMI_ALG_KDF_mst {
403     UINT8     marshalType;
404     UINT8     modifiers;
405     UINT8     errorCode;
406     UINT32     values[4];
407 } TPMI_ALG_KDF_mst;
408
409 typedef const struct TPMI_ALG_SIG_SCHEME_mst {
410     UINT8     marshalType;
411     UINT8     modifiers;
412     UINT8     errorCode;
413     UINT32     values[4];
414 } TPMI_ALG_SIG_SCHEME_mst;
415
416 typedef const struct TPMI_ECC_KEY_EXCHANGE_mst {
417     UINT8     marshalType;
418     UINT8     modifiers;
419     UINT8     errorCode;
420     UINT32     values[4];
421 } TPMI_ECC_KEY_EXCHANGE_mst;
422
423 typedef const struct TPMI_ST_COMMAND_TAG_mst {
424     UINT8     marshalType;
425     UINT8     modifiers;
426     UINT8     errorCode;
427     UINT8     entries;
428     UINT32     values[2];
429 } TPMI_ST_COMMAND_TAG_mst;
430
431 typedef const struct TPMI_ALG_MAC_SCHEME_mst {
432     UINT8     marshalType;
433     UINT8     modifiers;
434     UINT8     errorCode;
435     UINT32     values[5];
436 } TPMI_ALG_MAC_SCHEME_mst;
437
438 typedef const struct TPMI_ALG_CIPHER_MODE_mst {
439     UINT8     marshalType;
440     UINT8     modifiers;
441     UINT8     errorCode;
442     UINT32     values[4];
443 } TPMI_ALG_CIPHER_MODE_mst;
444
445 typedef const struct TPMS_EMPTY_mst
446 {
447     UINT8     marshalType;
448     UINT8     elements;
449     UINT16     values[3];
450 } TPMS_EMPTY_mst;

```

```

451
452 typedef const struct TPMS_ALGORITHM_DESCRIPTION_mst
453 {
454     UINT8    marshalType;
455     UINT8    elements;
456     UINT16   values[6];
457 } TPMS_ALGORITHM_DESCRIPTION_mst;
458
459 typedef struct TPMU_HA_mst
460 {
461     BYTE      countOfselectors;
462     BYTE      modifiers;
463     UINT16    offsetOfUnmarshalTypes;
464     UINT32    selectors[9];
465     UINT16    marshalingTypes[9];
466 } TPMU_HA_mst;
467
468 typedef const struct TPMT_HA_mst
469 {
470     UINT8    marshalType;
471     UINT8    elements;
472     UINT16   values[6];
473 } TPMT_HA_mst;
474
475 typedef const struct TPMS_PCR_SELECT_mst
476 {
477     UINT8    marshalType;
478     UINT8    elements;
479     UINT16   values[6];
480 } TPMS_PCR_SELECT_mst;
481
482 typedef const struct TPMS_PCR_SELECTION_mst
483 {
484     UINT8    marshalType;
485     UINT8    elements;
486     UINT16   values[9];
487 } TPMS_PCR_SELECTION_mst;
488
489 typedef const struct TPMT_TK_CREATION_mst
490 {
491     UINT8    marshalType;
492     UINT8    elements;
493     UINT16   values[9];
494 } TPMT_TK_CREATION_mst;
495
496 typedef const struct TPMT_TK_VERIFIED_mst
497 {
498     UINT8    marshalType;
499     UINT8    elements;
500     UINT16   values[9];
501 } TPMT_TK_VERIFIED_mst;
502
503 typedef const struct TPMT_TK_AUTH_mst
504 {
505     UINT8    marshalType;
506     UINT8    elements;
507     UINT16   values[9];
508 } TPMT_TK_AUTH_mst;
509
510 typedef const struct TPMT_TK_HASHCHECK_mst
511 {
512     UINT8    marshalType;
513     UINT8    elements;
514     UINT16   values[9];
515 } TPMT_TK_HASHCHECK_mst;
516

```

```

517 typedef const struct TPMS_ALG_PROPERTY_mst
518 {
519     UINT8    marshalType;
520     UINT8    elements;
521     UINT16   values[6];
522 } TPMS_ALG_PROPERTY_mst;
523
524 typedef const struct TPMS_TAGGED_PROPERTY_mst
525 {
526     UINT8    marshalType;
527     UINT8    elements;
528     UINT16   values[6];
529 } TPMS_TAGGED_PROPERTY_mst;
530
531 typedef const struct TPMS_TAGGED_PCR_SELECT_mst
532 {
533     UINT8    marshalType;
534     UINT8    elements;
535     UINT16   values[9];
536 } TPMS_TAGGED_PCR_SELECT_mst;
537
538 typedef const struct TPMS_TAGGED_POLICY_mst
539 {
540     UINT8    marshalType;
541     UINT8    elements;
542     UINT16   values[6];
543 } TPMS_TAGGED_POLICY_mst;
544
545 typedef const struct TPMS_ACT_DATA_mst
546 {
547     UINT8    marshalType;
548     UINT8    elements;
549     UINT16   values[9];
550 } TPMS_ACT_DATA_mst;
551
552 typedef struct TPMU_CAPABILITIES_mst
553 {
554     BYTE      countOfselectors;
555     BYTE      modifiers;
556     UINT16    offsetOfUnmarshalTypes;
557     UINT32    selectors[11];
558     UINT16    marshalingTypes[11];
559 } TPMU_CAPABILITIES_mst;
560
561 typedef const struct TPMS_CAPABILITY_DATA_mst
562 {
563     UINT8    marshalType;
564     UINT8    elements;
565     UINT16   values[6];
566 } TPMS_CAPABILITY_DATA_mst;
567
568 typedef const struct TPMS_CLOCK_INFO_mst
569 {
570     UINT8    marshalType;
571     UINT8    elements;
572     UINT16   values[12];
573 } TPMS_CLOCK_INFO_mst;
574
575 typedef const struct TPMS_TIME_INFO_mst
576 {
577     UINT8    marshalType;
578     UINT8    elements;
579     UINT16   values[6];
580 } TPMS_TIME_INFO_mst;
581
582 typedef const struct TPMS_TIME_ATTEST_INFO_mst

```

```

583 {
584     UINT8    marshalType;
585     UINT8    elements;
586     UINT16   values[6];
587 } TPMS_TIME_ATTEST_INFO_mst;
588
589 typedef const struct TPMS_CERTIFY_INFO_mst
590 {
591     UINT8    marshalType;
592     UINT8    elements;
593     UINT16   values[6];
594 } TPMS_CERTIFY_INFO_mst;
595
596 typedef const struct TPMS_QUOTE_INFO_mst
597 {
598     UINT8    marshalType;
599     UINT8    elements;
600     UINT16   values[6];
601 } TPMS_QUOTE_INFO_mst;
602
603 typedef const struct TPMS_COMMAND_AUDIT_INFO_mst
604 {
605     UINT8    marshalType;
606     UINT8    elements;
607     UINT16   values[12];
608 } TPMS_COMMAND_AUDIT_INFO_mst;
609
610 typedef const struct TPMS_SESSION_AUDIT_INFO_mst
611 {
612     UINT8    marshalType;
613     UINT8    elements;
614     UINT16   values[6];
615 } TPMS_SESSION_AUDIT_INFO_mst;
616
617 typedef const struct TPMS_CREATION_INFO_mst
618 {
619     UINT8    marshalType;
620     UINT8    elements;
621     UINT16   values[6];
622 } TPMS_CREATION_INFO_mst;
623
624 typedef const struct TPMS_NV_CERTIFY_INFO_mst
625 {
626     UINT8    marshalType;
627     UINT8    elements;
628     UINT16   values[9];
629 } TPMS_NV_CERTIFY_INFO_mst;
630
631 typedef const struct TPMS_NV_DIGEST_CERTIFY_INFO_mst
632 {
633     UINT8    marshalType;
634     UINT8    elements;
635     UINT16   values[6];
636 } TPMS_NV_DIGEST_CERTIFY_INFO_mst;
637
638 typedef const struct TPMS_ST_ATTEST_mst {
639     UINT8    marshalType;
640     UINT8    modifiers;
641     UINT8    errorCode;
642     UINT8    ranges;
643     UINT8    singles;
644     UINT32   values[3];
645 } TPMS_ST_ATTEST_mst;
646
647 typedef struct TPMU_ATTEST_mst
648 {

```



```

649     BYTE          countOfselectors;
650     BYTE          modifiers;
651     UINT16        offsetOfUnmarshalTypes;
652     UINT32        selectors[8];
653     UINT16        marshalingTypes[8];
654 } TPMU_ATTEST_mst;
655
656 typedef const struct TPMS_ATTEST_mst
657 {
658     UINT8         marshalType;
659     UINT8         elements;
660     UINT16        values[21];
661 } TPMS_ATTEST_mst;
662
663 typedef const struct TPMS_AUTH_COMMAND_mst
664 {
665     UINT8         marshalType;
666     UINT8         elements;
667     UINT16        values[12];
668 } TPMS_AUTH_COMMAND_mst;
669
670 typedef const struct TPMS_AUTH_RESPONSE_mst
671 {
672     UINT8         marshalType;
673     UINT8         elements;
674     UINT16        values[9];
675 } TPMS_AUTH_RESPONSE_mst;
676
677 typedef const struct TPMT_TDES_KEY_BITS_mst {
678     UINT8         marshalType;
679     UINT8         modifiers;
680     UINT8         errorCode;
681     UINT8         entries;
682     UINT32        values[1];
683 } TPMT_TDES_KEY_BITS_mst;
684
685 typedef const struct TPMT_AES_KEY_BITS_mst {
686     UINT8         marshalType;
687     UINT8         modifiers;
688     UINT8         errorCode;
689     UINT8         entries;
690     UINT32        values[3];
691 } TPMT_AES_KEY_BITS_mst;
692
693 typedef const struct TPMT_SM4_KEY_BITS_mst {
694     UINT8         marshalType;
695     UINT8         modifiers;
696     UINT8         errorCode;
697     UINT8         entries;
698     UINT32        values[1];
699 } TPMT_SM4_KEY_BITS_mst;
700
701 typedef const struct TPMT_CAMELLIA_KEY_BITS_mst {
702     UINT8         marshalType;
703     UINT8         modifiers;
704     UINT8         errorCode;
705     UINT8         entries;
706     UINT32        values[3];
707 } TPMT_CAMELLIA_KEY_BITS_mst;
708
709 typedef struct TPMU_SYM_KEY_BITS_mst
710 {
711     BYTE          countOfselectors;
712     BYTE          modifiers;
713     UINT16        offsetOfUnmarshalTypes;
714     UINT32        selectors[6];

```

```

715     UINT16      marshalingTypes[6];
716 } TPMU_SYM_KEY_BITS_mst;
717
718 typedef struct TPMU_SYM_MODE_mst
719 {
720     BYTE        countOfselectors;
721     BYTE        modifiers;
722     UINT16      offsetOfUnmarshalTypes;
723     UINT32      selectors[6];
724     UINT16      marshalingTypes[6];
725 } TPMU_SYM_MODE_mst;
726
727 typedef const struct TPMT_SYM_DEF_mst
728 {
729     UINT8       marshalType;
730     UINT8       elements;
731     UINT16      values[9];
732 } TPMT_SYM_DEF_mst;
733
734 typedef const struct TPMT_SYM_DEF_OBJECT_mst
735 {
736     UINT8       marshalType;
737     UINT8       elements;
738     UINT16      values[9];
739 } TPMT_SYM_DEF_OBJECT_mst;
740
741 typedef const struct TPMS_SYMCIPHER_PARMS_mst
742 {
743     UINT8       marshalType;
744     UINT8       elements;
745     UINT16      values[3];
746 } TPMS_SYMCIPHER_PARMS_mst;
747
748 typedef const struct TPMS_DERIVE_mst
749 {
750     UINT8       marshalType;
751     UINT8       elements;
752     UINT16      values[6];
753 } TPMS_DERIVE_mst;
754
755 typedef const struct TPMS_SENSITIVE_CREATE_mst
756 {
757     UINT8       marshalType;
758     UINT8       elements;
759     UINT16      values[6];
760 } TPMS_SENSITIVE_CREATE_mst;
761
762 typedef const struct TPMS_SCHEME_HASH_mst
763 {
764     UINT8       marshalType;
765     UINT8       elements;
766     UINT16      values[3];
767 } TPMS_SCHEME_HASH_mst;
768
769 typedef const struct TPMS_SCHEME_ECDA_mst
770 {
771     UINT8       marshalType;
772     UINT8       elements;
773     UINT16      values[6];
774 } TPMS_SCHEME_ECDA_mst;
775
776 typedef const struct TPMS_ALG_KEYEDHASH_SCHEME_mst {
777     UINT8       marshalType;
778     UINT8       modifiers;
779     UINT8       errorCode;
780     UINT32      values[4];

```

```

781 } TPMI_ALG_KEYEDHASH_SCHEME_mst;
782
783 typedef const struct TPMS_SCHEME_XOR_mst
784 {
785     UINT8    marshalType;
786     UINT8    elements;
787     UINT16   values[6];
788 } TPMS_SCHEME_XOR_mst;
789
790 typedef struct TPMU_SCHEME_KEYEDHASH_mst
791 {
792     BYTE      countOfselectors;
793     BYTE      modifiers;
794     UINT16    offsetOfUnmarshalTypes;
795     UINT32    selectors[3];
796     UINT16    marshalingTypes[3];
797 } TPMU_SCHEME_KEYEDHASH_mst;
798
799 typedef const struct TPMT_KEYEDHASH_SCHEME_mst
800 {
801     UINT8    marshalType;
802     UINT8    elements;
803     UINT16   values[6];
804 } TPMT_KEYEDHASH_SCHEME_mst;
805
806 typedef struct TPMU_SIG_SCHEME_mst
807 {
808     BYTE      countOfselectors;
809     BYTE      modifiers;
810     UINT16    offsetOfUnmarshalTypes;
811     UINT32    selectors[8];
812     UINT16    marshalingTypes[8];
813 } TPMU_SIG_SCHEME_mst;
814
815 typedef const struct TPMT_SIG_SCHEME_mst
816 {
817     UINT8    marshalType;
818     UINT8    elements;
819     UINT16   values[6];
820 } TPMT_SIG_SCHEME_mst;
821
822 typedef struct TPMU_KDF_SCHEME_mst
823 {
824     BYTE      countOfselectors;
825     BYTE      modifiers;
826     UINT16    offsetOfUnmarshalTypes;
827     UINT32    selectors[5];
828     UINT16    marshalingTypes[5];
829 } TPMU_KDF_SCHEME_mst;
830
831 typedef const struct TPMT_KDF_SCHEME_mst
832 {
833     UINT8    marshalType;
834     UINT8    elements;
835     UINT16   values[6];
836 } TPMT_KDF_SCHEME_mst;
837
838 typedef const struct TPMI_ALG_ASYNC_SCHEME_mst {
839     UINT8    marshalType;
840     UINT8    modifiers;
841     UINT8    errorCode;
842     UINT32    values[4];
843 } TPMI_ALG_ASYNC_SCHEME_mst;
844
845 typedef struct TPMU_ASYNC_SCHEME_mst
846 {

```

```

847     BYTE          countOfselectors;
848     BYTE          modifiers;
849     UINT16        offsetOfUnmarshalTypes;
850     UINT32        selectors[11];
851     UINT16        marshalingTypes[11];
852 } TPMU_ASYM_SCHEME_mst;
853
854 typedef const struct TPMI_ALG_RSA_SCHEME_mst {
855     UINT8          marshalType;
856     UINT8          modifiers;
857     UINT8          errorCode;
858     UINT32         values[4];
859 } TPMI_ALG_RSA_SCHEME_mst;
860
861 typedef const struct TPMT_RSA_SCHEME_mst
862 {
863     UINT8          marshalType;
864     UINT8          elements;
865     UINT16         values[6];
866 } TPMT_RSA_SCHEME_mst;
867
868 typedef const struct TPMI_ALG_RSA_DECRYPT_mst {
869     UINT8          marshalType;
870     UINT8          modifiers;
871     UINT8          errorCode;
872     UINT32         values[4];
873 } TPMI_ALG_RSA_DECRYPT_mst;
874
875 typedef const struct TPMT_RSA_DECRYPT_mst
876 {
877     UINT8          marshalType;
878     UINT8          elements;
879     UINT16         values[6];
880 } TPMT_RSA_DECRYPT_mst;
881
882 typedef const struct TPMI_RSA_KEY_BITS_mst {
883     UINT8          marshalType;
884     UINT8          modifiers;
885     UINT8          errorCode;
886     UINT8          entries;
887     UINT32         values[3];
888 } TPMI_RSA_KEY_BITS_mst;
889
890 typedef const struct TPMS_ECC_POINT_mst
891 {
892     UINT8          marshalType;
893     UINT8          elements;
894     UINT16         values[6];
895 } TPMS_ECC_POINT_mst;
896
897 typedef const struct TPMI_ALG_ECC_SCHEME_mst {
898     UINT8          marshalType;
899     UINT8          modifiers;
900     UINT8          errorCode;
901     UINT32         values[4];
902 } TPMI_ALG_ECC_SCHEME_mst;
903
904 typedef const struct TPMI_ECC_CURVE_mst {
905     UINT8          marshalType;
906     UINT8          modifiers;
907     UINT8          errorCode;
908     UINT32         values[3];
909 } TPMI_ECC_CURVE_mst;
910
911 typedef const struct TPMT_ECC_SCHEME_mst
912 {

```

```

913     UINT8     marshalType;
914     UINT8     elements;
915     UINT16    values[6];
916 } TPMT_ECC_SCHEME_mst;
917
918 typedef const struct TPMS_ALGORITHM_DETAIL_ECC_mst
919 {
920     UINT8     marshalType;
921     UINT8     elements;
922     UINT16    values[33];
923 } TPMS_ALGORITHM_DETAIL_ECC_mst;
924
925 typedef const struct TPMS_SIGNATURE_RSA_mst
926 {
927     UINT8     marshalType;
928     UINT8     elements;
929     UINT16    values[6];
930 } TPMS_SIGNATURE_RSA_mst;
931
932 typedef const struct TPMS_SIGNATURE_ECC_mst
933 {
934     UINT8     marshalType;
935     UINT8     elements;
936     UINT16    values[9];
937 } TPMS_SIGNATURE_ECC_mst;
938
939 typedef struct TPMU_SIGNATURE_mst
940 {
941     BYTE      countOfselectors;
942     BYTE      modifiers;
943     UINT16    offsetOfUnmarshalTypes;
944     UINT32    selectors[8];
945     UINT16    marshalingTypes[8];
946 } TPMU_SIGNATURE_mst;
947
948 typedef const struct TPMT_SIGNATURE_mst
949 {
950     UINT8     marshalType;
951     UINT8     elements;
952     UINT16    values[6];
953 } TPMT_SIGNATURE_mst;
954
955 typedef struct TPMU_ENCRYPTED_SECRET_mst
956 {
957     BYTE      countOfselectors;
958     BYTE      modifiers;
959     UINT16    offsetOfUnmarshalTypes;
960     UINT32    selectors[4];
961     UINT16    marshalingTypes[4];
962 } TPMU_ENCRYPTED_SECRET_mst;
963
964 typedef const struct TPMT_ALG_PUBLIC_mst {
965     UINT8     marshalType;
966     UINT8     modifiers;
967     UINT8     errorCode;
968     UINT32    values[4];
969 } TPMT_ALG_PUBLIC_mst;
970
971 typedef struct TPMU_PUBLIC_ID_mst
972 {
973     BYTE      countOfselectors;
974     BYTE      modifiers;
975     UINT16    offsetOfUnmarshalTypes;
976     UINT32    selectors[4];
977     UINT16    marshalingTypes[4];
978 } TPMU_PUBLIC_ID_mst;

```

```

979
980 typedef const struct TPMS_KEYEDHASH_PARMS_mst
981 {
982     UINT8    marshalType;
983     UINT8    elements;
984     UINT16    values[3];
985 } TPMS_KEYEDHASH_PARMS_mst;
986
987 typedef const struct TPMS_RSA_PARMS_mst
988 {
989     UINT8    marshalType;
990     UINT8    elements;
991     UINT16    values[12];
992 } TPMS_RSA_PARMS_mst;
993
994 typedef const struct TPMS_ECC_PARMS_mst
995 {
996     UINT8    marshalType;
997     UINT8    elements;
998     UINT16    values[12];
999 } TPMS_ECC_PARMS_mst;
1000
1001 typedef struct TPMU_PUBLIC_PARMS_mst
1002 {
1003     BYTE        countOfselectors;
1004     BYTE        modifiers;
1005     UINT16       offsetOfUnmarshalTypes;
1006     UINT32       selectors[4];
1007     UINT16       marshalingTypes[4];
1008 } TPMU_PUBLIC_PARMS_mst;
1009
1010 typedef const struct TPMT_PUBLIC_PARMS_mst
1011 {
1012     UINT8    marshalType;
1013     UINT8    elements;
1014     UINT16    values[6];
1015 } TPMT_PUBLIC_PARMS_mst;
1016
1017 typedef const struct TPMT_PUBLIC_mst
1018 {
1019     UINT8    marshalType;
1020     UINT8    elements;
1021     UINT16    values[18];
1022 } TPMT_PUBLIC_mst;
1023
1024 typedef struct TPMU_SENSITIVE_COMPOSITE_mst
1025 {
1026     BYTE        countOfselectors;
1027     BYTE        modifiers;
1028     UINT16       offsetOfUnmarshalTypes;
1029     UINT32       selectors[4];
1030     UINT16       marshalingTypes[4];
1031 } TPMU_SENSITIVE_COMPOSITE_mst;
1032
1033 typedef const struct TPMT_SENSITIVE_mst
1034 {
1035     UINT8    marshalType;
1036     UINT8    elements;
1037     UINT16    values[12];
1038 } TPMT_SENSITIVE_mst;
1039
1040 typedef const struct TPMS_NV_PIN_COUNTER_PARAMETERS_mst
1041 {
1042     UINT8    marshalType;
1043     UINT8    elements;
1044     UINT16    values[6];

```

```

1045 } TPMS_NV_PIN_COUNTER_PARAMETERS_mst;
1046
1047 typedef const struct TPMS_NV_PUBLIC_mst
1048 {
1049     UINT8    marshalType;
1050     UINT8    elements;
1051     UINT16   values[15];
1052 } TPMS_NV_PUBLIC_mst;
1053
1054 typedef const struct TPMS_CONTEXT_DATA_mst
1055 {
1056     UINT8    marshalType;
1057     UINT8    elements;
1058     UINT16   values[6];
1059 } TPMS_CONTEXT_DATA_mst;
1060
1061 typedef const struct TPMS_CONTEXT_mst
1062 {
1063     UINT8    marshalType;
1064     UINT8    elements;
1065     UINT16   values[12];
1066 } TPMS_CONTEXT_mst;
1067
1068 typedef const struct TPMS_CREATION_DATA_mst
1069 {
1070     UINT8    marshalType;
1071     UINT8    elements;
1072     UINT16   values[21];
1073 } TPMS_CREATION_DATA_mst;
1074
1075 typedef const struct TPM_AT_mst {
1076     UINT8    marshalType;
1077     UINT8    modifiers;
1078     UINT8    errorCode;
1079     UINT8    entries;
1080     UINT32   values[4];
1081 } TPM_AT_mst;
1082
1083 typedef const struct TPMS_AC_OUTPUT_mst
1084 {
1085     UINT8    marshalType;
1086     UINT8    elements;
1087     UINT16   values[6];
1088 } TPMS_AC_OUTPUT_mst;
1089
1090 typedef const struct Type02_mst {
1091     UINT8    marshalType;
1092     UINT8    modifiers;
1093     UINT8    errorCode;
1094     UINT32   values[2];
1095 } Type02_mst;
1096
1097 typedef const struct Type03_mst {
1098     UINT8    marshalType;
1099     UINT8    modifiers;
1100     UINT8    errorCode;
1101     UINT32   values[2];
1102 } Type03_mst;
1103
1104 typedef const struct Type04_mst {
1105     UINT8    marshalType;
1106     UINT8    modifiers;
1107     UINT8    errorCode;
1108     UINT32   values[2];
1109 } Type04_mst;
1110

```



```
1111 typedef const struct Type06_mst {
1112     UINT8      marshalType;
1113     UINT8      modifiers;
1114     UINT8      errorCode;
1115     UINT32     values[2];
1116 } Type06_mst;
1117
1118 typedef const struct Type08_mst {
1119     UINT8      marshalType;
1120     UINT8      modifiers;
1121     UINT8      errorCode;
1122     UINT32     values[2];
1123 } Type08_mst;
1124
1125 typedef const struct Type10_mst {
1126     UINT8      marshalType;
1127     UINT8      modifiers;
1128     UINT8      errorCode;
1129     UINT8      entries;
1130     UINT32     values[1];
1131 } Type10_mst;
1132
1133 typedef const struct Type11_mst {
1134     UINT8      marshalType;
1135     UINT8      modifiers;
1136     UINT8      errorCode;
1137     UINT8      entries;
1138     UINT32     values[1];
1139 } Type11_mst;
1140
1141 typedef const struct Type12_mst {
1142     UINT8      marshalType;
1143     UINT8      modifiers;
1144     UINT8      errorCode;
1145     UINT8      entries;
1146     UINT32     values[2];
1147 } Type12_mst;
1148
1149 typedef const struct Type13_mst {
1150     UINT8      marshalType;
1151     UINT8      modifiers;
1152     UINT8      errorCode;
1153     UINT8      entries;
1154     UINT32     values[1];
1155 } Type13_mst;
1156
1157 typedef const struct Type15_mst {
1158     UINT8      marshalType;
1159     UINT8      modifiers;
1160     UINT8      errorCode;
1161     UINT32     values[2];
1162 } Type15_mst;
1163
1164 typedef const struct Type17_mst {
1165     UINT8      marshalType;
1166     UINT8      modifiers;
1167     UINT8      errorCode;
1168     UINT32     values[2];
1169 } Type17_mst;
1170
1171 typedef const struct Type18_mst {
1172     UINT8      marshalType;
1173     UINT8      modifiers;
1174     UINT8      errorCode;
1175     UINT32     values[2];
1176 } Type18_mst;
```

```
1177
1178 typedef const struct Type19_mst {
1179     UINT8      marshalType;
1180     UINT8      modifiers;
1181     UINT8      errorCode;
1182     UINT32     values[2];
1183 } Type19_mst;
1184
1185 typedef const struct Type20_mst {
1186     UINT8      marshalType;
1187     UINT8      modifiers;
1188     UINT8      errorCode;
1189     UINT32     values[2];
1190 } Type20_mst;
1191
1192 typedef const struct Type22_mst {
1193     UINT8      marshalType;
1194     UINT8      modifiers;
1195     UINT8      errorCode;
1196     UINT32     values[2];
1197 } Type22_mst;
1198
1199 typedef const struct Type23_mst {
1200     UINT8      marshalType;
1201     UINT8      modifiers;
1202     UINT8      errorCode;
1203     UINT32     values[2];
1204 } Type23_mst;
1205
1206 typedef const struct Type24_mst {
1207     UINT8      marshalType;
1208     UINT8      modifiers;
1209     UINT8      errorCode;
1210     UINT32     values[2];
1211 } Type24_mst;
1212
1213 typedef const struct Type25_mst {
1214     UINT8      marshalType;
1215     UINT8      modifiers;
1216     UINT8      errorCode;
1217     UINT32     values[2];
1218 } Type25_mst;
1219
1220 typedef const struct Type26_mst {
1221     UINT8      marshalType;
1222     UINT8      modifiers;
1223     UINT8      errorCode;
1224     UINT32     values[2];
1225 } Type26_mst;
1226
1227 typedef const struct Type27_mst {
1228     UINT8      marshalType;
1229     UINT8      modifiers;
1230     UINT8      errorCode;
1231     UINT32     values[2];
1232 } Type27_mst;
1233
1234 typedef const struct Type29_mst {
1235     UINT8      marshalType;
1236     UINT8      modifiers;
1237     UINT8      errorCode;
1238     UINT32     values[2];
1239 } Type29_mst;
1240
1241 typedef const struct Type30_mst {
1242     UINT8      marshalType;
```

```

1243     UINT8      modifiers;
1244     UINT8      errorCode;
1245     UINT32     values[2];
1246 } Type30_mst;
1247
1248 typedef const struct Type33_mst {
1249     UINT8      marshalType;
1250     UINT8      modifiers;
1251     UINT8      errorCode;
1252     UINT32     values[2];
1253 } Type33_mst;
1254
1255 typedef const struct Type34_mst {
1256     UINT8      marshalType;
1257     UINT8      modifiers;
1258     UINT8      errorCode;
1259     UINT32     values[2];
1260 } Type34_mst;
1261
1262 typedef const struct Type35_mst {
1263     UINT8      marshalType;
1264     UINT8      modifiers;
1265     UINT8      errorCode;
1266     UINT32     values[2];
1267 } Type35_mst;
1268
1269 typedef const struct Type38_mst {
1270     UINT8      marshalType;
1271     UINT8      modifiers;
1272     UINT8      errorCode;
1273     UINT32     values[2];
1274 } Type38_mst;
1275
1276 typedef const struct Type41_mst {
1277     UINT8      marshalType;
1278     UINT8      modifiers;
1279     UINT8      errorCode;
1280     UINT32     values[2];
1281 } Type41_mst;
1282
1283 typedef const struct Type42_mst {
1284     UINT8      marshalType;
1285     UINT8      modifiers;
1286     UINT8      errorCode;
1287     UINT32     values[2];
1288 } Type42_mst;
1289
1290 typedef const struct Type44_mst {
1291     UINT8      marshalType;
1292     UINT8      modifiers;
1293     UINT8      errorCode;
1294     UINT32     values[2];
1295 } Type44_mst;

```

This structure combines all the individual marshaling structures to build something that can be referenced by offset rather than full address

```

1296 typedef const struct MarshalData_st {
1297     UintMarshal_mst      UINT8_DATA;
1298     UintMarshal_mst      UINT16_DATA;
1299     UintMarshal_mst      UINT32_DATA;
1300     UintMarshal_mst      UINT64_DATA;
1301     UintMarshal_mst      INT8_DATA;
1302     UintMarshal_mst      INT16_DATA;
1303     UintMarshal_mst      INT32_DATA;

```

1304	UIntMarshal_mst	INT64_DATA;
1305	UIntMarshal_mst	UINT0_DATA;
1306	TPM_ECC_CURVE_mst	TPM_ECC_CURVE_DATA;
1307	TPM_CLOCK_ADJUST_mst	TPM_CLOCK_ADJUST_DATA;
1308	TPM_EO_mst	TPM_EO_DATA;
1309	TPM_SU_mst	TPM_SU_DATA;
1310	TPM_SE_mst	TPM_SE_DATA;
1311	TPM_CAP_mst	TPM_CAP_DATA;
1312	AttributesMarshal_mst	TPMA_ALGORITHM_DATA;
1313	AttributesMarshal_mst	TPMA_OBJECT_DATA;
1314	AttributesMarshal_mst	TPMA_SESSION_DATA;
1315	AttributesMarshal_mst	TPMA_ACT_DATA;
1316	TPMI_YES_NO_mst	TPMI_YES_NO_DATA;
1317	TPMI_DH_OBJECT_mst	TPMI_DH_OBJECT_DATA;
1318	TPMI_DH_PARENT_mst	TPMI_DH_PARENT_DATA;
1319	TPMI_DH_PERSISTENT_mst	TPMI_DH_PERSISTENT_DATA;
1320	TPMI_DH_ENTITY_mst	TPMI_DH_ENTITY_DATA;
1321	TPMI_DH_PCR_mst	TPMI_DH_PCR_DATA;
1322	TPMI_SH_AUTH_SESSION_mst	TPMI_SH_AUTH_SESSION_DATA;
1323	TPMI_SH_HMAC_mst	TPMI_SH_HMAC_DATA;
1324	TPMI_SH_POLICY_mst	TPMI_SH_POLICY_DATA;
1325	TPMI_DH_CONTEXT_mst	TPMI_DH_CONTEXT_DATA;
1326	TPMI_DH_SAVED_mst	TPMI_DH_SAVED_DATA;
1327	TPMI_RH_HIERARCHY_mst	TPMI_RH_HIERARCHY_DATA;
1328	TPMI_RH_ENABLES_mst	TPMI_RH_ENABLES_DATA;
1329	TPMI_RH_HIERARCHY_AUTH_mst	TPMI_RH_HIERARCHY_AUTH_DATA;
1330	TPMI_RH_HIERARCHY_POLICY_mst	TPMI_RH_HIERARCHY_POLICY_DATA;
1331	TPMI_RH_PLATFORM_mst	TPMI_RH_PLATFORM_DATA;
1332	TPMI_RH_OWNER_mst	TPMI_RH_OWNER_DATA;
1333	TPMI_RH_ENDORSEMENT_mst	TPMI_RH_ENDORSEMENT_DATA;
1334	TPMI_RH_PROVISION_mst	TPMI_RH_PROVISION_DATA;
1335	TPMI_RH_CLEAR_mst	TPMI_RH_CLEAR_DATA;
1336	TPMI_RH_NV_AUTH_mst	TPMI_RH_NV_AUTH_DATA;
1337	TPMI_RH_LOCKOUT_mst	TPMI_RH_LOCKOUT_DATA;
1338	TPMI_RH_NV_INDEX_mst	TPMI_RH_NV_INDEX_DATA;
1339	TPMI_RH_AC_mst	TPMI_RH_AC_DATA;
1340	TPMI_RH_ACT_mst	TPMI_RH_ACT_DATA;
1341	TPMI_ALG_HASH_mst	TPMI_ALG_HASH_DATA;
1342	TPMI_ALG_ASYM_mst	TPMI_ALG_ASYM_DATA;
1343	TPMI_ALG_SYM_mst	TPMI_ALG_SYM_DATA;
1344	TPMI_ALG_SYM_OBJECT_mst	TPMI_ALG_SYM_OBJECT_DATA;
1345	TPMI_ALG_SYM_MODE_mst	TPMI_ALG_SYM_MODE_DATA;
1346	TPMI_ALG_KDF_mst	TPMI_ALG_KDF_DATA;
1347	TPMI_ALG_SIG_SCHEME_mst	TPMI_ALG_SIG_SCHEME_DATA;
1348	TPMI_ECC_KEY_EXCHANGE_mst	TPMI_ECC_KEY_EXCHANGE_DATA;
1349	TPMI_ST_COMMAND_TAG_mst	TPMI_ST_COMMAND_TAG_DATA;
1350	TPMI_ALG_MAC_SCHEME_mst	TPMI_ALG_MAC_SCHEME_DATA;
1351	TPMI_ALG_CIPHER_MODE_mst	TPMI_ALG_CIPHER_MODE_DATA;
1352	TPMS_EMPTY_mst	TPMS_EMPTY_DATA;
1353	TPMS_ALGORITHM_DESCRIPTION_mst	TPMS_ALGORITHM_DESCRIPTION_DATA;
1354	TPMU_HA_mst	TPMU_HA_DATA;
1355	TPMT_HA_mst	TPMT_HA_DATA;
1356	Tpm2bMarshal_mst	TPM2B_DIGEST_DATA;
1357	Tpm2bMarshal_mst	TPM2B_DATA_DATA;
1358	Tpm2bMarshal_mst	TPM2B_EVENT_DATA;
1359	Tpm2bMarshal_mst	TPM2B_MAX_BUFFER_DATA;
1360	Tpm2bMarshal_mst	TPM2B_MAX_NV_BUFFER_DATA;
1361	Tpm2bMarshal_mst	TPM2B_TIMEOUT_DATA;
1362	Tpm2bMarshal_mst	TPM2B_IV_DATA;
1363	NullUnionMarshal_mst	NULL_UNION_DATA;
1364	Tpm2bMarshal_mst	TPM2B_NAME_DATA;
1365	TPMS_PCR_SELECT_mst	TPMS_PCR_SELECT_DATA;
1366	TPMS_PCR_SELECTION_mst	TPMS_PCR_SELECTION_DATA;
1367	TPMT_TK_CREATION_mst	TPMT_TK_CREATION_DATA;
1368	TPMT_TK_VERIFIED_mst	TPMT_TK_VERIFIED_DATA;
1369	TPMT_TK_AUTH_mst	TPMT_TK_AUTH_DATA;

1370	TPMT_TK_HASHCHECK_mst	TPMT_TK_HASHCHECK_DATA;
1371	TPMS_ALG_PROPERTY_mst	TPMS_ALG_PROPERTY_DATA;
1372	TPMS_TAGGED_PROPERTY_mst	TPMS_TAGGED_PROPERTY_DATA;
1373	TPMS_TAGGED_PCR_SELECT_mst	TPMS_TAGGED_PCR_SELECT_DATA;
1374	TPMS_TAGGED_POLICY_mst	TPMS_TAGGED_POLICY_DATA;
1375	TPMS_ACT_DATA_mst	TPMS_ACT_DATA_DATA;
1376	ListMarshal_mst	TPML_CC_DATA;
1377	ListMarshal_mst	TPML_CCA_DATA;
1378	ListMarshal_mst	TPML_ALG_DATA;
1379	ListMarshal_mst	TPML_HANDLE_DATA;
1380	ListMarshal_mst	TPML_DIGEST_DATA;
1381	ListMarshal_mst	TPML_DIGEST_VALUES_DATA;
1382	ListMarshal_mst	TPML_PCR_SELECTION_DATA;
1383	ListMarshal_mst	TPML_ALG_PROPERTY_DATA;
1384	ListMarshal_mst	TPML_TAGGED_TPM_PROPERTY_DATA;
1385	ListMarshal_mst	TPML_TAGGED_PCR_PROPERTY_DATA;
1386	ListMarshal_mst	TPML_ECC_CURVE_DATA;
1387	ListMarshal_mst	TPML_TAGGED_POLICY_DATA;
1388	ListMarshal_mst	TPML_ACT_DATA_DATA;
1389	TPMU_CAPABILITIES_mst	TPMU_CAPABILITIES_DATA;
1390	TPMS_CAPABILITY_DATA_mst	TPMS_CAPABILITY_DATA_DATA;
1391	TPMS_CLOCK_INFO_mst	TPMS_CLOCK_INFO_DATA;
1392	TPMS_TIME_INFO_mst	TPMS_TIME_INFO_DATA;
1393	TPMS_TIME_ATTEST_INFO_mst	TPMS_TIME_ATTEST_INFO_DATA;
1394	TPMS_CERTIFY_INFO_mst	TPMS_CERTIFY_INFO_DATA;
1395	TPMS_QUOTE_INFO_mst	TPMS_QUOTE_INFO_DATA;
1396	TPMS_COMMAND_AUDIT_INFO_mst	TPMS_COMMAND_AUDIT_INFO_DATA;
1397	TPMS_SESSION_AUDIT_INFO_mst	TPMS_SESSION_AUDIT_INFO_DATA;
1398	TPMS_CREATION_INFO_mst	TPMS_CREATION_INFO_DATA;
1399	TPMS_NV_CERTIFY_INFO_mst	TPMS_NV_CERTIFY_INFO_DATA;
1400	TPMS_NV_DIGEST_CERTIFY_INFO_mst	TPMS_NV_DIGEST_CERTIFY_INFO_DATA;
1401	TPMI_ST_ATTEST_mst	TPMI_ST_ATTEST_DATA;
1402	TPMU_ATTEST_mst	TPMU_ATTEST_DATA;
1403	TPMS_ATTEST_mst	TPMS_ATTEST_DATA;
1404	Tpm2bMarshal_mst	TPM2B_ATTEST_DATA;
1405	TPMS_AUTH_COMMAND_mst	TPMS_AUTH_COMMAND_DATA;
1406	TPMS_AUTH_RESPONSE_mst	TPMS_AUTH_RESPONSE_DATA;
1407	TPMI_TDES_KEY_BITS_mst	TPMI_TDES_KEY_BITS_DATA;
1408	TPMI_AES_KEY_BITS_mst	TPMI_AES_KEY_BITS_DATA;
1409	TPMI_SM4_KEY_BITS_mst	TPMI_SM4_KEY_BITS_DATA;
1410	TPMI_CAMELLIA_KEY_BITS_mst	TPMI_CAMELLIA_KEY_BITS_DATA;
1411	TPMU_SYM_KEY_BITS_mst	TPMU_SYM_KEY_BITS_DATA;
1412	TPMU_SYM_MODE_mst	TPMU_SYM_MODE_DATA;
1413	TPMT_SYM_DEF_mst	TPMT_SYM_DEF_DATA;
1414	TPMT_SYM_DEF_OBJECT_mst	TPMT_SYM_DEF_OBJECT_DATA;
1415	Tpm2bMarshal_mst	TPM2B_SYM_KEY_DATA;
1416	TPMS_SYMCIPHER_PARMS_mst	TPMS_SYMCIPHER_PARMS_DATA;
1417	Tpm2bMarshal_mst	TPM2B_LABEL_DATA;
1418	TPMS_DERIVE_mst	TPMS_DERIVE_DATA;
1419	Tpm2bMarshal_mst	TPM2B_DERIVE_DATA;
1420	Tpm2bMarshal_mst	TPM2B_SENSITIVE_DATA_DATA;
1421	TPMS_SENSITIVE_CREATE_mst	TPMS_SENSITIVE_CREATE_DATA;
1422	Tpm2bsMarshal_mst	TPM2B_SENSITIVE_CREATE_DATA;
1423	TPMS_SCHEME_HASH_mst	TPMS_SCHEME_HASH_DATA;
1424	TPMS_SCHEME_ECDSA_mst	TPMS_SCHEME_ECDSA_DATA;
1425	TPMI_ALG_KEYEDHASH_SCHEME_mst	TPMI_ALG_KEYEDHASH_SCHEME_DATA;
1426	TPMS_SCHEME_XOR_mst	TPMS_SCHEME_XOR_DATA;
1427	TPMU_SCHEME_KEYEDHASH_mst	TPMU_SCHEME_KEYEDHASH_DATA;
1428	TPMT_KEYEDHASH_SCHEME_mst	TPMT_KEYEDHASH_SCHEME_DATA;
1429	TPMU_SIG_SCHEME_mst	TPMU_SIG_SCHEME_DATA;
1430	TPMT_SIG_SCHEME_mst	TPMT_SIG_SCHEME_DATA;
1431	TPMU_KDF_SCHEME_mst	TPMU_KDF_SCHEME_DATA;
1432	TPMT_KDF_SCHEME_mst	TPMT_KDF_SCHEME_DATA;
1433	TPMI_ALG_ASYM_SCHEME_mst	TPMI_ALG_ASYM_SCHEME_DATA;
1434	TPMU_ASYM_SCHEME_mst	TPMU_ASYM_SCHEME_DATA;
1435	TPMI_ALG_RSA_SCHEME_mst	TPMI_ALG_RSA_SCHEME_DATA;

1436	TPMT_RSA_SCHEME_mst	TPMT_RSA_SCHEME_DATA;
1437	TPMI_ALG_RSA_DECRYPT_mst	TPMI_ALG_RSA_DECRYPT_DATA;
1438	TPMT_RSA_DECRYPT_mst	TPMT_RSA_DECRYPT_DATA;
1439	Tpm2bMarshal_mst	TPM2B_PUBLIC_KEY_RSA_DATA;
1440	TPMI_RSA_KEY_BITS_mst	TPMI_RSA_KEY_BITS_DATA;
1441	Tpm2bMarshal_mst	TPM2B_PRIVATE_KEY_RSA_DATA;
1442	Tpm2bMarshal_mst	TPM2B_ECC_PARAMETER_DATA;
1443	TPMS_ECC_POINT_mst	TPMS_ECC_POINT_DATA;
1444	Tpm2bsMarshal_mst	TPM2B_ECC_POINT_DATA;
1445	TPMI_ALG_ECC_SCHEME_mst	TPMI_ALG_ECC_SCHEME_DATA;
1446	TPMI_ECC_CURVE_mst	TPMI_ECC_CURVE_DATA;
1447	TPMT_ECC_SCHEME_mst	TPMT_ECC_SCHEME_DATA;
1448	TPMS_ALGORITHM_DETAIL_ECC_mst	TPMS_ALGORITHM_DETAIL_ECC_DATA;
1449	TPMS_SIGNATURE_RSA_mst	TPMS_SIGNATURE_RSA_DATA;
1450	TPMS_SIGNATURE_ECC_mst	TPMS_SIGNATURE_ECC_DATA;
1451	TPMU_SIGNATURE_mst	TPMU_SIGNATURE_DATA;
1452	TPMT_SIGNATURE_mst	TPMT_SIGNATURE_DATA;
1453	TPMU_ENCRYPTED_SECRET_mst	TPMU_ENCRYPTED_SECRET_DATA;
1454	Tpm2bMarshal_mst	TPM2B_ENCRYPTED_SECRET_DATA;
1455	TPMI_ALG_PUBLIC_mst	TPMI_ALG_PUBLIC_DATA;
1456	TPMU_PUBLIC_ID_mst	TPMU_PUBLIC_ID_DATA;
1457	TPMS_KEYEDHASH_PARMS_mst	TPMS_KEYEDHASH_PARMS_DATA;
1458	TPMS_RSA_PARMS_mst	TPMS_RSA_PARMS_DATA;
1459	TPMS_ECC_PARMS_mst	TPMS_ECC_PARMS_DATA;
1460	TPMU_PUBLIC_PARMS_mst	TPMU_PUBLIC_PARMS_DATA;
1461	TPMT_PUBLIC_PARMS_mst	TPMT_PUBLIC_PARMS_DATA;
1462	TPMT_PUBLIC_mst	TPMT_PUBLIC_DATA;
1463	Tpm2bsMarshal_mst	TPM2B_PUBLIC_DATA;
1464	Tpm2bMarshal_mst	TPM2B_TEMPLATE_DATA;
1465	Tpm2bMarshal_mst	TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA;
1466	TPMU_SENSITIVE_COMPOSITE_mst	TPMU_SENSITIVE_COMPOSITE_DATA;
1467	TPMT_SENSITIVE_mst	TPMT_SENSITIVE_DATA;
1468	Tpm2bsMarshal_mst	TPM2B_SENSITIVE_DATA;
1469	Tpm2bMarshal_mst	TPM2B_PRIVATE_DATA;
1470	Tpm2bMarshal_mst	TPM2B_ID_OBJECT_DATA;
1471	TPMS_NV_PIN_COUNTER_PARAMETERS_mst	TPMS_NV_PIN_COUNTER_PARAMETERS_DATA;
1472	AttributesMarshal_mst	TPMA_NV_DATA;
1473	TPMS_NV_PUBLIC_mst	TPMS_NV_PUBLIC_DATA;
1474	Tpm2bsMarshal_mst	TPM2B_NV_PUBLIC_DATA;
1475	Tpm2bMarshal_mst	TPM2B_CONTEXT_SENSITIVE_DATA;
1476	TPMS_CONTEXT_DATA_mst	TPMS_CONTEXT_DATA_DATA;
1477	Tpm2bMarshal_mst	TPM2B_CONTEXT_DATA_DATA;
1478	TPMS_CONTEXT_mst	TPMS_CONTEXT_DATA;
1479	TPMS_CREATION_DATA_mst	TPMS_CREATION_DATA_DATA;
1480	Tpm2bsMarshal_mst	TPM2B_CREATION_DATA_DATA;
1481	TPM_AT_mst	TPM_AT_DATA;
1482	TPMS_AC_OUTPUT_mst	TPMS_AC_OUTPUT_DATA;
1483	ListMarshal_mst	TPML_AC_CAPABILITIES_DATA;
1484	MinMaxMarshal_mst	Type00_DATA;
1485	MinMaxMarshal_mst	Type01_DATA;
1486	Type02_mst	Type02_DATA;
1487	Type03_mst	Type03_DATA;
1488	Type04_mst	Type04_DATA;
1489	MinMaxMarshal_mst	Type05_DATA;
1490	Type06_mst	Type06_DATA;
1491	MinMaxMarshal_mst	Type07_DATA;
1492	Type08_mst	Type08_DATA;
1493	Type10_mst	Type10_DATA;
1494	Type11_mst	Type11_DATA;
1495	Type12_mst	Type12_DATA;
1496	Type13_mst	Type13_DATA;
1497	Type15_mst	Type15_DATA;
1498	Type17_mst	Type17_DATA;
1499	Type18_mst	Type18_DATA;
1500	Type19_mst	Type19_DATA;
1501	Type20_mst	Type20_DATA;

```

1502     Type22_mst                Type22_DATA;
1503     Type23_mst                Type23_DATA;
1504     Type24_mst                Type24_DATA;
1505     Type25_mst                Type25_DATA;
1506     Type26_mst                Type26_DATA;
1507     Type27_mst                Type27_DATA;
1508     MinMaxMarshal_mst        Type28_DATA;
1509     Type29_mst                Type29_DATA;
1510     Type30_mst                Type30_DATA;
1511     MinMaxMarshal_mst        Type31_DATA;
1512     MinMaxMarshal_mst        Type32_DATA;
1513     Type33_mst                Type33_DATA;
1514     Type34_mst                Type34_DATA;
1515     Type35_mst                Type35_DATA;
1516     MinMaxMarshal_mst        Type36_DATA;
1517     MinMaxMarshal_mst        Type37_DATA;
1518     Type38_mst                Type38_DATA;
1519     MinMaxMarshal_mst        Type39_DATA;
1520     MinMaxMarshal_mst        Type40_DATA;
1521     Type41_mst                Type41_DATA;
1522     Type42_mst                Type42_DATA;
1523     MinMaxMarshal_mst        Type43_DATA;
1524     Type44_mst                Type44_DATA;
1525 } MarshalData_st;
1526
1527 #endif // _TABLE_MARSHAL_TYPES_H_

```

9.10.8 Table Marshal Source

9.10.8.1 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 { int a; } External_Structure_t;
14
15 extern struct Exernal_Structure_t MarshalData;
16
17 #define IS_SUCCESS(UNMARSHAL_FUNCTION) \
18     (TPM_RC_SUCCESS == (result = (UNMARSHAL_FUNCTION)))
19
20 marshalIndex_t IntegerDispatch[] = {
21     UINT8_MARSHAL_REF, UINT16_MARSHAL_REF, UINT32_MARSHAL_REF, UINT64_MARSHAL_REF,
22     INT8_MARSHAL_REF, INT16_MARSHAL_REF, INT32_MARSHAL_REF, INT64_MARSHAL_REF
23 };
24
25 #if 1
26 #define GetDescriptor(reference) \
27     ((MarshalHeader_mst *) (((BYTE *) (&MarshalData)) + (reference & NULL_MASK)))
28 #else
29 static const MarshalHeader_mst *GetDescriptor(marshalIndex_t index)
30 {
31     const MarshalHeader_mst *mst = MarshalLookupTable[index & NULL_MASK];
32     return mst;
33 }

```



```
34 #endif
35
36 #define GetUnionDescriptor(_index_) \
37     ((UnionMarshal_mst *)GetDescriptor(_index_))
38 #define GetArrayDescriptor(_index_) \
39     ((ArrayMarshal_mst *) (ArrayLookupTable[_index_] & NULL_MASK))
```

DRAFT

9.10.8.1.1.1 GetUnmarshaledInteger()

Gets the unmarshaled value and normalizes it to a UIN32 for other processing (comparisons and such).

```

40 static UINT32 GetUnmarshaledInteger(
41     marshalIndex_t    type,
42     const void        *target
43 )
44 {
45     int    size = (type & SIZE_MASK);
46     //
47     if(size == FOUR_BYTES)
48         return *((UINT32 *)target);
49     if(type & IS_SIGNED)
50     {
51         if(size == TWO_BYTES)
52             return (UINT32)*((int16_t *)target);
53         return (UINT32)*((int8_t *)target);
54     }
55     if(size == TWO_BYTES)
56         return (UINT32)*((UINT16 *)target);
57     return (UINT32)*((UINT8 *)target);
58 }
59 static UINT32 GetSelector(
60     void            *structure,
61     const UINT16    *values,
62     UINT16          descriptor
63 )
64 {
65     unsigned        sel = GET_ELEMENT_NUMBER(descriptor);
66     // Get the offset of the value in the unmarshaled structure
67     const UINT16    *entry = &values[(sel * 3)];
68     //
69     return GetUnmarshaledInteger(GET_ELEMENT_SIZE(entry[0]),
70     ((UINT8 *)structure) + entry[2]);
71 }
72 static TPM_RC UnmarshalBytes(
73     UINT8            *target,           // IN/OUT: place to put the bytes
74     UINT8            **buffer,         // IN/OUT: source of the input data
75     INT32            *size,            // IN/OUT: remaining bytes in the input buffer
76     int              count             // IN: number of bytes to get
77 )
78 {
79     if((*size -= count) >= 0)
80     {
81         memcpy(target, *buffer, count);
82         *buffer += count;
83         return TPM_RC_SUCCESS;
84     }
85     return TPM_RC_INSUFFICIENT;
86 }

```

9.10.8.1.1.2 MarshalBytes()

Marshal an array of bytes.

```
87  static UINT16 MarshalBytes(  
88      UINT8          *source,  
89      UINT8          **buffer,  
90      INT32          *size,  
91      int32_t         count  
92  )  
93  {  
94      if(buffer != NULL)  
95      {  
96          if(size != NULL && (size -= count) < 0)  
97              return 0;  
98          memcpy(*buffer, source, count);  
99          *buffer += count;  
100     }  
101     return (UINT16)count;  
102 }
```

9.10.8.1.1.3 ArrayUnmarshal()

Unmarshal an array. The *index* is of the form: *type_ARRAY_MARSHAL_INDEX*.

```

103 static TPM_RC ArrayUnmarshal(
104     UINT16      index,           // IN: the type of the array
105     UINT8       *target,        // IN: target for the data
106     UINT8       **buffer,       // IN/OUT: place to get the data
107     INT32       *size,          // IN/OUT: remaining unmarshal data
108     UINT32      count,          // IN: number of values of 'index' to
109                                // unmarshal
110 )
111 {
112     marshalIndex_t  which = ArrayLookupTable[index & NULL_MASK].type;
113     UINT16          stride = ArrayLookupTable[index & NULL_MASK].stride;
114     TPM_RC          result;
115 //
116     if(stride == 1) // A byte array
117         result = UnmarshalBytes(target, buffer, size, count);
118     else
119     {
120         which |= index & NULL_FLAG;
121         for(result = TPM_RC_SUCCESS; count > 0; target += stride, count--)
122             if(!IS_SUCCESS(Unmarshal(which, target, buffer, size)))
123                 break;
124     }
125     return result;
126 }

```

9.10.8.1.1.4 ArrayMarshal()

```
127 static UINT16 ArrayMarshal(  
128     UINT16      index,           // IN: the type of the array  
129     UINT8       *source,         // IN: source of the data  
130     UINT8       **buffer,        // IN/OUT: place to put the data  
131     INT32       *size,           // IN/OUT: amount of space for the data  
132     UINT32      count            // IN: number of values of 'index' to marshal  
133 )  
134 {  
135     marshalIndex_t which = ArrayLookupTable[index & NULL_MASK].type;  
136     UINT16 stride = ArrayLookupTable[index & NULL_MASK].stride;  
137     UINT16 retVal;  
138     //  
139     if(stride == 1) // A byte array  
140         return MarshalBytes(source, buffer, size, count);  
141     which |= index & NULL_FLAG;  
142     for(retVal = 0  
143         ; count > 0  
144         ; source += stride, count--)  
145         retVal += Marshal(which, source, buffer, size);  
146  
147     return retVal;  
148 }
```

9.10.8.1.1.5 UnmarshalUnion()

```

149  TPM_RC
150  UnmarshalUnion(
151      UINT16      typeIndex,          // IN: the thing to unmarshal
152      void        *target,           // IN: where the data goes to
153      UINT8       **buffer,          // IN/OUT: the data source buffer
154      INT32       *size,             // IN/OUT: the remaining size
155      UINT32      selector
156  )
157  {
158      int          i;
159      UnionMarshal_mst *ut = GetUnionDescriptor(typeIndex);
160      marshalIndex_t selected;
161      //
162      for(i = 0; i < ut->countOfselectors; i++)
163      {
164          if(selector == ut->selectors[i])
165          {
166              UINT8 *offset = ((UINT8 *)ut) + ut->offsetOfUnmarshalTypes;
167              // Get the selected thing to unmarshal
168              selected = ((marshalIndex_t *)offset)[i];
169              if(ut->modifiers & IS_ARRAY_UNION)
170                  return UnmarshalBytes(target, buffer, size, selected);
171              else
172              {
173                  // Propagate NULL_FLAG if the null flag was
174                  // propagated to the structure containing the union
175                  selected |= (typeIndex & NULL_FLAG);
176                  return Unmarshal(selected, target, buffer, size);
177              }
178          }
179      }
180      // Didn't find the value.
181      return TPM_RC_SELECTOR;
182  }

```

9.10.8.1.1.6 MarshalUnion()

```

183  UINT16
184  MarshalUnion(
185      UINT16          typeIndex,          // IN: the thing to marshal
186      void            *source,            // IN: where the data comes from
187      UINT8           **buffer,           // IN/OUT: the data source buffer
188      INT32           *size,              // IN/OUT: the remaining size
189      UINT32          selector            // IN: the union selector
190  )
191  {
192      int              i;
193      UnionMarshal_mst *ut = GetUnionDescriptor(typeIndex);
194      marshalIndex_t   selected;
195      //
196      for(i = 0; i < ut->countOfselectors; i++)
197      {
198          if(selector == ut->selectors[i])
199          {
200              UINT8      *offset = ((UINT8 *)ut) + ut->offsetOfUnmarshalTypes;
201              // Get the selected thing to unmarshal
202              selected = ((marshalIndex_t *)offset)[i];
203              if(ut->modifiers & IS_ARRAY_UNION)
204                  return MarshalBytes(source, buffer, size, selected);
205              else
206                  return Marshal(selected, source, buffer, size);
207          }
208      }
209      if(size != NULL)
210          *size = -1;
211      return 0;
212  }
213  TPM_RC
214  UnmarshalInteger(
215      int              iSize,              // IN: Number of bytes in the integer
216      void            *target,            // OUT: receives the integer
217      UINT8           **buffer,           // IN/OUT: source of the data
218      INT32           *size,              // IN/OUT: amount of data available
219      UINT32          *value             // OUT: optional copy of 'target'
220  )
221  {
222      // This is just to save typing
223      #define _MB_ (*buffer)
224      // The size is a power of two so convert to regular integer
225      int      bytes = (1 << (iSize & SIZE_MASK));
226      //
227      // Check to see if there is enough data to fulfill the request
228      if((*size -= bytes) >= 0)
229      {
230          // The most common size
231          if(bytes == 4)
232          {
233              *((UINT32 *)target) = (UINT32)((((_MB_[0] << 8) | _MB_[1]) << 8)
234              | _MB_[2]) << 8) | _MB_[3]);
235              // If a copy is needed, copy it.
236              if(value != NULL)
237                  *value = *((UINT32 *)target);
238          }
239          else if(bytes == 2)
240          {
241              *((UINT16 *)target) = (UINT16)((_MB_[0] << 8) | _MB_[1]);
242              // If a copy is needed, copy with the appropriate sign extension
243              if(value != NULL)
244              {
245                  if(iSize & IS_SIGNED)

```



```

246         *value = (UINT32) (*(INT16 *)target);
247     else
248         *value = (UINT32) (*(UINT16 *)target);
249     }
250 }
251 else if(bytes == 1)
252 {
253     *((UINT8*)target) = (UINT8)_MB_[0];
254     // If a copy is needed, copy with the appropriate sign extension
255     if(value != NULL)
256     {
257         if(iSize & IS_SIGNED)
258             *value = (UINT32) (*(INT8 *)target);
259         else
260             *value = (UINT32) (*(UINT8 *)target);
261     }
262 }
263 else
264 {
265     // There is no input type that is a 64-bit value other than a UINT64. So
266     // there is no reason to do anything other than unmarshal it.
267     *((UINT64 *)target) = BYTE_ARRAY_TO_UINT64(*buffer);
268 }
269 *buffer += bytes;
270 return TPM_RC_SUCCESS;
271 #undef _MB_
272 }
273 return TPM_RC_INSUFFICIENT;
274 }

```

9.10.8.1.1.7 Unmarshal()

This is the function that performs unmarshaling of different numbered types. Each TPM type has a number. The number is used to lookup the address of the data structure that describes how to unmarshal that data type.

```

275 TPM_RC
276 Unmarshal(
277     UINT16      typeIndex,          // IN: the thing to marshal
278     void        *target,            // IN: where the data goes from
279     UINT8       **buffer,           // IN/OUT: the data source buffer
280     INT32       *size               // IN/OUT: the remaining size
281 )
282 {
283     const MarshalHeader_mst *sel;
284     TPM_RC result;
285     //
286     sel = GetDescriptor(typeIndex);
287     switch(sel->marshalType)
288     {
289     case UINT_MTYPE:
290     {
291         // A simple signed or unsigned integer value.
292         return UnmarshalInteger(sel->modifiers, target,
293                                buffer, size, NULL);
294     }
295     case VALUES_MTYPE:
296     {
297         // This is the general-purpose structure that can handle things like
298         // TPMI_DH_PARENT that has multiple ranges, multiple singles and a
299         // 'null' value. When things cover a large range with holes in the range
300         // they can be turned into multiple ranges. There is no option for a bit
301         // field.
302         // The structure is:
303         // typedef const struct ValuesMarshal_mst
304         // {
305         //     UINT8      marshalType;          // VALUES_MTYPE
306         //     UINT8      modifiers;
307         //     UINT8      errorCode;
308         //     UINT8      ranges;
309         //     UINT8      singles;
310         //     UINT32     values[1];
311         // } ValuesMarshal_mst;
312         // Unmarshal the base type
313         UINT32 val;
314         if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
315                                       buffer, size, &val)))
316         {
317             ValuesMarshal_mst *vmt = ((ValuesMarshal_mst *)sel);
318             const UINT32 *check = vmt->values;
319             //
320             // if the TAKES_NULL flag is set, then the first entry in the values
321             // list is the NULL value. It is not included in the 'ranges' or
322             // 'singles' count.
323             if((vmt->modifiers & TAKES_NULL) && (val == *check++))
324             {
325                 if((typeIndex & NULL_FLAG) == 0)
326                     result = (TPM_RC)(sel->errorCode);
327             }
328             // No NULL value or input is not the NULL value
329             else
330             {
331                 int i;
332                 //

```

```

333         // Check all the min-max ranges.
334         for(i = vmt->ranges - 1; i >= 0; check = &check[2], i--)
335             if((UINT32)(val - check[0]) <= check[1])
336                 break;
337         // if the input is in a selected range, then i >= 0
338         if(i < 0)
339         {
340             // Not in any range, so check sigles
341             for(i = vmt->singles - 1; i >= 0; i--)
342                 if(val == check[i])
343                     break;
344         }
345         // If input not in range and not in any single so return error
346         if(i < 0)
347             result = (TPM_RC)(sel->errorCode);
348     }
349 }
350 break;
351 }
352 case TABLE_MTYPE:
353 {
354     // This is a table with or without bit checking. The input is checked
355     // against each value in the table. If the value is in the table, and
356     // a bits table is present, then the bit field is checked to see if the
357     // indicated value is implemented. For example, if there is a table of
358     // allowed RSA key sizes and the 2nd entry matches, then the 2nd bit in
359     // the bit field is checked to see if that allowed size is implemented
360     // in this TPM.
361     // typedef const struct TableMarshal_mst
362     // {
363     //     UINT8      marshalType;          // TABLE_MTYPE
364     //     UINT8      modifiers;
365     //     UINT8      errorCode;
366     //     UINT8      singles;
367     //     UINT32     values[singles + 1 if TAKES_NULL];
368     // } TableMarshal_mst;
369
370     UINT32      val;
371
372     // Unmarshal the base type
373     if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
374                                     buffer, size, &val)))
375     {
376         TableMarshal_mst *tmt = ((TableMarshal_mst *)sel);
377         const UINT32 *check = tmt->values;
378
379         // If this type has a null value, then it is the first value in the
380         // list of values. It does not count in the count of values
381         if((tmt->modifiers & TAKES_NULL) && (val == *check++))
382         {
383             if((typeIndex & NULL_FLAG) == 0)
384                 result = (TPM_RC)(sel->errorCode);
385         }
386         else
387         {
388             int i;
389
390             // Process the singles
391             for(i = tmt->singles - 1; i >= 0; i--)
392             {
393                 // does the input value match the value in the table
394                 if(val == check[i])
395                 {
396                     // If there is an associated bit table, make sure that
397                     // the corresponding bit is SET
398                     if((HAS_BITS & tmt->modifiers)

```

```

399         && (!IS_BIT_SET32(i, &(check[tmt->singles])))
400         // if not SET, then this is a failure.
401         i = -1;
402         break;
403     }
404 }
405 // error if not found or bit not SET
406 if(i < 0)
407     result = (TPM_RC) (sel->errorCode);
408 }
409 }
410 break;
411 }
412 case MIN_MAX_MTYPE:
413 {
414     // A MIN_MAX is a range. It can have a bit field and a NULL value that is
415     // outside of the range. If the input value is in the min-max range then
416     // it is valid unless there is an associated bit field. Otherwise, it
417     // it is only valid if the corresponding value in the bit field is SET.
418     // The min value is 'values[0]' or 'values[1]' if there is a NULL value.
419     // The max value is the value after min. The max value is in the table as
420     // max minus min. This allows 'val' to be subtracted from min and then
421     // checked against max with one unsigned comparison. If present, the bit
422     // field will be the first 'values' after max.
423     // typedef const struct MinMaxMarshal_mst
424     // {
425     //     UINT8      marshalType;          // MIN_MAX_MTYPE
426     //     UINT8      modifiers;
427     //     UINT8      errorCode;
428     //     UINT32     values[2 + 1 if TAKES_NULL];
429     // } MinMaxMarshal_mst;
430     UINT32         val;
431     //
432     // A min-max has a range. It can have a bit-field that is indexed to the
433     // min value (something that matches min has a bit at 0. This is useful
434     // for algorithms. The min-max define a range of algorithms to be checked
435     // and the bit field can check to see if the algorithm in that range is
436     // allowed.
437     if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
438                                     buffer, size, &val)))
439     {
440         MinMaxMarshal_mst *mmt = (MinMaxMarshal_mst *)sel;
441         const UINT32      *check = mmt->values;
442         //
443         // If this type takes a NULL, see if it matches. This
444         if((mmt->modifiers & TAKES_NULL) && (val == *check++))
445         {
446             if((typeIndex & NULL_FLAG) == 0)
447                 result = (TPM_RC) (mmt->errorCode);
448             }
449             else
450             {
451                 val -= *check;
452                 if((val > check[1])
453                     || ((mmt->modifiers & HAS_BITS) &&
454                         !IS_BIT_SET32(val, &check[2])))
455                     result = (TPM_RC) (mmt->errorCode);
456             }
457         }
458         break;
459     }
460 case ATTRIBUTES_MTYPE:
461 {
462     // This is used for TPMA values.
463     UINT32         mask;
464     AttributesMarshal_mst *amt = (AttributesMarshal_mst *)sel;

```

```

465 //
466 if(IS_SUCCESS(UnmarshalInteger(sel->modifiers, target,
467                               buffer, size, &mask)))
468 {
469     if((mask & amt->attributeMask) != 0)
470         result = TPM_RC_RESERVED_BITS;
471 }
472 break;
473 }
474 case STRUCTURE_MTYPE:
475 {
476     // A structure (not a union). A structure has elements (one defined per
477     // row). Three UINT16 values are used for each row. The first indicates
478     // the type of the entry. The choices are: simple, union, or array. A
479     // simple type can be a simple integer or another structure. It can also
480     // be a specific "interface type." For example, when a structure entry is
481     // a value that is used to define the dimension of an array, the entry of
482     // the structure will reference a "synthetic" interface type, most often
483     // a min-max value. If the type of the entry is union or array, then the
484     // first value indicates which of the previous elements provides the union
485     // selector or the array dimension. That previous entry is referenced in
486     // the unmarshaled structure in memory (Not the marshaled buffer). The
487     // previous entry indicates the location in the structure of the value.
488     // The second entry of each structure entry indicated the index of the
489     // type associated with the entry. This is an index into the array of
490     // arrays or the union table (merged with the normal table in this
491     // implementation). The final entry is the offset in the unmarshaled
492     // structure where the value is located. This is the offsetof(STRUCTURE,
493     // element). This value is added to the input 'target' or 'source' value
494     // to determine where the value goes.
495     StructMarshal_mst *mst = (StructMarshal_mst *)sel;
496     int i;
497     const UINT16 *value = mst->values;
498 //
499 for(result = TPM_RC_SUCCESS, i = mst->elements
500     ; (TPM_RC_SUCCESS == result) && (i > 0)
501     ; value = &value[3], i--)
502 {
503     UINT16 descriptor = value[0];
504     marshalIndex_t index = value[1];
505     // The offset of the object in the structure is in the last value in
506     // the triplet. Add that value to the start of the structure
507     UINT8 *offset = ((UINT8 *)target) + value[2];
508 //
509 if ((ELEMENT_PROPAGATE & descriptor)
510     && (typeIndex & NULL_FLAG))
511     index |= NULL_FLAG;
512 switch(GET_ELEMENT_TYPE(descriptor))
513 {
514     case SIMPLE_STYPE:
515     {
516         result = Unmarshal(index, offset, buffer, size);
517         break;
518     }
519     case UNION_STYPE:
520     {
521         UINT32 choice;
522         //
523         // Get the selector or array dimension value
524         choice = GetSelector(target, mst->values, descriptor);
525         result = UnmarshalUnion(index, offset, buffer, size, choice);
526         break;
527     }
528     case ARRAY_STYPE:
529     {
530         UINT32 dimension;

```

```

531         //
532         dimension = GetSelector(target, mst->values, descriptor);
533         result = ArrayUnmarshal(index, offset, buffer,
534                                size, dimension);
535         break;
536     }
537     default:
538         result = TPM_RC_FAILURE;
539         break;
540 }
541 }
542 break;
543 }
544 case TPM2B_MTYPE:
545 {
546     // A primitive TPM2B. A size and byte buffer. The single value (other than
547     // the tag) references the synthetic 'interface' value for the size
548     // parameter.
549     Tpm2bMarshal_mst *m2bt = (Tpm2bMarshal_mst *)sel;
550     //
551     if(IS_SUCCESS(Unmarshal(m2bt->sizeIndex, target, buffer, size)))
552         result = UnmarshalBytes(((TPM2B *)target)->buffer,
553                                buffer, size, *((UINT16 *)target));
554     break;
555 }
556 case TPM2BS_MTYPE:
557 {
558     // This is used when a TPM2B contains a structure.
559     Tpm2bsMarshal_mst *m2bst = (Tpm2bsMarshal_mst *)sel;
560     INT32 count;
561     //
562     if(IS_SUCCESS(Unmarshal(m2bst->sizeIndex, target, buffer, size)))
563     {
564         // fetch the size value and convert it to a 32-bit count value
565         count = (int32_t)*((UINT16 *)target);
566         if(count == 0)
567         {
568             if(m2bst->modifiers & SIZE_EQUAL)
569                 result = TPM_RC_SIZE;
570         }
571         else if((*size -= count) >= 0)
572         {
573             marshalIndex_t index = m2bst->dataIndex;
574             //
575             // If this type propagates a null (PROPIGATE_NULL), propagate it
576             if ((m2bst->modifiers & PROPAGATE_NULL)
577                 && (typeIndex & typeIndex))
578                 index |= NULL_FLAG;
579             // The structure might not start two bytes after the start of the
580             // size field. The offset to the start of the structure is between
581             // 2 and 8 bytes. This is encoded into the low 4 bits of the
582             // modifiers byte
583             if(IS_SUCCESS(Unmarshal(index,
584                                   ((UINT8 *)target) + (m2bst->modifiers & OFFSET_MASK),
585                                   buffer, &count)))
586             {
587                 if(count != 0)
588                     result = TPM_RC_SIZE;
589             }
590         }
591         else
592             result = TPM_RC_INSUFFICIENT;
593     }
594     break;
595 }
596 case LIST_MTYPE:

```

```

597     {
598         // Used for a list. A list is a qualified 32-bit 'count' value followed
599         // by a type indicator.
600         ListMarshal_mst *mlt = (ListMarshal_mst *)sel;
601         marshalIndex_t index = mlt->arrayRef;
602         //
603         if(IS_SUCCESS(Unmarshal(mlt->sizeIndex, target, buffer, size)))
604         {
605             // If this type propagates a null (PROPAGATE_NULL), propagate it
606             if ((mlt->modifiers & PROPAGATE_NULL)
607                 && (typeIndex & NULL_FLAG))
608                 index |= NULL_FLAG;
609             result = ArrayUnmarshal(index,
610                                     ((UINT8 *)target) + (mlt->modifiers & OFFSET_MASK),
611                                     buffer, size, *((UINT32 *)target));
612         }
613         break;
614     }
615     case NULL_MTYPE:
616     {
617         result = TPM_RC_SUCCESS;
618         break;
619     }
620 #if 0
621     case COMPOSITE_MTYPE:
622     {
623         CompositeMarshal_mst *mct = (CompositeMarshal_mst *)sel;
624         int i;
625         UINT8 *buf = *buffer;
626         INT32 sz = *size;
627         //
628         result = TPM_RC_VALUE;
629         for(i = GET_ELEMENT_COUNT(mct->modifiers) - 1; i >= 0; i--)
630         {
631             marshalIndex_t index = mct->types[i];
632             //
633             // This type might take a null so set it in each called value, just
634             // in case it is needed in that value. Only one value in each
635             // composite should have the takes null SET.
636             index |= typeIndex & NULL_MASK;
637             result = Unmarshal(index, target, buffer, size);
638             if(result == TPM_RC_SUCCESS)
639                 break;
640             // Each of the composite values does its own unmarshaling. This
641             // has some execution overhead if it is unmarshaled multiple times
642             // but it saves code size in not having to reproduce the various
643             // unmarshaling types that can be in a composite. So, what this means
644             // is that the buffer pointer and size have to be reset for each
645             // unmarshaled value.
646             *buffer = buf;
647             *size = sz;
648         }
649         break;
650     }
651 #endif // 0
652     default:
653     {
654         result = TPM_RC_FAILURE;
655         break;
656     }
657 }
658 return result;
659 }

```


9.10.8.1.1.8 Marshal()

This is the function that drives marshaling of output. Because there is no validation of the output, there is a lot less code.

```

660  UINT16 Marshal(
661      UINT16          typeIndex,          // IN: the thing to marshal
662      void            *source,            // IN: where the data comes from
663      UINT8          **buffer,           // IN/OUT: the data source buffer
664      INT32          *size                // IN/OUT: the remaining size
665  )
666  {
667      #define _source ((UINT8 *)source)
668
669      const MarshalHeader_mst *sel;
670      UINT16                  retVal;
671      //
672      sel = GetDescriptor(typeIndex);
673      switch(sel->marshalType)
674      {
675          case VALUES_MTYPE:
676          case UINT_MTYPE:
677          case TABLE_MTYPE:
678          case MIN_MAX_MTYPE:
679          case ATTRIBUTES_MTYPE:
680          case COMPOSITE_MTYPE:
681              {
682                  #if BIG_ENDIAN_TPM
683                  #define MM16 0
684                  #define MM32 0
685                  #define MM64 0
686                  #else
687                      // These flip the constant index values so that they count in reverse order when doing
688                      // little-endian stuff
689                      #define MM16 1
690                      #define MM32 3
691                      #define MM64 7
692                  #endif
693                  // Just change the name and cast the type of the input parameters for typing purposes
694                  #define mb (*buffer)
695                  #define _source ((UINT8 *)source)
696                  retVal = (1 << (sel->modifiers & SIZE_MASK));
697                  if(buffer != NULL)
698                  {
699                      if((size == NULL) || ((*size -= retVal) >= 0))
700                      {
701                          if(retVal == 4)
702                          {
703                              mb[0 ^ MM32] = _source[0];
704                              mb[1 ^ MM32] = _source[1];
705                              mb[2 ^ MM32] = _source[2];
706                              mb[3 ^ MM32] = _source[3];
707                          }
708                          else if(retVal == 2)
709                          {
710                              mb[0 ^ MM16] = _source[0];
711                              mb[1 ^ MM16] = _source[1];
712                          }
713                          else if(retVal == 1)
714                              mb[0] = _source[0];
715                          else
716                          {
717                              mb[0 ^ MM64] = _source[0];
718                              mb[1 ^ MM64] = _source[1];

```

```

719         mb[2 ^ MM64] = _source[2];
720         mb[3 ^ MM64] = _source[3];
721         mb[4 ^ MM64] = _source[4];
722         mb[5 ^ MM64] = _source[5];
723         mb[6 ^ MM64] = _source[6];
724         mb[7 ^ MM64] = _source[7];
725     }
726     *buffer += retVal;
727 }
728 }
729 break;
730 }
731 case STRUCTURE_MTYPE:
732 {
733     // #define _mst ((StructMarshal_mst *)sel)
734     StructMarshal_mst *mst = ((StructMarshal_mst *)sel);
735     int i;
736     const UINT16 *value = mst->values;
737
738     //
739     for(retVal = 0, i = mst->elements; i > 0; value = &value[3], i--)
740     {
741         UINT16 des = value[0];
742         marshalIndex_t index = value[1];
743         UINT8 *offset = _source + value[2];
744         //
745         switch(GET_ELEMENT_TYPE(des))
746         {
747             case UNION_STYPE:
748             {
749                 UINT32 choice;
750                 //
751                 choice = GetSelector(source, mst->values, des);
752                 retVal += MarshalUnion(index, offset, buffer, size, choice);
753                 break;
754             }
755             case ARRAY_STYPE:
756             {
757                 UINT32 count;
758                 //
759                 count = GetSelector(source, mst->values, des);
760                 retVal += ArrayMarshal(index, offset, buffer, size, count);
761                 break;
762             }
763             case SIMPLE_STYPE:
764             default:
765             {
766                 // This is either another structure or a simple type
767                 retVal += Marshal(index, offset, buffer, size);
768                 break;
769             }
770         }
771     }
772     break;
773 }
774 case TPM2B_MTYPE:
775 {
776     // Get the number of bytes being marshaled
777     INT32 val = (int32_t)*((UINT16 *)source);
778     //
779     retVal = Marshal(UINT16_MARSHAL_REF, source, buffer, size);
780
781     // This is a standard 2B with a byte buffer
782     retVal += MarshalBytes(((TPM2B *)_source)->buffer, buffer, size, val);
783     break;
784 }

```

```

785     case TPM2BS_MTYPE: // A structure in a TPM2B
786     {
787         Tpm2bsMarshal_mst      *m2bst = (Tpm2bsMarshal_mst *)sel;
788         UINT8                  *offset;
789         UINT16                  amount;
790         UINT8                  *marshaledSize;
791     //
792     // Save the address of where the size should go
793     marshaledSize = *buffer;
794
795     // marshal the size (checks the space and advanced the pointer)
796     retVal = Marshal(UINT16_MARSHAL_REF, source, buffer, size);
797
798     // This gets the 'offsetof' the structure to marshal. It was placed in the
799     // modifiers byte because the offset from the start of the TPM2B to the
800     // start of the structure is going to be less than 8 and the modifiers
801     // byte isn't needed for anything else.
802     offset = _source + (m2bst->modifiers & SIGNED_MASK);
803
804     // Marshal the structure and get its size
805     amount = Marshal(m2bst->dataIndex, offset, buffer, size);
806
807     // put the size in the space used when the size was marshaled.
808     if(buffer != NULL)
809         UINT16_TO_BYTE_ARRAY(amount, marshaledSize);
810     retVal += amount;
811     break;
812 }
813 case LIST_MTYPE:
814 {
815     ListMarshal_mst *    mlt = ((ListMarshal_mst *)sel);
816     UINT8                *offset = _source + (mlt->modifiers & SIGNED_MASK);
817     retVal = Marshal(UINT32_MARSHAL_REF, source, buffer, size);
818     retVal += ArrayMarshal((marshalIndex_t)(mlt->arrayRef), offset,
819                             buffer, size, *((UINT32 *)source));
820     break;
821 }
822 case NULL_MTYPE:
823     retVal = 0;
824     break;
825 case ERROR_MTYPE:
826 default:
827 {
828     if(size != NULL)
829         *size = -1;
830     retVal = 0;
831     break;
832 }
833 }
834 return retVal;
835 }
836 }
837 #endif // TABLE_DRIVEN_MARSHAL

```

9.10.8.2 TableMarshalData.c

This file contains the data initializer used for the table-driven marshaling code.

```

1  #include "Tpm.h"
2
3  #if TABLE_DRIVEN_MARSHAL
4  #include "TableMarshal.h"
5  #include "Marshal.h"

```

The array marshaling table

```

6  ArrayMarshal_mst  ArrayLookupTable[] = {
7      ARRAY_MARSHAL_ENTRY(UINT8),
8      ARRAY_MARSHAL_ENTRY(TPM_CC),
9      ARRAY_MARSHAL_ENTRY(TPMA_CC),
10     ARRAY_MARSHAL_ENTRY(TPM_ALG_ID),
11     ARRAY_MARSHAL_ENTRY(TPM_HANDLE),
12     ARRAY_MARSHAL_ENTRY(TPM2B_DIGEST),
13     ARRAY_MARSHAL_ENTRY(TPMT_HA),
14     ARRAY_MARSHAL_ENTRY(TPMS_PCR_SELECTION),
15     ARRAY_MARSHAL_ENTRY(TPMS_ALG_PROPERTY),
16     ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_PROPERTY),
17     ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_PCR_SELECT),
18     ARRAY_MARSHAL_ENTRY(TPM_ECC_CURVE),
19     ARRAY_MARSHAL_ENTRY(TPMS_TAGGED_POLICY),
20     ARRAY_MARSHAL_ENTRY(TPMS_ACT_DATA),
21     ARRAY_MARSHAL_ENTRY(TPMS_AC_OUTPUT)};

```

The main marshaling structure

```

22  MarshalData_st MarshalData = {
23      // UINT8_DATA
24      {UINT_MTYPE, 0},
25      // UINT16_DATA
26      {UINT_MTYPE, 1},
27      // UINT32_DATA
28      {UINT_MTYPE, 2},
29      // UINT64_DATA
30      {UINT_MTYPE, 3},
31      // INT8_DATA
32      {UINT_MTYPE, 0 + IS_SIGNED},
33      // INT16_DATA
34      {UINT_MTYPE, 1 + IS_SIGNED},
35      // INT32_DATA
36      {UINT_MTYPE, 2 + IS_SIGNED},
37      // INT64_DATA
38      {UINT_MTYPE, 3 + IS_SIGNED},
39      // UINT0_DATA
40      {NULL_MTYPE, 0},
41      // TPM_ECC_CURVE_DATA
42      {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_CURVE,
43          {TPM_ECC_NONE,
44              RANGE(1, 32, UINT16),
45              (UINT32)((ECC_NIST_P192 << 0) | (ECC_NIST_P224 << 1) | (ECC_NIST_P256 << 2) |
46                  (ECC_NIST_P384 << 3) | (ECC_NIST_P521 << 4) | (ECC_BN_P256 << 15) |
47                  (ECC_BN_P638 << 16) | (ECC_SM2_P256 << 31))}},
48      // TPM_CLOCK_ADJUST_DATA
49      {MIN_MAX_MTYPE, ONE_BYTES|IS_SIGNED, (UINT8)TPM_RC_VALUE,
50          {RANGE(TPM_CLOCK_COARSE_SLOWER, TPM_CLOCK_COARSE_FASTER, INT8)}}},
51      // TPM_EO_DATA
52      {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE,
53          {RANGE(TPM_EO_EQ, TPM_EO_BITCLEAR, UINT16)}}},
54      // TPM_SU_DATA
55      {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 2,
56          {TPM_SU_CLEAR, TPM_SU_STATE}},
57      // TPM_SE_DATA
58      {TABLE_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE, 3,
59          {TPM_SE_HMAC, TPM_SE_POLICY, TPM_SE_TRIAL}},
60      // TPM_CAP_DATA
61      {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 1,
62          {RANGE(TPM_CAP_ALGS, TPM_CAP_ACT, UINT32),
63              TPM_CAP_VENDOR_PROPERTY}},
64      // TPMA_ALGORITHM_DATA
65      {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFFFF8F0},

```

```

66 // TPMA_OBJECT_DATA
67 {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFF0F309},
68 // TPMA_SESSION_DATA
69 {ATTRIBUTES_MTYPE, ONE_BYTES, 0x00000018},
70 // TPMA_ACT_DATA
71 {ATTRIBUTES_MTYPE, FOUR_BYTES, 0xFFFFFFFFFC},
72 // TPMI_YES_NO_DATA
73 {TABLE_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE, 2,
74  {NO, YES}},
75 // TPMI_DH_OBJECT_DATA
76 {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 0,
77  {TPM_RH_NULL,
78   RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
79   RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32)}},
80 // TPMI_DH_PARENT_DATA
81 {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 3,
82  {TPM_RH_NULL,
83   RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
84   RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32),
85   TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
86 // TPMI_DH_PERSISTENT_DATA
87 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
88  {RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32)}},
89 // TPMI_DH_ENTITY_DATA
90 {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 5, 4,
91  {TPM_RH_NULL,
92   RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32),
93   RANGE(PERSISTENT_FIRST, PERSISTENT_LAST, UINT32),
94   RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32),
95   RANGE(PCR_FIRST, PCR_LAST, UINT32),
96   RANGE(TPM_RH_AUTH_00, TPM_RH_AUTH_FF, UINT32),
97   TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
98 // TPMI_DH_PCR_DATA
99 {MIN_MAX_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE,
100  {TPM_RH_NULL,
101   RANGE(PCR_FIRST, PCR_LAST, UINT32)}},
102 // TPMI_SH_AUTH_SESSION_DATA
103 {VALUES_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 2, 0,
104  {TPM_RS_PW,
105   RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
106   RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32)}},
107 // TPMI_SH_HMAC_DATA
108 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
109  {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32)}},
110 // TPMI_SH_POLICY_DATA
111 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
112  {RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32)}},
113 // TPMI_DH_CONTEXT_DATA
114 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 3, 0,
115  {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
116   RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32),
117   RANGE(TRANSIENT_FIRST, TRANSIENT_LAST, UINT32)}},
118 // TPMI_DH_SAVED_DATA
119 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2, 3,
120  {RANGE(HMAC_SESSION_FIRST, HMAC_SESSION_LAST, UINT32),
121   RANGE(POLICY_SESSION_FIRST, POLICY_SESSION_LAST, UINT32),
122   0x80000000, 0x80000001, 0x80000002}},
123 // TPMI_RH_HIERARCHY_DATA
124 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 3,
125  {TPM_RH_NULL,
126   TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
127 // TPMI_RH_ENABLES_DATA
128 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 4,
129  {TPM_RH_NULL,
130   TPM_RH_OWNER, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM, TPM_RH_PLATFORM_NV}},
131 // TPMI_RH_HIERARCHY_AUTH_DATA

```

```

132 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 4,
133   {TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
134 // TPMI_RH_HIERARCHY_POLICY_DATA
135 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 4,
136   {RANGE(TPM_RH_ACT_0, TPM_RH_ACT_F, UINT32),
137     TPM_RH_OWNER, TPM_RH_LOCKOUT, TPM_RH_ENDORSEMENT, TPM_RH_PLATFORM}},
138 // TPMI_RH_PLATFORM_DATA
139 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1,
140   {TPM_RH_PLATFORM}},
141 // TPMI_RH_OWNER_DATA
142 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 1,
143   {TPM_RH_NULL,
144     TPM_RH_OWNER}},
145 // TPMI_RH_ENDORSEMENT_DATA
146 {TABLE_MTYPE, FOUR_BYTES|TAKES_NULL, (UINT8)TPM_RC_VALUE, 1,
147   {TPM_RH_NULL,
148     TPM_RH_ENDORSEMENT}},
149 // TPMI_RH_PROVISION_DATA
150 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2,
151   {TPM_RH_OWNER, TPM_RH_PLATFORM}},
152 // TPMI_RH_CLEAR_DATA
153 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 2,
154   {TPM_RH_LOCKOUT, TPM_RH_PLATFORM}},
155 // TPMI_RH_NV_AUTH_DATA
156 {VALUES_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1, 2,
157   {RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32),
158     TPM_RH_OWNER, TPM_RH_PLATFORM}},
159 // TPMI_RH_LOCKOUT_DATA
160 {TABLE_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE, 1,
161   {TPM_RH_LOCKOUT}},
162 // TPMI_RH_NV_INDEX_DATA
163 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
164   {RANGE(NV_INDEX_FIRST, NV_INDEX_LAST, UINT32)}},
165 // TPMI_RH_AC_DATA
166 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
167   {RANGE(AC_FIRST, AC_LAST, UINT32)}},
168 // TPMI_RH_ACT_DATA
169 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_VALUE,
170   {RANGE(TPM_RH_ACT_0, TPM_RH_ACT_F, UINT32)}},
171 // TPMI_ALG_HASH_DATA
172 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_HASH,
173   {TPM_ALG_NULL,
174     RANGE(4, 41, UINT16),
175     (UINT32)((ALG_SHA1 << 0) | (ALG_SHA256 << 7) | (ALG_SHA384 << 8) |
176       (ALG_SHA512 << 9) | (ALG_SM3_256 << 14)),
177     (UINT32)((ALG_SHA3_256 << 3) | (ALG_SHA3_384 << 4) | (ALG_SHA3_512 << 5))}},
178 // TPMI_ALG_ASYM_DATA
179 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_ASYMMETRIC,
180   {TPM_ALG_NULL,
181     RANGE(1, 35, UINT16),
182     (UINT32)((ALG_RSA << 0)),
183     (UINT32)((ALG_ECC << 2))}},
184 // TPMI_ALG_SYM_DATA
185 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
186   {TPM_ALG_NULL,
187     RANGE(3, 38, UINT16),
188     (UINT32)((ALG_TDES << 0) | (ALG_AES << 3) | (ALG_XOR << 7) | (ALG_SM4 << 16)),
189     (UINT32)((ALG_CAMELLIA << 3))}},
190 // TPMI_ALG_SYM_OBJECT_DATA
191 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
192   {TPM_ALG_NULL,
193     RANGE(3, 38, UINT16),
194     (UINT32)((ALG_TDES << 0) | (ALG_AES << 3) | (ALG_SM4 << 16)),
195     (UINT32)((ALG_CAMELLIA << 3))}},
196 // TPMI_ALG_SYM_MODE_DATA
197 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_MODE,

```



```

198     {TPM_ALG_NULL,
199     RANGE(63, 68, UINT16),
200     (UINT32)((ALG_CMAC << 0) | (ALG_CTR << 1) | (ALG_OFB << 2) |
201     (ALG_CBC << 3) | (ALG_CFB << 4) | (ALG_ECB << 5))}},
202 // TPMI_ALG_KDF_DATA
203 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_KDF,
204     {TPM_ALG_NULL,
205     RANGE(7, 34, UINT16),
206     (UINT32)((ALG_MGF1 << 0) | (ALG_KDF1_SP800_56A << 25) |
207     (ALG_KDF2 << 26) | (ALG_KDF1_SP800_108 << 27))}},
208 // TPMI_ALG_SIG_SCHEME_DATA
209 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
210     {TPM_ALG_NULL,
211     RANGE(5, 28, UINT16),
212     (UINT32)((ALG_HMAC << 0) | (ALG_RSASSA << 15) | (ALG_RSAPSS << 17) |
213     (ALG_ECDSA << 19) | (ALG_ECDSA << 21) | (ALG_SM2 << 22) |
214     (ALG_ECSCNORR << 23))}},
215 // TPMI_ECC_KEY_EXCHANGE_DATA
216 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SCHEME,
217     {TPM_ALG_NULL,
218     RANGE(25, 29, UINT16),
219     (UINT32)((ALG_ECDH << 0) | (ALG_SM2 << 2) | (ALG_ECMQV << 4))}},
220 // TPMI_ST_COMMAND_TAG_DATA
221 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_BAD_TAG, 2,
222     {TPM_ST_NO_SESSIONS, TPM_ST_SESSIONS}},
223 // TPMI_ALG_MAC_SCHEME_DATA
224 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_SYMMETRIC,
225     {TPM_ALG_NULL,
226     RANGE(4, 63, UINT16),
227     (UINT32)((ALG_SHA1 << 0) | (ALG_SHA256 << 7) | (ALG_SHA384 << 8) |
228     (ALG_SHA512 << 9) | (ALG_SM3_256 << 14)),
229     (UINT32)((ALG_SHA3_256 << 3) | (ALG_SHA3_384 << 4) | (ALG_SHA3_512 << 5) |
230     (ALG_CMAC << 27))}},
231 // TPMI_ALG_CIPHER_MODE_DATA
232 {MIN_MAX_MTYPE, TWO_BYTES|TAKES_NULL|HAS_BITS, (UINT8)TPM_RC_MODE,
233     {TPM_ALG_NULL,
234     RANGE(64, 68, UINT16),
235     (UINT32)((ALG_CTR << 0) | (ALG_OFB << 1) | (ALG_CBC << 2) | (ALG_CFB << 3) |
236     (ALG_ECB << 4))}},
237 // TPMS_EMPTY_DATA
238 {STRUCTURE_MTYPE, 1,
239     {SET_ELEMENT_TYPE(SIMPLE_TYPE), UINT0_MARSHAL_REF, 0}},
240 // TPMS_ALGORITHM_DESCRIPTION_DATA
241 {STRUCTURE_MTYPE, 2, {
242     SET_ELEMENT_TYPE(SIMPLE_TYPE)|SET_ELEMENT_SIZE(TWO_BYTES),
243     TPM_ALG_ID_MARSHAL_REF,
244     (UINT16)(offsetof(TPMS_ALGORITHM_DESCRIPTION, alg)),
245     SET_ELEMENT_TYPE(SIMPLE_TYPE)|SET_ELEMENT_SIZE(FOUR_BYTES),
246     TPMA_ALGORITHM_MARSHAL_REF,
247     (UINT16)(offsetof(TPMS_ALGORITHM_DESCRIPTION, attributes))}},
248 // TPMU_HA_DATA
249 {9, IS_ARRAY_UNION, (UINT16)(offsetof(TPMU_HA_mst, marshalingTypes)),
250     {(UINT32)TPM_ALG_SHA1, (UINT32)TPM_ALG_SHA256, (UINT32)TPM_ALG_SHA384,
251     (UINT32)TPM_ALG_SHA512, (UINT32)TPM_ALG_SM3_256, (UINT32)TPM_ALG_SHA3_256,
252     (UINT32)TPM_ALG_SHA3_384, (UINT32)TPM_ALG_SHA3_512, (UINT32)TPM_ALG_NULL},
253     {(UINT16)(SHA1_DIGEST_SIZE), (UINT16)(SHA256_DIGEST_SIZE),
254     (UINT16)(SHA384_DIGEST_SIZE), (UINT16)(SHA512_DIGEST_SIZE),
255     (UINT16)(SM3_256_DIGEST_SIZE), (UINT16)(SHA3_256_DIGEST_SIZE),
256     (UINT16)(SHA3_384_DIGEST_SIZE), (UINT16)(SHA3_512_DIGEST_SIZE),
257     (UINT16)(0)}
258 },
259 // TPMT_HA_DATA
260 {STRUCTURE_MTYPE, 2, {
261     SET_ELEMENT_TYPE(SIMPLE_TYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
262     TPMI_ALG_HASH_MARSHAL_REF,
263     (UINT16)(offsetof(TPMT_HA, hashAlg)),

```



```

264     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
265     TPMU_HA_MARSHAL_REF,
266     (UINT16) (offsetof(TPMT_HA, digest))},
267 // TPM2B_DIGEST_DATA
268 {TPM2B_MTYPE, Type00_MARSHAL_REF},
269 // TPM2B_DATA_DATA
270 {TPM2B_MTYPE, Type01_MARSHAL_REF},
271 // TPM2B_EVENT_DATA
272 {TPM2B_MTYPE, Type02_MARSHAL_REF},
273 // TPM2B_MAX_BUFFER_DATA
274 {TPM2B_MTYPE, Type03_MARSHAL_REF},
275 // TPM2B_MAX_NV_BUFFER_DATA
276 {TPM2B_MTYPE, Type04_MARSHAL_REF},
277 // TPM2B_TIMEOUT_DATA
278 {TPM2B_MTYPE, Type05_MARSHAL_REF},
279 // TPM2B_IV_DATA
280 {TPM2B_MTYPE, Type06_MARSHAL_REF},
281 // NULL_UNION_DATA
282 {0},
283 // TPM2B_NAME_DATA
284 {TPM2B_MTYPE, Type07_MARSHAL_REF},
285 // TPMS_PCR_SELECT_DATA
286 {STRUCTURE_MTYPE, 2, {
287     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
288     Type08_MARSHAL_REF,
289     (UINT16) (offsetof(TPMS_PCR_SELECT, sizeofSelect)),
290     SET_ELEMENT_TYPE(ARRAY_STYPE) | SET_ELEMENT_NUMBER(0),
291     UINT8_ARRAY_MARSHAL_INDEX,
292     (UINT16) (offsetof(TPMS_PCR_SELECT, pcrSelect))}},
293 // TPMS_PCR_SELECTION_DATA
294 {STRUCTURE_MTYPE, 3, {
295     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
296     TPMI_ALG_HASH_MARSHAL_REF,
297     (UINT16) (offsetof(TPMS_PCR_SELECTION, hash)),
298     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
299     Type08_MARSHAL_REF,
300     (UINT16) (offsetof(TPMS_PCR_SELECTION, sizeofSelect)),
301     SET_ELEMENT_TYPE(ARRAY_STYPE) | SET_ELEMENT_NUMBER(1),
302     UINT8_ARRAY_MARSHAL_INDEX,
303     (UINT16) (offsetof(TPMS_PCR_SELECTION, pcrSelect))}},
304 // TPMT_TK_CREATION_DATA
305 {STRUCTURE_MTYPE, 3, {
306     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
307     Type10_MARSHAL_REF,
308     (UINT16) (offsetof(TPMT_TK_CREATION, tag)),
309     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
310     TPMI_RH_HIERARCHY_MARSHAL_REF | NULL_FLAG,
311     (UINT16) (offsetof(TPMT_TK_CREATION, hierarchy)),
312     SET_ELEMENT_TYPE(SIMPLE_STYPE),
313     TPM2B_DIGEST_MARSHAL_REF,
314     (UINT16) (offsetof(TPMT_TK_CREATION, digest))}},
315 // TPMT_TK_VERIFIED_DATA
316 {STRUCTURE_MTYPE, 3, {
317     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
318     Type11_MARSHAL_REF,
319     (UINT16) (offsetof(TPMT_TK_VERIFIED, tag)),
320     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
321     TPMI_RH_HIERARCHY_MARSHAL_REF | NULL_FLAG,
322     (UINT16) (offsetof(TPMT_TK_VERIFIED, hierarchy)),
323     SET_ELEMENT_TYPE(SIMPLE_STYPE),
324     TPM2B_DIGEST_MARSHAL_REF,
325     (UINT16) (offsetof(TPMT_TK_VERIFIED, digest))}},
326 // TPMT_TK_AUTH_DATA
327 {STRUCTURE_MTYPE, 3, {
328     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
329     Type12_MARSHAL_REF,

```

```

330         (UINT16) (offsetof(TPMT_TK_AUTH, tag)),
331     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
332     TPMI_RH_HIERARCHY_MARSHAL_REF | NULL_FLAG,
333     (UINT16) (offsetof(TPMT_TK_AUTH, hierarchy)),
334     SET_ELEMENT_TYPE(SIMPLE_STYPE),
335     TPM2B_DIGEST_MARSHAL_REF,
336     (UINT16) (offsetof(TPMT_TK_AUTH, digest))}},
337 // TPMT_TK_HASHCHECK_DATA
338 {STRUCTURE_MTYPE, 3, {
339     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
340     Type13_MARSHAL_REF,
341     (UINT16) (offsetof(TPMT_TK_HASHCHECK, tag)),
342     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
343     TPMI_RH_HIERARCHY_MARSHAL_REF | NULL_FLAG,
344     (UINT16) (offsetof(TPMT_TK_HASHCHECK, hierarchy)),
345     SET_ELEMENT_TYPE(SIMPLE_STYPE),
346     TPM2B_DIGEST_MARSHAL_REF,
347     (UINT16) (offsetof(TPMT_TK_HASHCHECK, digest))}},
348 // TPMS_ALG_PROPERTY_DATA
349 {STRUCTURE_MTYPE, 2, {
350     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
351     TPM_ALG_ID_MARSHAL_REF,
352     (UINT16) (offsetof(TPMS_ALG_PROPERTY, alg)),
353     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
354     TPMA_ALGORITHM_MARSHAL_REF,
355     (UINT16) (offsetof(TPMS_ALG_PROPERTY, algProperties))}},
356 // TPMS_TAGGED_PROPERTY_DATA
357 {STRUCTURE_MTYPE, 2, {
358     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
359     TPM_PT_MARSHAL_REF,
360     (UINT16) (offsetof(TPMS_TAGGED_PROPERTY, property)),
361     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
362     UINT32_MARSHAL_REF,
363     (UINT16) (offsetof(TPMS_TAGGED_PROPERTY, value))}},
364 // TPMS_TAGGED_PCR_SELECT_DATA
365 {STRUCTURE_MTYPE, 3, {
366     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
367     TPM_PT_PCR_MARSHAL_REF,
368     (UINT16) (offsetof(TPMS_TAGGED_PCR_SELECT, tag)),
369     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
370     Type08_MARSHAL_REF,
371     (UINT16) (offsetof(TPMS_TAGGED_PCR_SELECT, sizeofSelect)),
372     SET_ELEMENT_TYPE(ARRAY_STYPE) | SET_ELEMENT_NUMBER(1),
373     UINT8_ARRAY_MARSHAL_INDEX,
374     (UINT16) (offsetof(TPMS_TAGGED_PCR_SELECT, pcrSelect))}},
375 // TPMS_TAGGED_POLICY_DATA
376 {STRUCTURE_MTYPE, 2, {
377     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
378     TPM_HANDLE_MARSHAL_REF,
379     (UINT16) (offsetof(TPMS_TAGGED_POLICY, handle)),
380     SET_ELEMENT_TYPE(SIMPLE_STYPE),
381     TPMT_HA_MARSHAL_REF,
382     (UINT16) (offsetof(TPMS_TAGGED_POLICY, policyHash))}},
383 // TPMS_ACT_DATA_DATA
384 {STRUCTURE_MTYPE, 3, {
385     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
386     TPM_HANDLE_MARSHAL_REF,
387     (UINT16) (offsetof(TPMS_ACT_DATA, handle)),
388     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
389     UINT32_MARSHAL_REF,
390     (UINT16) (offsetof(TPMS_ACT_DATA, timeout)),
391     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
392     TPMA_ACT_MARSHAL_REF,
393     (UINT16) (offsetof(TPMS_ACT_DATA, attributes))}},
394 // TPML_CC_DATA
395 {LIST_MTYPE,

```

```

396     (UINT8) (offsetof(TPML_CC, commandCodes)),
397     Type15_MARSHAL_REF,
398     TPM_CC_ARRAY_MARSHAL_INDEX},
399 // TPML_CCA_DATA
400 {LIST_MTYPE,
401     (UINT8) (offsetof(TPML_CCA, commandAttributes)),
402     Type15_MARSHAL_REF,
403     TPMA_CC_ARRAY_MARSHAL_INDEX},
404 // TPML_ALG_DATA
405 {LIST_MTYPE,
406     (UINT8) (offsetof(TPML_ALG, algorithms)),
407     Type17_MARSHAL_REF,
408     TPM_ALG_ID_ARRAY_MARSHAL_INDEX},
409 // TPML_HANDLE_DATA
410 {LIST_MTYPE,
411     (UINT8) (offsetof(TPML_HANDLE, handle)),
412     Type18_MARSHAL_REF,
413     TPM_HANDLE_ARRAY_MARSHAL_INDEX},
414 // TPML_DIGEST_DATA
415 {LIST_MTYPE,
416     (UINT8) (offsetof(TPML_DIGEST, digests)),
417     Type19_MARSHAL_REF,
418     TPM2B_DIGEST_ARRAY_MARSHAL_INDEX},
419 // TPML_DIGEST_VALUES_DATA
420 {LIST_MTYPE,
421     (UINT8) (offsetof(TPML_DIGEST_VALUES, digests)),
422     Type20_MARSHAL_REF,
423     TPMT_HA_ARRAY_MARSHAL_INDEX},
424 // TPML_PCR_SELECTION_DATA
425 {LIST_MTYPE,
426     (UINT8) (offsetof(TPML_PCR_SELECTION, pcrSelections)),
427     Type20_MARSHAL_REF,
428     TPMS_PCR_SELECTION_ARRAY_MARSHAL_INDEX},
429 // TPML_ALG_PROPERTY_DATA
430 {LIST_MTYPE,
431     (UINT8) (offsetof(TPML_ALG_PROPERTY, algProperties)),
432     Type22_MARSHAL_REF,
433     TPMS_ALG_PROPERTY_ARRAY_MARSHAL_INDEX},
434 // TPML_TAGGED_TPM_PROPERTY_DATA
435 {LIST_MTYPE,
436     (UINT8) (offsetof(TPML_TAGGED_TPM_PROPERTY, tpmProperty)),
437     Type23_MARSHAL_REF,
438     TPMS_TAGGED_PROPERTY_ARRAY_MARSHAL_INDEX},
439 // TPML_TAGGED_PCR_PROPERTY_DATA
440 {LIST_MTYPE,
441     (UINT8) (offsetof(TPML_TAGGED_PCR_PROPERTY, pcrProperty)),
442     Type24_MARSHAL_REF,
443     TPMS_TAGGED_PCR_SELECT_ARRAY_MARSHAL_INDEX},
444 // TPML_ECC_CURVE_DATA
445 {LIST_MTYPE,
446     (UINT8) (offsetof(TPML_ECC_CURVE, eccCurves)),
447     Type25_MARSHAL_REF,
448     TPM_ECC_CURVE_ARRAY_MARSHAL_INDEX},
449 // TPML_TAGGED_POLICY_DATA
450 {LIST_MTYPE,
451     (UINT8) (offsetof(TPML_TAGGED_POLICY, policies)),
452     Type26_MARSHAL_REF,
453     TPMS_TAGGED_POLICY_ARRAY_MARSHAL_INDEX},
454 // TPML_ACT_DATA_DATA
455 {LIST_MTYPE,
456     (UINT8) (offsetof(TPML_ACT_DATA, actData)),
457     Type27_MARSHAL_REF,
458     TPMS_ACT_DATA_ARRAY_MARSHAL_INDEX},
459 // TPMU_CAPABILITIES_DATA
460 {11, 0, (UINT16) (offsetof(TPMU_CAPABILITIES_mst, marshalingTypes)),
461     {(UINT32) TPM_CAP_ALGS, (UINT32) TPM_CAP_HANDLES,

```

```

462     (UINT32)TPM_CAP_COMMANDS,          (UINT32)TPM_CAP_PP_COMMANDS,
463     (UINT32)TPM_CAP_AUDIT_COMMANDS,    (UINT32)TPM_CAP_PCERS,
464     (UINT32)TPM_CAP_TPM_PROPERTIES,    (UINT32)TPM_CAP_PCR_PROPERTIES,
465     (UINT32)TPM_CAP_ECC_CURVES,        (UINT32)TPM_CAP_AUTH_POLICIES,
466     (UINT32)TPM_CAP_ACT},
467     { (UINT16) (TPML_ALG_PROPERTY_MARSHAL_REF),
468       (UINT16) (TPML_HANDLE_MARSHAL_REF),
469       (UINT16) (TPML_CCA_MARSHAL_REF),
470       (UINT16) (TPML_CC_MARSHAL_REF),
471       (UINT16) (TPML_CC_MARSHAL_REF),
472       (UINT16) (TPML_PCR_SELECTION_MARSHAL_REF),
473       (UINT16) (TPML_TAGGED_TPM_PROPERTY_MARSHAL_REF),
474       (UINT16) (TPML_TAGGED_PCR_PROPERTY_MARSHAL_REF),
475       (UINT16) (TPML_ECC_CURVE_MARSHAL_REF),
476       (UINT16) (TPML_TAGGED_POLICY_MARSHAL_REF),
477       (UINT16) (TPML_ACT_DATA_MARSHAL_REF) }
478 },
479 // TPMS_CAPABILITY_DATA_DATA
480 {STRUCTURE_MTYPE, 2, {
481     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
482     TPM_CAP_MARSHAL_REF,
483     (UINT16) (offsetof(TPMS_CAPABILITY_DATA, capability)),
484     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
485     TPMU_CAPABILITIES_MARSHAL_REF,
486     (UINT16) (offsetof(TPMS_CAPABILITY_DATA, data))}},
487 // TPMS_CLOCK_INFO_DATA
488 {STRUCTURE_MTYPE, 4, {
489     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
490     UINT64_MARSHAL_REF,
491     (UINT16) (offsetof(TPMS_CLOCK_INFO, clock)),
492     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
493     UINT32_MARSHAL_REF,
494     (UINT16) (offsetof(TPMS_CLOCK_INFO, resetCount)),
495     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
496     UINT32_MARSHAL_REF,
497     (UINT16) (offsetof(TPMS_CLOCK_INFO, restartCount)),
498     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
499     TPMI_YES_NO_MARSHAL_REF,
500     (UINT16) (offsetof(TPMS_CLOCK_INFO, safe))}},
501 // TPMS_TIME_INFO_DATA
502 {STRUCTURE_MTYPE, 2, {
503     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
504     UINT64_MARSHAL_REF,
505     (UINT16) (offsetof(TPMS_TIME_INFO, time)),
506     SET_ELEMENT_TYPE(SIMPLE_STYPE),
507     TPMS_CLOCK_INFO_MARSHAL_REF,
508     (UINT16) (offsetof(TPMS_TIME_INFO, clockInfo))}},
509 // TPMS_TIME_ATTEST_INFO_DATA
510 {STRUCTURE_MTYPE, 2, {
511     SET_ELEMENT_TYPE(SIMPLE_STYPE),
512     TPMS_TIME_INFO_MARSHAL_REF,
513     (UINT16) (offsetof(TPMS_TIME_ATTEST_INFO, time)),
514     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
515     UINT64_MARSHAL_REF,
516     (UINT16) (offsetof(TPMS_TIME_ATTEST_INFO, firmwareVersion))}},
517 // TPMS_CERTIFY_INFO_DATA
518 {STRUCTURE_MTYPE, 2, {
519     SET_ELEMENT_TYPE(SIMPLE_STYPE),
520     TPM2B_NAME_MARSHAL_REF,
521     (UINT16) (offsetof(TPMS_CERTIFY_INFO, name)),
522     SET_ELEMENT_TYPE(SIMPLE_STYPE),
523     TPM2B_NAME_MARSHAL_REF,
524     (UINT16) (offsetof(TPMS_CERTIFY_INFO, qualifiedName))}},
525 // TPMS_QUOTE_INFO_DATA
526 {STRUCTURE_MTYPE, 2, {
527     SET_ELEMENT_TYPE(SIMPLE_STYPE),

```

```

528         TPML_PCR_SELECTION_MARSHAL_REF,
529         (UINT16) (offsetof(TPMS_QUOTE_INFO, pcrSelect)),
530     SET_ELEMENT_TYPE(SIMPLE_STYPE),
531     TPM2B_DIGEST_MARSHAL_REF,
532     (UINT16) (offsetof(TPMS_QUOTE_INFO, pcrDigest))}},
533 // TPMS_COMMAND_AUDIT_INFO_DATA
534 {STRUCTURE_MTYPE, 4, {
535     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
536     UINT64_MARSHAL_REF,
537     (UINT16) (offsetof(TPMS_COMMAND_AUDIT_INFO, auditCounter)),
538     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
539     TPM_ALG_ID_MARSHAL_REF,
540     (UINT16) (offsetof(TPMS_COMMAND_AUDIT_INFO, digestAlg)),
541     SET_ELEMENT_TYPE(SIMPLE_STYPE),
542     TPM2B_DIGEST_MARSHAL_REF,
543     (UINT16) (offsetof(TPMS_COMMAND_AUDIT_INFO, auditDigest)),
544     SET_ELEMENT_TYPE(SIMPLE_STYPE),
545     TPM2B_DIGEST_MARSHAL_REF,
546     (UINT16) (offsetof(TPMS_COMMAND_AUDIT_INFO, commandDigest))}},
547 // TPMS_SESSION_AUDIT_INFO_DATA
548 {STRUCTURE_MTYPE, 2, {
549     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
550     TPMI_YES_NO_MARSHAL_REF,
551     (UINT16) (offsetof(TPMS_SESSION_AUDIT_INFO, exclusiveSession)),
552     SET_ELEMENT_TYPE(SIMPLE_STYPE),
553     TPM2B_DIGEST_MARSHAL_REF,
554     (UINT16) (offsetof(TPMS_SESSION_AUDIT_INFO, sessionDigest))}},
555 // TPMS_CREATION_INFO_DATA
556 {STRUCTURE_MTYPE, 2, {
557     SET_ELEMENT_TYPE(SIMPLE_STYPE),
558     TPM2B_NAME_MARSHAL_REF,
559     (UINT16) (offsetof(TPMS_CREATION_INFO, objectName)),
560     SET_ELEMENT_TYPE(SIMPLE_STYPE),
561     TPM2B_DIGEST_MARSHAL_REF,
562     (UINT16) (offsetof(TPMS_CREATION_INFO, creationHash))}},
563 // TPMS_NV_CERTIFY_INFO_DATA
564 {STRUCTURE_MTYPE, 3, {
565     SET_ELEMENT_TYPE(SIMPLE_STYPE),
566     TPM2B_NAME_MARSHAL_REF,
567     (UINT16) (offsetof(TPMS_NV_CERTIFY_INFO, indexName)),
568     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
569     UINT16_MARSHAL_REF,
570     (UINT16) (offsetof(TPMS_NV_CERTIFY_INFO, offset)),
571     SET_ELEMENT_TYPE(SIMPLE_STYPE),
572     TPM2B_MAX_NV_BUFFER_MARSHAL_REF,
573     (UINT16) (offsetof(TPMS_NV_CERTIFY_INFO, nvContents))}},
574 // TPMS_NV_DIGEST_CERTIFY_INFO_DATA
575 {STRUCTURE_MTYPE, 2, {
576     SET_ELEMENT_TYPE(SIMPLE_STYPE),
577     TPM2B_NAME_MARSHAL_REF,
578     (UINT16) (offsetof(TPMS_NV_DIGEST_CERTIFY_INFO, indexName)),
579     SET_ELEMENT_TYPE(SIMPLE_STYPE),
580     TPM2B_DIGEST_MARSHAL_REF,
581     (UINT16) (offsetof(TPMS_NV_DIGEST_CERTIFY_INFO, nvDigest))}},
582 // TPMI_ST_ATTEST_DATA
583 {VALUES_MTYPE, TWO_BYTES, (UINT8) TPM_RC_VALUE, 1, 1,
584     {RANGE(TPM_ST_ATTEST_NV, TPM_ST_ATTEST_CREATION, UINT16),
585     TPM_ST_ATTEST_NV_DIGEST}},
586 // TPMU_ATTEST_DATA
587 {8, 0, (UINT16) (offsetof(TPMU_ATTEST_mst, marshalingTypes)),
588     {(UINT32) TPM_ST_ATTEST_CERTIFY, (UINT32) TPM_ST_ATTEST_CREATION,
589     (UINT32) TPM_ST_ATTEST_QUOTE, (UINT32) TPM_ST_ATTEST_COMMAND_AUDIT,
590     (UINT32) TPM_ST_ATTEST_SESSION_AUDIT, (UINT32) TPM_ST_ATTEST_TIME,
591     (UINT32) TPM_ST_ATTEST_NV, (UINT32) TPM_ST_ATTEST_NV_DIGEST},
592     {(UINT16) (TPMS_CERTIFY_INFO_MARSHAL_REF),
593     (UINT16) (TPMS_CREATION_INFO_MARSHAL_REF)},

```



```

594     (UINT16) (TPMS_QUOTE_INFO_MARSHAL_REF),
595     (UINT16) (TPMS_COMMAND_AUDIT_INFO_MARSHAL_REF),
596     (UINT16) (TPMS_SESSION_AUDIT_INFO_MARSHAL_REF),
597     (UINT16) (TPMS_TIME_ATTEST_INFO_MARSHAL_REF),
598     (UINT16) (TPMS_NV_CERTIFY_INFO_MARSHAL_REF),
599     (UINT16) (TPMS_NV_DIGEST_CERTIFY_INFO_MARSHAL_REF)}
600 },
601 // TPMS_ATTEST_DATA
602 {STRUCTURE_MTYPE, 7, {
603     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
604     TPM_CONSTANTS32_MARSHAL_REF,
605     (UINT16) (offsetof(TPMS_ATTEST, magic)),
606     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
607     TPMI_ST_ATTEST_MARSHAL_REF,
608     (UINT16) (offsetof(TPMS_ATTEST, type)),
609     SET_ELEMENT_TYPE(SIMPLE_STYPE),
610     TPM2B_NAME_MARSHAL_REF,
611     (UINT16) (offsetof(TPMS_ATTEST, qualifiedSigner)),
612     SET_ELEMENT_TYPE(SIMPLE_STYPE),
613     TPM2B_DATA_MARSHAL_REF,
614     (UINT16) (offsetof(TPMS_ATTEST, extraData)),
615     SET_ELEMENT_TYPE(SIMPLE_STYPE),
616     TPMS_CLOCK_INFO_MARSHAL_REF,
617     (UINT16) (offsetof(TPMS_ATTEST, clockInfo)),
618     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
619     UINT64_MARSHAL_REF,
620     (UINT16) (offsetof(TPMS_ATTEST, firmwareVersion)),
621     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(1),
622     TPMU_ATTEST_MARSHAL_REF,
623     (UINT16) (offsetof(TPMS_ATTEST, attested))}},
624 // TPM2B_ATTEST_DATA
625 {TPM2B_MTYPE, Type28_MARSHAL_REF},
626 // TPMS_AUTH_COMMAND_DATA
627 {STRUCTURE_MTYPE, 4, {
628     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
629     TPMI_SH_AUTH_SESSION_MARSHAL_REF | NULL_FLAG,
630     (UINT16) (offsetof(TPMS_AUTH_COMMAND, sessionHandle)),
631     SET_ELEMENT_TYPE(SIMPLE_STYPE),
632     TPM2B_NONCE_MARSHAL_REF,
633     (UINT16) (offsetof(TPMS_AUTH_COMMAND, nonce)),
634     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
635     TPMA_SESSION_MARSHAL_REF,
636     (UINT16) (offsetof(TPMS_AUTH_COMMAND, sessionAttributes)),
637     SET_ELEMENT_TYPE(SIMPLE_STYPE),
638     TPM2B_AUTH_MARSHAL_REF,
639     (UINT16) (offsetof(TPMS_AUTH_COMMAND, hmac))}},
640 // TPMS_AUTH_RESPONSE_DATA
641 {STRUCTURE_MTYPE, 3, {
642     SET_ELEMENT_TYPE(SIMPLE_STYPE),
643     TPM2B_NONCE_MARSHAL_REF,
644     (UINT16) (offsetof(TPMS_AUTH_RESPONSE, nonce)),
645     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
646     TPMA_SESSION_MARSHAL_REF,
647     (UINT16) (offsetof(TPMS_AUTH_RESPONSE, sessionAttributes)),
648     SET_ELEMENT_TYPE(SIMPLE_STYPE),
649     TPM2B_AUTH_MARSHAL_REF,
650     (UINT16) (offsetof(TPMS_AUTH_RESPONSE, hmac))}},
651 // TPMI_TDES_KEY_BITS_DATA
652 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1,
653     {128*TDES_128}},
654 // TPMI_AES_KEY_BITS_DATA
655 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
656     {192*AES_192, 128*AES_128, 256*AES_256}},
657 // TPMI_SM4_KEY_BITS_DATA
658 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 1,
659     {128*SM4_128}},

```

```

660 // TPMI_CAMELLIA_KEY_BITS_DATA
661 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_VALUE, 3,
662   {192*CAMELLIA_192, 128*CAMELLIA_128, 256*CAMELLIA_256}},
663 // TPMU_SYM_KEY_BITS_DATA
664 {6, 0, (UINT16)(offsetof(TPMU_SYM_KEY_BITS_mst, marshalingTypes)),
665   {(UINT32)TPM_ALG_TDES, (UINT32)TPM_ALG_AES, (UINT32)TPM_ALG_SM4,
666    (UINT32)TPM_ALG_CAMELLIA, (UINT32)TPM_ALG_XOR, (UINT32)TPM_ALG_NULL},
667   {(UINT16)(TPMI_TDES_KEY_BITS_MARSHAL_REF),
668    (UINT16)(TPMI_AES_KEY_BITS_MARSHAL_REF),
669    (UINT16)(TPMI_SM4_KEY_BITS_MARSHAL_REF),
670    (UINT16)(TPMI_CAMELLIA_KEY_BITS_MARSHAL_REF),
671    (UINT16)(TPMI_ALG_HASH_MARSHAL_REF),
672    (UINT16)(UINT0_MARSHAL_REF)}},
673 },
674 // TPMU_SYM_MODE_DATA
675 {6, 0, (UINT16)(offsetof(TPMU_SYM_MODE_mst, marshalingTypes)),
676   {(UINT32)TPM_ALG_TDES, (UINT32)TPM_ALG_AES, (UINT32)TPM_ALG_SM4,
677    (UINT32)TPM_ALG_CAMELLIA, (UINT32)TPM_ALG_XOR, (UINT32)TPM_ALG_NULL},
678   {(UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
679    (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
680    (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
681    (UINT16)(TPMI_ALG_SYM_MODE_MARSHAL_REF|NULL_FLAG),
682    (UINT16)(UINT0_MARSHAL_REF),
683    (UINT16)(UINT0_MARSHAL_REF)}},
684 },
685 // TPMT_SYM_DEF_DATA
686 {STRUCTURE_MTYPE, 3, {
687   SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
688   TPMI_ALG_SYM_MARSHAL_REF,
689   (UINT16)(offsetof(TPMT_SYM_DEF, algorithm)),
690   SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
691   TPMU_SYM_KEY_BITS_MARSHAL_REF,
692   (UINT16)(offsetof(TPMT_SYM_DEF, keyBits)),
693   SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
694   TPMU_SYM_MODE_MARSHAL_REF,
695   (UINT16)(offsetof(TPMT_SYM_DEF, mode))}},
696 // TPMT_SYM_DEF_OBJECT_DATA
697 {STRUCTURE_MTYPE, 3, {
698   SET_ELEMENT_TYPE(SIMPLE_STYPE)|SET_ELEMENT_SIZE(TWO_BYTES)|ELEMENT_PROPAGATE,
699   TPMI_ALG_SYM_OBJECT_MARSHAL_REF,
700   (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, algorithm)),
701   SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
702   TPMU_SYM_KEY_BITS_MARSHAL_REF,
703   (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, keyBits)),
704   SET_ELEMENT_TYPE(UNION_STYPE)|SET_ELEMENT_NUMBER(0),
705   TPMU_SYM_MODE_MARSHAL_REF,
706   (UINT16)(offsetof(TPMT_SYM_DEF_OBJECT, mode))}},
707 // TPM2B_SYM_KEY_DATA
708 {TPM2B_MTYPE, Type29_MARSHAL_REF},
709 // TPMS_SYMCIPHER_PARMS_DATA
710 {STRUCTURE_MTYPE, 1, {
711   SET_ELEMENT_TYPE(SIMPLE_STYPE),
712   TPMT_SYM_DEF_OBJECT_MARSHAL_REF,
713   (UINT16)(offsetof(TPMS_SYMCIPHER_PARMS, sym))}},
714 // TPM2B_LABEL_DATA
715 {TPM2B_MTYPE, Type30_MARSHAL_REF},
716 // TPMS_DERIVE_DATA
717 {STRUCTURE_MTYPE, 2, {
718   SET_ELEMENT_TYPE(SIMPLE_STYPE),
719   TPM2B_LABEL_MARSHAL_REF,
720   (UINT16)(offsetof(TPMS_DERIVE, label)),
721   SET_ELEMENT_TYPE(SIMPLE_STYPE),
722   TPM2B_LABEL_MARSHAL_REF,
723   (UINT16)(offsetof(TPMS_DERIVE, context))}},
724 // TPM2B_DERIVE_DATA
725 {TPM2B_MTYPE, Type31_MARSHAL_REF},

```



```

726 // TPM2B_SENSITIVE_DATA_DATA
727 {TPM2B_MTYPE, Type32_MARSHAL_REF},
728 // TPMS_SENSITIVE_CREATE_DATA
729 {STRUCTURE_MTYPE, 2, {
730     SET_ELEMENT_TYPE(SIMPLE_TYPE),
731     TPM2B_AUTH_MARSHAL_REF,
732     (UINT16) (offsetof(TPMS_SENSITIVE_CREATE, userAuth)),
733     SET_ELEMENT_TYPE(SIMPLE_TYPE),
734     TPM2B_SENSITIVE_DATA_MARSHAL_REF,
735     (UINT16) (offsetof(TPMS_SENSITIVE_CREATE, data))}},
736 // TPM2B_SENSITIVE_CREATE_DATA
737 {TPM2BS_MTYPE,
738     (UINT8) (offsetof(TPM2B_SENSITIVE_CREATE, sensitive)) | SIZE_EQUAL,
739     UINT16_MARSHAL_REF,
740     TPMS_SENSITIVE_CREATE_MARSHAL_REF},
741 // TPMS_SCHEME_HASH_DATA
742 {STRUCTURE_MTYPE, 1, {
743     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
744     TPMI_ALG_HASH_MARSHAL_REF,
745     (UINT16) (offsetof(TPMS_SCHEME_HASH, hashAlg))}},
746 // TPMS_SCHEME_ECDSA_DATA
747 {STRUCTURE_MTYPE, 2, {
748     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
749     TPMI_ALG_HASH_MARSHAL_REF,
750     (UINT16) (offsetof(TPMS_SCHEME_ECDSA, hashAlg)),
751     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
752     UINT16_MARSHAL_REF,
753     (UINT16) (offsetof(TPMS_SCHEME_ECDSA, count))}},
754 // TPMI_ALG_KEYEDHASH_SCHEME_DATA
755 {MIN_MAX_MTYPE, TWO_BYTES | TAKES_NULL | HAS_BITS, (UINT8) TPM_RC_VALUE,
756     {TPM_ALG_NULL,
757     RANGE(5, 10, UINT16),
758     (UINT32) ((ALG_HMAC << 0) | (ALG_XOR << 5))}},
759 // TPMS_SCHEME_XOR_DATA
760 {STRUCTURE_MTYPE, 2, {
761     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
762     TPMI_ALG_HASH_MARSHAL_REF,
763     (UINT16) (offsetof(TPMS_SCHEME_XOR, hashAlg)),
764     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
765     TPMI_ALG_KDF_MARSHAL_REF | NULL_FLAG,
766     (UINT16) (offsetof(TPMS_SCHEME_XOR, kdf))}},
767 // TPMU_SCHEME_KEYEDHASH_DATA
768 {3, 0, (UINT16) (offsetof(TPMU_SCHEME_KEYEDHASH_mst, marshalingTypes)),
769     {(UINT32) TPM_ALG_HMAC, (UINT32) TPM_ALG_XOR, (UINT32) TPM_ALG_NULL},
770     {(UINT16) (TPMS_SCHEME_HMAC_MARSHAL_REF),
771     (UINT16) (TPMS_SCHEME_XOR_MARSHAL_REF),
772     (UINT16) (UINT0_MARSHAL_REF)}},
773 },
774 // TPMT_KEYEDHASH_SCHEME_DATA
775 {STRUCTURE_MTYPE, 2, {
776     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
777     TPMI_ALG_KEYEDHASH_SCHEME_MARSHAL_REF,
778     (UINT16) (offsetof(TPMT_KEYEDHASH_SCHEME, scheme)),
779     SET_ELEMENT_TYPE(UNION_TYPE) | SET_ELEMENT_NUMBER(0),
780     TPMU_SCHEME_KEYEDHASH_MARSHAL_REF,
781     (UINT16) (offsetof(TPMT_KEYEDHASH_SCHEME, details))}},
782 // TPMU_SIG_SCHEME_DATA
783 {8, 0, (UINT16) (offsetof(TPMU_SIG_SCHEME_mst, marshalingTypes)),
784     {(UINT32) TPM_ALG_ECDSA, (UINT32) TPM_ALG_RSASSA,
785     (UINT32) TPM_ALG_RSAPSS, (UINT32) TPM_ALG_ECDSA,
786     (UINT32) TPM_ALG_SM2, (UINT32) TPM_ALG_ECSCNORR,
787     (UINT32) TPM_ALG_HMAC, (UINT32) TPM_ALG_NULL},
788     {(UINT16) (TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF),
789     (UINT16) (TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF),
790     (UINT16) (TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF),
791     (UINT16) (TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF)},

```

```

792     (UINT16) (TPMS_SIG_SCHEME_SM2_MARSHAL_REF),
793     (UINT16) (TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF),
794     (UINT16) (TPMS_SCHEME_HMAC_MARSHAL_REF),
795     (UINT16) (UINT0_MARSHAL_REF) }
796 },
797 // TPMT_SIG_SCHEME_DATA
798 {STRUCTURE_MTYPE, 2, {
799     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
800     TPMI_ALG_SIG_SCHEME_MARSHAL_REF,
801     (UINT16) (offsetof(TPMT_SIG_SCHEME, scheme)),
802     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
803     TPMU_SIG_SCHEME_MARSHAL_REF,
804     (UINT16) (offsetof(TPMT_SIG_SCHEME, details))}},
805 // TPMU_KDF_SCHEME_DATA
806 {5, 0, (UINT16) (offsetof(TPMU_KDF_SCHEME_mst, marshalingTypes)),
807     { (UINT32) TPM_ALG_MGF1, (UINT32) TPM_ALG_KDF1_SP800_56A,
808       (UINT32) TPM_ALG_KDF2, (UINT32) TPM_ALG_KDF1_SP800_108,
809       (UINT32) TPM_ALG_NULL},
810     { (UINT16) (TPMS_KDF_SCHEME_MGF1_MARSHAL_REF),
811       (UINT16) (TPMS_KDF_SCHEME_KDF1_SP800_56A_MARSHAL_REF),
812       (UINT16) (TPMS_KDF_SCHEME_KDF2_MARSHAL_REF),
813       (UINT16) (TPMS_KDF_SCHEME_KDF1_SP800_108_MARSHAL_REF),
814       (UINT16) (UINT0_MARSHAL_REF) }
815 },
816 // TPMT_KDF_SCHEME_DATA
817 {STRUCTURE_MTYPE, 2, {
818     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
819     TPMI_ALG_KDF_MARSHAL_REF,
820     (UINT16) (offsetof(TPMT_KDF_SCHEME, scheme)),
821     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
822     TPMU_KDF_SCHEME_MARSHAL_REF,
823     (UINT16) (offsetof(TPMT_KDF_SCHEME, details))}},
824 // TPMI_ALG_ASYM_SCHEME_DATA
825 {MIN_MAX_MTYPE, TWO_BYTES | TAKES_NULL | HAS_BITS, (UINT8) TPM_RC_VALUE,
826     {TPM_ALG_NULL,
827       RANGE(20, 29, UINT16),
828       (UINT32) ((ALG_RSASSA << 0) | (ALG_RSAES << 1) | (ALG_RSAPSS << 2) |
829                (ALG_OAEP << 3) | (ALG_ECDSA << 4) | (ALG_ECDH << 5) |
830                (ALG_ECDAA << 6) | (ALG_SM2 << 7) | (ALG_ECSCHNORR << 8) |
831                (ALG_ECMQV << 9))}},
832 // TPMU_ASYM_SCHEME_DATA
833 {11, 0, (UINT16) (offsetof(TPMU_ASYM_SCHEME_mst, marshalingTypes)),
834     { (UINT32) TPM_ALG_ECDH, (UINT32) TPM_ALG_ECMQV,
835       (UINT32) TPM_ALG_ECDAA, (UINT32) TPM_ALG_RSASSA,
836       (UINT32) TPM_ALG_RSAPSS, (UINT32) TPM_ALG_ECDSA,
837       (UINT32) TPM_ALG_SM2, (UINT32) TPM_ALG_ECSCHNORR,
838       (UINT32) TPM_ALG_RSAES, (UINT32) TPM_ALG_OAEP,
839       (UINT32) TPM_ALG_NULL},
840     { (UINT16) (TPMS_KEY_SCHEME_ECDH_MARSHAL_REF),
841       (UINT16) (TPMS_KEY_SCHEME_ECMQV_MARSHAL_REF),
842       (UINT16) (TPMS_SIG_SCHEME_ECDAA_MARSHAL_REF),
843       (UINT16) (TPMS_SIG_SCHEME_RSASSA_MARSHAL_REF),
844       (UINT16) (TPMS_SIG_SCHEME_RSAPSS_MARSHAL_REF),
845       (UINT16) (TPMS_SIG_SCHEME_ECDSA_MARSHAL_REF),
846       (UINT16) (TPMS_SIG_SCHEME_SM2_MARSHAL_REF),
847       (UINT16) (TPMS_SIG_SCHEME_ECSCHNORR_MARSHAL_REF),
848       (UINT16) (TPMS_ENC_SCHEME_RSAES_MARSHAL_REF),
849       (UINT16) (TPMS_ENC_SCHEME_OAEP_MARSHAL_REF),
850       (UINT16) (UINT0_MARSHAL_REF) }
851 },
852 // TPMI_ALG_RSA_SCHEME_DATA
853 {MIN_MAX_MTYPE, TWO_BYTES | TAKES_NULL | HAS_BITS, (UINT8) TPM_RC_VALUE,
854     {TPM_ALG_NULL,
855       RANGE(20, 23, UINT16),
856       (UINT32) ((ALG_RSASSA << 0) | (ALG_RSAES << 1) | (ALG_RSAPSS << 2) | (ALG_OAEP << 3))}},
857 // TPMT_RSA_SCHEME_DATA

```

```

858 {STRUCTURE_MTYPE, 2, {
859     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
860     TPMI_ALG_RSA_SCHEME_MARSHAL_REF,
861     (UINT16) (offsetof(TPMT_RSA_SCHEME, scheme)),
862     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
863     TPMU_ASYM_SCHEME_MARSHAL_REF,
864     (UINT16) (offsetof(TPMT_RSA_SCHEME, details))}},
865 // TPMI_ALG_RSA_DECRYPT_DATA
866 {MIN_MAX_MTYPE, TWO_BYTES | TAKES_NULL | HAS_BITS, (UINT8) TPM_RC_VALUE,
867     {TPM_ALG_NULL,
868     RANGE(21, 23, UINT16),
869     (UINT32) ((ALG_RSAES << 0) | (ALG_OAEP << 2))}},
870 // TPMT_RSA_DECRYPT_DATA
871 {STRUCTURE_MTYPE, 2, {
872     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
873     TPMI_ALG_RSA_DECRYPT_MARSHAL_REF,
874     (UINT16) (offsetof(TPMT_RSA_DECRYPT, scheme)),
875     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
876     TPMU_ASYM_SCHEME_MARSHAL_REF,
877     (UINT16) (offsetof(TPMT_RSA_DECRYPT, details))}},
878 // TPM2B_PUBLIC_KEY_RSA_DATA
879 {TPM2B_MTYPE, Type33_MARSHAL_REF},
880 // TPMI_RSA_KEY_BITS_DATA
881 {TABLE_MTYPE, TWO_BYTES, (UINT8) TPM_RC_VALUE, 3,
882     {3072 * RSA_3072, 1024 * RSA_1024, 2048 * RSA_2048}},
883 // TPM2B_PRIVATE_KEY_RSA_DATA
884 {TPM2B_MTYPE, Type34_MARSHAL_REF},
885 // TPM2B_ECC_PARAMETER_DATA
886 {TPM2B_MTYPE, Type35_MARSHAL_REF},
887 // TPMS_ECC_POINT_DATA
888 {STRUCTURE_MTYPE, 2, {
889     SET_ELEMENT_TYPE(SIMPLE_STYPE),
890     TPM2B_ECC_PARAMETER_MARSHAL_REF,
891     (UINT16) (offsetof(TPMS_ECC_POINT, x)),
892     SET_ELEMENT_TYPE(SIMPLE_STYPE),
893     TPM2B_ECC_PARAMETER_MARSHAL_REF,
894     (UINT16) (offsetof(TPMS_ECC_POINT, y))}},
895 // TPM2B_ECC_POINT_DATA
896 {TPM2BS_MTYPE,
897     (UINT8) (offsetof(TPM2B_ECC_POINT, point)) | SIZE_EQUAL,
898     UINT16_MARSHAL_REF,
899     TPMS_ECC_POINT_MARSHAL_REF},
900 // TPMI_ALG_ECC_SCHEME_DATA
901 {MIN_MAX_MTYPE, TWO_BYTES | TAKES_NULL | HAS_BITS, (UINT8) TPM_RC_SCHEME,
902     {TPM_ALG_NULL,
903     RANGE(24, 29, UINT16),
904     (UINT32) ((ALG_ECDSA << 0) | (ALG_ECDH << 1) | (ALG_ECDAA << 2) |
905     (ALG_SM2 << 3) | (ALG_ECSCHNORR << 4) | (ALG_ECMQV << 5))}},
906 // TPMI_ECC_CURVE_DATA
907 {MIN_MAX_MTYPE, TWO_BYTES | HAS_BITS, (UINT8) TPM_RC_CURVE,
908     {RANGE(1, 32, UINT16),
909     (UINT32) ((ECC_NIST_P192 << 0) | (ECC_NIST_P224 << 1) | (ECC_NIST_P256 << 2) |
910     (ECC_NIST_P384 << 3) | (ECC_NIST_P521 << 4) | (ECC_BN_P256 << 15) |
911     (ECC_BN_P638 << 16) | (ECC_SM2_P256 << 31))}},
912 // TPMT_ECC_SCHEME_DATA
913 {STRUCTURE_MTYPE, 2, {
914     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
915     TPMI_ALG_ECC_SCHEME_MARSHAL_REF,
916     (UINT16) (offsetof(TPMT_ECC_SCHEME, scheme)),
917     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
918     TPMU_ASYM_SCHEME_MARSHAL_REF,
919     (UINT16) (offsetof(TPMT_ECC_SCHEME, details))}},
920 // TPMS_ALGORITHM_DETAIL_ECC_DATA
921 {STRUCTURE_MTYPE, 11, {
922     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
923     TPM_ECC_CURVE_MARSHAL_REF,

```

```

924         (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, curveID)),
925     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
926     UINT16_MARSHAL_REF,
927     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, keySize)),
928     SET_ELEMENT_TYPE(SIMPLE_TYPE),
929     TPMT_KDF_SCHEME_MARSHAL_REF | NULL_FLAG,
930     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, kdf)),
931     SET_ELEMENT_TYPE(SIMPLE_TYPE),
932     TPMT_ECC_SCHEME_MARSHAL_REF | NULL_FLAG,
933     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, sign)),
934     SET_ELEMENT_TYPE(SIMPLE_TYPE),
935     TPM2B_ECC_PARAMETER_MARSHAL_REF,
936     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, p)),
937     SET_ELEMENT_TYPE(SIMPLE_TYPE),
938     TPM2B_ECC_PARAMETER_MARSHAL_REF,
939     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, a)),
940     SET_ELEMENT_TYPE(SIMPLE_TYPE),
941     TPM2B_ECC_PARAMETER_MARSHAL_REF,
942     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, b)),
943     SET_ELEMENT_TYPE(SIMPLE_TYPE),
944     TPM2B_ECC_PARAMETER_MARSHAL_REF,
945     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, gX)),
946     SET_ELEMENT_TYPE(SIMPLE_TYPE),
947     TPM2B_ECC_PARAMETER_MARSHAL_REF,
948     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, gY)),
949     SET_ELEMENT_TYPE(SIMPLE_TYPE),
950     TPM2B_ECC_PARAMETER_MARSHAL_REF,
951     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, n)),
952     SET_ELEMENT_TYPE(SIMPLE_TYPE),
953     TPM2B_ECC_PARAMETER_MARSHAL_REF,
954     (UINT16) (offsetof(TPMS_ALGORITHM_DETAIL_ECC, h))}},
955 // TPMS_SIGNATURE_RSA_DATA
956 {STRUCTURE_MTYPE, 2, {
957     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
958     TPMI_ALG_HASH_MARSHAL_REF,
959     (UINT16) (offsetof(TPMS_SIGNATURE_RSA, hash)),
960     SET_ELEMENT_TYPE(SIMPLE_TYPE),
961     TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF,
962     (UINT16) (offsetof(TPMS_SIGNATURE_RSA, sig))}},
963 // TPMS_SIGNATURE_ECC_DATA
964 {STRUCTURE_MTYPE, 3, {
965     SET_ELEMENT_TYPE(SIMPLE_TYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
966     TPMI_ALG_HASH_MARSHAL_REF,
967     (UINT16) (offsetof(TPMS_SIGNATURE_ECC, hash)),
968     SET_ELEMENT_TYPE(SIMPLE_TYPE),
969     TPM2B_ECC_PARAMETER_MARSHAL_REF,
970     (UINT16) (offsetof(TPMS_SIGNATURE_ECC, signatureR)),
971     SET_ELEMENT_TYPE(SIMPLE_TYPE),
972     TPM2B_ECC_PARAMETER_MARSHAL_REF,
973     (UINT16) (offsetof(TPMS_SIGNATURE_ECC, signatureS))}},
974 // TPMU_SIGNATURE_DATA
975 {8, 0, (UINT16) (offsetof(TPMU_SIGNATURE_mst, marshalingTypes)),
976     {(UINT32)TPM_ALG_ECDSA, (UINT32)TPM_ALG_RSASSA,
977     (UINT32)TPM_ALG_RSAPSS, (UINT32)TPM_ALG_ECDSA,
978     (UINT32)TPM_ALG_SM2, (UINT32)TPM_ALG_ECSCHNORR,
979     (UINT32)TPM_ALG_HMAC, (UINT32)TPM_ALG_NULL},
980     {(UINT16) (TPMS_SIGNATURE_ECDSA_MARSHAL_REF),
981     (UINT16) (TPMS_SIGNATURE_RSASSA_MARSHAL_REF),
982     (UINT16) (TPMS_SIGNATURE_RSAPSS_MARSHAL_REF),
983     (UINT16) (TPMS_SIGNATURE_ECDSA_MARSHAL_REF),
984     (UINT16) (TPMS_SIGNATURE_SM2_MARSHAL_REF),
985     (UINT16) (TPMS_SIGNATURE_ECSCHNORR_MARSHAL_REF),
986     (UINT16) (TPMT_HA_MARSHAL_REF),
987     (UINT16) (UINT0_MARSHAL_REF)}
988 },
989 // TPMT_SIGNATURE_DATA

```

```

990 {STRUCTURE_MTYPE, 2, {
991     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
992     TPMI_ALG_SIG_SCHEME_MARSHAL_REF,
993     (UINT16) (offsetof(TPMT_SIGNATURE, sigAlg)),
994     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
995     TPMU_SIGNATURE_MARSHAL_REF,
996     (UINT16) (offsetof(TPMT_SIGNATURE, signature))}},
997 // TPMU_ENCRYPTED_SECRET_DATA
998 {4, IS_ARRAY_UNION, (UINT16) (offsetof(TPMU_ENCRYPTED_SECRET_mst, marshalingTypes)),
999     {(UINT32)TPM_ALG_ECC, (UINT32)TPM_ALG_RSA,
1000      (UINT32)TPM_ALG_SYMCIPHER, (UINT32)TPM_ALG_KEYEDHASH},
1001     {(UINT16) (sizeof(TPMS_ECC_POINT)), (UINT16) (MAX_RSA_KEY_BYTES),
1002      (UINT16) (sizeof(TPM2B_DIGEST)), (UINT16) (sizeof(TPM2B_DIGEST))}
1003 },
1004 // TPM2B_ENCRYPTED_SECRET_DATA
1005 {TPM2B_MTYPE, Type36_MARSHAL_REF},
1006 // TPMI_ALG_PUBLIC_DATA
1007 {MIN_MAX_MTYPE, TWO_BYTES | HAS_BITS, (UINT8)TPM_RC_TYPE,
1008  {RANGE(1, 37, UINT16),
1009   (UINT32) ((ALG_RSA << 0) | (ALG_KEYEDHASH << 7)),
1010   (UINT32) ((ALG_ECC << 2) | (ALG_SYMCIPHER << 4))}},
1011 // TPMU_PUBLIC_ID_DATA
1012 {4, 0, (UINT16) (offsetof(TPMU_PUBLIC_ID_mst, marshalingTypes)),
1013  {(UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER,
1014   (UINT32)TPM_ALG_RSA, (UINT32)TPM_ALG_ECC},
1015  {(UINT16) (TPM2B_DIGEST_MARSHAL_REF),
1016   (UINT16) (TPM2B_DIGEST_MARSHAL_REF),
1017   (UINT16) (TPM2B_PUBLIC_KEY_RSA_MARSHAL_REF),
1018   (UINT16) (TPMS_ECC_POINT_MARSHAL_REF)}
1019 },
1020 // TPMS_KEYEDHASH_PARMS_DATA
1021 {STRUCTURE_MTYPE, 1, {
1022     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1023     TPMT_KEYEDHASH_SCHEME_MARSHAL_REF | NULL_FLAG,
1024     (UINT16) (offsetof(TPMS_KEYEDHASH_PARMS, scheme))}},
1025 // TPMS_RSA_PARMS_DATA
1026 {STRUCTURE_MTYPE, 4, {
1027     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1028     TPMT_SYM_DEF_OBJECT_MARSHAL_REF | NULL_FLAG,
1029     (UINT16) (offsetof(TPMS_RSA_PARMS, symmetric)),
1030     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1031     TPMT_RSA_SCHEME_MARSHAL_REF | NULL_FLAG,
1032     (UINT16) (offsetof(TPMS_RSA_PARMS, scheme)),
1033     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1034     TPMI_RSA_KEY_BITS_MARSHAL_REF,
1035     (UINT16) (offsetof(TPMS_RSA_PARMS, keyBits)),
1036     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1037     UINT32_MARSHAL_REF,
1038     (UINT16) (offsetof(TPMS_RSA_PARMS, exponent))}},
1039 // TPMS_ECC_PARMS_DATA
1040 {STRUCTURE_MTYPE, 4, {
1041     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1042     TPMT_SYM_DEF_OBJECT_MARSHAL_REF | NULL_FLAG,
1043     (UINT16) (offsetof(TPMS_ECC_PARMS, symmetric)),
1044     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1045     TPMT_ECC_SCHEME_MARSHAL_REF | NULL_FLAG,
1046     (UINT16) (offsetof(TPMS_ECC_PARMS, scheme)),
1047     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1048     TPMI_ECC_CURVE_MARSHAL_REF,
1049     (UINT16) (offsetof(TPMS_ECC_PARMS, curveID)),
1050     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1051     TPMT_KDF_SCHEME_MARSHAL_REF | NULL_FLAG,
1052     (UINT16) (offsetof(TPMS_ECC_PARMS, kdf))}},
1053 // TPMU_PUBLIC_PARMS_DATA
1054 {4, 0, (UINT16) (offsetof(TPMU_PUBLIC_PARMS_mst, marshalingTypes)),
1055  {(UINT32)TPM_ALG_KEYEDHASH, (UINT32)TPM_ALG_SYMCIPHER,

```



```

1056     (UINT32)TPM_ALG_RSA,          (UINT32)TPM_ALG_ECC},
1057     {(UINT16) (TPMS_KEYEDHASH_PARMS_MARSHAL_REF),
1058      (UINT16) (TPMS_SYMCIPHER_PARMS_MARSHAL_REF),
1059      (UINT16) (TPMS_RSA_PARMS_MARSHAL_REF),
1060      (UINT16) (TPMS_ECC_PARMS_MARSHAL_REF)}
1061 },
1062 // TPMT_PUBLIC_PARMS_DATA
1063 {STRUCTURE_MTYPE, 2, {
1064     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1065     TPMI_ALG_PUBLIC_MARSHAL_REF,
1066     (UINT16) (offsetof(TPMT_PUBLIC_PARMS, type)),
1067     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
1068     TPMU_PUBLIC_PARMS_MARSHAL_REF,
1069     (UINT16) (offsetof(TPMT_PUBLIC_PARMS, parameters))}},
1070 // TPMT_PUBLIC_DATA
1071 {STRUCTURE_MTYPE, 6, {
1072     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1073     TPMI_ALG_PUBLIC_MARSHAL_REF,
1074     (UINT16) (offsetof(TPMT_PUBLIC, type)),
1075     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES) | ELEMENT_PROPAGATE,
1076     TPMI_ALG_HASH_MARSHAL_REF,
1077     (UINT16) (offsetof(TPMT_PUBLIC, nameAlg)),
1078     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1079     TPMA_OBJECT_MARSHAL_REF,
1080     (UINT16) (offsetof(TPMT_PUBLIC, objectAttributes)),
1081     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1082     TPM2B_DIGEST_MARSHAL_REF,
1083     (UINT16) (offsetof(TPMT_PUBLIC, authPolicy)),
1084     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
1085     TPMU_PUBLIC_PARMS_MARSHAL_REF,
1086     (UINT16) (offsetof(TPMT_PUBLIC, parameters)),
1087     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
1088     TPMU_PUBLIC_ID_MARSHAL_REF,
1089     (UINT16) (offsetof(TPMT_PUBLIC, unique))}},
1090 // TPM2B_PUBLIC_DATA
1091 {TPM2BS_MTYPE,
1092     (UINT8) (offsetof(TPM2B_PUBLIC, publicArea)) | SIZE_EQUAL | ELEMENT_PROPAGATE,
1093     UINT16_MARSHAL_REF,
1094     TPMT_PUBLIC_MARSHAL_REF},
1095 // TPM2B_TEMPLATE_DATA
1096 {TPM2B_MTYPE, Type37_MARSHAL_REF},
1097 // TPM2B_PRIVATE_VENDOR_SPECIFIC_DATA
1098 {TPM2B_MTYPE, Type38_MARSHAL_REF},
1099 // TPMU_SENSITIVE_COMPOSITE_DATA
1100 {4, 0, (UINT16) (offsetof(TPMU_SENSITIVE_COMPOSITE_mst, marshalingTypes)),
1101     {(UINT32)TPM_ALG_RSA,          (UINT32)TPM_ALG_ECC,
1102      (UINT32)TPM_ALG_KEYEDHASH,    (UINT32)TPM_ALG_SYMCIPHER},
1103     {(UINT16) (TPM2B_PRIVATE_KEY_RSA_MARSHAL_REF),
1104      (UINT16) (TPM2B_ECC_PARAMETER_MARSHAL_REF),
1105      (UINT16) (TPM2B_SENSITIVE_DATA_MARSHAL_REF),
1106      (UINT16) (TPM2B_SYM_KEY_MARSHAL_REF)}
1107 },
1108 // TPMT_SENSITIVE_DATA
1109 {STRUCTURE_MTYPE, 4, {
1110     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1111     TPMI_ALG_PUBLIC_MARSHAL_REF,
1112     (UINT16) (offsetof(TPMT_SENSITIVE, sensitiveType)),
1113     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1114     TPM2B_AUTH_MARSHAL_REF,
1115     (UINT16) (offsetof(TPMT_SENSITIVE, authValue)),
1116     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1117     TPM2B_DIGEST_MARSHAL_REF,
1118     (UINT16) (offsetof(TPMT_SENSITIVE, seedValue)),
1119     SET_ELEMENT_TYPE(UNION_STYPE) | SET_ELEMENT_NUMBER(0),
1120     TPMU_SENSITIVE_COMPOSITE_MARSHAL_REF,
1121     (UINT16) (offsetof(TPMT_SENSITIVE, sensitive))}},

```

```

1122 // TPM2B_SENSITIVE_DATA
1123 {TPM2BS_MTYPE,
1124     (UINT8) (offsetof(TPM2B_SENSITIVE, sensitiveArea)),
1125     UINT16_MARSHAL_REF,
1126     TPMT_SENSITIVE_MARSHAL_REF},
1127 // TPM2B_PRIVATE_DATA
1128 {TPM2B_MTYPE, Type39_MARSHAL_REF},
1129 // TPM2B_ID_OBJECT_DATA
1130 {TPM2B_MTYPE, Type40_MARSHAL_REF},
1131 // TPMS_NV_PIN_COUNTER_PARAMETERS_DATA
1132 {STRUCTURE_MTYPE, 2, {
1133     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1134     UINT32_MARSHAL_REF,
1135     (UINT16) (offsetof(TPMS_NV_PIN_COUNTER_PARAMETERS, pinCount)),
1136     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1137     UINT32_MARSHAL_REF,
1138     (UINT16) (offsetof(TPMS_NV_PIN_COUNTER_PARAMETERS, pinLimit))}},
1139 // TPMA_NV_DATA
1140 {ATTRIBUTES_MTYPE, FOUR_BYTES, 0x01F00300},
1141 // TPMS_NV_PUBLIC_DATA
1142 {STRUCTURE_MTYPE, 5, {
1143     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1144     TPMI_RH_NV_INDEX_MARSHAL_REF,
1145     (UINT16) (offsetof(TPMS_NV_PUBLIC, nvIndex)),
1146     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1147     TPMI_ALG_HASH_MARSHAL_REF,
1148     (UINT16) (offsetof(TPMS_NV_PUBLIC, nameAlg)),
1149     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1150     TPMA_NV_MARSHAL_REF,
1151     (UINT16) (offsetof(TPMS_NV_PUBLIC, attributes)),
1152     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1153     TPM2B_DIGEST_MARSHAL_REF,
1154     (UINT16) (offsetof(TPMS_NV_PUBLIC, authPolicy)),
1155     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1156     Type41_MARSHAL_REF,
1157     (UINT16) (offsetof(TPMS_NV_PUBLIC, dataSize))}},
1158 // TPM2B_NV_PUBLIC_DATA
1159 {TPM2BS_MTYPE,
1160     (UINT8) (offsetof(TPM2B_NV_PUBLIC, nvPublic)) | SIZE_EQUAL,
1161     UINT16_MARSHAL_REF,
1162     TPMS_NV_PUBLIC_MARSHAL_REF},
1163 // TPM2B_CONTEXT_SENSITIVE_DATA
1164 {TPM2B_MTYPE, Type42_MARSHAL_REF},
1165 // TPMS_CONTEXT_DATA_DATA
1166 {STRUCTURE_MTYPE, 2, {
1167     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1168     TPM2B_DIGEST_MARSHAL_REF,
1169     (UINT16) (offsetof(TPMS_CONTEXT_DATA, integrity)),
1170     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1171     TPM2B_CONTEXT_SENSITIVE_MARSHAL_REF,
1172     (UINT16) (offsetof(TPMS_CONTEXT_DATA, encrypted))}},
1173 // TPM2B_CONTEXT_DATA_DATA
1174 {TPM2B_MTYPE, Type43_MARSHAL_REF},
1175 // TPMS_CONTEXT_DATA
1176 {STRUCTURE_MTYPE, 4, {
1177     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(EIGHT_BYTES),
1178     UINT64_MARSHAL_REF,
1179     (UINT16) (offsetof(TPMS_CONTEXT, sequence)),
1180     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1181     TPMI_DH_SAVED_MARSHAL_REF,
1182     (UINT16) (offsetof(TPMS_CONTEXT, savedHandle)),
1183     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1184     TPMI_RH_HIERARCHY_MARSHAL_REF | NULL_FLAG,
1185     (UINT16) (offsetof(TPMS_CONTEXT, hierarchy)),
1186     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1187     TPM2B_CONTEXT_DATA_MARSHAL_REF,

```



```

1188         (UINT16) (offsetof(TPMS_CONTEXT, contextBlob))}},
1189 // TPMS_CREATION_DATA_DATA
1190 {STRUCTURE_MTYPE, 7, {
1191     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1192     TPML_PCR_SELECTION_MARSHAL_REF,
1193     (UINT16) (offsetof(TPMS_CREATION_DATA, pcrSelect)),
1194     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1195     TPM2B_DIGEST_MARSHAL_REF,
1196     (UINT16) (offsetof(TPMS_CREATION_DATA, pcrDigest)),
1197     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(ONE_BYTES),
1198     TPMA_LOCALITY_MARSHAL_REF,
1199     (UINT16) (offsetof(TPMS_CREATION_DATA, locality)),
1200     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(TWO_BYTES),
1201     TPM_ALG_ID_MARSHAL_REF,
1202     (UINT16) (offsetof(TPMS_CREATION_DATA, parentNameAlg)),
1203     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1204     TPM2B_NAME_MARSHAL_REF,
1205     (UINT16) (offsetof(TPMS_CREATION_DATA, parentName)),
1206     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1207     TPM2B_NAME_MARSHAL_REF,
1208     (UINT16) (offsetof(TPMS_CREATION_DATA, parentQualifiedName)),
1209     SET_ELEMENT_TYPE(SIMPLE_STYPE),
1210     TPM2B_DATA_MARSHAL_REF,
1211     (UINT16) (offsetof(TPMS_CREATION_DATA, outsideInfo))}},
1212 // TPM2B_CREATION_DATA_DATA
1213 {TPM2BS_MTYPE,
1214     (UINT8) (offsetof(TPM2B_CREATION_DATA, creationData)) | SIZE_EQUAL,
1215     UINT16_MARSHAL_REF,
1216     TPMS_CREATION_DATA_MARSHAL_REF},
1217 // TPM_AT_DATA
1218 {TABLE_MTYPE, FOUR_BYTES, (UINT8) TPM_RC_VALUE, 4,
1219     {TPM_AT_ANY, TPM_AT_ERROR, TPM_AT_PV1, TPM_AT_VEND}},
1220 // TPMS_AC_OUTPUT_DATA
1221 {STRUCTURE_MTYPE, 2, {
1222     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1223     TPM_AT_MARSHAL_REF,
1224     (UINT16) (offsetof(TPMS_AC_OUTPUT, tag)),
1225     SET_ELEMENT_TYPE(SIMPLE_STYPE) | SET_ELEMENT_SIZE(FOUR_BYTES),
1226     UINT32_MARSHAL_REF,
1227     (UINT16) (offsetof(TPMS_AC_OUTPUT, data))}},
1228 // TPML_AC_CAPABILITIES_DATA
1229 {LIST_MTYPE,
1230     (UINT8) (offsetof(TPML_AC_CAPABILITIES, acCapabilities)),
1231     Type44_MARSHAL_REF,
1232     TPMS_AC_OUTPUT_ARRAY_MARSHAL_INDEX},
1233 // Type00_DATA
1234 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1235     {RANGE(0, sizeof(TPMU_HA), UINT16)}},
1236 // Type01_DATA
1237 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1238     {RANGE(0, sizeof(TPMT_HA), UINT16)}},
1239 // Type02_DATA
1240 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1241     {RANGE(0, 1024, UINT16)}},
1242 // Type03_DATA
1243 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1244     {RANGE(0, MAX_DIGEST_BUFFER, UINT16)}},
1245 // Type04_DATA
1246 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1247     {RANGE(0, MAX_NV_BUFFER_SIZE, UINT16)}},
1248 // Type05_DATA
1249 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1250     {RANGE(0, sizeof(UINT64), UINT16)}},
1251 // Type06_DATA
1252 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8) TPM_RC_SIZE,
1253     {RANGE(0, MAX_SYM_BLOCK_SIZE, UINT16)}},

```

```

1254 // Type07_DATA
1255 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1256   {RANGE(0, sizeof(TPMU_NAME), UINT16)}}},
1257 // Type08_DATA
1258 {MIN_MAX_MTYPE, ONE_BYTES, (UINT8)TPM_RC_VALUE,
1259   {RANGE(PCR_SELECT_MIN, PCR_SELECT_MAX, UINT8)}}},
1260 // Type10_DATA
1261 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1262   {TPM_ST_CREATION}}},
1263 // Type11_DATA
1264 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1265   {TPM_ST_VERIFIED}}},
1266 // Type12_DATA
1267 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 2,
1268   {TPM_ST_AUTH_SECRET, TPM_ST_AUTH_SIGNED}}},
1269 // Type13_DATA
1270 {TABLE_MTYPE, TWO_BYTES, (UINT8)TPM_RC_TAG, 1,
1271   {TPM_ST_HASHCHECK}}},
1272 // Type15_DATA
1273 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1274   {RANGE(0, MAX_CAP_CC, UINT32)}}},
1275 // Type17_DATA
1276 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1277   {RANGE(0, MAX_ALG_LIST_SIZE, UINT32)}}},
1278 // Type18_DATA
1279 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1280   {RANGE(0, MAX_CAP_HANDLES, UINT32)}}},
1281 // Type19_DATA
1282 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1283   {RANGE(2, 8, UINT32)}}},
1284 // Type20_DATA
1285 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1286   {RANGE(0, HASH_COUNT, UINT32)}}},
1287 // Type22_DATA
1288 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1289   {RANGE(0, MAX_CAP_ALGS, UINT32)}}},
1290 // Type23_DATA
1291 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1292   {RANGE(0, MAX_TPM_PROPERTIES, UINT32)}}},
1293 // Type24_DATA
1294 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1295   {RANGE(0, MAX_PCR_PROPERTIES, UINT32)}}},
1296 // Type25_DATA
1297 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1298   {RANGE(0, MAX_ECC_CURVES, UINT32)}}},
1299 // Type26_DATA
1300 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1301   {RANGE(0, MAX_TAGGED_POLICIES, UINT32)}}},
1302 // Type27_DATA
1303 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1304   {RANGE(0, MAX_ACT_DATA, UINT32)}}},
1305 // Type28_DATA
1306 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1307   {RANGE(0, sizeof(TPMS_ATTEST), UINT16)}}},
1308 // Type29_DATA
1309 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1310   {RANGE(0, MAX_SYM_KEY_BYTES, UINT16)}}},
1311 // Type30_DATA
1312 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1313   {RANGE(0, LABEL_MAX_BUFFER, UINT16)}}},
1314 // Type31_DATA
1315 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1316   {RANGE(0, sizeof(TPMS_DERIVE), UINT16)}}},
1317 // Type32_DATA
1318 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1319   {RANGE(0, sizeof(TPMU_SENSITIVE_CREATE), UINT16)}}},

```

```

1320 // Type33_DATA
1321 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1322   {RANGE(0, MAX_RSA_KEY_BYTES, UINT16)}}},
1323 // Type34_DATA
1324 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1325   {RANGE(0, RSA_PRIVATE_SIZE, UINT16)}}},
1326 // Type35_DATA
1327 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1328   {RANGE(0, MAX_ECC_KEY_BYTES, UINT16)}}},
1329 // Type36_DATA
1330 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1331   {RANGE(0, sizeof(TPMU_ENCRYPTED_SECRET), UINT16)}}},
1332 // Type37_DATA
1333 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1334   {RANGE(0, sizeof(TPMT_PUBLIC), UINT16)}}},
1335 // Type38_DATA
1336 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1337   {RANGE(0, PRIVATE_VENDOR_SPECIFIC_BYTES, UINT16)}}},
1338 // Type39_DATA
1339 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1340   {RANGE(0, sizeof(_PRIVATE), UINT16)}}},
1341 // Type40_DATA
1342 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1343   {RANGE(0, sizeof(TPMS_ID_OBJECT), UINT16)}}},
1344 // Type41_DATA
1345 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1346   {RANGE(0, MAX_NV_INDEX_SIZE, UINT16)}}},
1347 // Type42_DATA
1348 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1349   {RANGE(0, MAX_CONTEXT_SIZE, UINT16)}}},
1350 // Type43_DATA
1351 {MIN_MAX_MTYPE, TWO_BYTES, (UINT8)TPM_RC_SIZE,
1352   {RANGE(0, sizeof(TPMS_CONTEXT_DATA), UINT16)}}},
1353 // Type44_DATA
1354 {MIN_MAX_MTYPE, FOUR_BYTES, (UINT8)TPM_RC_SIZE,
1355   {RANGE(0, MAX_AC_CAPABILITIES, UINT32)}}
1356 };
1357 #endif // TABLE_DRIVEN_MARSHAL

```

9.11 MathOnByteBuffers.c

9.11.1 Introduction

This file contains implementation of the math functions that are performed with canonical integers in byte buffers. The canonical integer is big-endian bytes.

```
1  #include "Tpm.h"
```

9.11.2 Functions

9.11.2.1 UnsignedCmpB()

This function compare two unsigned values. The values are byte-aligned, big-endian numbers (e.g, a hash).

Return Value	Meaning
1	if (a > b)
0	if (a = b)
-1	if (a < b)

```
2  LIB_EXPORT int
3  UnsignedCompareB(
4      UINT32      aSize,          // IN: size of a
5      const BYTE  *a,            // IN: a
6      UINT32      bSize,          // IN: size of b
7      const BYTE  *b,            // IN: b
8  )
9  {
10     UINT32      i;
11     if(aSize > bSize)
12         return 1;
13     else if(aSize < bSize)
14         return -1;
15     else
16     {
17         for(i = 0; i < aSize; i++)
18         {
19             if(a[i] != b[i])
20                 return (a[i] > b[i]) ? 1 : -1;
21         }
22     }
23     return 0;
24 }
```

9.11.2.2 SignedCompareB()

Compare two signed integers:

Return Value	Meaning
1	if $a > b$
0	if $a = b$
-1	if $a < b$

```

25  int
26  SignedCompareB(
27      const UINT32    aSize,          // IN: size of a
28      const BYTE      *a,            // IN: a buffer
29      const UINT32    bSize,          // IN: size of b
30      const BYTE      *b,            // IN: b buffer
31  )
32  {
33      int    signA, signB;           // sign of a and b
34
35      // For positive or 0, sign_a is 1
36      // for negative, sign_a is 0
37      signA = ((a[0] & 0x80) == 0) ? 1 : 0;
38
39      // For positive or 0, sign_b is 1
40      // for negative, sign_b is 0
41      signB = ((b[0] & 0x80) == 0) ? 1 : 0;
42
43      if(signA != signB)
44      {
45          return signA - signB;
46      }
47      if(signA == 1)
48          // do unsigned compare function
49          return UnsignedCompareB(aSize, a, bSize, b);
50      else
51          // do unsigned compare the other way
52          return 0 - UnsignedCompareB(aSize, a, bSize, b);
53  }

```

9.11.2.3 ModExpB()

This function is used to do modular exponentiation in support of RSA. The most typical uses are: $c = m^e \bmod n$ (RSA encrypt) and $m = c^d \bmod n$ (RSA decrypt). When doing decryption, the e parameter of the function will contain the private exponent d instead of the public exponent e .

If the results will not fit in the provided buffer, an error is returned (CRYPT_ERROR_UNDERFLOW). If the results is smaller than the buffer, the results is de-normalized.

This version is intended for use with RSA and requires that m be less than n .

Error Return	Meaning
TPM_RC_SIZE	number to exponentiate is larger than the modulus
TPM_RC_NO_RESULT	result will not fit into the provided buffer

```

54  TPM_RC
55  ModExpB(
56      UINT32    cSize,          // IN: the size of the output buffer. It will
57                                // need to be the same size as the modulus
58      BYTE      *c,            // OUT: the buffer to receive the results
59                                // (c->size must be set to the maximum size
60                                // for the returned value)
61      const UINT32    mSize,
62      const BYTE      *m,          // IN: number to exponentiate

```

```

63     const UINT32      eSize,
64     const BYTE        *e,                // IN: power
65     const UINT32      nSize,
66     const BYTE        *n                // IN: modulus
67 )
68 {
69     BN_MAX(bnC);
70     BN_MAX(bnM);
71     BN_MAX(bnE);
72     BN_MAX(bnN);
73     NUMBYTES      tSize = (NUMBYTES)nSize;
74     TPM_RC        retVal = TPM_RC_SUCCESS;
75
76     // Convert input parameters
77     BnFromBytes(bnM, m, (NUMBYTES)mSize);
78     BnFromBytes(bnE, e, (NUMBYTES)eSize);
79     BnFromBytes(bnN, n, (NUMBYTES)nSize);
80
81     // Make sure that the output is big enough to hold the result
82     // and that 'm' is less than 'n' (the modulus)
83     if(cSize < nSize)
84         ERROR_RETURN(TPM_RC_NO_RESULT);
85     if(BnUnsignedCmp(bnM, bnN) >= 0)
86         ERROR_RETURN(TPM_RC_SIZE);
87     BnModExp(bnC, bnM, bnE, bnN);
88     BnToBytes(bnC, c, &tSize);
89 Exit:
90     return retVal;
91 }

```

9.11.2.4 DivideB()

Divide an integer (n) by an integer (d) producing a quotient (q) and a remainder (r). If q or r is not needed, then the pointer to them may be set to NULL.

Error Return	Meaning
TPM_RC_NO_RESULT	q or r is too small to receive the result

```

92 LIB_EXPORT TPM_RC
93 DivideB(
94     const TPM2B      *n,                // IN: numerator
95     const TPM2B      *d,                // IN: denominator
96     TPM2B             *q,                // OUT: quotient
97     TPM2B             *r                // OUT: remainder
98 )
99 {
100     BN_MAX_INITIALIZED(bnN, n);
101     BN_MAX_INITIALIZED(bnD, d);
102     BN_MAX(bnQ);
103     BN_MAX(bnR);
104     //
105     // Do divide with converted values
106     BnDiv(bnQ, bnR, bnN, bnD);
107
108     // Convert the BIGNUM result back to 2B format using the size of the original
109     // number
110     if(q != NULL)
111         if(!BnTo2B(bnQ, q, q->size))
112             return TPM_RC_NO_RESULT;
113     if(r != NULL)
114         if(!BnTo2B(bnR, r, r->size))
115             return TPM_RC_NO_RESULT;
116     return TPM_RC_SUCCESS;

```

```
117 }
```

9.11.2.5 AdjustNumberB()

Remove/add leading zeros from a number in a TPM2B. Will try to make the number by adding or removing leading zeros. If the number is larger than the requested size, it will make the number as small as possible. Setting *requestedSize* to zero is equivalent to requesting that the number be normalized.

```
118 UINT16
119 AdjustNumberB(
120     TPM2B      *num,
121     UINT16      requestedSize
122 )
123 {
124     BYTE      *from;
125     UINT16      i;
126     // See if number is already the requested size
127     if(num->size == requestedSize)
128         return requestedSize;
129     from = num->buffer;
130     if (num->size > requestedSize)
131     {
132         // This is a request to shift the number to the left (remove leading zeros)
133         // Find the first non-zero byte. Don't look past the point where removing
134         // more zeros would make the number smaller than requested, and don't throw
135         // away any significant digits.
136         for(i = num->size; *from == 0 && i > requestedSize; from++, i--);
137         if(i < num->size)
138         {
139             num->size = i;
140             MemoryCopy(num->buffer, from, i);
141         }
142     }
143     // This is a request to shift the number to the right (add leading zeros)
144     else
145     {
146         MemoryCopy(&num->buffer[requestedSize - num->size], num->buffer, num->size);
147         MemorySet(num->buffer, 0, requestedSize - num->size);
148         num->size = requestedSize;
149     }
150     return num->size;
151 }
```

9.11.2.6 ShiftLeft()

This function shifts a byte buffer (a TPM2B) one byte to the left. That is, the most significant bit of the most significant byte is lost.

```
152 TPM2B *
153 ShiftLeft(
154     TPM2B      *value      // IN/OUT: value to shift and shifted value out
155 )
156 {
157     UINT16      count = value->size;
158     BYTE      *buffer = value->buffer;
159     if(count > 0)
160     {
161         for(count -= 1; count > 0; buffer++, count--)
162         {
163             buffer[0] = (buffer[0] << 1) + ((buffer[1] & 0x80) ? 1 : 0);
164         }
165         *buffer <<= 1;
166     }
```



```
166     }  
167     return value;  
168 }
```

DRAFT

9.12 Memory.c

9.12.1 Description

This file contains a set of miscellaneous memory manipulation routines. Many of the functions have the same semantics as functions defined in `string.h`. Those functions are not used directly in the TPM because they are not *safe*.

This version uses `string.h` after adding guards. This is because the math libraries invariably use those functions so it is not practical to prevent those library functions from being pulled into the build.

9.12.2 Includes and Data Definitions

```
1 #include "Tpm.h"
2 #include "Memory_fp.h"
```

9.12.3 Functions

9.12.3.1 MemoryCopy()

This is an alias for `memmove`. This is used in place of `memcpy` because some of the moves may overlap and rather than try to make sure that `memmove` is used when necessary, it is always used.

```
3 void
4 MemoryCopy(
5     void        *dest,
6     const void  *src,
7     int         sSize
8 )
9 {
10     if(dest != src)
11         memmove(dest, src, sSize);
12 }
```

9.12.3.2 MemoryEqual()

This function indicates if two buffers have the same values in the indicated number of bytes.

Return Value	Meaning
TRUE(1)	all octets are the same
FALSE(0)	all octets are not the same

```
13 BOOL
14 MemoryEqual(
15     const void    *buffer1,    // IN: compare buffer1
16     const void    *buffer2,    // IN: compare buffer2
17     unsigned int   size        // IN: size of bytes being compared
18 )
19 {
20     BYTE          equal = 0;
21     const BYTE    *b1 = (BYTE *)buffer1;
22     const BYTE    *b2 = (BYTE *)buffer2;
23     //
24     // Compare all bytes so that there is no leakage of information
25     // due to timing differences.
26     for(; size > 0; size--)
```

```

27     equal |= (*b1++ ^ *b2++);
28     return (equal == 0);
29 }

```

9.12.3.3 MemoryCopy2B()

This function copies a TPM2B. This can be used when the TPM2B types are the same or different.

This function returns the number of octets in the data buffer of the TPM2B.

```

30 LIB_EXPORT INT16
31 MemoryCopy2B(
32     TPM2B      *dest,           // OUT: receiving TPM2B
33     const TPM2B *source,        // IN: source TPM2B
34     unsigned int dSize          // IN: size of the receiving buffer
35 )
36 {
37     pAssert(dest != NULL);
38     if(source == NULL)
39         dest->size = 0;
40     else
41     {
42         pAssert(source->size <= dSize);
43         MemoryCopy(dest->buffer, source->buffer, source->size);
44         dest->size = source->size;
45     }
46     return dest->size;
47 }

```

9.12.3.4 MemoryConcat2B()

This function will concatenate the buffer contents of a TPM2B to the buffer contents of another TPM2B and adjust the size accordingly ($a := (a \parallel b)$).

```

48 void
49 MemoryConcat2B(
50     TPM2B      *aInOut,        // IN/OUT: destination 2B
51     TPM2B      *bIn,           // IN: second 2B
52     unsigned int aMaxSize       // IN: The size of aInOut.buffer (max values for
53                                 // aInOut.size)
54 )
55 {
56     pAssert(bIn->size <= aMaxSize - aInOut->size);
57     MemoryCopy(&aInOut->buffer[aInOut->size], &bIn->buffer, bIn->size);
58     aInOut->size = aInOut->size + bIn->size;
59     return;
60 }

```

9.12.3.5 MemoryEqual2B()

This function will compare two TPM2B structures. To be equal, they need to be the same size and the buffer contexts need to be the same in all octets.

Return Value	Meaning
TRUE(1)	size and buffer contents are the same
FALSE(0)	size or buffer contents are not the same

```

61 BOOL
62 MemoryEqual2B(

```

```

63     const TPM2B      *aIn,          // IN: compare value
64     const TPM2B      *bIn          // IN: compare value
65 )
66 {
67     if(aIn->size != bIn->size)
68         return FALSE;
69     return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);
70 }

```

9.12.3.6 MemorySet()

This function will set all the octets in the specified memory range to the specified octet value.

NOTE A previous version had an additional parameter (*dSize*) that was intended to make sure that the destination would not be overrun. The problem is that, in use, all that was happening was that the value of size was used for *dSize* so there was no benefit in the extra parameter.

```

71 void
72 MemorySet(
73     void          *dest,
74     int           value,
75     size_t        size
76 )
77 {
78     memset(dest, value, size);
79 }

```

9.12.3.7 MemoryPad2B()

Function to pad a TPM2B with zeros and adjust the size.

```

80 void
81 MemoryPad2B(
82     TPM2B      *b,
83     UINT16      newSize
84 )
85 {
86     MemorySet(&b->buffer[b->size], 0, newSize - b->size);
87     b->size = newSize;
88 }

```

9.12.3.8 Uint16ToByteArray()

Function to write an integer to a byte array

```

89 void
90 Uint16ToByteArray(
91     UINT16      i,
92     BYTE        *a
93 )
94 {
95     a[1] = (BYTE) (i); i >>= 8;
96     a[0] = (BYTE) (i);
97 }

```

9.12.3.9 Uint32ToByteArray()

Function to write an integer to a byte array

```

98 void

```

```

99  UInt32ToByteArray(
100      UINT32          i,
101      BYTE            *a
102  )
103  {
104      a[3] = (BYTE) (i); i >>= 8;
105      a[2] = (BYTE) (i); i >>= 8;
106      a[1] = (BYTE) (i); i >>= 8;
107      a[0] = (BYTE) (i);
108  }

```

9.12.3.10 UInt64ToByteArray()

Function to write an integer to a byte array

```

109  void
110  UInt64ToByteArray(
111      UINT64          i,
112      BYTE            *a
113  )
114  {
115      a[7] = (BYTE) (i); i >>= 8;
116      a[6] = (BYTE) (i); i >>= 8;
117      a[5] = (BYTE) (i); i >>= 8;
118      a[4] = (BYTE) (i); i >>= 8;
119      a[3] = (BYTE) (i); i >>= 8;
120      a[2] = (BYTE) (i); i >>= 8;
121      a[1] = (BYTE) (i); i >>= 8;
122      a[0] = (BYTE) (i);
123  }

```

9.12.3.11 ByteArrayToUInt8()

Function to write a **UINT8** to a byte array. This is included for completeness and to allow certain macro expansions

```

124  UINT8
125  ByteArrayToUInt8(
126      BYTE            *a
127  )
128  {
129      return *a;
130  }

```

9.12.3.12 ByteArrayToUInt16()

Function to write an integer to a byte array

```

131  UINT16
132  ByteArrayToUInt16(
133      BYTE            *a
134  )
135  {
136      return ((UINT16) a[0] << 8) + a[1];
137  }

```

9.12.3.13 ByteArrayToUInt32()

Function to write an integer to a byte array

```
138  UINT32  
139  ByteArrayToUint32(  
140      BYTE          *a  
141  )  
142  {  
143      return (UINT32)((((UINT32)a[0] << 8) + a[1]) << 8) + (UINT32)a[2]) << 8) + a[3];  
144  }
```

9.12.3.14 ByteArrayToUint64()

Function to write an integer to a byte array

```
145  UINT64  
146  ByteArrayToUint64(  
147      BYTE          *a  
148  )  
149  {  
150      return (((UINT64)BYTE_ARRAY_TO_UINT32(a)) << 32) + BYTE_ARRAY_TO_UINT32(&a[4]);  
151  }
```

9.13 Power.c

9.13.1 Description

This file contains functions that receive the simulated power state transitions of the TPM.

9.13.2 Includes and Data Definitions

```
1  #define POWER_C
2  #include "Tpm.h"
```

9.13.3 Functions

9.13.3.1 TPMInit()

This function is used to process a power on event.

```
3  void
4  TPMInit(
5      void
6  )
7  {
8      // Set state as not initialized. This means that Startup is required
9      g_initialized = FALSE;
10     return;
11 }
```

9.13.3.2 TPMRegisterStartup()

This function registers the fact that the TPM has been initialized (a TPM2_Startup() has completed successfully).

```
12 BOOL
13 TPMRegisterStartup(
14     void
15 )
16 {
17     g_initialized = TRUE;
18     return TRUE;
19 }
```

9.13.3.3 TPMIsStarted()

Indicates if the TPM has been initialized (a TPM2_Startup() has completed successfully after a _TPM_Init).

Return Value	Meaning
TRUE(1)	TPM has been initialized
FALSE(0)	TPM has not been initialized

```
20 BOOL
21 TPMIsStarted(
22     void
23 )
24 {
```



```
25     return g_initialized;  
26 }
```

DRAFT

9.14 PropertyCap.c

9.14.1 Description

This file contains the functions that are used for accessing the TPM_CAP_TPM_PROPERTY values.

9.14.2 Includes

```
1  #include "Tpm.h"
```

9.14.3 Functions

9.14.3.1 TPMPropertyIsDefined()

This function accepts a property selection and, if so, sets *value* to the value of the property.

All the fixed values are vendor dependent or determined by a platform-specific specification. The values in the table below are examples and should be changed by the vendor.

Return Value	Meaning
TRUE(1)	referenced property exists and <i>value</i> set
FALSE(0)	referenced property does not exist

```
2  static BOOL
3  TPMPropertyIsDefined(
4      TPM_PT          property,      // IN: property
5      UINT32          *value         // OUT: property value
6  )
7  {
8      switch(property)
9      {
10         case TPM_PT_FAMILY_INDICATOR:
11             // from the title page of the specification
12             // For this specification, the value is "2.0".
13             *value = TPM_SPEC_FAMILY;
14             break;
15         case TPM_PT_LEVEL:
16             // from the title page of the specification
17             *value = TPM_SPEC_LEVEL;
18             break;
19         case TPM_PT_REVISION:
20             // from the title page of the specification
21             *value = TPM_SPEC_VERSION;
22             break;
23         case TPM_PT_DAY_OF_YEAR:
24             // computed from the date value on the title page of the specification
25             *value = TPM_SPEC_DAY_OF_YEAR;
26             break;
27         case TPM_PT_YEAR:
28             // from the title page of the specification
29             *value = TPM_SPEC_YEAR;
30             break;
31         case TPM_PT_MANUFACTURER:
32             // vendor ID unique to each TPM manufacturer
33             *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
34             break;
35         case TPM_PT_VENDOR_STRING_1:
36             // first four characters of the vendor ID string
37             *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
```

```

38         break;
39     case TPM_PT_VENDOR_STRING_2:
40         // second four characters of the vendor ID string
41 #ifdef VENDOR_STRING_2
42         *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_2);
43 #else
44         *value = 0;
45 #endif
46         break;
47     case TPM_PT_VENDOR_STRING_3:
48         // third four characters of the vendor ID string
49 #ifdef VENDOR_STRING_3
50         *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_3);
51 #else
52         *value = 0;
53 #endif
54         break;
55     case TPM_PT_VENDOR_STRING_4:
56         // fourth four characters of the vendor ID string
57 #ifdef VENDOR_STRING_4
58         *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
59 #else
60         *value = 0;
61 #endif
62         break;
63     case TPM_PT_VENDOR_TPM_TYPE:
64         // vendor-defined value indicating the TPM model
65         *value = 1;
66         break;
67     case TPM_PT_FIRMWARE_VERSION_1:
68         // more significant 32-bits of a vendor-specific value
69         *value = gp.firmwareV1;
70         break;
71     case TPM_PT_FIRMWARE_VERSION_2:
72         // less significant 32-bits of a vendor-specific value
73         *value = gp.firmwareV2;
74         break;
75     case TPM_PT_INPUT_BUFFER:
76         // maximum size of TPM2B_MAX_BUFFER
77         *value = MAX_DIGEST_BUFFER;
78         break;
79     case TPM_PT_HR_TRANSIENT_MIN:
80         // minimum number of transient objects that can be held in TPM
81         // RAM
82         *value = MAX_LOADED_OBJECTS;
83         break;
84     case TPM_PT_HR_PERSISTENT_MIN:
85         // minimum number of persistent objects that can be held in
86         // TPM NV memory
87         // In this implementation, there is no minimum number of
88         // persistent objects.
89         *value = MIN_EVICT_OBJECTS;
90         break;
91     case TPM_PT_HR_LOADED_MIN:
92         // minimum number of authorization sessions that can be held in
93         // TPM RAM
94         *value = MAX_LOADED_SESSIONS;
95         break;
96     case TPM_PT_ACTIVE_SESSIONS_MAX:
97         // number of authorization sessions that may be active at a time
98         *value = MAX_ACTIVE_SESSIONS;
99         break;
100    case TPM_PT_PCR_COUNT:
101        // number of PCR implemented
102        *value = IMPLEMENTATION_PCR;
103        break;

```

```

104     case TPM_PT_PCR_SELECT_MIN:
105         // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
106         *value = PCR_SELECT_MIN;
107         break;
108     case TPM_PT_CONTEXT_GAP_MAX:
109         // maximum allowed difference (unsigned) between the contextID
110         // values of two saved session contexts
111         *value = ((UINT32)1 << (sizeof(CONTEXT_SLOT) * 8)) - 1;
112         break;
113     case TPM_PT_NV_COUNTERS_MAX:
114         // maximum number of NV indexes that are allowed to have the
115         // TPMA_NV_COUNTER attribute SET
116         // In this implementation, there is no limitation on the number
117         // of counters, except for the size of the NV Index memory.
118         *value = 0;
119         break;
120     case TPM_PT_NV_INDEX_MAX:
121         // maximum size of an NV index data area
122         *value = MAX_NV_INDEX_SIZE;
123         break;
124     case TPM_PT_MEMORY:
125         // a TPMA_MEMORY indicating the memory management method for the TPM
126     {
127         union
128         {
129             TPMA_MEMORY    att;
130             UINT32         u32;
131         } attributes = { TPMA_ZERO_INITIALIZER() };
132         SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, sharedNV);
133         SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, objectCopiedToRam);
134
135         // Note: For a LSB0 machine, the bits in a bit field are in the correct
136         // order even if the machine is MSB0. For a MSB0 machine, a TPMA will
137         // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
138         // be NO) so the bits are manipulate correctly.
139         *value = attributes.u32;
140         break;
141     }
142     case TPM_PT_CLOCK_UPDATE:
143         // interval, in seconds, between updates to the copy of
144         // TPMS_TIME_INFO .clock in NV
145         *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
146         break;
147     case TPM_PT_CONTEXT_HASH:
148         // algorithm used for the integrity hash on saved contexts and
149         // for digesting the fuData of TPM2_FirmwareRead()
150         *value = CONTEXT_INTEGRITY_HASH_ALG;
151         break;
152     case TPM_PT_CONTEXT_SYM:
153         // algorithm used for encryption of saved contexts
154         *value = CONTEXT_ENCRYPT_ALG;
155         break;
156     case TPM_PT_CONTEXT_SYM_SIZE:
157         // size of the key used for encryption of saved contexts
158         *value = CONTEXT_ENCRYPT_KEY_BITS;
159         break;
160     case TPM_PT_ORDERLY_COUNT:
161         // maximum difference between the volatile and non-volatile
162         // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
163         *value = MAX_ORDERLY_COUNT;
164         break;
165     case TPM_PT_MAX_COMMAND_SIZE:
166         // maximum value for 'commandSize'
167         *value = MAX_COMMAND_SIZE;
168         break;
169     case TPM_PT_MAX_RESPONSE_SIZE:

```

```

170         // maximum value for 'responseSize'
171         *value = MAX_RESPONSE_SIZE;
172         break;
173     case TPM_PT_MAX_DIGEST:
174         // maximum size of a digest that can be produced by the TPM
175         *value = sizeof(TPMU_HA);
176         break;
177     case TPM_PT_MAX_OBJECT_CONTEXT:
178         // Header has 'sequence', 'handle' and 'hierarchy'
179         #define SIZE_OF_CONTEXT_HEADER \
180             sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) + sizeof(TPMI_RH_HIERARCHY)
181         #define SIZE_OF_CONTEXT_INTEGRITY (sizeof(UINT16) + CONTEXT_INTEGRITY_HASH_SIZE)
182         #define SIZE_OF_FINGERPRINT      sizeof(UINT64)
183         #define SIZE_OF_CONTEXT_BLOB_OVERHEAD \
184             (sizeof(UINT16) + SIZE_OF_CONTEXT_INTEGRITY + SIZE_OF_FINGERPRINT)
185         #define SIZE_OF_CONTEXT_OVERHEAD \
186             (SIZE_OF_CONTEXT_HEADER + SIZE_OF_CONTEXT_BLOB_OVERHEAD)
187     #if 0
188         // maximum size of a TPMS_CONTEXT that will be returned by
189         // TPM2_ContextSave for object context
190         *value = 0;
191         // adding sequence, saved handle and hierarchy
192         *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
193             sizeof(TPMI_RH_HIERARCHY);
194         // add size field in TPM2B_CONTEXT
195         *value += sizeof(UINT16);
196         // add integrity hash size
197         *value += sizeof(UINT16) +
198             CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
199         // Add fingerprint size, which is the same as sequence size
200         *value += sizeof(UINT64);
201         // Add OBJECT structure size
202         *value += sizeof(OBJECT);
203     #else
204         // the maximum size of a TPMS_CONTEXT that will be returned by
205         // TPM2_ContextSave for object context
206         *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(OBJECT);
207     #endif
208     break;
209     case TPM_PT_MAX_SESSION_CONTEXT:
210     #if 0
211         // the maximum size of a TPMS_CONTEXT that will be returned by
212         // TPM2_ContextSave for object context
213         *value = 0;
214         // adding sequence, saved handle and hierarchy
215         *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
216             sizeof(TPMI_RH_HIERARCHY);
217         // Add size field in TPM2B_CONTEXT
218         *value += sizeof(UINT16);
219         // Add integrity hash size
220         *value += sizeof(UINT16) +
221             CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
222         // Add fingerprint size, which is the same as sequence size
223         *value += sizeof(UINT64);
224         // Add SESSION structure size
225         *value += sizeof(SESSION);
226     #else
227         // the maximum size of a TPMS_CONTEXT that will be returned by
228         // TPM2_ContextSave for object context
229         *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(SESSION);
230     #endif
231     break;
232     case TPM_PT_PS_FAMILY_INDICATOR:
233         // platform specific values for the TPM_PT_PS parameters from
234         // the relevant platform-specific specification
235

```

```

236         // In this reference implementation, all of these values are 0.
237         *value = PLATFORM_FAMILY;
238         break;
239     case TPM_PT_PS_LEVEL:
240         // level of the platform-specific specification
241         *value = PLATFORM_LEVEL;
242         break;
243     case TPM_PT_PS_REVISION:
244         // specification Revision times 100 for the platform-specific
245         // specification
246         *value = PLATFORM_VERSION;
247         break;
248     case TPM_PT_PS_DAY_OF_YEAR:
249         // platform-specific specification day of year using TCG calendar
250         *value = PLATFORM_DAY_OF_YEAR;
251         break;
252     case TPM_PT_PS_YEAR:
253         // platform-specific specification year using the CE
254         *value = PLATFORM_YEAR;
255         break;
256     case TPM_PT_SPLIT_MAX:
257         // number of split signing operations supported by the TPM
258         *value = 0;
259 #if ALG_ECC
260         *value = sizeof(gr.commitArray) * 8;
261 #endif
262         break;
263     case TPM_PT_TOTAL_COMMANDS:
264         // total number of commands implemented in the TPM
265         // Since the reference implementation does not have any
266         // vendor-defined commands, this will be the same as the
267         // number of library commands.
268     {
269 #if COMPRESSED_LISTS
270         (*value) = COMMAND_COUNT;
271 #else
272         COMMAND_INDEX      commandIndex;
273         *value = 0;
274
275         // scan all implemented commands
276         for(commandIndex = GetClosestCommandIndex(0);
277            commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
278            commandIndex = GetNextCommandIndex(commandIndex))
279         {
280             (*value)++;    // count of all implemented
281         }
282 #endif
283         break;
284     }
285     case TPM_PT_LIBRARY_COMMANDS:
286         // number of commands from the TPM library that are implemented
287     {
288 #if COMPRESSED_LISTS
289         *value = LIBRARY_COMMAND_ARRAY_SIZE;
290 #else
291         COMMAND_INDEX      commandIndex;
292         *value = 0;
293
294         // scan all implemented commands
295         for(commandIndex = GetClosestCommandIndex(0);
296            commandIndex < LIBRARY_COMMAND_ARRAY_SIZE;
297            commandIndex = GetNextCommandIndex(commandIndex))
298         {
299             (*value)++;
300         }
301 #endif

```

```

302         break;
303     }
304     case TPM_PT_VENDOR_COMMANDS:
305         // number of vendor commands that are implemented
306         *value = VENDOR_COMMAND_ARRAY_SIZE;
307         break;
308     case TPM_PT_NV_BUFFER_MAX:
309         // Maximum data size in an NV write command
310         *value = MAX_NV_BUFFER_SIZE;
311         break;
312     case TPM_PT_MODES:
313 #if FIPS_COMPLIANT
314         *value = 1;
315 #else
316         *value = 0;
317 #endif
318         break;
319     case TPM_PT_MAX_CAP_BUFFER:
320         *value = MAX_CAP_BUFFER;
321         break;
322
323     // Start of variable commands
324     case TPM_PT_PERMANENT:
325         // TPMA_PERMANENT
326         {
327             union {
328                 TPMA_PERMANENT attr;
329                 UINT32 u32;
330             } flags = { TPMA_ZERO_INITIALIZER() };
331             if(gp.ownerAuth.t.size != 0)
332                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, ownerAuthSet);
333             if(gp.endorsementAuth.t.size != 0)
334                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, endorsementAuthSet);
335             if(gp.lockoutAuth.t.size != 0)
336                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, lockoutAuthSet);
337             if(gp.disableClear)
338                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, disableClear);
339             if(gp.failedTries >= gp.maxTries)
340                 SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, inLockout);
341             // In this implementation, EPS is always generated by TPM
342             SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, tpmGeneratedEPS);
343
344             // Note: For a LSb0 machine, the bits in a bit field are in the correct
345             // order even if the machine is MSB0. For a MSb0 machine, a TPMA will
346             // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
347             // be NO) so the bits are manipulate correctly.
348             *value = flags.u32;
349             break;
350         }
351     case TPM_PT_STARTUP_CLEAR:
352         // TPMA_STARTUP_CLEAR
353         {
354             union {
355                 TPMA_STARTUP_CLEAR attr;
356                 UINT32 u32;
357             } flags = { TPMA_ZERO_INITIALIZER() };
358
359             //
360             if(g_phEnable)
361                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnable);
362             if(gc.shEnable)
363                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, shEnable);
364             if(gc.ehEnable)
365                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, ehEnable);
366             if(gc.phEnableNV)
367                 SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnableNV);
368             if(g_prevOrderlyState != SU_NONE_VALUE)

```



```

368         SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, orderly);
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_HR_NV_INDEX:
378         // number of NV indexes currently defined
379         *value = NvCapGetIndexNumber();
380         break;
381     case TPM_PT_HR_LOADED:
382         // number of authorization sessions currently loaded into TPM
383         // RAM
384         *value = SessionCapGetLoadedNumber();
385         break;
386     case TPM_PT_HR_LOADED_AVAIL:
387         // number of additional authorization sessions, of any type,
388         // that could be loaded into TPM RAM
389         *value = SessionCapGetLoadedAvail();
390         break;
391     case TPM_PT_HR_ACTIVE:
392         // number of active authorization sessions currently being
393         // tracked by the TPM
394         *value = SessionCapGetActiveNumber();
395         break;
396     case TPM_PT_HR_ACTIVE_AVAIL:
397         // number of additional authorization sessions, of any type,
398         // that could be created
399         *value = SessionCapGetActiveAvail();
400         break;
401     case TPM_PT_HR_TRANSIENT_AVAIL:
402         // estimate of the number of additional transient objects that
403         // could be loaded into TPM RAM
404         *value = ObjectCapGetTransientAvail();
405         break;
406     case TPM_PT_HR_PERSISTENT:
407         // number of persistent objects currently loaded into TPM
408         // NV memory
409         *value = NvCapGetPersistentNumber();
410         break;
411     case TPM_PT_HR_PERSISTENT_AVAIL:
412         // number of additional persistent objects that could be loaded
413         // into NV memory
414         *value = NvCapGetPersistentAvail();
415         break;
416     case TPM_PT_NV_COUNTERS:
417         // number of defined NV indexes that have NV TPMA_NV_COUNTER
418         // attribute SET
419         *value = NvCapGetCounterNumber();
420         break;
421     case TPM_PT_NV_COUNTERS_AVAIL:
422         // number of additional NV indexes that can be defined with their
423         // TPMA_NV_COUNTER attribute SET
424         *value = NvCapGetCounterAvail();
425         break;
426     case TPM_PT_ALGORITHM_SET:
427         // region code for the TPM
428         *value = gp.algorithmSet;
429         break;
430     case TPM_PT_LOADED_CURVES:
431 #if ALG_ECC
432         // number of loaded ECC curves
433         *value = ECC_CURVE_COUNT;

```

```

434 #else // ALG_ECC
435     *value = 0;
436 #endif // ALG_ECC
437     break;
438     case TPM_PT_LOCKOUT_COUNTER:
439         // current value of the lockout counter
440         *value = gp.failedTries;
441         break;
442     case TPM_PT_MAX_AUTH_FAIL:
443         // number of authorization failures before DA lockout is invoked
444         *value = gp.maxTries;
445         break;
446     case TPM_PT_LOCKOUT_INTERVAL:
447         // number of seconds before the value reported by
448         // TPM_PT_LOCKOUT_COUNTER is decremented
449         *value = gp.recoveryTime;
450         break;
451     case TPM_PT_LOCKOUT_RECOVERY:
452         // number of seconds after a lockoutAuth failure before use of
453         // lockoutAuth may be attempted again
454         *value = gp.lockoutRecovery;
455         break;
456     case TPM_PT_NV_WRITE_RECOVERY:
457         // number of milliseconds before the TPM will accept another command
458         // that will modify NV.
459         // This should make a call to the platform code that is doing rate
460         // limiting of NV. Rate limiting is not implemented in the reference
461         // code so no call is made.
462         *value = 0;
463         break;
464     case TPM_PT_AUDIT_COUNTER_0:
465         // high-order 32 bits of the command audit counter
466         *value = (UINT32) (gp.auditCounter >> 32);
467         break;
468     case TPM_PT_AUDIT_COUNTER_1:
469         // low-order 32 bits of the command audit counter
470         *value = (UINT32) (gp.auditCounter);
471         break;
472     default:
473         // property is not defined
474         return FALSE;
475         break;
476 }
477 return TRUE;
478 }

```

9.14.3.2 TPMCapGetProperties()

This function is used to get the TPM_PT values. The search of properties will start at *property* and continue until *propertyList* has as many values as will fit, or the last property has been reported, or the list has as many values as requested in *count*.

Return Value	Meaning
YES	more properties are available
NO	no more properties to be reported

```

479 TPMI_YES_NO
480 TPMCapGetProperties(
481     TPM_PT                property,    // IN: the starting TPM property
482     UINT32                count,      // IN: maximum number of returned
483                                     // properties
484     TPML_TAGGED_TPM_PROPERTY *propertyList // OUT: property list

```

```

485     )
486 {
487     TPMI_YES_NO    more = NO;
488     UINT32         i;
489     UINT32         nextGroup;
490
491     // initialize output property list
492     propertyList->count = 0;
493
494     // maximum count of properties we may return is MAX_PCR_PROPERTIES
495     if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;
496
497     // if property is less than PT_FIXED, start from PT_FIXED
498     if(property < PT_FIXED)
499         property = PT_FIXED;
500     // There is only the fixed and variable groups with the variable group coming
501     // last
502     if(property >= (PT_VAR + PT_GROUP))
503         return more;
504
505     // Don't read past the end of the selected group
506     nextGroup = ((property / PT_GROUP) * PT_GROUP) + PT_GROUP;
507
508     // Scan through the TPM properties of the requested group.
509     for(i = property; i < nextGroup; i++)
510     {
511         UINT32         value;
512         // if we have hit the end of the group, quit
513         if(i != property && ((i % PT_GROUP) == 0))
514             break;
515         if(TPMPropertyIsDefined((TPM_PT)i, &value))
516         {
517             if(propertyList->count < count)
518             {
519                 // If the list is not full, add this property
520                 propertyList->tpmProperty[propertyList->count].property =
521                     (TPM_PT)i;
522                 propertyList->tpmProperty[propertyList->count].value = value;
523                 propertyList->count++;
524             }
525             else
526             {
527                 // If the return list is full but there are more properties
528                 // available, set the indication and exit the loop.
529                 more = YES;
530                 break;
531             }
532         }
533     }
534     return more;
535 }

```

9.15 Response.c

9.15.1 Description

This file contains the common code for building a response header, including setting the size of the structure. *command* may be NULL if result is not TPM_RC_SUCCESS.

9.15.2 Includes and Defines

```
1  #include "Tpm.h"
```

9.15.3 BuildResponseHeader()

Adds the response header to the response. It will update *command*→*parameterSize* to indicate the total size of the response.

```
2  void
3  BuildResponseHeader(
4      COMMAND      *command,      // IN: main control structure
5      BYTE          *buffer,      // OUT: the output buffer
6      TPM_RC        result        // IN: the response code
7  )
8  {
9      TPM_ST        tag;
10     UINT32         size;
11
12     if(result != TPM_RC_SUCCESS)
13     {
14         tag = TPM_ST_NO_SESSIONS;
15         size = 10;
16     }
17     else
18     {
19         tag = command->tag;
20         // Compute the overall size of the response
21         size = STD_RESPONSE_HEADER + command->handleNum * sizeof(TPM_HANDLE);
22         size += command->parameterSize;
23         size += (command->tag == TPM_ST_SESSIONS) ?
24             command->authSize + sizeof(UINT32) : 0;
25     }
26     TPM_ST_Marshal(&tag, &buffer, NULL);
27     UINT32_Marshal(&size, &buffer, NULL);
28     TPM_RC_Marshal(&result, &buffer, NULL);
29     if(result == TPM_RC_SUCCESS)
30     {
31         if(command->handleNum > 0)
32             TPM_HANDLE_Marshal(&command->handles[0], &buffer, NULL);
33         if(tag == TPM_ST_SESSIONS)
34             UINT32_Marshal((UINT32 *) &command->parameterSize, &buffer, NULL);
35     }
36     command->parameterSize = size;
37 }
```

9.16 ResponseCodeProcessing.c

9.16.1 Description

This file contains the miscellaneous functions for processing response codes.

NOTE Currently, there is only one.

9.16.2 Includes and Defines

```
1  #include "Tpm.h"
```

9.16.3 RcSafeAddToResult()

Adds a modifier to a response code as long as the response code allows a modifier and no modifier has already been added.

```
2  TPM_RC  
3  RcSafeAddToResult(  
4      TPM_RC      responseCode,  
5      TPM_RC      modifier  
6  )  
7  {  
8      if((_responseCode & RC_FMT1) && !(responseCode & 0xf40))  
9          return responseCode + modifier;  
10     else  
11         return responseCode;  
12 }
```

9.17 TpmFail.c

9.17.1 Includes, Defines, and Types

```

1  #define      TPM_FAIL_C
2  #include     "Tpm.h"
3  #include     <assert.h>

```

On MS C compiler, can save the alignment state and set the alignment to 1 for the duration of the TpmTypes.h include. This will avoid a lot of alignment warnings from the compiler for the unaligned structures. The alignment of the structures is not important as this function does not use any of the structures in TpmTypes.h and only include it for the #defines of the capabilities, properties, and command code values.

```

4  #include "TpmTypes.h"

```

9.17.2 Typedefs

These defines are used primarily for sizing of the local response buffer.

```

5  typedef struct
6  {
7      TPM_ST      tag;
8      UINT32      size;
9      TPM_RC      code;
10 } HEADER;
11
12 typedef struct
13 {
14     BYTE      tag[sizeof(TPM_ST)];
15     BYTE      size[sizeof(UINT32)];
16     BYTE      code[sizeof(TPM_RC)];
17 } PACKED_HEADER;
18
19 typedef struct
20 {
21     BYTE      size[sizeof(UINT16)];
22     struct
23     {
24         BYTE      function[sizeof(UINT32)];
25         BYTE      line[sizeof(UINT32)];
26         BYTE      code[sizeof(UINT32)];
27     } values;
28     BYTE      returnCode[sizeof(TPM_RC)];
29 } GET_TEST_RESULT_PARAMETERS;
30
31 typedef struct
32 {
33     BYTE      moreData[sizeof(TPMI_YES_NO)];
34     BYTE      capability[sizeof(TPM_CAP)]; // Always TPM_CAP_TPM_PROPERTIES
35     BYTE      tpmProperty[sizeof(TPML_TAGGED_TPM_PROPERTY)];
36 } GET_CAPABILITY_PARAMETERS;
37
38 typedef struct
39 {
40     BYTE      header[sizeof(PACKED_HEADER)];
41     BYTE      getTestResult[sizeof(GET_TEST_RESULT_PARAMETERS)];
42 } TEST_RESPONSE;
43
44 typedef struct
45 {

```

```

46     BYTE          header[sizeof(PACKED_HEADER)];
47     BYTE          getCap[sizeof(GET_CAPABILITY_PARAMETERS)];
48 } CAPABILITY_RESPONSE;
49
50 typedef union
51 {
52     BYTE          test[sizeof(TEST_RESPONSE)];
53     BYTE          cap[sizeof(CAPABILITY_RESPONSE)];
54 } RESPONSES;

```

Buffer to hold the responses. This may be a little larger than required due to padding that a compiler might add.

NOTE This is not in Global.c because of the specialized data definitions above. Since the data contained in this structure is not relevant outside of the execution of a single command (when the TPM is in failure mode. There is no compelling reason to move all the typedefs to Global.h and this structure to Global.c.

```

55 #ifndef __IGNORE_STATE__ // Don't define this value
56 static BYTE response[sizeof(RESPONSES)];
57 #endif

```

9.17.3 Local Functions

9.17.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```

58 static INT32
59 MarshalUint16(
60     UINT16          integer,
61     BYTE            **buffer
62 )
63 {
64     UINT16_TO_BYTE_ARRAY(integer, *buffer);
65     *buffer += 2;
66     return 2;
67 }

```

9.17.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```

68 static INT32
69 MarshalUint32(
70     UINT32          integer,
71     BYTE            **buffer
72 )
73 {
74     UINT32_TO_BYTE_ARRAY(integer, *buffer);
75     *buffer += 4;
76     return 4;
77 }

```

9.17.3.3 Unmarshal32()

```

78 static BOOL Unmarshal32(
79     UINT32          *target,
80     BYTE            **buffer,
81     INT32           *size
82 )

```



```

83  {
84      if((*size -= 4) < 0)
85          return FALSE;
86      *target = BYTE_ARRAY_TO_UINT32(*buffer);
87      *buffer += 4;
88      return TRUE;
89  }

```

9.17.3.4 Unmarshal16()

```

90  static BOOL Unmarshal16(
91      UINT16      *target,
92      BYTE        **buffer,
93      INT32       *size
94  )
95  {
96      if((*size -= 2) < 0)
97          return FALSE;
98      *target = BYTE_ARRAY_TO_UINT16(*buffer);
99      *buffer += 2;
100     return TRUE;
101 }

```

9.17.4 Public Functions

9.17.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```

102  #if SIMULATION
103  LIB_EXPORT void
104  SetForceFailureMode(
105      void
106  )
107  {
108      g_forceFailureMode = TRUE;
109      return;
110  }
111  #endif

```

9.17.4.2 TpmLogFailure()

This function saves the failure values when the code will continue to operate. It is similar to TpmFail() but returns to the caller. The assumption is that the caller will propagate a failure back up the stack.

```

112  void
113  TpmLogFailure(
114      #if FAIL_TRACE
115          const char    *function,
116          int           line,
117      #endif
118          int           code
119  )
120  {
121      // Save the values that indicate where the error occurred.
122      // On a 64-bit machine, this may truncate the address of the string
123      // of the function name where the error occurred.
124      #if FAIL_TRACE
125          s_failFunction = (UINT32)(ptrdiff_t)function;
126          s_failLine = line;
127      #else

```

```

128     s_failFunction = 0;
129     s_failLine = 0;
130 #endif
131     s_failCode = code;
132
133     // We are in failure mode
134     g_inFailureMode = TRUE;
135
136     return;
137 }

```

9.17.4.3 TpmFail()

This function is called by TPM.lib when a failure occurs. It will set up the failure values to be returned on TPM2_GetTestResult().

```

138 NORETURN void
139 TpmFail(
140 #if FAIL_TRACE
141     const char    *function,
142     int           line,
143 #endif
144     int           code
145 )
146 {
147     // Save the values that indicate where the error occurred.
148     // On a 64-bit machine, this may truncate the address of the string
149     // of the function name where the error occurred.
150 #if FAIL_TRACE
151     s_failFunction = (UINT32)(ptrdiff_t)function;
152     s_failLine = line;
153 #else
154     s_failFunction = (UINT32)(ptrdiff_t)NULL;
155     s_failLine = 0;
156 #endif
157     s_failCode = code;
158
159     // We are in failure mode
160     g_inFailureMode = TRUE;
161
162     // if asserts are enabled, then do an assert unless the failure mode code
163     // is being tested.
164 #if SIMULATION
165 #   ifndef NDEBUG
166     assert(g_forceFailureMode);
167 #   endif
168     // Clear this flag
169     g_forceFailureMode = FALSE;
170 #endif
171     // Jump to the failure mode code.
172     // Note: only get here if asserts are off or if we are testing failure mode
173     _plat__Fail();
174 }

```

9.17.4.4 TpmFailureMode()

This function is called by the interface code when the platform is in failure mode.

```

175 void
176 TpmFailureMode(
177     unsigned int    inRequestSize,    // IN: command buffer size
178     unsigned char   *inRequest,      // IN: command buffer
179     unsigned int    *outResponseSize, // OUT: response buffer size

```

```

180     unsigned char    **outResponse    // OUT: response buffer
181 )
182 {
183     UINT32            marshalSize;
184     UINT32            capability;
185     HEADER            header;        // unmarshaled command header
186     UINT32            pt;            // unmarshaled property type
187     UINT32            count;        // unmarshaled property count
188     UINT8             *buffer = inRequest;
189     INT32             size = inRequestSize;
190
191     // If there is no command buffer, then just return TPM_RC_FAILURE
192     if(inRequestSize == 0 || inRequest == NULL)
193         goto FailureModeReturn;
194     // If the header is not correct for TPM2_GetCapability() or
195     // TPM2_GetTestResult() then just return the in failure mode response;
196     if(! (Unmarshal16(&header.tag, &buffer, &size)
197         && Unmarshal32(&header.size, &buffer, &size)
198         && Unmarshal32(&header.code, &buffer, &size)))
199         goto FailureModeReturn;
200     if(header.tag != TPM_ST_NO_SESSIONS
201         || header.size < 10)
202         goto FailureModeReturn;
203     switch(header.code)
204     {
205     case TPM_CC_GetTestResult:
206         // make sure that the command size is correct
207         if(header.size != 10)
208             goto FailureModeReturn;
209         buffer = &response[10];
210         marshalSize = MarshalUint16(3 * sizeof(UINT32), &buffer);
211         marshalSize += MarshalUint32(s_failFunction, &buffer);
212         marshalSize += MarshalUint32(s_failLine, &buffer);
213         marshalSize += MarshalUint32(s_failCode, &buffer);
214         if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
215             marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
216         else
217             marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
218         break;
219     case TPM_CC_GetCapability:
220         // make sure that the size of the command is exactly the size
221         // returned for the capability, property, and count
222         if(header.size != (10 + (3 * sizeof(UINT32))))
223             // also verify that this is requesting TPM properties
224             || !Unmarshal32(&capability, &buffer, &size)
225             || capability != TPM_CAP_TPM_PROPERTIES
226             || !Unmarshal32(&pt, &buffer, &size)
227             || !Unmarshal32(&count, &buffer, &size))
228             goto FailureModeReturn;
229         // If in failure mode because of an unrecoverable read error, and the
230         // property is 0 and the count is 0, then this is an indication to
231         // re-manufacture the TPM. Do the re-manufacture but stay in failure
232         // mode until the TPM is reset.
233         // Note: this behavior is not required by the specification and it is
234         // OK to leave the TPM permanently bricked due to an unrecoverable NV
235         // error.
236         if(count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
237         {
238             g_manufactured = FALSE;
239             TPM_Manufacture(0);
240         }
241         if(count > 0)
242             count = 1;
243         else if(pt > TPM_PT_FIRMWARE_VERSION_2)
244             count = 0;
245         if(pt < TPM_PT_MANUFACTURER)

```

```

246         pt = TPM_PT_MANUFACTURER;
247         // set up for return
248         buffer = &response[10];
249         // if the request was for a PT less than the last one
250         // then we indicate more, otherwise, not.
251         if(pt < TPM_PT_FIRMWARE_VERSION_2)
252             *buffer++ = YES;
253         else
254             *buffer++ = NO;
255         marshalSize = 1;
256
257         // indicate the capability type
258         marshalSize += MarshalUint32(capability, &buffer);
259         // indicate the number of values that are being returned (0 or 1)
260         marshalSize += MarshalUint32(count, &buffer);
261         // indicate the property
262         marshalSize += MarshalUint32(pt, &buffer);
263
264         if(count > 0)
265             switch(pt)
266             {
267                 case TPM_PT_MANUFACTURER:
268                     // the vendor ID unique to each TPM manufacturer
269 #ifndef MANUFACTURER
270                     pt = *(UINT32*)MANUFACTURER;
271 #else
272                     pt = 0;
273 #endif
274                     break;
275                 case TPM_PT_VENDOR_STRING_1:
276                     // the first four characters of the vendor ID string
277 #ifndef VENDOR_STRING_1
278                     pt = *(UINT32*)VENDOR_STRING_1;
279 #else
280                     pt = 0;
281 #endif
282                     break;
283                 case TPM_PT_VENDOR_STRING_2:
284                     // the second four characters of the vendor ID string
285 #ifndef VENDOR_STRING_2
286                     pt = *(UINT32*)VENDOR_STRING_2;
287 #else
288                     pt = 0;
289 #endif
290                     break;
291                 case TPM_PT_VENDOR_STRING_3:
292                     // the third four characters of the vendor ID string
293 #ifndef VENDOR_STRING_3
294                     pt = *(UINT32*)VENDOR_STRING_3;
295 #else
296                     pt = 0;
297 #endif
298                     break;
299                 case TPM_PT_VENDOR_STRING_4:
300                     // the fourth four characters of the vendor ID string
301 #ifndef VENDOR_STRING_4
302                     pt = *(UINT32*)VENDOR_STRING_4;
303 #else
304                     pt = 0;
305 #endif
306                     break;
307                 case TPM_PT_VENDOR_TPM_TYPE:
308                     // vendor-defined value indicating the TPM model
309                     // We just make up a number here
310                     pt = 1;
311                     break;

```

```

312         case TPM_PT_FIRMWARE_VERSION_1:
313             // the more significant 32-bits of a vendor-specific value
314             // indicating the version of the firmware
315 #ifdef FIRMWARE_V1
316             pt = FIRMWARE_V1;
317 #else
318             pt = 0;
319 #endif
320             break;
321         default: // TPM_PT_FIRMWARE_VERSION_2:
322             // the less significant 32-bits of a vendor-specific value
323             // indicating the version of the firmware
324 #ifdef FIRMWARE_V2
325             pt = FIRMWARE_V2;
326 #else
327             pt = 0;
328 #endif
329             break;
330     }
331     marshalSize += MarshalUint32(pt, &buffer);
332     break;
333     default: // default for switch (cc)
334         goto FailureModeReturn;
335 }
336 // Now do the header
337 buffer = response;
338 marshalSize = marshalSize + 10; // Add the header size to the
339                                // stuff already marshaled
340 MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
341 MarshalUint32(marshalSize, &buffer); // responseSize
342 MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code
343
344 *outResponseSize = marshalSize;
345 *outResponse = (unsigned char *)&response;
346 return;
347 FailureModeReturn:
348 buffer = response;
349 marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
350 marshalSize += MarshalUint32(10, &buffer);
351 marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
352 *outResponseSize = marshalSize;
353 *outResponse = (unsigned char *)&response;
354 return;
355 }

```

9.17.4.5 UnmarshalFail()

This is a stub that is used to catch an attempt to unmarshal an entry that is not defined. Don't ever expect this to be called but...

```

356 void
357 UnmarshalFail(
358     void            *type,
359     BYTE            **buffer,
360     INT32           *size
361 )
362 {
363     NOT_REFERENCED(type);
364     NOT_REFERENCED(buffer);
365     NOT_REFERENCED(size);
366     FAIL(FATAL_ERROR_INTERNAL);
367 }

```

10 Cryptographic Functions

10.1 Headers

10.1.1 BnValues.h

10.1.1.1 Introduction

This file contains the definitions needed for defining the internal BIGNUM structure.

A BIGNUM is a pointer to a structure. The structure has three fields. The last field is an array (*d*) of `crypt_uword_t`. Each word is in machine format (big- or little-endian) with the words in ascending significance (i.e. words in little-endian order). This is the order that seems to be used in every big number library in the worlds, so...

The first field in the structure (*allocated*) is the number of words in *d*. This is the upper limit on the size of the number that can be held in the structure. This differs from libraries like OpenSSL as this is not intended to deal with numbers of arbitrary size; just numbers that are needed to deal with the algorithms that are defined in the TPM implementation.

The second field in the structure (*size*) is the number of significant words in *n*. When this number is zero, the number is zero. The word at *used-1* should never be zero. All words between *d[size]* and *d[allocated-1]* should be zero.

10.1.1.2 Defines

```

1  #ifndef BN_NUMBERS_H
2  #define BN_NUMBERS_H
3
4  #if RADIX_BITS == 64
5  # define RADIX_LOG2      6
6  #elif RADIX_BITS == 32
7  #define RADIX_LOG2      5
8  #else
9  # error "Unsupported radix"
10 #endif
11
12 #define RADIX_MOD(x)      ((x) & ((1 << RADIX_LOG2) - 1))
13 #define RADIX_DIV(x)      ((x) >> RADIX_LOG2)
14 #define RADIX_MASK      (((crypt_uword_t)1) << RADIX_LOG2) - 1)
15
16 #define BITS_TO_CRYPT_WORDS(bits)      RADIX_DIV((bits) + (RADIX_BITS - 1))
17 #define BYTES_TO_CRYPT_WORDS(bytes)     BITS_TO_CRYPT_WORDS(bytes * 8)
18 #define SIZE_IN_CRYPT_WORDS(thing)      BYTES_TO_CRYPT_WORDS(sizeof(thing))
19
20 #if RADIX_BITS == 64
21 #define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_64(x)
22     typedef uint64_t      crypt_uword_t;
23     typedef int64_t       crypt_word_t;
24 # define TO_CRYPT_WORD_64      BIG_ENDIAN_BYTES_TO_UINT64
25 # define TO_CRYPT_WORD_32(a, b, c, d) TO_CRYPT_WORD_64(0, 0, 0, 0, a, b, c, d)
26 #elif RADIX_BITS == 32
27 # define SWAP_CRYPT_WORD(x)  REVERSE_ENDIAN_32((x))
28     typedef uint32_t      crypt_uword_t;
29     typedef int32_t       crypt_word_t;
30 # define TO_CRYPT_WORD_64(a, b, c, d, e, f, g, h) \
31     BIG_ENDIAN_BYTES_TO_UINT32(e, f, g, h), \
32     BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d)
33 #endif
34
```

```

35 #define MAX_CRYPT_UWORD (~((crypt_uword_t)0))
36 #define MAX_CRYPT_WORD ((crypt_word_t)(MAX_CRYPT_UWORD >> 1))
37 #define MIN_CRYPT_WORD (~MAX_CRYPT_WORD)
38
39 #define LARGEST_NUMBER (MAX((ALG_RSA * MAX_RSA_KEY_BYTES), \
40 MAX((ALG_ECC * MAX_ECC_KEY_BYTES), MAX_DIGEST_SIZE)))
41 #define LARGEST_NUMBER_BITS (LARGEST_NUMBER * 8)
42
43 #define MAX_ECC_PARAMETER_BYTES (MAX_ECC_KEY_BYTES * ALG_ECC)

```

These are the basic big number formats. This is convertible to the library- specific format without too much difficulty. For the math performed using these numbers, the value is always positive.

```

44 #define BN_STRUCT_DEF(count) struct { \
45     crypt_uword_t    allocated; \
46     crypt_uword_t    size; \
47     crypt_uword_t    d[count]; \
48 }
49 typedef BN_STRUCT_DEF(1) bignum_t;
50 #ifndef bigNum
51 typedef bignum_t      *bigNum;
52 typedef const bignum_t *bigConst;
53 #endif
54
55 extern const bignum_t  BnConstZero;

```

The Functions to access the properties of a big number. Get number of allocated words

```

56 #define BnGetAllocated(x)    (unsigned)((x)->allocated)

```

Get number of words used

```

57 #define BnGetSize(x)        ((x)->size)

```

Get a pointer to the data array

```

58 #define BnGetArray(x)       ((crypt_uword_t *)&((x)->d[0]))

```

Get the nth word of a BIGNUM (zero-based)

```

59 #define BnGetWord(x, i)     (crypt_uword_t)((x)->d[i])

```

Some things that are done often.

Test to see if a bignum_t is equal to zero

```

60 #define BnEqualZero(bn)     (BnGetSize(bn) == 0)

```

Test to see if a bignum_t is equal to a word type

```

61 #define BnEqualWord(bn, word) \
62     ((BnGetSize(bn) == 1) && (BnGetWord(bn, 0) == (crypt_uword_t)word))

```

Determine if a BIGNUM is even. A zero is even. Although the indication that a number is zero is that its size is zero, all words of the number are 0 so this test works on zero.

```

63 #define BnIsEven(n)         ((BnGetWord(n, 0) & 1) == 0)

```

The macros below are used to define BIGNUM values of the required size. The values are allocated on the stack so they can be treated like simple local values.

This will call the initialization function for a defined `bignum_t`. This sets the allocated and used fields and clears the words of *n*.

```

64 #define BN_INIT(name)                                     \
65     (bigNum)BnInit((bigNum) &(name),                      \
66         BYTES_TO_CRYPT_WORDS(sizeof(name.d)))

```

In some cases, a function will need the address of the structure associated with a variable. The structure for a BIGNUM variable of *name* is *name_*. Generally, when the structure is created, it is initialized and a parameter is created with a pointer to the structure. The pointer has the *name* and the structure it points to is *name_*.

```

67 #define BN_ADDRESS(name) (bigNum) &name##_
68
69 #define BN_CONST(name, words, initializer)               \
70 typedef const struct name##_type {                       \
71     crypt_ushort_t    allocated;                          \
72     crypt_ushort_t    size;                              \
73     crypt_ushort_t    d[words < 1 ? 1 : words];          \
74 } name##_type;                                           \
75 name##_type name = {(words < 1 ? 1 : words), words, {initializer}};
76
77 #define BN_STRUCT_ALLOCATION(bits) (BITS_TO_CRYPT_WORDS(bits) + 1)

```

Create a structure of the correct size.

```

78 #define BN_STRUCT(bits)                                  \
79     BN_STRUCT_DEF(BN_STRUCT_ALLOCATION(bits))

```

Define a BIGNUM type with a specific allocation

```

80 #define BN_TYPE(name, bits)                              \
81     typedef BN_STRUCT(bits) bn_##name##_t

```

This creates a local BIGNUM variable of a specific size and initializes it from a TPM2B input parameter.

```

82 #define BN_INITIALIZED(name, bits, initializer)          \
83     BN_STRUCT(bits) name##_;                             \
84     bigNum name = BnFrom2B(BN_INIT(name##_),             \
85         (const TPM2B *)initializer)

```

Create a local variable that can hold a number with *bits*

```

86 #define BN_VAR(name, bits)                               \
87     BN_STRUCT(bits) _##name;                             \
88     bigNum name = BN_INIT(_##name)

```

Create a type that can hold the largest number defined by the implementation.

```

89 #define BN_MAX(name) BN_VAR(name, LARGEST_NUMBER_BITS)
90 #define BN_MAX_INITIALIZED(name, initializer)            \
91     BN_INITIALIZED(name, LARGEST_NUMBER_BITS, initializer)

```

A word size value is useful

```

92 #define BN_WORD(name) BN_VAR(name, RADIX_BITS)

```

This is used to create a word-size BIGNUM and initialize it with an input parameter to a function.

```

93 #define BN_WORD_INITIALIZED(name, initial)               \
94     BN_STRUCT(RADIX_BITS) name##_;

```

```

95     bigNum          name = BnInitializeWord((bigNum) &name##_ ,
96                                     BN_STRUCT_ALLOCATION(RADIX_BITS), initial) \

```

ECC-Specific Values

This is the format for a point. It is always in affine format. The Z value is carried as part of the point, primarily to simplify the interface to the support library. Rather than have the interface layer have to create space for the point each time it is used... The x, y, and z values are pointers to *bigNum* values and not in-line versions of the numbers. This is a relic of the days when there was no standard TPM format for the numbers

```

97 typedef struct _bn_point_t
98 {
99     bigNum          x;
100    bigNum          y;
101    bigNum          z;
102 } bn_point_t;
103
104 typedef bn_point_t      *bigPoint;
105 typedef const bn_point_t *pointConst;
106
107 typedef struct constant_point_t
108 {
109     bigConst        x;
110     bigConst        y;
111     bigConst        z;
112 } constant_point_t;
113
114 #define ECC_BITS      (MAX_ECC_KEY_BYTES * 8)
115 BN_TYPE(ecc, ECC_BITS);
116 #define ECC_NUM(name)      BN_VAR(name, ECC_BITS)
117 #define ECC_INITIALIZED(name, initializer) \
118     BN_INITIALIZED(name, ECC_BITS, initializer)
119
120 #define POINT_INSTANCE(name, bits) \
121     BN_STRUCT (bits)      name##_x = \
122         {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}}; \
123     BN_STRUCT ( bits )    name##_y = \
124         {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}}; \
125     BN_STRUCT ( bits )    name##_z = \
126         {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}}; \
127     bn_point_t name##_
128
129 #define POINT_INITIALIZER(name) \
130     BnInitializePoint(&name##_ , (bigNum) &name##_x, \
131         (bigNum) &name##_y, (bigNum) &name##_z)
132
133 #define POINT_INITIALIZED(name, initValue) \
134     POINT_INSTANCE(name, MAX_ECC_KEY_BITS); \
135     bigPoint      name = BnPointFrom2B( \
136         POINT_INITIALIZER(name), \
137         initValue)
138
139 #define POINT_VAR(name, bits) \
140     POINT_INSTANCE (name, bits); \
141     bigPoint      name = POINT_INITIALIZER(name)
142
143 #define POINT(name)      POINT_VAR(name, MAX_ECC_KEY_BITS)

```

Structure for the curve parameters. This is an analog to the TPMS_ALGORITHM_DETAIL_ECC

```

144 typedef struct
145 {
146     bigConst        prime;      // a prime number

```

```

147     bigConst      order;      // the order of the curve
148     bigConst      h;          // cofactor
149     bigConst      a;          // linear coefficient
150     bigConst      b;          // constant term
151     constant_point_t base;     // base point
152 } ECC_CURVE_DATA;

```

Access macros for the ECC_CURVE structure. The parameter C is a pointer to an ECC_CURVE_DATA structure. In some libraries, the curve structure contains a pointer to an ECC_CURVE_DATA structure as well as some other bits. For those cases, the AccessCurveData() macro is used in the code to first get the pointer to the ECC_CURVE_DATA for access. In some cases, the macro does nothing.

```

153 #define CurveGetPrime(C)      ((C)->prime)
154 #define CurveGetOrder(C)     ((C)->order)
155 #define CurveGetCofactor(C) ((C)->h)
156 #define CurveGet_a(C)        ((C)->a)
157 #define CurveGet_b(C)        ((C)->b)
158 #define CurveGetG(C)         ((pointConst) & ((C)->base))
159 #define CurveGetGx(C)        ((C)->base.x)
160 #define CurveGetGy(C)        ((C)->base.y)

```

Convert bytes in initializers This is used for CryptEccData.c.

```

161 #define BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d) \
162 ( \
163 + ((UINT32)(a) << 24) \
164 + ((UINT32)(b) << 16) \
165 + ((UINT32)(c) << 8) \
166 + ((UINT32)(d)) \
167 )
168 #define BIG_ENDIAN_BYTES_TO_UINT64(a, b, c, d, e, f, g, h) \
169 ( \
170 + ((UINT64)(a) << 56) \
171 + ((UINT64)(b) << 48) \
172 + ((UINT64)(c) << 40) \
173 + ((UINT64)(d) << 32) \
174 + ((UINT64)(e) << 24) \
175 + ((UINT64)(f) << 16) \
176 + ((UINT64)(g) << 8) \
177 + ((UINT64)(h)) \
178 )
179 #ifndef RADIX_BYTES
180 #   if RADIX_BITS == 32
181 #       define RADIX_BYTES 4
182 #   elif RADIX_BITS == 64
183 #       define RADIX_BYTES 8
184 #   else
185 #       error "RADIX_BITS must either be 32 or 64"
186 #   endif
187 #endif

```

These macros are used for data initialization of big number ECC constants. These two macros combine a macro for data definition with a macro for structure initialization. The *a* parameter is a macro that gives numbers to each of the bytes of the initializer and defines where each of the numbered bytes will show up in the final structure. The *b* value is a structure that contains the requisite number of bytes in big endian order. So, the MJOIN and JOIN macros will combine a macro defining a data layout with a macro defining the data to be placed. Generally, these macros will only need expansion when CryptEccData.c gets compiled.

```

188 #define JOINED(a,b) a b
189 #define MJOIN(a,b) a b
190
191 #define B4_TO_BN(a, b, c, d) (((((a << 8) + b) << 8) + c) + d)

```

```

192 #if RADIX_BYTES == 64
193 #define B8_TO_BN(a, b, c, d, e, f, g, h) \
194     (UINT64) (((((((((a) << 8) | b) << 8) | c) << 8) | d) << 8) \
195             e) << 8) | f) << 8) | g) << 8) | h)
196 #define B1_TO_BN(a) B8_TO_BN(0, 0, 0, 0, 0, 0, 0, a)
197 #define B2_TO_BN(a, b) B8_TO_BN(0, 0, 0, 0, 0, 0, a, b)
198 #define B3_TO_BN(a, b, c) B8_TO_BN(0, 0, 0, 0, 0, a, b, c)
199 #define B4_TO_BN(a, b, c, d) B8_TO_BN(0, 0, 0, 0, a, b, c, d)
200 #define B5_TO_BN(a, b, c, d, e) B8_TO_BN(0, 0, 0, a, b, c, d, e)
201 #define B6_TO_BN(a, b, c, d, e, f) B8_TO_BN(0, 0, a, b, c, d, e, f)
202 #define B7_TO_BN(a, b, c, d, e, f, g) B8_TO_BN(0, a, b, c, d, e, f, g)
203 #else
204 #define B1_TO_BN(a) B4_TO_BN(0, 0, 0, a)
205 #define B2_TO_BN(a, b) B4_TO_BN(0, 0, a, b)
206 #define B3_TO_BN(a, b, c) B4_TO_BN(0, a, b, c)
207 #define B4_TO_BN(a, b, c, d) (((a << 8) + b) << 8) + c) + d)
208 #define B5_TO_BN(a, b, c, d, e) B4_TO_BN(b, c, d, e), B1_TO_BN(a)
209 #define B6_TO_BN(a, b, c, d, e, f) B4_TO_BN(c, d, e, f), B2_TO_BN(a, b)
210 #define B7_TO_BN(a, b, c, d, e, f, g) B4_TO_BN(d, e, f, g), B3_TO_BN(a, b, c)
211 #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)
212
213 #endif

```

Add implementation dependent definitions for other ECC Values and for linkages.

```

214 #include LIB_INCLUDE(MATH_LIB, Math)
215
216 #endif // _BN_NUMBERS_H

```

10.1.2 CryptEcc.h

10.1.2.1 Introduction

This file contains structure definitions used for ECC. The structures in this file are only used internally. The ECC-related structures that cross the TPM interface are defined in TpmTypes.h

```
1  #ifndef _CRYPT_ECC_H
2  #define _CRYPT_ECC_H
```

10.1.2.2 Structures

```
3  typedef struct ECC_CURVE
4  {
5      const TPM_ECC_CURVE      curveId;
6      const UINT16             keySizeBits;
7      const TPMT_KDF_SCHEME    kdf;
8      const TPMT_ECC_SCHEME    sign;
9      const ECC_CURVE_DATA     *curveData; // the address of the curve data
10     const BYTE                *OID;
11 } ECC_CURVE;
```

10.1.2.2.1 Macros

This macro is used to instance an ECC_CURVE_DATA structure for the curve. This structure is referenced by the ECC_CURVE structure

```
12 #define CURVE_DATA_DEF(CURVE) \
13 const ECC_CURVE_DATA CURVE = { \
14     (bigNum)&CURVE##_p_DATA, (bigNum)&CURVE##_n_DATA, (bigNum)&CURVE##_h_DATA, \
15     (bigNum)&CURVE##_a_DATA, (bigNum)&CURVE##_b_DATA, \
16     { (bigNum)&CURVE##_gX_DATA, (bigNum)&CURVE##_gY_DATA, (bigNum)&BN_ONE } }; \
17 \
18 extern const ECC_CURVE eccCurves[ECC_CURVE_COUNT]; \
19 \
20 #define CURVE_DEF(CURVE) \
21 { \
22     TPM_ECC_##CURVE, \
23     CURVE##_KEY_SIZE, \
24     CURVE##_KDF, \
25     CURVE##_SIGN, \
26     &##CURVE, \
27     OID_ECC_##CURVE \
28 } \
29 #define CURVE_NAME(N) \
30 \
31 #endif
```

10.1.3 CryptHash.h

10.1.3.1 Introduction

This header contains the hash structure definitions used in the TPM code to define the amount of space to be reserved for the hash state. This allows the TPM code to not have to import all of the symbols used by the hash computations. This lets the build environment of the TPM code not to have include the header files associated with the CryptoEngine() code.

```
1  #ifndef _CRYPT_HASH_H
2  #define _CRYPT_HASH_H
```

10.1.3.2 Hash-related Structures

```
3  union SMAC_STATES;
```

These definitions add the high-level methods for processing state that may be an SMAC

```
4  typedef void(* SMAC_DATA_METHOD)(
5      union SMAC_STATES *state,
6      UINT32 size,
7      const BYTE *buffer
8  );
9
10 typedef UINT16(* SMAC_END_METHOD)(
11     union SMAC_STATES *state,
12     UINT32 size,
13     BYTE *buffer
14 );
15
16 typedef struct sequenceMethods {
17     SMAC_DATA_METHOD data;
18     SMAC_END_METHOD end;
19 } SMAC_METHODS;
20
21 #define SMAC_IMPLEMENTED (CC_MAC || CC_MAC_Start)
```

These definitions are here because the SMAC state is in the union of hash states.

```
22 typedef struct tpmCmacState {
23     TPM_ALG_ID symAlg;
24     UINT16 keySizeBits;
25     INT16 bcount; // current count of bytes accumulated in IV
26     TPM2B_IV iv; // IV buffer
27     TPM2B_SYM_KEY symKey;
28 } tpmCmacState_t;
29
30 typedef union SMAC_STATES {
31     #if ALG_CMAL
32     tpmCmacState_t cmac;
33     #endif
34     UINT64 pad;
35 } SMAC_STATES;
36
37 typedef struct SMAC_STATE {
38     SMAC_METHODS smacMethods;
39     SMAC_STATES state;
40 } SMAC_STATE;
41
42 #if ALG_SHA1
43 # define IF_IMPLEMENTED_SHA1(op) op(SHA1, Sha1)
```

```

44 #else
45 #   define IF_IMPLEMENTED_SHA1(op)
46 #endif
47 #if ALG_SHA256
48 #   define IF_IMPLEMENTED_SHA256(op)    op(SHA256, Sha256)
49 #else
50 #   define IF_IMPLEMENTED_SHA256(op)
51 #endif
52 #if ALG_SHA384
53 #   define IF_IMPLEMENTED_SHA384(op)    op(SHA384, Sha384)
54 #else
55 #   define IF_IMPLEMENTED_SHA384(op)
56 #endif
57 #if ALG_SHA512
58 #   define IF_IMPLEMENTED_SHA512(op)    op(SHA512, Sha512)
59 #else
60 #   define IF_IMPLEMENTED_SHA512(op)
61 #endif
62 #if ALG_SM3_256
63 #   define IF_IMPLEMENTED_SM3_256(op)    op(SM3_256, Sm3_256)
64 #else
65 #   define IF_IMPLEMENTED_SM3_256(op)
66 #endif
67 #if ALG_SHA3_256
68 #   define IF_IMPLEMENTED_SHA3_256(op)    op(SHA3_256, Sha3_256)
69 #else
70 #   define IF_IMPLEMENTED_SHA3_256(op)
71 #endif
72 #if ALG_SHA3_384
73 #   define IF_IMPLEMENTED_SHA3_384(op)    op(SHA3_384, Sha3_384)
74 #else
75 #   define IF_IMPLEMENTED_SHA3_384(op)
76 #endif
77 #if ALG_SHA3_512
78 #   define IF_IMPLEMENTED_SHA3_512(op)    op(SHA3_512, Sha3_512)
79 #else
80 #   define IF_IMPLEMENTED_SHA3_512(op)
81 #endif
82
83 #define FOR_EACH_HASH(op) \
84     IF_IMPLEMENTED_SHA1(op) \
85     IF_IMPLEMENTED_SHA256(op) \
86     IF_IMPLEMENTED_SHA384(op) \
87     IF_IMPLEMENTED_SM3_256(op) \
88     IF_IMPLEMENTED_SHA3_256(op) \
89     IF_IMPLEMENTED_SHA3_384(op) \
90     IF_IMPLEMENTED_SHA3_512(op)
91
92 #define HASH_TYPE(HASH, Hash)    tpmHashState##HASH##_t    Hash;
93 typedef union
94 {
95     FOR_EACH_HASH(HASH_TYPE)
96     // Additions for symmetric block cipher MAC
97     #if SMAC_IMPLEMENTED
98         SMAC_STATE                smac;
99     #endif
100     // to force structure alignment to be no worse than HASH_ALIGNMENT
101     #if HASH_ALIGNMENT == 8
102         uint64_t                align;
103     #else
104         uint32_t                align;
105     #endif
106 } ANY_HASH_STATE;
107
108 typedef ANY_HASH_STATE *PANY_HASH_STATE;
109 typedef const ANY_HASH_STATE *PCANY_HASH_STATE;

```



```

110
111 #define ALIGNED_SIZE(x, b) (((x) + (b) - 1) / (b)) * (b))

```

MAX_HASH_STATE_SIZE will change with each implementation. It is assumed that a hash state will not be larger than twice the block size plus some overhead (in this case, 16 bytes). The overall size needs to be as large as any of the hash contexts. The structure needs to start on an alignment boundary and be an even multiple of the alignment

```

112 #define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
113 #define MAX_HASH_STATE_SIZE_ALIGNED
114     ALIGNED_SIZE(MAX_HASH_STATE_SIZE, HASH_ALIGNMENT)

```

This is an aligned byte array that will hold any of the hash contexts.

```

115 typedef ANY_HASH_STATE ALIGNED_HASH_STATE;

```

The header associated with the hash library is expected to define the methods which include the calling sequence. When not compiling CryptHash.c, the methods are not defined so we need placeholder functions for the structures

```

116 #ifndef HASH_START_METHOD_DEF
117 #   define HASH_START_METHOD_DEF    void (HASH_START_METHOD) (void)
118 #endif
119 #ifndef HASH_DATA_METHOD_DEF
120 #   define HASH_DATA_METHOD_DEF     void (HASH_DATA_METHOD) (void)
121 #endif
122 #ifndef HASH_END_METHOD_DEF
123 #   define HASH_END_METHOD_DEF      void (HASH_END_METHOD) (void)
124 #endif
125 #ifndef HASH_STATE_COPY_METHOD_DEF
126 #   define HASH_STATE_COPY_METHOD_DEF void (HASH_STATE_COPY_METHOD) (void)
127 #endif
128 #ifndef HASH_STATE_EXPORT_METHOD_DEF
129 #   define HASH_STATE_EXPORT_METHOD_DEF void (HASH_STATE_EXPORT_METHOD) (void)
130 #endif
131 #ifndef HASH_STATE_IMPORT_METHOD_DEF
132 #   define HASH_STATE_IMPORT_METHOD_DEF void (HASH_STATE_IMPORT_METHOD) (void)
133 #endif

```

Define the prototypical function call for each of the methods. This defines the order in which the parameters are passed to the underlying function.

```

134 typedef HASH_START_METHOD_DEF;
135 typedef HASH_DATA_METHOD_DEF;
136 typedef HASH_END_METHOD_DEF;
137 typedef HASH_STATE_COPY_METHOD_DEF;
138 typedef HASH_STATE_EXPORT_METHOD_DEF;
139 typedef HASH_STATE_IMPORT_METHOD_DEF;
140
141 typedef struct _HASH_METHODS
142 {
143     HASH_START_METHOD    *start;
144     HASH_DATA_METHOD     *data;
145     HASH_END_METHOD      *end;
146     HASH_STATE_COPY_METHOD *copy;    // Copy a hash block
147     HASH_STATE_EXPORT_METHOD *copyOut; // Copy a hash block from a hash
148                                         // context
149     HASH_STATE_IMPORT_METHOD *copyIn; // Copy a hash block to a proper hash
150                                         // context
151 } HASH_METHODS, *PHASH_METHODS;
152
153 #define HASH_TPM2B(HASH, Hash)    TPM2B_TYPE(HASH##_DIGEST, HASH##_DIGEST_SIZE);
154

```

```
155 FOR_EACH_HASH(HASH_TPM2B)
```

When the TPM implements RSA, the hash-dependent OID pointers are part of the HASH_DEF. These macros conditionally add the OID reference to the HASH_DEF and the HASH_DEF_TEMPLATE.

```
156 #if ALG_RSA
157 #define PKCS1_HASH_REF    const BYTE *PKCS1;
158 #define PKCS1_OID(NAME)  , OID_PKCS1_ ##NAME
159 #else
160 #define PKCS1_HASH_REF
161 #define PKCS1_OID(NAME)
162 #endif
```

When the TPM implements ECC, the hash-dependent OID pointers are part of the HASH_DEF. These macros conditionally add the OID reference to the HASH_DEF and the HASH_DEF_TEMPLATE.

```
163 #if ALG_ECDSA
164 #define ECDSA_HASH_REF    const BYTE *ECDSA;
165 #define ECDSA_OID(NAME)  , OID_ECDSA_ ##NAME
166 #else
167 #define ECDSA_HASH_REF
168 #define ECDSA_OID(NAME)
169 #endif
170
171 typedef const struct HASH_DEF
172 {
173     HASH_METHODS        method;
174     uint16_t             blockSize;
175     uint16_t             digestSize;
176     uint16_t             contextSize;
177     uint16_t             hashAlg;
178     const BYTE           *OID;
179     PKCS1_HASH_REF       // PKCS1 OID
180     ECDSA_HASH_REF       // ECDSA OID
181 } HASH_DEF, *PHASH_DEF;
```

Macro to fill in the HASH_DEF for an algorithm. For SHA1, the instance would be: HASH_DEF_TEMPLATE(Sha1, SHA1) This handles the difference in capitalization for the various pieces.

```
182 #define HASH_DEF_TEMPLATE(HASH, Hash)
183     HASH_DEF    Hash##_Def= {
184         { (HASH_START_METHOD *) &tpmHashStart_ ##HASH,
185         (HASH_DATA_METHOD *) &tpmHashData_ ##HASH,
186         (HASH_END_METHOD *) &tpmHashEnd_ ##HASH,
187         (HASH_STATE_COPY_METHOD *) &tpmHashStateCopy_ ##HASH,
188         (HASH_STATE_EXPORT_METHOD *) &tpmHashStateExport_ ##HASH,
189         (HASH_STATE_IMPORT_METHOD *) &tpmHashStateImport_ ##HASH,
190         },
191         HASH##_BLOCK_SIZE,        /*block size */
192         HASH##_DIGEST_SIZE,       /*data size */
193         sizeof(tpmHashState ##HASH##_t),
194         TPM_ALG_ ##HASH, OID_ ##HASH
195         PKCS1_OID(HASH) ECDSA_OID(HASH) };
\
\
\
\
\
\
\
\
\
```

These definitions are for the types that can be in a hash state structure. These types are used in the cryptographic utilities. This is a define rather than an enum so that the size of this field can be explicit.

```
196 typedef BYTE    HASH_STATE_TYPE;
197 #define HASH_STATE_EMPTY    ((HASH_STATE_TYPE) 0)
198 #define HASH_STATE_HASH     ((HASH_STATE_TYPE) 1)
199 #define HASH_STATE_HMAC     ((HASH_STATE_TYPE) 2)
200 #if CC_MAC || CC_MAC_Start
201 #define HASH_STATE_SMAC     ((HASH_STATE_TYPE) 3)
```

202 **#endif**

This is the structure that is used for passing a context into the hashing functions. It should be the same size as the function context used within the hashing functions. This is checked when the hash function is initialized. This version uses a new layout for the contexts and a different definition. The state buffer is an array of HASH_UNIT values so that a decent compiler will put the structure on a HASH_UNIT boundary. If the structure is not properly aligned, the code that manipulates the structure will copy to a properly aligned structure before it is used and copy the result back. This just makes things slower.

NOTE This version of the state had the pointer to the update method in the state. This is to allow the SMAC functions to use the same structure without having to replicate the entire HASH_DEF structure.

```
203  typedef struct _HASH_STATE
204  {
205      HASH_STATE_TYPE      type;                // type of the context
206      TPM_ALG_ID           hashAlg;
207      PHASH_DEF            def;
208      ANY_HASH_STATE       state;
209  } HASH_STATE, *PHASH_STATE;
210  typedef const HASH_STATE *PCHASH_STATE;
```

10.1.3.3 HMAC State Structures

An HMAC_STATE structure contains an opaque HMAC stack state. A caller would use this structure when performing incremental HMAC operations. This structure contains a hash state and an HMAC key and allows slightly better stack optimization than adding an HMAC key to each hash state.

```
211  typedef struct hmacState
212  {
213      HASH_STATE           hashState;           // the hash state
214      TPM2B_HASH_BLOCK     hmacKey;            // the HMAC key
215  } HMAC_STATE, *PHMAC_STATE;
```

This is for the external hash state. This implementation assumes that the size of the exported hash state is no larger than the internal hash state.

```
216  typedef struct
217  {
218      BYTE                 buffer[sizeof(HASH_STATE)];
219  } EXPORT_HASH_STATE, *PEXPORT_HASH_STATE;
220
221  typedef const EXPORT_HASH_STATE *PCEXPORT_HASH_STATE;
222
223  #endif // _CRYPT_HASH_H
```

10.1.4 CryptRand.h

10.1.4.1 Introduction

This file contains constant definition shared by CryptUtil() and the parts of the Crypto Engine.

```
1  #ifndef _CRYPT_RAND_H
2  #define _CRYPT_RAND_H
```

10.1.4.2 DRBG Structures and Defines

Values and structures for the random number generator. These values are defined in this header file so that the size of the RNG state can be known to TPM.lib. This allows the allocation of some space in NV memory for the state to be stored on an orderly shutdown.

The DRBG based on a symmetric block cipher is defined by three values,

- 1) the key size
- 2) the block size (the IV size)
- 3) the symmetric algorithm

```
3  #define DRBG_KEY_SIZE_BITS      AES_MAX_KEY_SIZE_BITS
4  #define DRBG_IV_SIZE_BITS      (AES_MAX_BLOCK_SIZE * 8)
5  #define DRBG_ALGORITHM         TPM_ALG_AES
6
7  typedef tpmKeyScheduleAES      DRBG_KEY_SCHEDULE;
8  #define DRBG_ENCRYPT_SETUP(key, keySizeInBits, schedule) \
9      TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule) \
10 #define DRBG_ENCRYPT(keySchedule, in, out) \
11     TpmCryptEncryptAES(SWIZZLE(keySchedule, in, out))
12
13 #if ((DRBG_KEY_SIZE_BITS % RADIX_BITS) != 0) \
14 || ((DRBG_IV_SIZE_BITS % RADIX_BITS) != 0)
15 #error "Key size and IV for DRBG must be even multiples of the radix"
16 #endif
17 #if (DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
18 #error "Key size for DRBG must be even multiple of the cypher block size"
19 #endif
```

Derived values

```
20 #define DRBG_MAX_REQUESTS_PER_RESEED (1 << 48)
21 #define DRBG_MAX_REQUEST_SIZE (1 << 32)
22
23 #define pDRBG_KEY(seed) ((DRBG_KEY *) &(((BYTE *) (seed))[0]))
24 #define pDRBG_IV(seed) ((DRBG_IV *) &(((BYTE *) (seed))[DRBG_KEY_SIZE_BYTES]))
25
26 #define DRBG_KEY_SIZE_WORDS      (BITS_TO_CRYPT_WORDS(DRBG_KEY_SIZE_BITS))
27 #define DRBG_KEY_SIZE_BYTES      (DRBG_KEY_SIZE_WORDS * RADIX_BYTES)
28
29 #define DRBG_IV_SIZE_WORDS      (BITS_TO_CRYPT_WORDS(DRBG_IV_SIZE_BITS))
30 #define DRBG_IV_SIZE_BYTES      (DRBG_IV_SIZE_WORDS * RADIX_BYTES)
31
32 #define DRBG_SEED_SIZE_WORDS      (DRBG_KEY_SIZE_WORDS + DRBG_IV_SIZE_WORDS)
33 #define DRBG_SEED_SIZE_BYTES      (DRBG_KEY_SIZE_BYTES + DRBG_IV_SIZE_BYTES)
34
35 typedef union
36 {
37     BYTE          bytes[DRBG_KEY_SIZE_BYTES];
38     crypt_uword_t words[DRBG_KEY_SIZE_WORDS];
```

```

39 } DRBG_KEY;
40
41 typedef union
42 {
43     BYTE            bytes[DRBG_IV_SIZE_BYTES];
44     crypt_uword_t   words[DRBG_IV_SIZE_WORDS];
45 } DRBG_IV;
46
47 typedef union
48 {
49     BYTE            bytes[DRBG_SEED_SIZE_BYTES];
50     crypt_uword_t   words[DRBG_SEED_SIZE_WORDS];
51 } DRBG_SEED;
52
53 #define CTR_DRBG_MAX_REQUESTS_PER_RESEED    ((UINT64)1 << 20)
54 #define CTR_DRBG_MAX_BYTES_PER_REQUEST     (1 << 16)
55
56 # define CTR_DRBG_MIN_ENTROPY_INPUT_LENGTH  DRBG_SEED_SIZE_BYTES
57 # define CTR_DRBG_MAX_ENTROPY_INPUT_LENGTH  DRBG_SEED_SIZE_BYTES
58 # define CTR_DRBG_MAX_ADDITIONAL_INPUT_LENGTH DRBG_SEED_SIZE_BYTES
59
60 #define TESTING            (1 << 0)
61 #define ENTROPY            (1 << 1)
62 #define TESTED             (1 << 2)
63
64 #define IsTestStateSet(BIT)    ((g_cryptoSelfTestState.rng & BIT) != 0)
65 #define SetTestStateBit(BIT)   (g_cryptoSelfTestState.rng |= BIT)
66 #define ClearTestStateBit(BIT) (g_cryptoSelfTestState.rng &= ~BIT)
67
68 #define IsSelfTest()           IsTestStateSet(TESTING)
69 #define SetSelfTest()          SetTestStateBit(TESTING)
70 #define ClearSelfTest()        ClearTestStateBit(TESTING)
71
72 #define IsEntropyBad()         IsTestStateSet(ENTROPY)
73 #define SetEntropyBad()        SetTestStateBit(ENTROPY)
74 #define ClearEntropyBad()      ClearTestStateBit(ENTROPY)
75
76 #define IsDrbgTested()         IsTestStateSet(TESTED)
77 #define SetDrbgTested()        SetTestStateBit(TESTED)
78 #define ClearDrbgTested()      ClearTestStateBit(TESTED)
79
80 typedef struct
81 {
82     UINT64      reseedCounter;
83     UINT32      magic;
84     DRBG_SEED   seed; // contains the key and IV for the counter mode DRBG
85     UINT32      lastValue[4]; // used when the TPM does continuous self-test
86                                     // for FIPS compliance of DRBG
87 } DRBG_STATE, *pDRBG_STATE;
88 #define DRBG_MAGIC    ((UINT32) 0x47425244) // "DRBG" backwards so that it displays
89
90 typedef struct KDF_STATE {
91     UINT64      counter;
92     UINT32      magic;
93     UINT32      limit;
94     TPM2B       *seed;
95     const TPM2B *label;
96     TPM2B       *context;
97     TPM_ALG_ID   hash;
98     TPM_ALG_ID   kdf;
99     UINT16       digestSize;
100    TPM2B_DIGEST  residual;
101 } KDF_STATE, *pKDR_STATE;
102 #define KDF_MAGIC    ((UINT32) 0x4048444a) // "KDF " backwards

```

Make sure that any other structures added to this union start with a 64-bit counter and a 32-bit magic number

```

103 typedef union
104 {
105     DRBG_STATE      drbg;
106     KDF_STATE       kdf;
107 } RAND_STATE;

```

This is the state used when the library uses a random number generator. A special function is installed for the library to call. That function picks up the state from this location and uses it for the generation of the random number.

```

108 extern RAND_STATE      *s_random;

```

When instrumenting RSA key sieve

```

109 #if RSA_INSTRUMENT
110 #define PRIME_INDEX(x) ((x) == 512 ? 0 : (x) == 1024 ? 1 : 2)
111 # define INSTRUMENT_SET(a, b) ((a) = (b))
112 # define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
113 # define INSTRUMENT_INC(a)    (a) = (a) + 1
114
115 extern UINT32 PrimeIndex;
116 extern UINT32 failedAtIteration[10];
117 extern UINT32 PrimeCounts[3];
118 extern UINT32 MillerRabinTrials[3];
119 extern UINT32 totalFieldsSieved[3];
120 extern UINT32 bitsInFieldAfterSieve[3];
121 extern UINT32 emptyFieldsSieved[3];
122 extern UINT32 noPrimeFields[3];
123 extern UINT32 primesChecked[3];
124 extern UINT16 lastSievePrime;
125 #else
126 # define INSTRUMENT_SET(a, b)
127 # define INSTRUMENT_ADD(a, b)
128 # define INSTRUMENT_INC(a)
129 #endif
130
131 #endif // _CRYPT_RAND_H

```

10.1.5 CryptRsa.h

This file contains the RSA-related structures and defines.

```
1  #ifndef _CRYPT_RSA_H
2  #define _CRYPT_RSA_H
```

These values are used in the *bigNum* representation of various RSA values.

```
3  BN_TYPE(rsa, MAX_RSA_KEY_BITS);
4  #define BN_RSA(name)      BN_VAR(name, MAX_RSA_KEY_BITS)
5  #define BN_RSA_INITIALIZED(name, initializer) \
6      BN_INITIALIZED(name, MAX_RSA_KEY_BITS, initializer)
7
8  #define BN_PRIME(name)      BN_VAR(name, (MAX_RSA_KEY_BITS / 2))
9  BN_TYPE(prime, (MAX_RSA_KEY_BITS / 2));
10 #define BN_PRIME_INITIALIZED(name, initializer) \
11     BN_INITIALIZED(name, MAX_RSA_KEY_BITS / 2, initializer)
12
13 #if !CRT_FORMAT_RSA
14 #   error This version only works with CRT formatted data
15 #endif // !CRT_FORMAT_RSA
16
17 typedef struct privateExponent
18 {
19     bigNum      P;
20     bigNum      Q;
21     bigNum      dP;
22     bigNum      dQ;
23     bigNum      qInv;
24     bn_prime_t  entries[5];
25 } privateExponent;
26
27 #define NEW_PRIVATE_EXPONENT(X) \
28     privateExponent  _##X; \
29     privateExponent  *X = RsaInitializeExponent(&(_##X))
30
31 #endif // _CRYPT_RSA_H
```


10.1.6 CryptTest.h

This file contains constant definitions used for self-test.

```
1  #ifndef _CRYPT_TEST_H
2  #define _CRYPT_TEST_H
```

This is the definition of a bit array with one bit per algorithm.

NOTE Since bit numbering starts at zero, when TPM_ALG_LAST is a multiple of 8, ALGORITHM_VECTOR will need to have byte for the single bit in the last byte. So, for example, when TPM_ALG_LAST is 8, ALGORITHM_VECTOR will need 2 bytes.

```
3  #define ALGORITHM_VECTOR_BYTES ((TPM_ALG_LAST + 8) / 8)
4  typedef BYTE    ALGORITHM_VECTOR[ALGORITHM_VECTOR_BYTES];
5
6  #ifdef TEST_SELF_TEST
7  LIB_EXPORT     extern ALGORITHM_VECTOR    LibToTest;
8  #endif
```

This structure is used to contain self-test tracking information for the cryptographic modules. Each of the major modules is given a 32-bit value in which it may maintain its own self test information. The convention for this state is that when all of the bits in this structure are 0, all functions need to be tested.

```
9  typedef struct
10 {
11     UINT32    rng;
12     UINT32    hash;
13     UINT32    sym;
14     #if ALG_RSA
15     UINT32    rsa;
16     #endif
17     #if ALG_ECC
18     UINT32    ecc;
19     #endif
20 } CRYPTO_SELF_TEST_STATE;
21
22 #endif // _CRYPT_TEST_H
```

10.1.7 HashTestData.h

Hash Test Vectors

```

1  TPM2B_TYPE(HASH_TEST_KEY, 128); // Twice the largest digest size
2  TPM2B_HASH_TEST_KEY      c_hashTestKey = {{128, {
3      0xa0,0xed,0x5c,0x9a,0xd2,0x4a,0x21,0x40,0x1a,0xd0,0x81,0x47,0x39,0x63,0xf9,0x50,
4      0xdc,0x59,0x47,0x11,0x40,0x13,0x99,0x92,0xc0,0x72,0xa4,0x0f,0xe2,0x33,0xe4,0x63,
5      0x9b,0xb6,0x76,0xc3,0x1e,0x6f,0x13,0xee,0xcc,0x99,0x71,0xa5,0xc0,0xcf,0x9a,0x40,
6      0xcf,0xdb,0x66,0x70,0x05,0x63,0x54,0x12,0x25,0xf4,0xe0,0x1b,0x23,0x35,0xe3,0x70,
7      0x7d,0x19,0x5f,0x00,0xe4,0xf1,0x61,0x73,0x05,0xd8,0x58,0x7f,0x60,0x61,0x84,0x36,
8      0xec,0xbe,0x96,0x1b,0x69,0x00,0xf0,0x9a,0x6e,0xe3,0x26,0x73,0x0d,0x17,0x5b,0x33,
9      0x41,0x44,0x9d,0x90,0xab,0xd9,0x6b,0x7d,0x48,0x99,0x25,0x93,0x29,0x14,0x2b,0xce,
10     0x93,0x8d,0x8c,0xaf,0x31,0x0e,0x9c,0x57,0xd8,0x5b,0x57,0x20,0x1b,0x9f,0x2d,0xa5
11     }}};
12
13 TPM2B_TYPE(HASH_TEST_DATA, 256); // Twice the largest block size
14 TPM2B_HASH_TEST_DATA      c_hashTestData = {{256, {
15     0x88,0xac,0xc3,0xe5,0x5f,0x66,0x9d,0x18,0x80,0xc9,0x7a,0x9c,0xa4,0x08,0x90,0x98,
16     0x0f,0x3a,0x53,0x92,0x4c,0x67,0x4e,0xb7,0x37,0xec,0x67,0x87,0xb6,0xbe,0x10,0xca,
17     0x11,0x5b,0x4a,0x0b,0x45,0xc3,0x32,0x68,0x48,0x69,0xce,0x25,0x1b,0xc8,0xaf,0x44,
18     0x79,0x22,0x83,0xc8,0xfb,0xe2,0x63,0x94,0xa2,0x3c,0x59,0x3e,0x3e,0xc6,0x64,0x2c,
19     0x1f,0x8c,0x11,0x93,0x24,0xa3,0x17,0xc5,0x2f,0x37,0xcf,0x95,0x97,0x8e,0x63,0x39,
20     0x68,0xd5,0xca,0xba,0x18,0x37,0x69,0x6e,0x4f,0x19,0xfd,0x8a,0xc0,0x8d,0x87,0x3a,
21     0xbc,0x31,0x42,0x04,0x05,0xef,0xb5,0x02,0xef,0x1e,0x92,0x4b,0xb7,0x73,0x2c,0x8c,
22     0xeb,0x23,0x13,0x81,0x34,0xb9,0xb5,0xc1,0x17,0x37,0x39,0xf8,0x3e,0xe4,0x4c,0x06,
23     0xa8,0x81,0x52,0x2f,0xef,0xc9,0x9c,0x69,0x89,0xbc,0x85,0x9c,0x30,0x16,0x02,0xca,
24     0xe3,0x61,0xd4,0x0f,0xed,0x34,0x1b,0xca,0xc1,0x1b,0xd1,0xfa,0xc1,0xa2,0xe0,0xdf,
25     0x52,0x2f,0x0b,0x4b,0x9f,0x0e,0x45,0x54,0xb9,0x17,0xb6,0xaf,0xd6,0xd5,0xca,0x90,
26     0x29,0x57,0x7b,0x70,0x50,0x94,0x5c,0x8e,0xf6,0x4e,0x21,0x8b,0xc6,0x8b,0xa6,0xbc,
27     0xb9,0x64,0xd4,0x4d,0xf3,0x68,0xd8,0xac,0xde,0xd8,0xd8,0xb5,0x6d,0xcd,0x93,0xeb,
28     0x28,0xa4,0xe2,0x5c,0x44,0xef,0xf0,0xe1,0x6f,0x38,0x1a,0x3c,0xe6,0xef,0xa2,0x9d,
29     0xb9,0xa8,0x05,0x2a,0x95,0xec,0x5f,0xdb,0xb0,0x25,0x67,0x9c,0x86,0x7a,0x8e,0xea,
30     0x51,0xcc,0xc3,0xd3,0xff,0x6e,0xf0,0xed,0xa3,0xae,0xf9,0x5d,0x33,0x70,0xf2,0x11
31     }}};
32
33 #if ALG_SHA1 == YES
34 TPM2B_TYPE(SHA1, 20);
35 TPM2B_SHA1      c_SHA1_digest = {{20, {
36     0xee,0x2c,0xef,0x93,0x76,0xbd,0xf8,0x91,0xbc,0xe6,0xe5,0x57,0x53,0x77,0x01,0xb5,
37     0x70,0x95,0xe5,0x40
38     }}};
39 #endif
40
41 #if ALG_SHA256 == YES
42 TPM2B_TYPE(SHA256, 32);
43 TPM2B_SHA256      c_SHA256_digest = {{32, {
44     0x64,0xe8,0xe0,0xc3,0xa9,0xa4,0x51,0x49,0x10,0x55,0x8d,0x31,0x71,0xe5,0x2f,0x69,
45     0x3a,0xdc,0xc7,0x11,0x32,0x44,0x61,0xbd,0x34,0x39,0x57,0xb0,0xa8,0x75,0x86,0x1b
46     }}};
47 #endif
48
49 #if ALG_SHA384 == YES
50 TPM2B_TYPE(SHA384, 48);
51 TPM2B_SHA384      c_SHA384_digest = {{48, {
52     0x37,0x75,0x29,0xb5,0x20,0x15,0x6e,0xa3,0x7e,0xa3,0x0d,0xcd,0x80,0xa8,0xa3,0x3d,
53     0xeb,0xe8,0xad,0x4e,0x1c,0x77,0x94,0x5a,0xaf,0x6c,0xd0,0xc1,0xfa,0x43,0x3f,0xc7,
54     0xb8,0xf1,0x01,0xc0,0x60,0xbf,0xf2,0x87,0xe8,0x71,0x9e,0x51,0x97,0xa0,0x09,0x8d
55     }}};
56 #endif
57
58 #if ALG_SHA512 == YES
59 TPM2B_TYPE(SHA512, 64);
60 TPM2B_SHA512      c_SHA512_digest = {{64, {
61     0xe2,0x7b,0x10,0x3d,0x5e,0x48,0x58,0x44,0x67,0xac,0xa3,0x81,0x8c,0x1d,0xc5,0x71,

```

```
62     0x66,0x92,0x8a,0x89,0xaa,0xd4,0x35,0x51,0x60,0x37,0x31,0xd7,0xba,0xe7,0x93,0x0b,  
63     0x16,0x4d,0xb3,0xc8,0x34,0x98,0x3c,0xd3,0x53,0xde,0x5e,0xe8,0x0c,0xbc,0xaf,0xc9,  
64     0x24,0x2c,0xcc,0xed,0xdb,0xde,0xba,0x1f,0x14,0x14,0x5a,0x95,0x80,0xde,0x66,0xbd  
65     }  
66 #endif  
67  
68 TPM2B_TYPE(EMPTY, 1);  
69  
70 #if ALG_SM3_256 == YES  
71 TPM2B_EMPTY c_SM3_256_digest = {{0, {0}}};  
72 #endif  
73  
74 #if ALG_SHA3_256 == YES  
75 TPM2B_EMPTY c_SHA3_256_digest = {{0, {0}}};  
76 #endif  
77  
78 #if ALG_SHA3_384 == YES  
79 TPM2B_EMPTY c_SHA3_384_digest = {{0, {0}}};  
80 #endif  
81  
82 #if ALG_SHA3_512 == YES  
83 TPM2B_EMPTY c_SHA3_512_digest = {{0, {0}}};  
84 #endif
```

10.1.8 KdfTestData.h

Hash Test Vectors

```

1  #define TEST_KDF_KEY_SIZE    20
2
3  TPM2B_TYPE(KDF_TEST_KEY, TEST_KDF_KEY_SIZE);
4  TPM2B_KDF_TEST_KEY    c_kdfTestKeyIn = {{TEST_KDF_KEY_SIZE, {
5      0x27, 0x1F, 0xA0, 0x8B, 0xBD, 0xC5, 0x06, 0x0E, 0xC3, 0xDF,
6      0xA9, 0x28, 0xFF, 0x9B, 0x73, 0x12, 0x3A, 0x12, 0xDA, 0x0C }}};
7
8  TPM2B_TYPE(KDF_TEST_LABEL, 17);
9  TPM2B_KDF_TEST_LABEL    c_kdfTestLabel = {{17, {
10     0x4B, 0x44, 0x46, 0x53, 0x45, 0x4C, 0x46, 0x54,
11     0x45, 0x53, 0x54, 0x4C, 0x41, 0x42, 0x45, 0x4C, 0x00 }}};
12
13  TPM2B_TYPE(KDF_TEST_CONTEXT, 8);
14  TPM2B_KDF_TEST_CONTEXT    c_kdfTestContextU = {{8, {
15     0xCE, 0x24, 0x4F, 0x39, 0x5D, 0xCA, 0x73, 0x91 }}};
16
17  TPM2B_KDF_TEST_CONTEXT    c_kdfTestContextV = {{8, {
18     0xDA, 0x50, 0x40, 0x31, 0xDD, 0xF1, 0x2E, 0x83 }}};
19
20  #if ALG_SHA512 == ALG_YES
21      TPM2B_KDF_TEST_KEY    c_kdfTestKeyOut = {{20, {
22          0x8b, 0xe2, 0xc1, 0xb8, 0x5b, 0x78, 0x56, 0x9b, 0x9f, 0xa7,
23          0x59, 0xf5, 0x85, 0x7c, 0x56, 0xd6, 0x84, 0x81, 0x0f, 0xd3 }}};
24      #define KDF_TEST_ALG    TPM_ALG_SHA512
25
26  #elif ALG_SHA384 == ALG_YES
27      TPM2B_KDF_TEST_KEY    c_kdfTestKeyOut = {{20, {
28          0x1d, 0xce, 0x70, 0xc9, 0x11, 0x3e, 0xb2, 0xdb, 0xa4, 0x7b,
29          0xd9, 0xcf, 0xc7, 0x2b, 0xf4, 0x6f, 0x45, 0xb0, 0x93, 0x12 }}};
30      #define KDF_TEST_ALG    TPM_ALG_SHA384
31
32  #elif ALG_SHA256 == ALG_YES
33      TPM2B_KDF_TEST_KEY    c_kdfTestKeyOut = {{20, {
34          0xbb, 0x02, 0x59, 0xe1, 0xc8, 0xba, 0x60, 0x7e, 0x6a, 0x2c,
35          0xd7, 0x04, 0xb6, 0x9a, 0x90, 0x2e, 0x9a, 0xde, 0x84, 0xc4 }}};
36      #define KDF_TEST_ALG    TPM_ALG_SHA256
37
38  #elif ALG_SHA1 == ALG_YES
39      TPM2B_KDF_TEST_KEY    c_kdfTestKeyOut = {{20, {
40          0x55, 0xb5, 0xa7, 0x18, 0x4a, 0xa0, 0x74, 0x23, 0xc4, 0x7d,
41          0xae, 0x76, 0x6c, 0x26, 0xa2, 0x37, 0x7d, 0x7c, 0xf8, 0x51 }}};
42      #define KDF_TEST_ALG    TPM_ALG_SHA1
43  #endif

```

10.1.9 RsaTestData.h

RSA Test Vectors

```

1  #define RSA_TEST_KEY_SIZE    256
2
3  typedef struct
4  {
5      UINT16      size;
6      BYTE        buffer[RSA_TEST_KEY_SIZE];
7  } TPM2B_RSA_TEST_KEY;
8
9  typedef TPM2B_RSA_TEST_KEY  TPM2B_RSA_TEST_VALUE;
10
11 typedef struct
12 {
13     UINT16      size;
14     BYTE        buffer[RSA_TEST_KEY_SIZE / 2];
15 } TPM2B_RSA_TEST_PRIME;
16
17 const TPM2B_RSA_TEST_KEY  c_rsaPublicModulus = {256, {
18     0x91,0x12,0xf5,0x07,0x9d,0x5f,0x6b,0x1c,0x90,0xf6,0xcc,0x87,0xde,0x3a,0x7a,0x15,
19     0xdc,0x54,0x07,0x6c,0x26,0x8f,0x25,0xef,0x7e,0x66,0xc0,0xe3,0x82,0x12,0x2f,0xab,
20     0x52,0x82,0x1e,0x85,0xbc,0x53,0xba,0x2b,0x01,0xad,0x01,0xc7,0x8d,0x46,0x4f,0x7d,
21     0xdd,0x7e,0xdc,0xb0,0xad,0xf6,0x0c,0xa1,0x62,0x92,0x97,0x8a,0x3e,0x6f,0x7e,0x3e,
22     0xf6,0x9a,0xcc,0xf9,0xa9,0x86,0x77,0xb6,0x85,0x43,0x42,0x04,0x13,0x65,0xe2,0xad,
23     0x36,0xc9,0xbf,0xc1,0x97,0x84,0x6f,0xee,0x7c,0xda,0x58,0xd2,0xae,0x07,0x00,0xaf,
24     0xc5,0x5f,0x4d,0x3a,0x98,0xb0,0xed,0x27,0x7c,0xc2,0xce,0x26,0x5d,0x87,0xe1,0xe3,
25     0xa9,0x69,0x88,0x4f,0x8c,0x08,0x31,0x18,0xae,0x93,0x16,0xe3,0x74,0xde,0xd3,0xf6,
26     0x16,0xaf,0xa3,0xac,0x37,0x91,0x8d,0x10,0xc6,0x6b,0x64,0x14,0x3a,0xd9,0xfc,0xe4,
27     0xa0,0xf2,0xd1,0x01,0x37,0x4f,0x4a,0xeb,0xe5,0xec,0x98,0xc5,0xd9,0x4b,0x30,0xd2,
28     0x80,0x2a,0x5a,0x18,0x5a,0x7d,0xd4,0x3d,0xb7,0x62,0x98,0xce,0x6d,0xa2,0x02,0x6e,
29     0x45,0xaa,0x95,0x73,0xe0,0xaa,0x75,0x57,0xb1,0x3d,0x1b,0x05,0x75,0x23,0x6b,0x20,
30     0x69,0x9e,0x14,0xb0,0x7f,0xac,0xae,0xd2,0xc7,0x48,0x3b,0xe4,0x56,0x11,0x34,0x1e,
31     0x05,0x1a,0x30,0x20,0xef,0x68,0x93,0x6b,0x9d,0x7e,0xdd,0xba,0x96,0x50,0xcc,0x1c,
32     0x81,0xb4,0x59,0xb9,0x74,0x36,0xd9,0x97,0xdc,0x8f,0x17,0x82,0x72,0xb3,0x59,0xf6,
33     0x23,0xfa,0x84,0xf7,0x6d,0xf2,0x05,0xff,0xf1,0xb9,0xcc,0xe9,0xa2,0x82,0x01,0xfb}};
34
35 const TPM2B_RSA_TEST_PRIME  c_rsaPrivatePrime = {RSA_TEST_KEY_SIZE / 2, {
36     0xb7,0xa0,0x90,0xc7,0x92,0x09,0xde,0x71,0x03,0x37,0x4a,0xb5,0x2f,0xda,0x61,0xb8,
37     0x09,0x1b,0xba,0x99,0x70,0x45,0xc1,0x0b,0x15,0x12,0x71,0x8a,0xb3,0x2a,0x4d,0x5a,
38     0x41,0x9b,0x73,0x89,0x80,0x0a,0x8f,0x18,0x4c,0x8b,0xa2,0x5b,0xda,0xbd,0x43,0xbe,
39     0xdc,0x76,0x4d,0x71,0x0f,0xb9,0xfc,0x7a,0x09,0xfe,0x4f,0xac,0x63,0xd9,0x2e,0x50,
40     0x3a,0xa1,0x37,0xc6,0xf2,0xa1,0x89,0x12,0xe7,0x72,0x64,0x2b,0xba,0xc1,0x1f,0xca,
41     0x9d,0xb7,0xaa,0x3a,0xa9,0xd3,0xa6,0x6f,0x73,0x02,0xbb,0x85,0x5d,0x9a,0xb9,0x5c,
42     0x08,0x83,0x22,0x20,0x49,0x91,0x5f,0x4b,0x86,0xbc,0x3f,0x76,0x43,0x08,0x97,0xbf,
43     0x82,0x55,0x36,0x2d,0x8b,0x6e,0x9e,0xfb,0xc1,0x67,0x6a,0x43,0xa2,0x46,0x81,0x71}};
44
45 const BYTE  c_RsaTestValue[RSA_TEST_KEY_SIZE] = {
46     0x2a,0x24,0x3a,0xbb,0x50,0x1d,0xd4,0x2a,0xf9,0x18,0x32,0x34,0xa2,0x0f,0xea,0x5c,
47     0x91,0x77,0xe9,0xe1,0x09,0x83,0xdc,0x5f,0x71,0x64,0x5b,0xeb,0x57,0x79,0xa0,0x41,
48     0xc9,0xe4,0x5a,0x0b,0xf4,0x9f,0xdb,0x84,0x04,0xa6,0x48,0x24,0xf6,0x3f,0x66,0x1f,
49     0xa8,0x04,0x5c,0xf0,0x7a,0x6b,0x4a,0x9c,0x7e,0x21,0xb6,0xda,0x6b,0x65,0x9c,0x3a,
50     0x68,0x50,0x13,0x1e,0xa4,0xb7,0xca,0xec,0xd3,0xcc,0xb2,0x9b,0x8c,0x87,0xa4,0x6a,
51     0xba,0xc2,0x06,0x3f,0x40,0x48,0x7b,0xa8,0xb8,0x2c,0x03,0x14,0x33,0xf3,0x1d,0xe9,
52     0xbd,0x6f,0x54,0x66,0xb4,0x69,0x5e,0xbc,0x80,0x7c,0xe9,0x6a,0x43,0x7f,0xb8,0x6a,
53     0xa0,0x5f,0x5d,0x7a,0x20,0xfd,0x7a,0x39,0xe1,0xea,0x0e,0x94,0x91,0x28,0x63,0x7a,
54     0xac,0xc9,0xa5,0x3a,0x6d,0x31,0x7b,0x7c,0x54,0x56,0x99,0x56,0xbb,0xb7,0xa1,0x2d,
55     0xd2,0x5c,0x91,0x5f,0x1c,0xd3,0x06,0x7f,0x34,0x53,0x2f,0x4c,0xd1,0x8b,0xd2,0x9e,
56     0xdc,0xc3,0x94,0x0a,0xe1,0x0f,0xa5,0x15,0x46,0x2a,0x8e,0x10,0xc2,0xfe,0xb7,0x5e,
57     0x2d,0x0d,0xd1,0x25,0xfc,0xe4,0xf7,0x02,0x19,0xfe,0xb6,0xe4,0x95,0x9c,0x17,0x4a,
58     0x9b,0xdb,0xab,0xc7,0x79,0xe3,0x5e,0x40,0xd0,0x56,0x6d,0x25,0x0a,0x72,0x65,0x80,
59     0x92,0x9a,0xa8,0x07,0x70,0x32,0x14,0xfb,0xfe,0x08,0xeb,0x13,0xb4,0x07,0x68,0xb4,
60     0x58,0x39,0xbe,0x8e,0x78,0x3a,0x59,0x3f,0x9c,0x4c,0xe9,0xa8,0x64,0x68,0xf7,0xb9,
61     0x6e,0x20,0xf5,0xcb,0xca,0x47,0xf2,0x17,0xaa,0x8b,0xbc,0x13,0x14,0x84,0xf6,0xab};

```

```

62
63 const TPM2B_RSA_TEST_VALUE    c_RsaepKvt = {RSA_TEST_KEY_SIZE, {
64     0x73,0xbd,0x65,0x49,0xda,0x7b,0xb8,0x50,0x9e,0x87,0xf0,0x0a,0x8a,0x07,0xb6,
65     0x00,0x82,0x10,0x14,0x60,0xd8,0x01,0xfc,0xc5,0x18,0xea,0x49,0x5f,0x13,0xcf,0x65,
66     0x66,0x30,0x6c,0x60,0x3f,0x24,0x3c,0xfb,0xe2,0x31,0x16,0x99,0x7e,0x31,0x98,0xab,
67     0x93,0xb8,0x07,0x53,0xcc,0xdb,0x7f,0x44,0xd9,0xee,0x5d,0xe8,0x5f,0x97,0x5f,0xe8,
68     0x1f,0x88,0x52,0x24,0x7b,0xac,0x62,0x95,0xb7,0x7d,0xf5,0xf8,0x9f,0x5a,0xa8,0x24,
69     0x9a,0x76,0x71,0x2a,0x35,0x2a,0xa1,0x08,0xbb,0x95,0xe3,0x64,0xdc,0xdb,0xc2,0x33,
70     0xa9,0x5f,0xbe,0x4c,0xc4,0xcc,0x28,0xc9,0x25,0xff,0xee,0x17,0x15,0x9a,0x50,0x90,
71     0x0e,0x15,0xb4,0xea,0x6a,0x09,0xe6,0xff,0xa4,0xe4,0xee,0xc7,0x7e,0xce,0xa9,0x73,0xe4,
72     0xa0,0x56,0xbd,0x53,0x2a,0xe4,0xc0,0x2b,0xa8,0x9b,0x09,0x30,0x72,0x62,0x0f,0xf9,
73     0xf6,0xa1,0x52,0xd2,0x8a,0x37,0xee,0xa5,0xc8,0x47,0xe1,0x99,0x21,0x47,0xeb,0xdd,
74     0x37,0xaa,0xe4,0xbd,0x55,0x46,0x5a,0x5a,0x5d,0xfb,0x7b,0xfc,0xff,0xbf,0x26,0x71,
75     0xf6,0x1e,0xad,0xbc,0xbf,0x33,0xca,0xe1,0x92,0x8f,0x2a,0x89,0x6c,0x45,0x24,0xd1,
76     0xa6,0x52,0x56,0x24,0x5e,0x90,0x47,0xe5,0xcb,0x12,0xb0,0x32,0xf9,0xa6,0xbb,0xea,
77     0x37,0xa9,0xbd,0xef,0x23,0xef,0x63,0x07,0x6c,0xc4,0x4e,0x64,0x3c,0xc6,0x11,0x84,
78     0x7d,0x65,0xd6,0x7a,0x17,0x58,0xa5,0xf7,0x74,0x3b,0x42,0xe3,0xd2,0xda,0x5f,
79     0x6f,0xe0,0x1e,0x4b,0xcf,0x46,0xe2,0xdf,0x3e,0x41,0x8e,0x0e,0xb0,0x3f,0x8b,0x65}};
80
81 #define    OAEP_TEST_LABEL        "OAEP Test Value"
82
83 #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
84
85 const TPM2B_RSA_TEST_VALUE    c_OaepKvt = {RSA_TEST_KEY_SIZE, {
86     0x32,0x68,0x84,0x0b,0x9c,0xc9,0x25,0x26,0xd9,0xc0,0xd0,0xb1,0xde,0x60,0x55,0xae,
87     0x33,0xe5,0xcf,0x6c,0x85,0xbe,0x0d,0x71,0x11,0xe1,0x45,0x60,0xbb,0x42,0x3d,0xf3,
88     0xb1,0x18,0x84,0x7b,0xc6,0x5d,0xce,0x1d,0x5f,0x9a,0x97,0xcf,0xb1,0x97,0x9a,0x85,
89     0x7c,0xa7,0xa1,0x63,0x23,0xb6,0x74,0x0f,0x1a,0xee,0x29,0x51,0xeb,0x50,0x8f,0x3c,
90     0x8e,0x4e,0x31,0x38,0xdc,0x11,0xfc,0x9a,0x4e,0xaf,0x93,0xc9,0x7f,0x6e,0x35,0xf3,
91     0xc9,0xe4,0x89,0x14,0x53,0xe2,0xc2,0x1a,0xf7,0x6b,0x9b,0xf0,0x7a,0xa4,0x69,0x52,
92     0xe0,0x24,0x8f,0xea,0x31,0xa7,0x5c,0x43,0xb0,0x65,0xc9,0xfe,0xba,0xfe,0x80,0x9e,
93     0xa5,0xc0,0xf5,0x8d,0xce,0x41,0xf9,0x83,0x0d,0x8e,0x0f,0xef,0x3d,0x1f,0x6a,0xcc,
94     0x8a,0x3d,0x3b,0xdf,0x22,0x38,0xd7,0x34,0x58,0x7b,0x55,0xc9,0xf6,0xbc,0x7c,0x4c,
95     0x3f,0xd7,0xde,0x4e,0x30,0xa9,0x69,0xf3,0x5f,0x56,0x8f,0xc2,0xe7,0x75,0x79,0xb8,
96     0xa5,0xc8,0x0d,0xc0,0xcd,0xb6,0xc9,0x63,0xad,0x7c,0xe4,0x8f,0x39,0x60,0x4d,0x7d,
97     0xdb,0x34,0x49,0x2a,0x47,0xde,0xc0,0x42,0x4a,0x19,0x94,0x2e,0x50,0x21,0x03,0x47,
98     0xff,0x73,0xb3,0xb7,0x89,0xcc,0x7b,0x2c,0xeb,0x03,0xa7,0x9a,0x06,0xfd,0xed,0x19,
99     0xbb,0x82,0xa0,0x13,0xe9,0xfa,0xac,0x06,0x5f,0xc5,0xa9,0x2b,0xda,0x88,0x23,0xa2,
100    0x5d,0xc2,0x7f,0xda,0xc8,0x5a,0x94,0x31,0xc1,0x21,0xd7,0x1e,0x6b,0xd7,0x89,0xb1,
101    0x93,0x80,0xab,0xd1,0x37,0xf2,0x6f,0x50,0xcd,0x2a,0xea,0xb1,0xc4,0xcd,0xcb,0xb5}};
102
103 const TPM2B_RSA_TEST_VALUE    c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
104     0x29,0xa4,0x2f,0xbb,0x8a,0x14,0x05,0x1e,0x3c,0x72,0x76,0x77,0x38,0xe7,0x73,0xe3,
105     0x6e,0x24,0x4b,0x38,0xd2,0x1a,0xcf,0x23,0x58,0x78,0x36,0x82,0x23,0x6e,0x6b,0xef,
106     0x2c,0x3d,0xf2,0xe8,0xd6,0xc6,0x87,0x8e,0x78,0x9b,0x27,0x39,0xc0,0xd6,0xef,0x4d,
107     0x0b,0xfc,0x51,0x27,0x18,0xf3,0x51,0x5e,0x4d,0x96,0x3a,0xe2,0x15,0xe2,0x7e,0x42,
108     0xf4,0x16,0xd5,0xc6,0x52,0x5d,0x17,0x44,0x76,0x09,0x7a,0xcf,0xe3,0x30,0xe3,0x84,
109     0xf6,0x6f,0x3a,0x33,0xfb,0x32,0x0d,0x1d,0xe7,0x7c,0x80,0x82,0x4f,0xed,0xda,0x87,
110     0x11,0x9c,0xc3,0x7e,0x85,0xbd,0x18,0x58,0x08,0x2b,0x23,0x37,0xe7,0x9d,0xd0,0xd1,
111     0x79,0xe2,0x05,0xbd,0xf5,0x4f,0x0e,0x0f,0xdb,0x4a,0x74,0xeb,0x09,0x01,0xb3,0xca,
112     0xbd,0xa6,0x7b,0x09,0xb1,0x13,0x77,0x30,0x4d,0x87,0x41,0x06,0x57,0x2e,0x5f,0x36,
113     0x6e,0xfc,0x35,0x69,0xfe,0x0a,0x24,0x6c,0x98,0x8c,0xda,0x97,0xf4,0xfb,0xc7,0x83,
114     0x2d,0x3e,0x7d,0xc0,0x5c,0x34,0xfd,0x11,0x2a,0x12,0xa7,0xae,0x4a,0xde,0xc8,0x4e,
115     0xcf,0xf4,0x85,0x63,0x77,0xc6,0x33,0x34,0xe0,0x27,0xe4,0x9e,0x91,0x0b,0x4b,0x85,
116     0xf0,0xb0,0x79,0xaa,0x7c,0xc6,0xff,0x3b,0xbc,0x04,0x73,0xb8,0x95,0xd7,0x31,0x54,
117     0x3b,0x56,0xec,0x52,0x15,0xd7,0x3e,0x62,0xf5,0x82,0x99,0x3e,0x2a,0xc0,0x4b,0x2e,
118     0x06,0x57,0x6d,0x3f,0x3e,0x77,0x1f,0x2b,0x2d,0xc5,0xb9,0x3b,0x68,0x56,0x73,0x70,
119     0x32,0x6b,0x6b,0x65,0x25,0x76,0x45,0x6c,0x45,0xf1,0x6c,0x59,0xfc,0x94,0xa7,0x15}};
120
121 const TPM2B_RSA_TEST_VALUE    c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
122     0x01,0xfe,0xd5,0x83,0x0b,0x15,0xba,0x90,0x2c,0xdf,0xf7,0x26,0xb7,0x8f,0xb1,0xd7,
123     0x0b,0xfd,0x83,0xf9,0x95,0xd5,0xd7,0xb5,0xc5,0xc5,0x4a,0xde,0xd5,0xe6,0x20,0x78,
124     0xca,0x73,0x77,0x3d,0x61,0x36,0x48,0xae,0x3e,0x8f,0xee,0x43,0x29,0x96,0xdf,0x3f,
125     0x1c,0x97,0x5a,0xbe,0xe5,0xa2,0x7e,0x5b,0xd0,0xc0,0x29,0x39,0x83,0x81,0x77,0x24,
126     0x43,0xdb,0x3c,0x64,0x4d,0xf0,0x23,0xe4,0xae,0x0f,0x78,0x31,0x8c,0xda,0x0c,0xec,
127     0xf1,0xdf,0x09,0xf2,0x14,0x6a,0x4d,0xaf,0x36,0x81,0x6e,0xbd,0xbe,0x36,0x79,0x88,

```

```

128 0x98,0xb6,0x6f,0x5a,0xad,0xcf,0x7c,0xee,0xe0,0xdd,0x00,0xbe,0x59,0x97,0x88,0x00,
129 0x34,0xc0,0x8b,0x48,0x42,0x05,0x04,0x5a,0xb7,0x85,0x38,0xa0,0x35,0xd7,0x3b,0x51,
130 0xb8,0x7b,0x81,0x83,0xee,0xff,0x76,0x6f,0x50,0x39,0x4d,0xab,0x89,0x63,0x07,0x6d,
131 0xf5,0xe5,0x01,0x10,0x56,0xfe,0x93,0x06,0x8f,0xd3,0xc9,0x41,0xab,0xc9,0xdf,0x6e,
132 0x59,0xa8,0xc3,0x1d,0xbf,0x96,0x4a,0x59,0x80,0x3c,0x90,0x3a,0x59,0x56,0x4c,0x6d,
133 0x44,0x6d,0xeb,0xdc,0x73,0xcd,0xc1,0xec,0xb8,0x41,0xbf,0x89,0x8c,0x03,0x69,0x4c,
134 0xaf,0x3f,0xc1,0xc5,0xc7,0xe7,0x7d,0xa7,0x83,0x39,0x70,0xa2,0x6b,0x83,0xbc,0xbe,
135 0xf5,0xbf,0x1c,0xee,0x6e,0xa3,0x22,0x1e,0x25,0x2f,0x16,0x68,0x69,0x5a,0x1d,0xfa,
136 0x2c,0x3a,0x0f,0x67,0xe1,0x77,0x12,0xe8,0x3d,0xba,0xaa,0xef,0x96,0x9c,0x1f,0x64,
137 0x32,0xf4,0xa7,0xb3,0x3f,0x7d,0x61,0xbb,0x9a,0x27,0xad,0xfb,0x2f,0x33,0xc4,0x70}};
138
139 const TPM2B_RSA_TEST_VALUE c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
140 0x67,0x4e,0xdd,0xc2,0xd2,0x6d,0xe0,0x03,0xc4,0xc2,0x41,0xd3,0xd4,0x61,0x30,0xd0,
141 0xe1,0x68,0x31,0x4a,0xda,0xd9,0xc2,0x5d,0xaa,0xa2,0x7b,0xfb,0x44,0x02,0xf5,0xd6,
142 0xd8,0x2e,0xcd,0x13,0x36,0xc9,0x4b,0xdb,0x1a,0x4b,0x66,0x1b,0x4f,0x9c,0xb7,0x17,
143 0xac,0x53,0x37,0x4f,0x21,0xbd,0x0c,0x66,0xac,0x06,0x65,0x52,0x9f,0x04,0xf6,0xa5,
144 0x22,0x5b,0xf7,0xe6,0x0d,0x3c,0x9f,0x41,0x19,0x09,0x88,0x7c,0x41,0x4c,0x2f,0x9c,
145 0x8b,0x3c,0xdd,0x7c,0x28,0x78,0x24,0xd2,0x09,0xa6,0x5b,0xf7,0x3c,0x88,0x7e,0x73,
146 0x5a,0x2d,0x36,0x02,0x4f,0x65,0xb0,0xcb,0xc8,0xdc,0xac,0xa2,0xda,0x8b,0x84,0x91,
147 0x71,0xe4,0x30,0x8b,0xb6,0x12,0xf2,0xf0,0xd0,0xa0,0x38,0xcf,0x75,0xb7,0x20,0xcb,
148 0x35,0x51,0x52,0x6b,0xc4,0xf4,0x21,0x95,0xc2,0xf7,0x9a,0x13,0xc1,0x1a,0x7b,0x8f,
149 0x77,0xda,0x19,0x48,0xbb,0x6d,0x14,0x5d,0xba,0x65,0xb4,0x9e,0x43,0x42,0x58,0x98,
150 0x0b,0x91,0x46,0xd8,0x4c,0xf3,0x4c,0xaf,0x2e,0x02,0xa6,0xb2,0x49,0x12,0x62,0x43,
151 0x4e,0xa8,0xac,0xbf,0xf0,0xfa,0x37,0x24,0xea,0x69,0x1c,0xf5,0xae,0xfa,0x08,0x82,
152 0x30,0xc3,0xc0,0xf8,0x9a,0x89,0x33,0xe1,0x40,0x6d,0x18,0x5c,0x7b,0x90,0x48,0xbf,
153 0x37,0xdb,0xea,0xfb,0x0e,0xd4,0x2e,0x11,0xfa,0xa9,0x86,0xff,0x00,0x0b,0x7b,0xca,
154 0x09,0x64,0x6a,0x8f,0x0c,0x0e,0x09,0x14,0x36,0x4a,0x74,0x31,0x18,0x5b,0x18,0xeb,
155 0xea,0x83,0xc3,0x66,0x68,0xa6,0x7d,0x43,0x06,0x0f,0x99,0x60,0xce,0x65,0x08,0xf6}};
156
157 #endif // SHA1
158
159 #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
160
161 const TPM2B_RSA_TEST_VALUE c_OaepKvt = {RSA_TEST_KEY_SIZE, {
162 0x33,0x20,0x6e,0x21,0xc3,0xf6,0xcd,0xf8,0xd7,0x5d,0x9f,0xe9,0x05,0x14,0x8c,0x7c,
163 0xbb,0x69,0x24,0x9e,0x52,0x8f,0xaf,0x84,0x73,0x21,0x2c,0x85,0xa5,0x30,0x4d,0xb6,
164 0xb8,0xfa,0x15,0x9b,0xc7,0x8f,0xc9,0x7a,0x72,0x4b,0x85,0xa4,0x1c,0xc5,0xd8,0xe4,
165 0x92,0xb3,0xec,0xd9,0xa8,0xca,0x5e,0x74,0x73,0x89,0x7f,0xb4,0xac,0x7e,0x68,0x12,
166 0xb2,0x53,0x27,0x4b,0xbf,0xd0,0x71,0x69,0x46,0x9f,0xef,0xf4,0x70,0x60,0xf8,0xd7,
167 0xae,0xc7,0x5a,0x27,0x38,0x25,0x2d,0x25,0xab,0x96,0x56,0x66,0x3a,0x23,0x40,0xa8,
168 0xdb,0xbc,0x86,0xe8,0xf3,0xd2,0x58,0x0b,0x44,0xfc,0x94,0x1e,0xb7,0x5d,0xb4,0x57,
169 0xb5,0xf3,0x56,0xee,0x9b,0xc7,0x97,0x91,0x29,0x36,0xe3,0x06,0x13,0xa2,0xea,0xd6,
170 0xd6,0x0b,0x86,0x0b,0x1a,0x27,0xe6,0x22,0xc4,0x7b,0xff,0xde,0x0f,0xbf,0x79,0xc8,
171 0x1b,0xed,0xf1,0x27,0x62,0xb5,0x8b,0xf9,0xd9,0x76,0x90,0xf6,0xcc,0x83,0x0f,0xce,
172 0xce,0x2e,0x63,0x7a,0x9b,0xf4,0x48,0x5b,0xd7,0x81,0x2c,0x3a,0xdb,0x59,0x0d,0x4d,
173 0x9e,0x46,0xe9,0x9e,0x92,0x22,0x27,0x1c,0xb0,0x67,0x8a,0xe6,0x8a,0x16,0x8a,0xdf,
174 0x95,0x76,0x24,0x82,0xad,0xf1,0xbc,0x97,0xbf,0xd3,0x5e,0x6e,0x14,0x0c,0x5b,0x25,
175 0xfe,0x58,0xfa,0x64,0xe5,0x14,0x46,0xb7,0x58,0xc6,0x3f,0x7f,0x42,0xd2,0x8e,0x45,
176 0x13,0x41,0x85,0x12,0x2e,0x96,0x19,0xd0,0x5e,0x7d,0x34,0x06,0x32,0x2b,0xc8,0xd9,
177 0x0d,0x6c,0x06,0x36,0xa0,0xff,0x47,0x57,0x2c,0x25,0xbc,0x8a,0xa5,0xe2,0xc7,0xe3}};
178
179 const TPM2B_RSA_TEST_VALUE c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
180 0x39,0xfc,0x10,0x5d,0xf4,0x45,0x3d,0x94,0x53,0x06,0x89,0x24,0xe7,0xe8,0xfd,0x03,
181 0xac,0xfd,0xbd,0xb2,0x28,0xd3,0x4a,0x52,0xc5,0xd4,0xdb,0x17,0xd4,0x24,0x05,0xc4,
182 0xeb,0x6a,0xce,0x1d,0xbb,0x37,0xcb,0x09,0xd8,0x6c,0x83,0x19,0x93,0xd4,0xe2,0x88,
183 0x88,0x9b,0xaf,0x92,0x16,0xc4,0x15,0xbd,0x49,0x13,0x22,0xb7,0x84,0xcf,0x23,0xf2,
184 0x6f,0xc0,0x3e,0x3e,0x04,0x09,0x31,0x2d,0x99,0xdf,0xe6,0x74,0x70,0x30,0xde,
185 0x8c,0xad,0x32,0x86,0xe2,0x7c,0x12,0x90,0x21,0xf3,0x86,0xb7,0xe2,0x64,0xca,0x98,
186 0xcc,0x64,0x4b,0xef,0x57,0x4f,0x5a,0x16,0x6e,0xd7,0x2f,0x5b,0xf6,0x07,0xad,0x33,
187 0xb4,0x8f,0x3b,0x3a,0x8b,0xd9,0x06,0x2b,0xed,0x3c,0x3c,0x76,0xf6,0x21,0x31,0xe3,
188 0xfb,0x2c,0x45,0x61,0x42,0xba,0xe0,0xc3,0x72,0x63,0xd0,0x6b,0x8f,0x36,0x26,0xfb,
189 0x9e,0x89,0x0e,0x44,0x9a,0xc1,0x84,0x5e,0x84,0x8d,0xb6,0xea,0xf1,0x0d,0x66,0xc7,
190 0xdb,0x44,0xbd,0x19,0x7c,0x05,0xbe,0xc4,0xab,0x88,0x32,0xbe,0xc7,0x63,0x31,0xe6,
191 0x38,0xd4,0xe5,0xb8,0x4b,0xf5,0x0e,0x55,0x9a,0x3a,0xe6,0x0a,0xec,0xee,0xe2,0xa8,
192 0x88,0x04,0xf2,0xb8,0xaa,0x5a,0xd8,0x97,0x5d,0xa0,0xa8,0x42,0xfb,0xd9,0xde,0x80,
193 0xae,0x4c,0xb3,0xa1,0x90,0x47,0x57,0x03,0x10,0x78,0xa6,0x8f,0x11,0xba,0x4b,0xce,

```



```
194 0x2d,0x56,0xa4,0xe1,0xbd,0xf8,0xa0,0xa4,0xd5,0x48,0x3c,0x63,0x20,0x00,0x38,0xa0,
195 0xd1,0xe6,0x12,0xe9,0x1d,0xd8,0x49,0xe3,0xd5,0x24,0xb5,0xc5,0x3a,0x1f,0xb0,0xd4}};
196
197 const TPM2B_RSA_TEST_VALUE c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
198 0x74,0x89,0x29,0x3e,0x1b,0xac,0xc6,0x85,0xca,0xf0,0x63,0x43,0x30,0x7d,0x1c,0x9b,
199 0x2f,0xbd,0x4d,0x69,0x39,0x5e,0x85,0xe2,0xef,0x86,0x0a,0xc6,0x6b,0xa6,0x08,0x19,
200 0x6c,0x56,0x38,0x24,0x55,0x92,0x84,0x9b,0x1b,0x8b,0x04,0xcf,0x24,0x14,0x24,0x13,
201 0x0e,0x8b,0x82,0x6f,0x96,0xc8,0x9a,0x68,0xfc,0x4c,0x02,0xf0,0xdc,0xcd,0x36,0x25,
202 0x31,0xd5,0x82,0xcf,0xc9,0x69,0x72,0xf6,0x1d,0xab,0x68,0x20,0x2e,0x2d,0x19,0x49,
203 0xf0,0x2e,0xad,0xd2,0xda,0xaf,0xff,0xb6,0x92,0x83,0x5b,0x8a,0x06,0x2d,0xc0,0x32,
204 0x11,0x32,0x3b,0x77,0x17,0xf6,0x50,0xfb,0xf8,0x57,0xc9,0xc7,0x9b,0x9e,0xc6,0xd1,
205 0xa9,0x55,0xf0,0x22,0x35,0xda,0xca,0x3c,0x8e,0xc6,0x9a,0xd8,0x25,0xc8,0x5e,0x93,
206 0x0d,0xaa,0xa7,0x06,0xaf,0x11,0x29,0x99,0xe7,0x7c,0xee,0x49,0x82,0x30,0xba,0x2c,
207 0xe2,0x40,0x8f,0x0a,0xa6,0x7b,0x24,0x75,0xc5,0xcd,0x03,0x12,0xf4,0xb2,0x4b,0x3a,
208 0xd1,0x91,0x3c,0x20,0x0e,0x58,0x2b,0x31,0xf8,0x8b,0xee,0xbc,0x1f,0x95,0x35,0x58,
209 0x6a,0x73,0xee,0x99,0xb0,0x01,0x42,0x4f,0x66,0xc0,0x66,0xbb,0x35,0x86,0xeb,0xd9,
210 0x7b,0x55,0x77,0x9d,0x54,0x78,0x19,0x49,0xe8,0xcc,0xfd,0xb1,0xcb,0x49,0xc9,0xea,
211 0x20,0xab,0xed,0xb5,0xed,0xfe,0xb2,0xb5,0xa8,0xcf,0x05,0x06,0xd5,0x7d,0x2b,0xbb,
212 0x0b,0x65,0x6b,0x2b,0x6d,0x55,0x95,0x85,0x44,0x8b,0x12,0x05,0xf3,0x4b,0xd4,0x8e,
213 0x3d,0x68,0x2d,0x29,0x9c,0x05,0x79,0xd6,0xfc,0x72,0x90,0x6a,0xab,0x46,0x38,0x81}};
214
215 const TPM2B_RSA_TEST_VALUE c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
216 0x8a,0xb1,0x0a,0xb5,0xe4,0x02,0xf7,0xdd,0x45,0x2a,0xcc,0x2b,0x6b,0x8c,0x0e,0x9a,
217 0x92,0x4f,0x9b,0xc5,0xe4,0x8b,0x82,0xb9,0xb0,0xd9,0x87,0x8c,0xcb,0xf0,0xb0,0x59,
218 0xa5,0x92,0x21,0xa0,0xa7,0x61,0x5c,0xed,0xa8,0x6e,0x22,0x29,0x46,0xc7,0x86,0x37,
219 0x4b,0x1b,0x1e,0x94,0x93,0xc8,0x4c,0x17,0x7a,0xae,0x59,0x91,0xf8,0x83,0x84,0xc4,
220 0x8c,0x38,0xc2,0x35,0x0e,0x7e,0x50,0x67,0x76,0xe7,0xd3,0xec,0x6f,0x0d,0xa0,0x5c,
221 0x2f,0x0a,0x80,0x28,0xd3,0xc5,0x7d,0x2d,0x1a,0x0b,0x96,0xd6,0xe5,0x98,0x05,0x8c,
222 0x4d,0xa0,0x1f,0x8c,0xb6,0xfb,0xb1,0xcf,0xe9,0xcb,0x38,0x27,0x60,0x64,0x17,0xca,
223 0xf4,0x8b,0x61,0xb7,0x1d,0xb6,0x20,0x9d,0x40,0x2a,0x1c,0xfd,0x55,0x40,0x4b,0x95,
224 0x39,0x52,0x18,0x3b,0xab,0x44,0xe8,0x83,0x4b,0x7c,0x47,0xfb,0xed,0x06,0x9c,0xcd,
225 0x4f,0xba,0x81,0xd6,0xb7,0x31,0xcf,0x5c,0x23,0xf8,0x25,0xab,0x95,0x77,0x0a,0x8f,
226 0x46,0xef,0xfb,0x59,0xb8,0x04,0xd7,0x1e,0xf5,0xaf,0x6a,0x1a,0x26,0x9b,0xae,0xf4,
227 0xf5,0x7f,0x84,0x6f,0x3c,0xed,0xf8,0x24,0x0b,0x43,0xd1,0xba,0x74,0x89,0x4e,0x39,
228 0xfe,0xab,0xa5,0x16,0xa5,0x28,0xee,0x96,0x84,0x3e,0x16,0x6d,0x5f,0x4e,0x0b,0x7d,
229 0x94,0x16,0x1b,0x8c,0xf9,0xaa,0x9b,0xc0,0x49,0x02,0x4c,0x3e,0x62,0xff,0xfe,0xa2,
230 0x20,0x33,0x5e,0xa6,0xdd,0xda,0x15,0x2d,0xb7,0xcd,0xda,0xff,0xb1,0x0b,0x45,0x7b,
231 0xd3,0xa0,0x42,0x29,0xab,0xa9,0x73,0xe9,0xa4,0xd9,0x8d,0xac,0xa1,0x88,0x2c,0x2d}};
232
233 #endif // SHA256
234
235 #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
236
237 const TPM2B_RSA_TEST_VALUE c_OaepKvt = {RSA_TEST_KEY_SIZE, {
238 0x0f,0x3c,0x42,0x4d,0x8c,0x91,0x96,0x05,0x3c,0xfd,0x59,0x3b,0x7f,0x29,0xbc,0x03,
239 0x67,0xc1,0xff,0x74,0xe7,0x09,0xf4,0x13,0x45,0xbe,0x13,0x1d,0xc9,0x86,0x94,0xfe,
240 0xed,0xa6,0xe8,0x3a,0xcb,0x89,0x4d,0xec,0x86,0x63,0x4c,0xdb,0xf1,0x95,0xee,0xc1,
241 0x46,0xc5,0x3b,0xd8,0xf8,0xa2,0x41,0x6a,0x60,0x8b,0x9e,0x5e,0x7f,0x20,0x16,0xe3,
242 0x69,0xb6,0x2d,0x92,0xfc,0x60,0xa2,0x74,0x88,0xd5,0xc7,0xa6,0xd1,0xff,0xe3,0x45,
243 0x02,0x51,0x39,0xd9,0xf3,0x56,0x0b,0x91,0x80,0xe0,0x6c,0xa8,0xc3,0x78,0xef,0x34,
244 0x22,0x8c,0xf5,0xfb,0x47,0x98,0x5d,0x57,0x8e,0x3a,0xb9,0xff,0x92,0x04,0xc7,0xc2,
245 0x6e,0xfa,0x14,0xc1,0xb9,0x68,0x15,0x5c,0x12,0xe8,0xa8,0xbe,0xea,0xe8,0x8d,0x9b,
246 0x48,0x28,0x35,0xdb,0x4b,0x52,0xc1,0x2d,0x85,0x47,0x83,0xd0,0xe9,0xae,0x90,0x6e,
247 0x65,0xd4,0x34,0x7f,0x81,0xce,0x69,0xf0,0x96,0x62,0xf7,0xec,0x41,0xd5,0xc2,0xe3,
248 0x4b,0xba,0x9c,0x8a,0x02,0xce,0xf0,0x5d,0x14,0xf7,0x09,0x42,0x8e,0x4a,0x27,0xfe,
249 0x3e,0x66,0x42,0x99,0x03,0xe1,0x69,0xbd,0xdb,0x7f,0x9b,0x70,0xeb,0x4e,0x9c,0xac,
250 0x45,0x67,0x91,0x9f,0x75,0x10,0xc6,0xfc,0x14,0xe1,0x28,0xc1,0xe0,0xe0,0x7e,0xc0,
251 0x5c,0x1d,0xee,0xe8,0xff,0x45,0x79,0x51,0x86,0x08,0xe6,0x39,0xac,0xb5,0xfd,0xb8,
252 0xf1,0xdd,0x2e,0xf4,0xb2,0x1a,0x69,0x0d,0xd9,0x98,0x8e,0xdb,0x85,0x61,0x70,0x20,
253 0x82,0x91,0x26,0x87,0x80,0xc4,0x6a,0xd8,0x3b,0x91,0x4d,0xd3,0x33,0x84,0xad,0xb7}};
254
255 const TPM2B_RSA_TEST_VALUE c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
256 0x44,0xd5,0x9f,0xbc,0x48,0x03,0x3d,0x9f,0x22,0x91,0x2a,0xab,0x3c,0x31,0x71,0xab,
257 0x86,0x3f,0x0f,0x6f,0x59,0x5b,0x93,0x27,0xbc,0xbc,0xcd,0x29,0x38,0x43,0x2a,0x3b,
258 0x3b,0xd2,0xb3,0x45,0x40,0xba,0x15,0xb4,0x45,0xe3,0x56,0xab,0xff,0xb3,0x20,0x26,
259 0x39,0xcc,0x48,0xc5,0x5d,0x41,0x0d,0x2f,0x57,0x7f,0x9d,0x16,0x2e,0x26,0x57,0xc7,
```

```

260     0x6b,0xf3,0x36,0x54,0xbd,0xb6,0x1d,0x46,0x4e,0x13,0x50,0xd7,0x61,0x9d,0x8d,0x7b,
261     0xeb,0x21,0x9f,0x79,0xf3,0xfd,0xe0,0x1b,0xa8,0xed,0x6d,0x29,0x33,0x0d,0x65,0x94,
262     0x24,0x1e,0x62,0x88,0x6b,0x2b,0x4e,0x39,0xf5,0x80,0x39,0xca,0x76,0x95,0xbc,0x7c,
263     0x27,0x1d,0xdd,0x3a,0x11,0xf1,0x3e,0x54,0x03,0xb7,0x43,0x91,0x99,0x33,0xfe,0x9d,
264     0x14,0x2c,0x87,0x9a,0x95,0x18,0x1f,0x02,0x04,0x6a,0xe2,0xb7,0x81,0x14,0x13,0x45,
265     0x16,0xfb,0xe4,0xb7,0x8f,0xab,0x2b,0xd7,0x60,0x34,0x8a,0x55,0xbc,0x01,0x8c,0x49,
266     0x02,0x29,0xf1,0x9c,0x94,0x98,0x44,0xd0,0x94,0xcb,0xd4,0x85,0x4c,0x3b,0x77,0x72,
267     0x99,0xd5,0x4b,0xc6,0x3b,0xe4,0xd2,0xc8,0xe9,0x6a,0x23,0x18,0x3b,0x3b,0x5e,0x32,
268     0xec,0x70,0x84,0x5d,0xbb,0x6a,0x8f,0x0c,0x5f,0x55,0xa5,0x30,0x34,0x48,0xbb,0xc2,
269     0xdf,0x12,0xb9,0xad,0x36,0x3f,0xf0,0x24,0x16,0x48,0x04,0x4a,0x7f,0xfd,0x9f,
270     0x4c,0xea,0xfe,0x1d,0x83,0xd0,0x81,0xad,0x25,0x6c,0x5f,0x45,0x36,0x91,0xf0,0xd5,
271     0x8b,0x53,0x0a,0xdf,0xec,0x9f,0x04,0x58,0xc4,0x35,0xa0,0x78,0x1f,0x68,0xe0,0x22}};
272
273     const TPM2B_RSA_TEST_VALUE      c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
274         0x3f,0x3a,0x82,0x6d,0x42,0xe3,0x8b,0x4f,0x45,0x9c,0xda,0x6c,0xbe,0xbe,0xcd,0x00,
275         0x98,0xfb,0xbe,0x59,0x30,0xc6,0x3c,0xaa,0xb3,0x06,0x27,0xb5,0xda,0xfa,0xb2,0xc3,
276         0x43,0xb7,0xbd,0xe9,0xd3,0x23,0xed,0x80,0xce,0x74,0xb3,0xb8,0x77,0x8d,0xe6,0xd3,
277         0x3c,0xe5,0xf5,0xd7,0x80,0xcf,0x38,0x55,0x76,0xd7,0x87,0xa8,0xd6,0x3a,0xcf,0xfd,
278         0xd8,0x91,0x65,0xab,0x43,0x66,0x50,0xb7,0x9a,0x13,0x6b,0x45,0x80,0x76,0x86,0x22,
279         0x27,0x72,0xf7,0xbb,0x65,0x22,0x5c,0x55,0x60,0xd8,0x84,0x9f,0xf2,0x61,0x52,0xac,
280         0xf2,0x4f,0x5b,0x7b,0x21,0xe1,0xf5,0x4b,0x8f,0x01,0xf2,0x4b,0xcf,0xd3,0xfb,0x74,
281         0x5e,0x6e,0x96,0xb4,0xa8,0x0f,0x01,0x9b,0x26,0x54,0x0a,0x70,0x55,0x26,0xb7,0x0b,
282         0xe8,0x01,0x68,0x66,0x0d,0x6f,0xb5,0xfc,0x66,0xbd,0x9e,0x44,0xed,0x6a,0x1e,0x3c,
283         0x3b,0x61,0x5d,0xe8,0xdb,0x99,0x5b,0x67,0xbf,0x94,0xfb,0xe6,0x8c,0x4b,0x07,0xc3,
284         0x43,0x3a,0x0d,0xb1,0x1b,0x10,0x66,0x81,0xe2,0x0d,0xe7,0xd1,0xca,0x85,0xa7,0x50,
285         0x82,0x2d,0xbf,0xed,0xcf,0x43,0x6d,0xdb,0x2c,0x7b,0x73,0x20,0xfe,0x73,0x3f,0x19,
286         0xc6,0xdb,0x69,0xb8,0xc3,0xd3,0xf4,0xe5,0x64,0xf8,0x36,0x8e,0xd5,0xd8,0x09,0x2a,
287         0x5f,0x26,0x70,0xa1,0xd9,0x5b,0x14,0xf8,0x22,0xe9,0x9d,0x22,0x51,0xf4,0x52,0xc1,
288         0x6f,0x53,0xf5,0xca,0x0d,0xda,0x39,0x8c,0x29,0x42,0xe8,0x58,0x89,0xbb,0xd1,0x2e,
289         0xc5,0xdb,0x86,0x8d,0xaf,0xec,0x58,0x36,0x8d,0x8d,0x57,0x23,0xd5,0xdd,0xb9,0x24}};
290
291     const TPM2B_RSA_TEST_VALUE      c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
292         0x39,0x10,0x58,0x7d,0x6d,0xa8,0xd5,0x90,0x07,0xd6,0x2b,0x13,0xe9,0xd8,0x93,0x7e,
293         0xf3,0x5d,0x71,0xe0,0xf0,0x33,0x3a,0x4a,0x22,0xf3,0xe6,0x95,0xd3,0x8e,0x8c,0x41,
294         0xe7,0xb3,0x13,0xde,0x4a,0x45,0xd3,0xd1,0xfb,0xb1,0x3f,0x9b,0x39,0xa5,0x50,0x58,
295         0xef,0xb6,0x3a,0x43,0xdd,0x54,0xab,0xda,0x9d,0x32,0x49,0xe4,0x57,0x96,0xe5,0x1b,
296         0x1d,0x8f,0x33,0x8e,0x07,0x67,0x56,0x14,0xc1,0x18,0x78,0xa2,0x52,0xe6,0x2e,0x07,
297         0x81,0xbe,0xd8,0xca,0x76,0x63,0x68,0xc5,0x47,0xa2,0x92,0x5e,0x4c,0xfd,0x14,0xc7,
298         0x46,0x14,0xbe,0xc7,0x85,0xef,0xe6,0xb8,0x46,0xcb,0x3a,0x67,0x66,0x89,0xc6,0xee,
299         0x9d,0x64,0xf5,0x0d,0x09,0x80,0x9a,0x6f,0x0e,0xeb,0xe4,0xb9,0xe9,0xab,0x90,0x4f,
300         0xe7,0x5a,0xc8,0xca,0xf6,0x16,0x0a,0x82,0xbd,0xb7,0x76,0x59,0x08,0x2d,0xd9,0x40,
301         0x5d,0xaa,0xa5,0xef,0xfb,0xe3,0x81,0x2c,0x2c,0x5c,0xa8,0x16,0xbd,0x63,0x20,0xc2,
302         0x4d,0x3b,0x51,0xaa,0x62,0x1f,0x06,0xe5,0xbb,0x78,0x44,0x04,0x0c,0x5c,0xe1,0x1b,
303         0x6b,0x9d,0x21,0x10,0xaf,0x48,0x48,0x98,0x97,0x77,0xc2,0x73,0xb4,0x98,0x64,0xcc,
304         0x94,0x2c,0x29,0x28,0x45,0x36,0xd1,0xc5,0xd0,0x2f,0x97,0x27,0x92,0x65,0x22,0xbb,
305         0x63,0x79,0xea,0xf5,0xff,0x77,0x0f,0x4b,0x56,0x8a,0x9f,0xad,0x1a,0x97,0x67,0x39,
306         0x69,0xb8,0x4c,0x6c,0xc2,0x56,0xc5,0x7a,0xa8,0x14,0x5a,0x24,0x7a,0xa4,0x6e,0x55,
307         0xb2,0x86,0x1d,0xf4,0x62,0x5a,0x2d,0x87,0x6d,0xde,0x99,0x78,0x2d,0xef,0xd7,0xdc}};
308
309     #endif // SHA384
310
311     #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
312
313     const TPM2B_RSA_TEST_VALUE      c_OaepKvt = {RSA_TEST_KEY_SIZE, {
314         0x48,0x45,0xa7,0x70,0xb2,0x41,0xb7,0x48,0x5e,0x79,0x8c,0xdf,0x1c,0xc6,0x7e,0xbb,
315         0x11,0x80,0x82,0x52,0xbf,0x40,0x3d,0x90,0x03,0x6e,0x20,0x3a,0xb9,0x65,0xc8,0x51,
316         0x4c,0xbd,0x9c,0xa9,0x43,0x89,0xd0,0x57,0x0c,0xa3,0x69,0x22,0x7e,0x82,0x2a,0x1c,
317         0x1d,0x5a,0x80,0x84,0x81,0xbb,0x5e,0x5e,0xd0,0xc1,0x66,0x9a,0xac,0x00,0xba,0x14,
318         0xa2,0xe9,0xd0,0x3a,0x89,0x5a,0x63,0xe2,0xec,0x92,0x05,0xf4,0x47,0x66,0x12,0x7f,
319         0xdb,0xa7,0x3c,0x5b,0x67,0xe1,0x55,0xca,0x0a,0x27,0xbf,0x39,0x89,0x11,0x05,0xba,
320         0x9b,0x5a,0x9b,0x65,0x44,0xad,0x78,0xcf,0x8f,0x94,0xf6,0x9a,0xb4,0x52,0x39,0x0e,
321         0x00,0xba,0xbc,0xe0,0xbd,0x6f,0x81,0x2d,0x76,0x42,0x66,0x70,0x07,0x77,0xbf,0x09,
322         0x88,0x2a,0x0c,0xb1,0x56,0x3e,0xee,0xfd,0xdc,0xb6,0x3c,0x0d,0xc5,0xa4,0x0d,0x10,
323         0x32,0x80,0x3e,0x1e,0xfe,0x36,0x8f,0xb5,0x42,0xc1,0x21,0x7b,0xdf,0x4a,0xd2,
324         0x68,0x0c,0x01,0x9f,0x4a,0xfd,0xd4,0xec,0xf7,0x49,0x06,0xab,0xed,0xc6,0xd5,0x1b,
325         0x63,0x76,0x38,0xc8,0x6c,0xc7,0x4f,0xcb,0x29,0x8a,0x0e,0x6f,0x33,0xaf,0x69,0x31,

```

```

326     0x8e,0xa7,0xdd,0x9a,0x36,0xde,0x9b,0xf1,0x0b,0xfb,0x20,0xa0,0x6d,0x33,0x31,0xc9,
327     0x9e,0xb4,0x2e,0xc5,0x40,0x0e,0x60,0x71,0x36,0x75,0x05,0xf9,0x37,0xe0,0xca,0x8e,
328     0x8f,0x56,0xe0,0xea,0x9b,0xeb,0x17,0xf3,0xca,0x40,0xc3,0x48,0x01,0xba,0xdc,0xc6,
329     0x4b,0x2b,0x5b,0x7b,0x5c,0x81,0xa6,0xbb,0xc7,0x43,0xc0,0xbe,0xc0,0x30,0x7b,0x55}};
330
331     const TPM2B_RSA_TEST_VALUE      c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
332         0x74,0x83,0xfa,0x52,0x65,0x50,0x68,0xd0,0x82,0x05,0x72,0x70,0x78,0x1c,0xac,0x10,
333         0x23,0xc5,0x07,0xf8,0x93,0xd2,0xeb,0x65,0x87,0xbb,0x47,0xc2,0xfb,0x30,0x9e,0x61,
334         0x4c,0xac,0x04,0x57,0x5a,0x7c,0xeb,0x29,0x08,0x84,0x86,0x89,0x1e,0x8f,0x07,0x32,
335         0xa3,0x8b,0x70,0xe7,0xa2,0x9f,0x9c,0x42,0x71,0x3d,0x23,0x59,0x82,0x5e,0x8a,0xde,
336         0xd6,0xfb,0xd8,0xc5,0x8b,0xc0,0xdb,0x10,0x38,0x87,0xd3,0xbf,0x04,0xb0,0x66,0xb9,
337         0x85,0x81,0x54,0x4c,0x69,0xdc,0xba,0x78,0xf3,0x4a,0xdb,0x25,0xa2,0xf2,0x34,0x55,
338         0xdd,0xaa,0xa5,0xc4,0xed,0x55,0x06,0x0e,0x2a,0x30,0x77,0xab,0x82,0x79,0xf0,0xcd,
339         0x9d,0x6f,0x09,0xa0,0xc8,0x82,0xc9,0xe0,0x61,0xda,0x40,0xcd,0x17,0x59,0xc0,0xef,
340         0x95,0x6d,0xa3,0x6d,0x1c,0x2b,0xee,0x24,0xef,0xd8,0x4a,0x55,0x6c,0xd6,0x26,0x42,
341         0x32,0x17,0xfd,0x6a,0xb3,0x4f,0xde,0x07,0x2f,0x10,0xd4,0xac,0x14,0xea,0x89,0x68,
342         0xcc,0xd3,0x70,0xb7,0xcf,0x39,0x20,0x63,0x20,0x7b,0x44,0x8b,0x48,0x60,0xd5,
343         0x3a,0x2a,0x0a,0xe9,0x68,0xab,0x15,0x46,0x27,0x64,0xb5,0x82,0x06,0x29,0xe7,0x25,
344         0xca,0x46,0x48,0x6e,0x2a,0x34,0x57,0x4b,0x81,0x75,0xae,0xb6,0xfd,0x6f,0x51,0x5f,
345         0x04,0x59,0xc7,0x15,0x1f,0xe0,0x68,0xf7,0x36,0x2d,0xdf,0xc8,0x9d,0x05,0x27,0x2d,
346         0x3f,0x2b,0x59,0x5d,0xcb,0xf3,0xc4,0x92,0x6e,0x00,0xa8,0x8d,0xd0,0x69,0xe5,0x59,
347         0xda,0xba,0x4f,0x38,0xf5,0xa0,0x8b,0xf1,0x73,0xe9,0x0d,0xee,0x64,0xe5,0xa2,0xd8}};
348
349     const TPM2B_RSA_TEST_VALUE      c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
350         0x1b,0xca,0x8b,0x18,0x15,0x3b,0x95,0x5b,0x0a,0x89,0x10,0x03,0x7f,0x7c,0xa0,0xc9,
351         0x66,0x57,0x86,0x6a,0xc9,0xeb,0x82,0x71,0xf3,0x8d,0x6f,0xa9,0xa4,0x2d,0xd0,0x22,
352         0xdf,0xe9,0xc6,0x71,0x5b,0xf4,0x27,0x38,0x5b,0x2c,0x8a,0x54,0xcc,0x85,0x11,0x69,
353         0x6d,0x6f,0x42,0xe7,0x22,0xcb,0xd6,0xad,0x1a,0xc5,0xab,0x6a,0xa5,0xfc,0xa5,0x70,
354         0x72,0x4a,0x62,0x25,0xd0,0xa2,0x16,0x61,0xab,0xac,0x31,0xa0,0x46,0x24,0x4f,0xdd,
355         0x9a,0x36,0x55,0xb6,0x00,0x9e,0x23,0x50,0x0d,0x53,0x01,0xb3,0x46,0x56,0xb2,0x1d,
356         0x33,0x5b,0xca,0x41,0x7f,0x65,0x7e,0x00,0x5c,0x12,0xff,0x0a,0x70,0x5d,0x8c,0x69,
357         0x4a,0x02,0xee,0x72,0x30,0xa7,0x5c,0xa4,0xbb,0xbe,0x03,0x0c,0xe4,0x5f,0x33,0xb6,
358         0x78,0x91,0x9d,0xd8,0xec,0x34,0x03,0x2e,0x63,0x32,0xc7,0x2a,0x36,0x50,0xd5,0x8b,
359         0xe0,0x7f,0x54,0x4e,0xf4,0x29,0x11,0x1b,0xcd,0x0f,0x37,0xa5,0xbc,0x61,0x83,0x50,
360         0xfa,0x18,0x75,0xd9,0xfe,0xa7,0xe8,0x9b,0xc1,0x4f,0x96,0x37,0x81,0x71,0xdf,0x71,
361         0x8b,0x89,0x81,0xf4,0x95,0xb5,0x29,0x66,0x41,0x0c,0x73,0xd7,0x0b,0x21,0xb4,0xfb,
362         0xf9,0x63,0x2f,0xe9,0x7b,0x38,0xaa,0x20,0xc3,0x96,0xc0,0xb7,0xb2,0x24,0xa1,0xe0,
363         0x59,0x9c,0x10,0x9e,0x5a,0xf7,0xe3,0x02,0xe6,0x23,0xe2,0x44,0x21,0x3f,0x6e,0x5e,
364         0x79,0xb2,0x93,0x7d,0xce,0xed,0xe2,0xe1,0xab,0x98,0x07,0xa7,0xbd,0xbc,0xd8,0xf7,
365         0x06,0xeb,0xc5,0xa6,0x37,0x18,0x11,0x88,0xf7,0x63,0x39,0xb9,0x57,0x29,0xdc,0x03}};
366
367     const TPM2B_RSA_TEST_VALUE      c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
368         0x05,0x55,0x00,0x62,0x01,0xc6,0x04,0x31,0x55,0x73,0x3f,0x2a,0xf9,0xd4,0x0f,0xc1,
369         0x2b,0xeb,0xd8,0xc8,0xdb,0xb2,0xab,0x6c,0x26,0xde,0x2d,0x89,0xc2,0x2d,0x36,0x62,
370         0xc8,0x22,0x5d,0x58,0x03,0xb1,0x46,0x14,0xa5,0xd4,0xbc,0x25,0x6b,0x7f,0x8f,0x14,
371         0x7e,0x03,0x2f,0x3d,0xb8,0x39,0xa5,0x79,0x13,0x7e,0x22,0x2a,0xb9,0x3e,0x8f,0xaa,
372         0x01,0x7c,0x03,0x12,0x21,0x6c,0x2a,0xb4,0x39,0x98,0x6d,0xff,0x08,0x6c,0x59,0x2d,
373         0xdc,0xc6,0xf1,0x77,0x62,0x10,0xa6,0xcc,0xe2,0x71,0x8e,0x97,0x00,0x87,0x5b,0x0e,
374         0x20,0x00,0x3f,0x18,0x63,0x83,0xf0,0xe4,0x0a,0x64,0x8c,0xe9,0x8c,0x91,0xe7,0x89,
375         0x04,0x64,0x2c,0x8b,0x41,0xc8,0xac,0xf6,0x5a,0x75,0xe6,0xa5,0x76,0x43,0xcb,0xa5,
376         0x33,0x8b,0x07,0xc9,0x73,0x0f,0x45,0xa4,0xc3,0xac,0xc1,0xc3,0xe6,0xe7,0x21,0x66,
377         0x1c,0xba,0xbf,0xea,0x3e,0x39,0xfa,0xb2,0xe2,0x8f,0xfe,0x9c,0xb4,0x85,0x89,0x33,
378         0x2a,0x0c,0xc8,0x5d,0x58,0xe1,0x89,0x12,0xe9,0x4d,0x42,0xb3,0x1f,0x99,0x0c,0x3e,
379         0xd8,0xb2,0xeb,0xf5,0x88,0xfb,0xe1,0x4b,0x8e,0xdc,0xd3,0xa8,0xda,0xbe,0x04,0x45,
380         0xbf,0x56,0xc6,0x54,0x70,0x00,0xb8,0x66,0x46,0x3a,0xa3,0x1e,0xb6,0xeb,0x1a,0xa0,
381         0x0b,0xd3,0x9a,0x9a,0x52,0xda,0x60,0x69,0xb7,0xef,0x93,0x47,0x38,0xab,0x1a,0xa0,
382         0x22,0x6e,0x76,0x06,0xb6,0x74,0xaf,0x74,0x8f,0x51,0xc0,0x89,0x5a,0x4b,0xbe,0x6a,
383         0x91,0x18,0x25,0x7d,0xa6,0x77,0xe6,0xfd,0xc2,0x62,0x36,0x07,0xc6,0xef,0x79,0xc9}};
384
385     #endif // SHA512

```

10.1.10 SelfTest.h

10.1.10.1 Introduction

This file contains the structure definitions for the self-test. It also contains macros for use when the self-test is implemented.

```
1  #ifndef          _SELF_TEST_H_
2  #define          _SELF_TEST_H_
```

10.1.10.2 Defines

Was typing this a lot

```
3  #define SELF_TEST_FAILURE    FAIL(FATAL_ERROR_SELF_TEST)
```

Use the definition of key sizes to set algorithm values for key size.

```
4  #define AES_ENTRIES  (AES_128 + AES_192 + AES_256)
5  #define SM4_ENTRIES  (SM4_128)
6  #define CAMELLIA_ENTRIES  (CAMELLIA_128 + CAMELLIA_192 + CAMELLIA_256)
7  #define TDES_ENTRIES  (TDES_128 + TDES_192)
8
9  #define NUM_SYMS      (AES_ENTRIES + SM4_ENTRIES + CAMELLIA_ENTRIES + TDES_ENTRIES)
10
11 typedef UINT32        SYM_INDEX;
```

These two defines deal with the fact that the TPM_ALG_ID table does not delimit the symmetric mode values with a SYM_MODE_FIRST and SYM_MODE_LAST

```
12 #define SYM_MODE_FIRST    ALG_CTR_VALUE
13 #define SYM_MODE_LAST    ALG_ECB_VALUE
14
15 #define NUM_SYM_MODES    (SYM_MODE_LAST - SYM_MODE_FIRST + 1)
```

Define a type to hold a bit vector for the modes.

```
16 #if NUM_SYM_MODES <= 0
17 #error "No symmetric modes implemented"
18 #elif NUM_SYM_MODES <= 8
19 typedef BYTE        SYM_MODES;
20 #elif NUM_SYM_MODES <= 16
21 typedef UINT16      SYM_MODES;
22 #elif NUM_SYM_MODES <= 32
23 typedef UINT32      SYM_MODES;
24 #else
25 #error "Too many symmetric modes"
26 #endif
27
28 typedef struct SYMMETRIC_TEST_VECTOR {
29     const TPM_ALG_ID    alg;                // the algorithm
30     const UINT16        keyBits;           // bits in the key
31     const BYTE          *key;              // The test key
32     const UINT32        ivSize;            // block size of the algorithm
33     const UINT32        dataInOutSize;     // size to encrypt/decrypt
34     const BYTE          *dataIn;          // data to encrypt
35     const BYTE          *dataOut[NUM_SYM_MODES]; // data to decrypt
36 } SYMMETRIC_TEST_VECTOR;
37
38 #if ALG_SHA512
39 #define DEFAULT_TEST_HASH    ALG_SHA512_VALUE
```

```

40 # define DEFAULT_TEST_DIGEST_SIZE SHA512_DIGEST_SIZE
41 # define DEFAULT_TEST_HASH_BLOCK_SIZE SHA512_BLOCK_SIZE
42 #elif ALG_SHA384
43 # define DEFAULT_TEST_HASH ALG_SHA384_VALUE
44 # define DEFAULT_TEST_DIGEST_SIZE SHA384_DIGEST_SIZE
45 # define DEFAULT_TEST_HASH_BLOCK_SIZE SHA384_BLOCK_SIZE
46 #elif ALG_SHA256
47 # define DEFAULT_TEST_HASH ALG_SHA256_VALUE
48 # define DEFAULT_TEST_DIGEST_SIZE SHA256_DIGEST_SIZE
49 # define DEFAULT_TEST_HASH_BLOCK_SIZE SHA256_BLOCK_SIZE
50 #elif ALG_SHA1
51 # define DEFAULT_TEST_HASH ALG_SHA1_VALUE
52 # define DEFAULT_TEST_DIGEST_SIZE SHA1_DIGEST_SIZE
53 # define DEFAULT_TEST_HASH_BLOCK_SIZE SHA1_BLOCK_SIZE
54 #endif
55
56 #endif // _SELF_TEST_H_

```

10.1.11 SupportLibraryFunctionPrototypes_fp.h

10.1.11.1 Introduction

This file contains the function prototypes for the functions that need to be present in the selected math library. For each function listed, there should be a small stub function. That stub provides the interface between the TPM code and the support library. In most cases, the stub function will only need to do a format conversion between the TPM big number and the support library big number. The TPM big number format was chosen to make this relatively simple and fast.

Arithmetic operations return a BOOL to indicate if the operation completed successfully or not.

```

1 #ifndef SUPPORT_LIBRARY_FUNCTION_PROTOTYPES_H
2 #define SUPPORT_LIBRARY_FUNCTION_PROTOTYPES_H

```

10.1.11.2 SupportLibInit()

This function is called by CryptInit() so that necessary initializations can be performed on the cryptographic library.

```

3 LIB_EXPORT
4 int SupportLibInit(void);

```

10.1.11.3 MathLibraryCompatibilityCheck()

This function is only used during development to make sure that the library that is being referenced is using the same size of data structures as the TPM.

```

5 BOOL
6 MathLibraryCompatibilityCheck(
7     void
8 );

```

10.1.11.4 BnModMult()

Does $op1 * op2$ and divide by *modulus* returning the remainder of the divide.

```

9 LIB_EXPORT BOOL
10 BnModMult(bigNum result, bigConst op1, bigConst op2, bigConst modulus);

```

10.1.11.5 BnMult()

Multiplies two numbers and returns the result

```
11  LIB_EXPORT BOOL
12  BnMult(bigNum result, bigConst multiplicand, bigConst multiplier);
```

10.1.11.6 BnDiv()

This function divides two *bigNum* values. The function returns FALSE if there is an error in the operation.

```
13  LIB_EXPORT BOOL
14  BnDiv(bigNum quotient, bigNum remainder,
15        bigConst dividend, bigConst divisor);
```

10.1.11.7 BnMod()

```
16  #define BnMod(a, b)      BnDiv(NULL, (a), (a), (b))
```

10.1.11.8 BnGcd()

Get the greatest common divisor of two numbers. This function is only needed when the TPM implements RSA.

```
17  LIB_EXPORT BOOL
18  BnGcd(bigNum gcd, bigConst number1, bigConst number2);
```

10.1.11.9 BnModExp()

Do modular exponentiation using *bigNum* values. This function is only needed when the TPM implements RSA.

```
19  LIB_EXPORT BOOL
20  BnModExp(bigNum result, bigConst number,
21           bigConst exponent, bigConst modulus);
```

10.1.11.10 BnModInverse()

Modular multiplicative inverse. This function is only needed when the TPM implements RSA.

```
22  LIB_EXPORT BOOL BnModInverse(bigNum result, bigConst number,
23                               bigConst modulus);
```

10.1.11.11 BnEccModMult()

This function does a point multiply of the form $R = [d]S$. A return of FALSE indicates that the result was the point at infinity. This function is only needed if the TPM supports ECC.

```
24  LIB_EXPORT BOOL
25  BnEccModMult(bigPoint R, pointConst S, bigConst d, bigCurve E);
```

10.1.11.12 BnEccModMult2()

This function does a point multiply of the form $R = [d]S + [u]Q$. A return of FALSE indicates that the result was the point at infinity. This function is only needed if the TPM supports ECC.

```
26  LIB_EXPORT BOOL
27  BnEccModMult2(bigPoint R, pointConst S, bigConst d,
28                pointConst Q, bigConst u, bigCurve E);
```

10.1.11.13 BnEccAdd()

This function does a point add $R = S + Q$. A return of FALSE indicates that the result was the point at infinity. This function is only needed if the TPM supports ECC.

```
29  LIB_EXPORT BOOL
30  BnEccAdd(bigPoint R, pointConst S, pointConst Q, bigCurve E);
```

10.1.11.14 BnCurveInitialize()

This function is used to initialize the pointers of a *bnCurve_t* structure. The structure is a set of pointers to *bigNum* values. The curve-dependent values are set by a different function. This function is only needed if the TPM supports ECC.

```
31  LIB_EXPORT bigCurve
32  BnCurveInitialize(bigCurve E, TPM_ECC_CURVE curveId);
```

10.1.11.14.1 BnCurveFree()

This function will free the allocated components of the curve and end the frame in which the curve data exists

```
33  LIB_EXPORT void
34  BnCurveFree(bigCurve E);
35
36  #endif
```


10.1.12 SymmetricTestData.h

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

```

1  #if AES_128
2
3  const BYTE key_AES128 [] = {
4      0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
5      0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
6
7  const BYTE dataIn_AES128 [] = {
8      0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
9      0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
10     0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
11     0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
12
13  const BYTE dataOut_AES128_ECB [] = {
14      0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60,
15      0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97,
16      0xf5, 0xd3, 0xd5, 0x85, 0x03, 0xb9, 0x69, 0x9d,
17      0xe7, 0x85, 0x89, 0x5a, 0x96, 0xfd, 0xba, 0xaf};
18
19  const BYTE dataOut_AES128_CBC [] = {
20      0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46,
21      0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
22      0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee,
23      0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2};
24
25  const BYTE dataOut_AES128_CFB [] = {
26      0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
27      0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
28      0xc8, 0xa6, 0x45, 0x37, 0xa0, 0xb3, 0xa9, 0x3f,
29      0xcd, 0xe3, 0xcd, 0xad, 0x9f, 0x1c, 0xe5, 0x8b};
30
31  const BYTE dataOut_AES128_OFB [] = {
32      0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
33      0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
34      0x77, 0x89, 0x50, 0x8d, 0x16, 0x91, 0x8f, 0x03,
35      0xf5, 0x3c, 0x52, 0xda, 0xc5, 0x4e, 0xd8, 0x25};
36
37  const BYTE dataOut_AES128_CTR [] = {
38      0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
39      0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
40      0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff,
41      0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd, 0xff};
42  #endif
43
44  #if AES_192
45
46  const BYTE key_AES192 [] = {
47      0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
48      0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
49      0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b};
50
51  const BYTE dataIn_AES192 [] = {
52      0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
53      0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
54      0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
55      0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
56
57  const BYTE dataOut_AES192_ECB [] = {

```

```

58         0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f,
59         0xf7, 0x12, 0xa2, 0x14, 0x57, 0x1f, 0xa5, 0xcc,
60         0x97, 0x41, 0x04, 0x84, 0x6d, 0x0a, 0xd3, 0xad,
61         0x77, 0x34, 0xec, 0xb3, 0xec, 0xee, 0x4e, 0xef};
62
63     const BYTE dataOut_AES192_CBC [] = {
64         0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d,
65         0x71, 0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
66         0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4,
67         0xe5, 0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a};
68
69     const BYTE dataOut_AES192_CFB [] = {
70         0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
71         0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
72         0x67, 0xce, 0x7f, 0x7f, 0x81, 0x17, 0x36, 0x21,
73         0x96, 0x1a, 0x2b, 0x70, 0x17, 0x1d, 0x3d, 0x7a};
74
75     const BYTE dataOut_AES192_OFB [] = {
76         0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
77         0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
78         0xfc, 0xc2, 0x8b, 0x8d, 0x4c, 0x63, 0x83, 0x7c,
79         0x09, 0xe8, 0x17, 0x00, 0xc1, 0x10, 0x04, 0x01};
80
81     const BYTE dataOut_AES192_CTR [] = {
82         0x1a, 0xbc, 0x93, 0x24, 0x17, 0x52, 0x1c, 0xa2,
83         0x4f, 0x2b, 0x04, 0x59, 0xfe, 0x7e, 0x6e, 0x0b,
84         0x09, 0x03, 0x39, 0xec, 0x0a, 0xa6, 0xfa, 0xef,
85         0xd5, 0xcc, 0xc2, 0xc6, 0xf4, 0xce, 0x8e, 0x94};
86 #endif
87
88 #if AES_256
89
90     const BYTE key_AES256 [] = {
91         0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
92         0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
93         0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
94         0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4};
95
96     const BYTE dataIn_AES256 [] = {
97         0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
98         0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
99         0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
100        0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
101
102     const BYTE dataOut_AES256_ECB [] = {
103         0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c,
104         0x06, 0x4b, 0x5a, 0x7e, 0x3d, 0xb1, 0x81, 0xf8,
105         0x59, 0x1c, 0xcb, 0x10, 0xd4, 0x10, 0xed, 0x26,
106         0xdc, 0x5b, 0xa7, 0x4a, 0x31, 0x36, 0x28, 0x70};
107
108     const BYTE dataOut_AES256_CBC [] = {
109         0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba,
110         0x77, 0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
111         0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d,
112         0x67, 0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d};
113
114     const BYTE dataOut_AES256_CFB [] = {
115         0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
116         0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
117         0x39, 0xff, 0xed, 0x14, 0x3b, 0x28, 0xb1, 0xc8,
118         0x32, 0x11, 0x3c, 0x63, 0x31, 0xe5, 0x40, 0x7b};
119
120     const BYTE dataOut_AES256_OFB [] = {
121         0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
122         0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
123         0x4f, 0xeb, 0xdc, 0x67, 0x40, 0xd2, 0x0b, 0x3a,

```

```
124         0xc8, 0x8f, 0x6a, 0xd8, 0x2a, 0x4f, 0xb0, 0x8d};
125
126     const BYTE dataOut_AES256_CTR [] = {
127         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
```

DRAFT

10.1.13 SymmetricTest.h

10.1.13.1 Introduction

This file contains the structures and data definitions for the symmetric tests. This file references the header file that contains the actual test vectors. This organization was chosen so that the program that is used to generate the test vector values does not have to also re-generate this data.

```

1  #ifndef      SELF_TEST_DATA
2  #error      "This file may only be included in AlgorithmTests.c"
3  #endif
4
5  #ifndef      _SYMMETRIC_TEST_H
6  #define      _SYMMETRIC_TEST_H
7  #include      "SymmetricTestData.h"

```

10.1.13.2 Symmetric Test Structures

```

8  const SYMMETRIC_TEST_VECTOR  c_symTestValues[NUM_SYMS + 1] = {
9  #if ALG_AES && AES_128
10     {TPM_ALG_AES, 128, key_AES128, 16, sizeof(dataIn_AES128), dataIn_AES128,
11     {dataOut_AES128_CTR, dataOut_AES128_OFB, dataOut_AES128_CBC,
12     dataOut_AES128_CFB, dataOut_AES128_ECB}},
13 #endif
14 #if ALG_AES && AES_192
15     {TPM_ALG_AES, 192, key_AES192, 16, sizeof(dataIn_AES192), dataIn_AES192,
16     {dataOut_AES192_CTR, dataOut_AES192_OFB, dataOut_AES192_CBC,
17     dataOut_AES192_CFB, dataOut_AES192_ECB}},
18 #endif
19 #if ALG_AES && AES_256
20     {TPM_ALG_AES, 256, key_AES256, 16, sizeof(dataIn_AES256), dataIn_AES256,
21     {dataOut_AES256_CTR, dataOut_AES256_OFB, dataOut_AES256_CBC,
22     dataOut_AES256_CFB, dataOut_AES256_ECB}},
23 #endif
24     // There are no SM4 test values yet so...
25 #if ALG_SM4 && SM4_128 && 0
26     {TPM_ALG_SM4, 128, key_SM4128, 16, sizeof(dataIn_SM4128), dataIn_SM4128,
27     {dataOut_SM4128_CTR, dataOut_SM4128_OFB, dataOut_SM4128_CBC,
28     dataOut_SM4128_CFB, dataOut_AES128_ECB}},
29 #endif
30     {0}
31 };
32
33 #endif // _SYMMETRIC_TEST_H

```

10.1.14 EccTestData.h

This file contains the parameter data for ECC testing.

```

1  #ifndef SELF_TEST_DATA
2
3  TPM2B_TYPE(EC_TEST, 32);
4  const TPM_ECC_CURVE      c_testCurve = 00003;

```

The **static** key

```

5  const TPM2B_EC_TEST      c_ecTestKey_ds = {{32, {
6      0xdf,0x8d,0xa4,0xa3,0x88,0xf6,0x76,0x96,0x89,0xfc,0x2f,0x2d,0xa1,0xb4,0x39,0x7a,
7      0x78,0xc4,0x7f,0x71,0x8c,0xa6,0x91,0x85,0xc0,0xbf,0xf3,0x54,0x20,0x91,0x2f,0x73}}};
8
9  const TPM2B_EC_TEST      c_ecTestKey_QsX = {{32, {
10     0x17,0xad,0x2f,0xcb,0x18,0xd4,0xdb,0x3f,0x2c,0x53,0x13,0x82,0x42,0x97,0xff,0x8d,
11     0x99,0x50,0x16,0x02,0x35,0xa7,0x06,0xae,0x1f,0xda,0xe2,0x9c,0x12,0x77,0xc0,0xf9}}};
12
13  const TPM2B_EC_TEST      c_ecTestKey_QsY = {{32, {
14     0xa6,0xca,0xf2,0x18,0x45,0x96,0x6e,0x58,0xe6,0x72,0x34,0x12,0x89,0xcd,0xaa,0xad,
15     0xcb,0x68,0xb2,0x51,0xdc,0x5e,0xd1,0x6d,0x38,0x20,0x35,0x57,0xb2,0xfd,0xc7,0x52}}};

```

The **ephemeral** key

```

16  const TPM2B_EC_TEST      c_ecTestKey_de = {{32, {
17     0xb6,0xb5,0x33,0x5c,0xd1,0xee,0x52,0x07,0x99,0xea,0x2e,0x8f,0x8b,0x19,0x18,0x07,
18     0xc1,0xf8,0xdf,0xdd,0xb8,0x77,0x00,0xc7,0xd6,0x53,0x21,0xed,0x02,0x53,0xee,0xac}}};
19
20  const TPM2B_EC_TEST      c_ecTestKey_QeX = {{32, {
21     0xa5,0x1e,0x80,0xd1,0x76,0x3e,0x8b,0x96,0xce,0xcc,0x21,0x82,0xc9,0xa2,0xa2,0xed,
22     0x47,0x21,0x89,0x53,0x44,0xe9,0xc7,0x92,0xe7,0x31,0x48,0x38,0xe6,0xea,0x93,0x47}}};
23
24  const TPM2B_EC_TEST      c_ecTestKey_QeY = {{32, {
25     0x30,0xe6,0x4f,0x97,0x03,0xa1,0xcb,0x3b,0x32,0x2a,0x70,0x39,0x94,0xeb,0x4e,0xea,
26     0x55,0x88,0x81,0x3f,0xb5,0x00,0xb8,0x54,0x25,0xab,0xd4,0xda,0xfd,0x53,0x7a,0x18}}};

```

ECDH test results

```

27  const TPM2B_EC_TEST      c_ecTestEcdh_X = {{32, {
28     0x64,0x02,0x68,0x92,0x78,0xdb,0x33,0x52,0xed,0x3b,0xfa,0x3b,0x74,0xa3,0x3d,0x2c,
29     0x2f,0x9c,0x59,0x03,0x07,0xf8,0x22,0x90,0xed,0xe3,0x45,0xf8,0x2a,0x0a,0xd8,0x1d}}};
30
31  const TPM2B_EC_TEST      c_ecTestEcdh_Y = {{32, {
32     0x58,0x94,0x05,0x82,0xbe,0x5f,0x33,0x02,0x25,0x90,0x3a,0x33,0x90,0x89,0xe3,0xe5,
33     0x10,0x4a,0xbc,0x78,0xa5,0xc5,0x07,0x64,0xaf,0x91,0xbc,0xe6,0xff,0x85,0x11,0x40}}};
34
35  TPM2B_TYPE(TEST_VALUE, 64);
36  const TPM2B_TEST_VALUE    c_ecTestValue = {{64, {
37     0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
38     0xf4,0x45,0x7f,0x13,0xd8,0x13,0x81,0x0b,0xe5,0x76,0xbe,0xaa,0xb6,0x3f,0x8d,0x4d,
39     0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xcb,0x0c,0xc7,0x11,0x8d,0xc3,0xff,
40     0x62,0x69,0xa2,0xbe,0x46,0x90,0xe7,0x7d,0x81,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}}};
41
42  #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
43
44  const TPM2B_EC_TEST      c_TestEcDsa_r = {{32, {
45     0x57,0xf3,0x36,0xb7,0xec,0xc2,0xdd,0x76,0x0e,0xe2,0x81,0x21,0x49,0xc5,0x66,0x11,
46     0x4b,0x8a,0x4f,0x17,0x62,0x82,0xcc,0x06,0xf6,0x64,0x78,0xef,0x6b,0x7c,0xf2,0x6c}}};
47  const TPM2B_EC_TEST      c_TestEcDsa_s = {{32, {
48     0x1b,0xed,0x23,0x72,0x8f,0x17,0x5f,0x47,0x2e,0xa7,0x97,0x2c,0x51,0x57,0x20,0x70,
49     0x6f,0x89,0x74,0x8a,0xa8,0xf4,0x26,0xf4,0x96,0xa1,0xb8,0x3e,0xe5,0x35,0xc5,0x94}}};
50

```

```

1  const TPM2B_EC_TEST      c_TestEcSchnorr_r = {{32, {
2      0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x1b, 0x08, 0x9f, 0xde,
3      0xef, 0x62, 0xe3, 0xf1, 0x14, 0xcb, 0x54, 0x28, 0x13, 0x76, 0xfc, 0x6d, 0x69, 0x22, 0xb5, 0x3e}}};
4  const TPM2B_EC_TEST      c_TestEcSchnorr_s = {{32, {
5      0xd9, 0xd3, 0x20, 0xfb, 0x4d, 0x16, 0xf2, 0xe6, 0xe2, 0x45, 0x07, 0x45, 0x1c, 0x92, 0x92, 0x92,
6      0xa9, 0x6b, 0x48, 0xf8, 0xd1, 0x98, 0x29, 0x4d, 0xd3, 0x8f, 0x56, 0xf2, 0xbb, 0x2e, 0x22, 0x3b}}};
7
8  #endif // SHA1
9
10 #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
11
12 const TPM2B_EC_TEST      c_TestEcDsa_r = {{32, {
13     0x04, 0x7d, 0x54, 0xeb, 0x04, 0x6f, 0x56, 0xec, 0xa2, 0x6c, 0x38, 0x8c, 0xeb, 0x43, 0x0b, 0x71,
14     0xf8, 0xf2, 0xf4, 0xa5, 0xe0, 0x1d, 0x3c, 0xa2, 0x39, 0x31, 0xe4, 0xe7, 0x36, 0x3b, 0xb5, 0x5f}}};
15 const TPM2B_EC_TEST      c_TestEcDsa_s = {{32, {
16     0x8f, 0xd0, 0x12, 0xd9, 0x24, 0x75, 0xf6, 0xc4, 0x3b, 0xb5, 0x46, 0x75, 0x3a, 0x41, 0x8d, 0x80,
17     0x23, 0x99, 0x38, 0xd7, 0xe2, 0x40, 0xca, 0x9a, 0x19, 0x2a, 0xfc, 0x54, 0x75, 0xd3, 0x4a, 0x6e}}};
18
19 const TPM2B_EC_TEST      c_TestEcSchnorr_r = {{32, {
20     0xf7, 0xb9, 0x15, 0x4c, 0x34, 0xf6, 0x41, 0x19, 0xa3, 0xd2, 0xf1, 0xbd, 0xf4, 0x13, 0x6a, 0x4f,
21     0x63, 0xb8, 0x4d, 0xb5, 0xc8, 0xcd, 0xde, 0x85, 0x95, 0xa5, 0x39, 0x0a, 0x14, 0x49, 0x3d, 0x2f}}};
22 const TPM2B_EC_TEST      c_TestEcSchnorr_s = {{32, {
23     0xfe, 0xbe, 0x17, 0xaa, 0x31, 0x22, 0x9f, 0xd0, 0xd2, 0xf5, 0x25, 0x04, 0x92, 0xb0, 0xaa, 0x4e,
24     0xcc, 0x1c, 0xb6, 0x79, 0xd6, 0x42, 0xb3, 0x4e, 0x3f, 0xbb, 0xfe, 0x5f, 0xd0, 0xd0, 0x8b, 0xc3}}};
25
26 #endif // SHA256
27
28 #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
29
30 const TPM2B_EC_TEST      c_TestEcDsa_r = {{32, {
31     0xf5, 0x74, 0x6d, 0xd6, 0xc6, 0x56, 0x86, 0xbb, 0xba, 0x1c, 0xba, 0x75, 0x65, 0xee, 0x64, 0x31,
32     0xce, 0x04, 0xe3, 0x9f, 0x24, 0x3f, 0xbd, 0xfe, 0x04, 0xcd, 0xab, 0x7e, 0xfe, 0xad, 0xcb, 0x82}}};
33 const TPM2B_EC_TEST      c_TestEcDsa_s = {{32, {
34     0xc2, 0x4f, 0x32, 0xa1, 0x06, 0xc0, 0x85, 0x4f, 0xc6, 0xd8, 0x31, 0x66, 0x91, 0x9f, 0x79, 0xcd,
35     0x5b, 0xe5, 0x7b, 0x94, 0xa1, 0x91, 0x38, 0xac, 0xd4, 0x20, 0xa2, 0x10, 0xf0, 0xd5, 0x9d, 0xbf}}};
36
37 const TPM2B_EC_TEST      c_TestEcSchnorr_r = {{32, {
38     0x1e, 0xb8, 0xe1, 0xbf, 0xa1, 0x9e, 0x39, 0x1e, 0x58, 0xa2, 0xe6, 0x59, 0xd0, 0x1a, 0x6a, 0x03,
39     0x6a, 0x1f, 0x1c, 0x4f, 0x36, 0x19, 0xc1, 0xec, 0x30, 0xa4, 0x85, 0x1b, 0xe9, 0x74, 0x35, 0x66}}};
40 const TPM2B_EC_TEST      c_TestEcSchnorr_s = {{32, {
41     0xb9, 0xe6, 0xe3, 0x7e, 0xcb, 0xb9, 0xea, 0xf1, 0xcc, 0xf4, 0x48, 0x44, 0x4a, 0xda, 0xc8, 0xd7,
42     0x87, 0xb4, 0xba, 0x40, 0xfe, 0x5b, 0x68, 0x11, 0x14, 0xcf, 0xa0, 0x0e, 0x85, 0x46, 0x99, 0x01}}};
43
44 #endif // SHA384
45
46 #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
47
48 const TPM2B_EC_TEST      c_TestEcDsa_r = {{32, {
49     0xc9, 0x71, 0xa6, 0xb4, 0xaf, 0x46, 0x26, 0x8c, 0x27, 0x00, 0x06, 0x3b, 0x00, 0x0f, 0xa3, 0x17,
50     0x72, 0x48, 0x40, 0x49, 0x4d, 0x51, 0x4f, 0xa4, 0xcb, 0x7e, 0x86, 0xe9, 0xe7, 0xb4, 0x79, 0xb2}}};
51 const TPM2B_EC_TEST      c_TestEcDsa_s = {{32, {
52     0x87, 0xbc, 0xc0, 0xed, 0x74, 0x60, 0x9e, 0xfa, 0x4e, 0xe8, 0x16, 0xf3, 0xf9, 0x6b, 0x26, 0x07,
53     0x3c, 0x74, 0x31, 0x7e, 0xf0, 0x62, 0x46, 0xdc, 0xd6, 0x45, 0x22, 0x47, 0x3e, 0x0c, 0xa0, 0x02}}};
54
55 const TPM2B_EC_TEST      c_TestEcSchnorr_r = {{32, {
56     0xcc, 0x07, 0xad, 0x65, 0x91, 0xdd, 0xa0, 0x10, 0x23, 0xae, 0x53, 0xec, 0xdf, 0xf1, 0x50, 0x90,
57     0x16, 0x96, 0xf4, 0x45, 0x09, 0x73, 0x9c, 0x84, 0xb5, 0x5c, 0x5f, 0x08, 0x51, 0xcb, 0x60, 0x01}}};
58 const TPM2B_EC_TEST      c_TestEcSchnorr_s = {{32, {
59     0x55, 0x20, 0x21, 0x54, 0xe2, 0x49, 0x07, 0x47, 0x71, 0xf4, 0x99, 0x15, 0x54, 0xf3, 0xab, 0x14,
60     0xdb, 0x8e, 0xda, 0x79, 0xb6, 0x02, 0x0e, 0xe3, 0x5e, 0x6f, 0x2c, 0xb6, 0x05, 0xbd, 0x14, 0x10}}};
61
62 #endif // SHA512
63
64 #endif // SELF TEST DATA

```

10.1.15 CryptSym.h

10.1.15.1 Introduction

This file contains the implementation of the symmetric block cipher modes allowed for a TPM. These functions only use the single block encryption functions of the selected symmetric cryptographic library.

10.1.15.2 Includes, Defines, and Typedefs

```

1  #ifndef CRYPT_SYM_H
2  #define CRYPT_SYM_H
3
4  #if ALG_AES
5  #   define IF_IMPLEMENTED_AES(op)    op(AES, aes)
6  #else
7  #   define IF_IMPLEMENTED_AES(op)
8  #endif
9  #if ALG_SM4
10 #   define IF_IMPLEMENTED_SM4(op)    op(SM4, sm4)
11 #else
12 #   define IF_IMPLEMENTED_SM4(op)
13 #endif
14 #if ALG_CAMELLIA
15 #   define IF_IMPLEMENTED_CAMELLIA(op)    op(CAMELLIA, camellia)
16 #else
17 #   define IF_IMPLEMENTED_CAMELLIA(op)
18 #endif
19 #if ALG_TDES
20 #   define IF_IMPLEMENTED_TDES(op)    op(TDES, tdes)
21 #else
22 #   define IF_IMPLEMENTED_TDES(op)
23 #endif
24
25 #define FOR_EACH_SYM(op) \
26     IF_IMPLEMENTED_AES(op) \
27     IF_IMPLEMENTED_SM4(op) \
28     IF_IMPLEMENTED_CAMELLIA(op) \
29     IF_IMPLEMENTED_TDES(op)

```

Macros for creating the key schedule union

```

30 #define KEY_SCHEDULE(SYM, sym)    tpmKeySchedule##SYM sym;
31 #define TDES    DES[3]
32 typedef union tpmCryptKeySchedule_t {
33     FOR_EACH_SYM(KEY_SCHEDULE)
34
35     #if SYMMETRIC_ALIGNMENT == 8
36         uint64_t    alignment;
37     #else
38         uint32_t    alignment;
39     #endif
40 } tpmCryptKeySchedule_t;

```

Each block cipher within a library is expected to conform to the same calling conventions with three parameters (*keySchedule*, *in*, and *out*) in the same order. That means that all algorithms would use the same order of the same parameters. The code is written assuming the (*keySchedule*, *in*, and *out*) order. However, if the library uses a different order, the order can be changed with a SWIZZLE macro that puts the parameters in the correct order. Note that all algorithms have to use the same order and number of parameters because the code to build the calling list is common for each call to encrypt or decrypt with the algorithm chosen by setting a function pointer to select the algorithm that is used.


```

41  #   define ENCRYPT(keySchedule, in, out)                \
42      encrypt(SWIZZLE(keySchedule, in, out))
43
44  #   define DECRYPT(keySchedule, in, out)                \
45      decrypt(SWIZZLE(keySchedule, in, out))

```

Note that the macros rely on *encrypt* as local values in the functions that use these macros. Those parameters are set by the macro that set the key schedule to be used for the call.

```

46  #define ENCRYPT_CASE(ALG, alg)                          \
47      case TPM_ALG_##ALG:                                \
48          TpmCryptSetEncryptKey##ALG(key, keySizeInBits, &keySchedule.alg); \
49          encrypt = (TpmCryptSetSymKeyCall_t)TpmCryptEncrypt##ALG; \
50          break;
51  #define DECRYPT_CASE(ALG, alg)                          \
52      case TPM_ALG_##ALG:                                \
53          TpmCryptSetDecryptKey##ALG(key, keySizeInBits, &keySchedule.alg); \
54          decrypt = (TpmCryptSetSymKeyCall_t)TpmCryptDecrypt##ALG; \
55          break;
56
57  #endif // CRYPT_SYM_H

```

10.1.16 OIDs.h

```

1  #ifndef _OIDS_H_
2  #define _OIDS_H_

```

All the OIDs in this file are defined as DER-encoded values with a leading tag 0x06 (ASN1_OBJECT_IDENTIFIER), followed by a single length byte. This allows the OID size to be determined by looking at octet[1] of the OID (total size is OID[1] + 2).

These macros allow OIDs to be defined (or not) depending on whether the associated hash algorithm is implemented.

NOTE When one of these macros is used, the NAME needs `_` on each side. The exception is when the macro is used for the hash OID when only a single `_` is used.

```

3  #ifndef ALG_SHA1
4  #   define ALG_SHA1 NO
5  #endif
6  #if ALG_SHA1
7  #define SHA1_OID(NAME)    MAKE_OID(NAME##SHA1)
8  #else
9  #define SHA1_OID(NAME)
10 #endif
11 #ifndef ALG_SHA256
12 #   define ALG_SHA256 NO
13 #endif
14 #if ALG_SHA256
15 #define SHA256_OID(NAME)  MAKE_OID(NAME##SHA256)
16 #else
17 #define SHA256_OID(NAME)
18 #endif
19 #ifndef ALG_SHA384
20 #   define ALG_SHA384 NO
21 #endif
22 #if ALG_SHA384
23 #define SHA384_OID(NAME)  MAKE_OID(NAME##SHA384)
24 #else
25 #define SHA384_OID(NAME)
26 #endif
27 #ifndef ALG_SHA512
28 #   define ALG_SHA512 NO
29 #endif
30 #if ALG_SHA512
31 #define SHA512_OID(NAME)  MAKE_OID(NAME##SHA512)
32 #else
33 #define SHA512_OID(NAME)
34 #endif
35 #ifndef ALG_SM3_256
36 #   define ALG_SM3_256 NO
37 #endif
38 #if ALG_SM3_256
39 #define SM3_256_OID(NAME) MAKE_OID(NAME##SM3_256)
40 #else
41 #define SM3_256_OID(NAME)
42 #endif
43 #ifndef ALG_SHA3_256
44 #   define ALG_SHA3_256 NO
45 #endif
46 #if ALG_SHA3_256
47 #define SHA3_256_OID(NAME) MAKE_OID(NAME##SHA3_256)
48 #else
49 #define SHA3_256_OID(NAME)
50 #endif
51 #ifndef ALG_SHA3_384

```

```

52 #   define ALG_SHA3_384 NO
53 #endif
54 #if ALG_SHA3_384
55 #define SHA3_384_OID(NAME) MAKE_OID(NAME##SHA3_384)
56 #else
57 #define SHA3_384_OID(NAME)
58 #endif
59 #ifndef ALG_SHA3_512
60 #   define ALG_SHA3_512 NO
61 #endif
62 #if ALG_SHA3_512
63 #define SHA3_512_OID(NAME) MAKE_OID(NAME##SHA3_512)
64 #else
65 #define SHA3_512_OID(NAME)
66 #endif

```

These are encoded to take one additional byte of algorithm selector

```

67 #define NIST_HASH      0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 2
68 #define NIST_SIG       0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 3

```

These hash OIDs used in a lot of places.

```

69 #define OID_SHA1_VALUE      0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A
70 SHA1_OID(_);               // Expands to:
71                             //   MAKE_OID(_SHA1)
72                             // which expands to:
73                             //   EXTERN const BYTE OID_SHA1[] INITIALIZER({OID_SHA1_VALUE})
74                             // which, depending on the setting of EXTERN and
75                             // INITIALIZER, expands to either:
76                             //   extern const BYTE   OID_SHA1[]
77                             // or
78                             //   const BYTE         OID_SHA1[] = {OID_SHA1_VALUE}
79                             // which is:
80                             //   const BYTE         OID_SHA1[] = {0x06, 0x05, 0x2B, 0x0E,
81                             //                                     0x03, 0x02, 0x1A}
82 #define OID_SHA256_VALUE   NIST_HASH, 1
83 SHA256_OID(_);
84
85 #define OID_SHA384_VALUE   NIST_HASH, 2
86 SHA384_OID(_);
87
88 #define OID_SHA512_VALUE   NIST_HASH, 3
89 SHA512_OID(_);
90
91 #define OID_SM3_256_VALUE   0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, \
92                             0x83, 0x11
93 SM3_256_OID(_);           // (1.2.156.10197.1.401)
94
95 #define OID_SHA3_256_VALUE   NIST_HASH, 8
96 SHA3_256_OID(_);
97
98 #define OID_SHA3_384_VALUE   NIST_HASH, 9
99 SHA3_384_OID(_);
100
101 #define OID_SHA3_512_VALUE   NIST_HASH, 10
102 SHA3_512_OID(_);

```

These are used for RSA-PSS

```

103 #if ALG_RSA
104
105 #define OID_MGF1_VALUE      0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
106                             0x01, 0x01, 0x08

```

```

107 MAKE_OID(_MGF1);
108
109 #define OID_RSAPSS_VALUE      0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
110                               0x01, 0x01, 0x0A
111 MAKE_OID(_RSAPSS);

```

This is the OID to designate the public part of an RSA key.

```

112 #define OID_PKCS1_PUB_VALUE  0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
113                               0x01, 0x01, 0x01
114 MAKE_OID(_PKCS1_PUB);

```

These are used for RSA PKCS1 signature Algorithms

```

115 #define OID_PKCS1_SHA1_VALUE  0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,      \
116                               0x0D, 0x01, 0x01, 0x05
117 SHA1_OID(_PKCS1_);           // (1.2.840.113549.1.1.5)
118
119 #define OID_PKCS1_SHA256_VALUE 0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,      \
120                               0x0D, 0x01, 0x01, 0x0B
121 SHA256_OID(_PKCS1_);         // (1.2.840.113549.1.1.11)
122
123 #define OID_PKCS1_SHA384_VALUE 0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,      \
124                               0x0D, 0x01, 0x01, 0x0C
125 SHA384_OID(_PKCS1_);         // (1.2.840.113549.1.1.12)
126
127 #define OID_PKCS1_SHA512_VALUE 0x06,0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,      \
128                               0x0D, 0x01, 0x01, 0x0D
129 SHA512_OID(_PKCS1_);         // (1.2.840.113549.1.1.13)
130
131 #define OID_PKCS1_SM3_256_VALUE 0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55,      \
132                               0x01, 0x83, 0x78
133 SM3_256_OID(_PKCS1_);        // 1.2.156.10197.1.504
134
135 #define OID_PKCS1_SHA3_256_VALUE NIST_SIG, 14
136 SHA3_256_OID(_PKCS1_);
137 #define OID_PKCS1_SHA3_384_VALUE NIST_SIG, 15
138 SHA3_384_OID(_PKCS1_);
139 #define OID_PKCS1_SHA3_512_VALUE NIST_SIG, 16
140 SHA3_512_OID(_PKCS1_);
141
142 #endif // ALG_RSA
143
144 #if ALG_ECDSA
145
146 #define OID_ECDSA_SHA1_VALUE  0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
147                               0x01
148 SHA1_OID(_ECDSA_);           // (1.2.840.10045.4.1) SHA1 digest signed by an ECDSA key.
149
150 #define OID_ECDSA_SHA256_VALUE 0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
151                               0x03, 0x02
152 SHA256_OID(_ECDSA_);         // (1.2.840.10045.4.3.2) SHA256 digest signed by an ECDSA key.
153
154 #define OID_ECDSA_SHA384_VALUE 0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
155                               0x03, 0x03
156 SHA384_OID(_ECDSA_);         // (1.2.840.10045.4.3.3) SHA384 digest signed by an ECDSA key.
157
158 #define OID_ECDSA_SHA512_VALUE 0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
159                               0x03, 0x04
160 SHA512_OID(_ECDSA_);         // (1.2.840.10045.4.3.4) SHA512 digest signed by an ECDSA key.
161
162 #define OID_ECDSA_SM3_256_VALUE 0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, \
163                               0x83, 0x75
164 SM3_256_OID(_ECDSA_);        // 1.2.156.10197.1.501
165

```

```

166 #define OID_ECDSA_SHA3_256_VALUE      NIST_SIG, 10
167 SHA3_256_OID( ECDSA );
168 #define OID_ECDSA_SHA3_384_VALUE      NIST_SIG, 11
169 SHA3_384_OID( ECDSA );
170 #define OID_ECDSA_SHA3_512_VALUE      NIST_SIG, 12
171 SHA3_512_OID( ECDSA );
172
173 #endif // ALG_ECDSA
174
175 #if ALG_ECC
176
177 #define OID_ECC_PUBLIC_VALUE           0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x02, \
178                                         0x01
179 MAKE_OID( _ECC_PUBLIC );
180
181 #define OID_ECC_NIST_P192_VALUE        0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, \
182                                         0x01, 0x01
183 #if ECC_NIST_P192
184 MAKE_OID( _ECC_NIST_P192 ); // (1.2.840.10045.3.1.1) 'nistP192'
185 #endif // ECC_NIST_P192
186
187 #define OID_ECC_NIST_P224_VALUE        0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x21
188 #if ECC_NIST_P224
189 MAKE_OID( _ECC_NIST_P224 ); // (1.3.132.0.33) 'nistP224'
190 #endif // ECC_NIST_P224
191
192 #define OID_ECC_NIST_P256_VALUE        0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, \
193                                         0x01, 0x07
194 #if ECC_NIST_P256
195 MAKE_OID( _ECC_NIST_P256 ); // (1.2.840.10045.3.1.7) 'nistP256'
196 #endif // ECC_NIST_P256
197
198 #define OID_ECC_NIST_P384_VALUE        0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x22
199 #if ECC_NIST_P384
200 MAKE_OID( _ECC_NIST_P384 ); // (1.3.132.0.34) 'nistP384'
201 #endif // ECC_NIST_P384
202
203 #define OID_ECC_NIST_P521_VALUE        0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x23
204 #if ECC_NIST_P521
205 MAKE_OID( _ECC_NIST_P521 ); // (1.3.132.0.35) 'nistP521'
206 #endif // ECC_NIST_P521

```

No OIDs defined for these anonymous curves

```

207 #define OID_ECC_BN_P256_VALUE          0x00
208 #if ECC_BN_P256
209 MAKE_OID( _ECC_BN_P256 );
210 #endif // ECC_BN_P256
211
212 #define OID_ECC_BN_P638_VALUE          0x00
213 #if ECC_BN_P638
214 MAKE_OID( _ECC_BN_P638 );
215 #endif // ECC_BN_P638
216
217 #define OID_ECC_SM2_P256_VALUE          0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, \
218                                         0x82, 0x2D
219 #if ECC_SM2_P256
220 MAKE_OID( _ECC_SM2_P256 ); // Don't know where I found this OID. It needs checking
221 #endif // ECC_SM2_P256
222
223 #if ECC_BN_P256
224 #define OID_ECC_BN_P256              NULL
225 #endif // ECC_BN_P256
226
227 #endif // ALG_ECC

```

```

228
229 #define OID_SIZE(OID)      (OID[1] + 2)
230
231 #endif // !_OIDS_H_

```

10.1.17 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

```

Entropy is the size of the state. The state is the size of the key plus the IV. The IV is a block. If Key = 256 and Block = 128 then State = 384

```

6  #   define DRBG_TEST_INITIATE_ENTROPY      \
7      0x0d, 0x15, 0xaa, 0x80, 0xb1, 0x6c, 0x3a, 0x10, \
8      0x90, 0x6c, 0xfe, 0xdb, 0x79, 0x5d, 0xae, 0x0b, \
9      0x5b, 0x81, 0x04, 0x1c, 0x5c, 0x5b, 0xfa, 0xcb, \
10     0x37, 0x3d, 0x44, 0x40, 0xd9, 0x12, 0x0f, 0x7e, \
11     0x3d, 0x6c, 0xf9, 0x09, 0x86, 0xcf, 0x52, 0xd8, \
12     0x5d, 0x3e, 0x94, 0x7d, 0x8c, 0x06, 0x1f, 0x91
13
14 #   define DRBG_TEST_RESEED_ENTROPY      \
15     0x6e, 0xe7, 0x93, 0xa3, 0x39, 0x55, 0xd7, 0x2a, \
16     0xd1, 0x2f, 0xd8, 0x0a, 0x8a, 0x3f, 0xcf, 0x95, \
17     0xed, 0x3b, 0x4d, 0xac, 0x57, 0x95, 0xfe, 0x25, \
18     0xcf, 0x86, 0x9f, 0x7c, 0x27, 0x57, 0x3b, 0xbc, \
19     0x56, 0xf1, 0xac, 0xae, 0x13, 0xa6, 0x50, 0x42, \
20     0xb3, 0x40, 0x09, 0x3c, 0x46, 0x4a, 0x7a, 0x22
21
22 #   define DRBG_TEST_GENERATED_INTERM      \
23     0x28, 0xe0, 0xeb, 0xb8, 0x21, 0x01, 0x66, 0x50, \
24     0x8c, 0x8f, 0x65, 0xf2, 0x20, 0x7b, 0xd0, 0xa3
25
26 #   define DRBG_TEST_GENERATED      \
27     0x94, 0x6f, 0x51, 0x82, 0xd5, 0x45, 0x10, 0xb9, \
28     0x46, 0x12, 0x48, 0xf5, 0x71, 0xca, 0x06, 0xc9
29 #elif DRBG_KEY_SIZE_BITS == 128
30
31 #   define DRBG_TEST_INITIATE_ENTROPY      \
32     0x8f, 0xc1, 0x1b, 0xdb, 0x5a, 0xab, 0xb7, 0xe0, \
33     0x93, 0xb6, 0x14, 0x28, 0xe0, 0x90, 0x73, 0x03, \
34     0xcb, 0x45, 0x9f, 0x3b, 0x60, 0x0d, 0xad, 0x87, \
35     0x09, 0x55, 0xf2, 0x2d, 0xa8, 0x0a, 0x44, 0xf8
36
37 #   define DRBG_TEST_RESEED_ENTROPY      \
38     0x0c, 0xd5, 0x3c, 0xd5, 0xec, 0xcd, 0x5a, 0x10, \
39     0xd7, 0xea, 0x26, 0x61, 0x11, 0x25, 0x9b, 0x05, \
40     0x57, 0x4f, 0xc6, 0xdd, 0xd8, 0xbe, 0xd8, 0xbd, \
41     0x72, 0x37, 0x8c, 0xf8, 0x2f, 0x1d, 0xba, 0x2a
42
43 #define DRBG_TEST_GENERATED_INTERM      \
44     0xdc, 0x3c, 0xf6, 0xbf, 0x5b, 0xd3, 0x41, 0x13, \
45     0x5f, 0x2c, 0x68, 0x11, 0xa1, 0x07, 0x1c, 0x87
46
47 #   define DRBG_TEST_GENERATED      \
48     0xb6, 0x18, 0x50, 0xde, 0xcf, 0xd7, 0x10, 0x6d, \
49     0x44, 0x76, 0x9a, 0x8e, 0x6e, 0x8c, 0x1a, 0xd4
50
51 #endif
52
53 #endif //      _MSBN_DRBG_TEST_VECTORS_H

```

10.1.18 TpmAsn1.h

10.1.18.1 Introduction

This file contains the macro and structure definitions for the X509 commands and functions.

```
1  #ifndef _TPMASN1_H_
2  #define _TPMASN1_H_
```

10.1.18.2 Includes

```
3  #include "Tpm.h"
4  #include "OIDs.h"
```

10.1.18.3 Defined Constants

10.1.18.3.1 ASN.1 Universal Types (Class 00b)

```
5  #define ASN1_EOC                0x00
6  #define ASN1_BOOLEAN            0x01
7  #define ASN1_INTEGER            0x02
8  #define ASN1_BITSTRING          0x03
9  #define ASN1_OCTET_STRING       0x04
10 #define ASN1_NULL                0x05
11 #define ASN1_OBJECT_IDENTIFIER  0x06
12 #define ASN1_OBJECT_DESCRIPTOR  0x07
13 #define ASN1_EXTERNAL            0x08
14 #define ASN1_REAL                0x09
15 #define ASN1_ENUMERATED          0x0A
16 #define ASN1_EMBEDDED           0x0B
17 #define ASN1_UTF8String          0x0C
18 #define ASN1_RELATIVE_OID        0x0D
19 #define ASN1_SEQUENCE            0x10    // Primitive + Constructed + 0x10
20 #define ASN1_SET                 0x11    // Primitive + Constructed + 0x11
21 #define ASN1_NumericString        0x12
22 #define ASN1_PrintableString      0x13
23 #define ASN1_T61String           0x14
24 #define ASN1_VideoString         0x15
25 #define ASN1_IA5String           0x16
26 #define ASN1_UTCTime             0x17
27 #define ASN1_GeneralizeTime       0x18
28 #define ASN1_VisibleString       0x1A
29 #define ASN1_GeneralString       0x1B
30 #define ASN1_UniversalString      0x1C
31 #define ASN1_CHARACTER_STRING    0x1D
32 #define ASN1_BMPString           0x1E
33 #define ASN1_CONSTRUCTED         0x20
34
35 #define ASN1_APPLICATION_SPECIFIC 0xA0
36
37 #define ASN1_CONSTRUCTED_SEQUENCE (ASN1_SEQUENCE + ASN1_CONSTRUCTED)
38
39 #define MAX_DEPTH                10    // maximum push depth for marshaling context.
```

10.1.18.4 Macros

10.1.18.4.1 Unmarshaling Macros

```
40 #ifndef VERIFY
```



```

41 #define VERIFY(_X_) {if(!(_X_)) goto Error; }
42 #endif

```

Checks the validity of the size making sure that there is no wrap around

```

43 #define CHECK_SIZE(context, length) \
44     VERIFY( ((length) + (context)->offset) >= (context)->offset) \
45     && ((length) + (context)->offset) <= (context)->size))
46 #define NEXT_OCTET(context) ((context)->buffer[(context)->offset++])
47 #define PEEK_NEXT(context) ((context)->buffer[(context)->offset])

```

10.1.18.4.2 Marshaling Macros

Marshaling works in reverse order. The offset is set to the top of the buffer and, as the buffer is filled, *offset* counts down to zero. When the full thing is encoded it can be moved to the top of the buffer. This happens when the last context is closed.

```

48 #define CHECK_SPACE(context, length)    VERIFY(context->offset > length)

```

10.1.18.5 Structures

```

49 typedef struct ASN1UnmarshalContext {
50     BYTE          *buffer;    // pointer to the buffer
51     INT16         size;       // size of the buffer (a negative number indicates
52                               // a parsing failure).
53     INT16         offset;     // current offset into the buffer (a negative number
54                               // indicates a parsing failure). Not used
55     BYTE          tag;        // The last unmarshaled tag
56 } ASN1UnmarshalContext;
57
58 typedef struct ASN1MarshalContext {
59     BYTE          *buffer;    // pointer to the start of the buffer
60     INT16         offset;     // place on the top where the last entry was added
61                               // items are added from the bottom up.
62     INT16         end;        // the end offset of the current value
63     INT16         depth;     // how many pushed end values.
64     INT16         ends[MAX_DEPTH];
65 } ASN1MarshalContext;
66
67 #endif // _TPMASN1_H_

```

10.1.19 X509.h

10.1.19.1 Introduction

This file contains the macro and structure definitions for the X509 commands and functions.

```

1 #ifndef _X509_H_
2 #define _X509_H_

```

10.1.19.2 Includes

```

3 #include "Tpm.h"
4 #include "TpmASN1.h"

```

10.1.19.3 Defined Constants

10.1.19.3.1 X509 Application-specific types

```

5  #define X509_SELECTION          0xA0
6  #define X509_ISSUER_UNIQUE_ID  0xA1
7  #define X509_SUBJECT_UNIQUE_ID 0xA2
8  #define X509_EXTENSIONS        0xA3

```

These defines give the order in which values appear in the TBScertificate of an x.509 certificate. These values are used to index into an array of

```

9  #define ENCODED_SIZE_REF        0
10 #define VERSION_REF             (ENCODED_SIZE_REF + 1)
11 #define SERIAL_NUMBER_REF       (VERSION_REF + 1)
12 #define SIGNATURE_REF           (SERIAL_NUMBER_REF + 1)
13 #define ISSUER_REF               (SIGNATURE_REF + 1)
14 #define VALIDITY_REF            (ISSUER_REF + 1)
15 #define SUBJECT_KEY_REF          (VALIDITY_REF + 1)
16 #define SUBJECT_PUBLIC_KEY_REF   (SUBJECT_KEY_REF + 1)
17 #define EXTENSIONS_REF          (SUBJECT_PUBLIC_KEY_REF + 1)
18 #define REF_COUNT               (EXTENSIONS_REF + 1)

```

10.1.19.4 Structures

Used to access the fields of a TBSSignature some of which are in the *in_CertifyX509* structure and some of which are in the *out_CertifyX509* structure.

```

19 typedef struct stringRef
20 {
21     BYTE      *buf;
22     INT16     len;
23 } stringRef;

```

This is defined to avoid bit by bit comparisons within a UINT32

```

24 typedef union x509KeyUsageUnion {
25     TPMA_X509_KEY_USAGE    x509;
26     UINT32                 integer;
27 } x509KeyUsageUnion;

```

10.1.19.5 Global X509 Constants

These values are instantiated by X509_spt.c and referenced by other X509-related files.

This is the DER-encoded value for the Key Usage OID (2.5.29.15). This is the full OID, not just the numeric value

```

28 #define OID_KEY_USAGE_EXTENSION_VALUE 0x06, 0x03, 0x55, 0x1D, 0x0F
29 MAKE_OID(_KEY_USAGE_EXTENSION);

```

This is the DER-encoded value for the TCG-defined TPMA_OBJECT OID (2.23.133.10.1.1.1)

```

30 #define OID_TCG_TPMA_OBJECT_VALUE    0x06, 0x07, 0x67, 0x81, 0x05, 0x0a, 0x01, \
31                                         0x01, 0x01
32 MAKE_OID(_TCG_TPMA_OBJECT);
33
34 #ifdef _X509_SPT_

```

If a bit is SET in KEY_USAGE_SIGN is also SET in *keyUsage* then the associated key has to have *sign* SET.

```

35 const x509KeyUsageUnion KEY_USAGE_SIGN =
36 { TPM_X509_KEY_USAGE_INITIALIZER(
37     /* bits_at_0      */ 0, /* decipheronly */ 0, /* encipheronly */ 0,
38     /* crlsign        */ 1, /* keycertsign  */ 1, /* keyagreement */ 0,
39     /* dataencipherment */ 0, /* keyencipherment */ 0, /* nonrepudiation */ 0,
40     /* digitalsignature */ 1) };

```

If a bit is SET in KEY_USAGE_DECRYPT is also SET in *keyUsage* then the associated key has to have *decrypt* SET.

```

41 const x509KeyUsageUnion KEY_USAGE_DECRYPT =
42 { TPM_X509_KEY_USAGE_INITIALIZER(
43     /* bits_at_0      */ 0, /* decipheronly */ 1, /* encipheronly */ 1,
44     /* crlsign        */ 0, /* keycertsign  */ 0, /* keyagreement */ 1,
45     /* dataencipherment */ 1, /* keyencipherment */ 1, /* nonrepudiation */ 0,
46     /* digitalsignature */ 0) };
47 #else
48 extern x509KeyUsageUnion KEY_USAGE_SIGN;
49 extern x509KeyUsageUnion KEY_USAGE_DECRYPT;
50 #endif
51
52 #endif // _X509_H_

```

10.1.20 TpmAlgorithmDefines.h

This file contains the algorithm values from the TCG Algorithm Registry.

```

1 #ifndef TPM_ALGORITHM_DEFINES_H
2 #define _TPM_ALGORITHM_DEFINES_H_

```

TPM 2.0 Part 2: Table 3 - Definition of Base Types

```

3 #define ECC_CURVES \
4     {TPM_ECC_BN_P256, TPM_ECC_BN_P638, TPM_ECC_NIST_P192, \
5     TPM_ECC_NIST_P224, TPM_ECC_NIST_P256, TPM_ECC_NIST_P384, \
6     TPM_ECC_NIST_P521, TPM_ECC_SM2_P256}
7 #define ECC_CURVE_COUNT \
8     (ECC_BN_P256 + ECC_BN_P638 + ECC_NIST_P192 + ECC_NIST_P224 + \
9     ECC_NIST_P256 + ECC_NIST_P384 + ECC_NIST_P521 + ECC_SM2_P256)
10 #define MAX_ECC_KEY_BITS \
11     MAX(ECC_BN_P256 * 256, MAX(ECC_BN_P638 * 638, \
12     MAX(ECC_NIST_P192 * 192, MAX(ECC_NIST_P224 * 224, \
13     MAX(ECC_NIST_P256 * 256, MAX(ECC_NIST_P384 * 384, \
14     MAX(ECC_NIST_P521 * 521, MAX(ECC_SM2_P256 * 256, \
15     0)))))))
16 #define MAX_ECC_KEY_BYTES BITS_TO_BYTES(MAX_ECC_KEY_BITS)

```

Vendor -Specific: Table 6 - Defines for PLATFORM Values

```

17 #define PLATFORM_FAMILY TPM_SPEC_FAMILY
18 #define PLATFORM_LEVEL TPM_SPEC_LEVEL
19 #define PLATFORM_VERSION TPM_SPEC_VERSION
20 #define PLATFORM_YEAR TPM_SPEC_YEAR
21 #define PLATFORM_DAY_OF_YEAR TPM_SPEC_DAY_OF_YEAR

```

TCG Algorithm Registry: Table 3 - Defines for RSA Asymmetric Cipher Algorithm Constants

```

22 #define RSA_KEY_SIZES_BITS \
23     (1024 * RSA_1024), (2048 * RSA_2048), (3072 * RSA_3072), \

```

```

24      (4096 * RSA_4096), (16384 * RSA_16384)
25  #if RSA_16384
26  #   define RSA_MAX_KEY_SIZE_BITS    16384
27  #elif RSA_4096
28  #   define RSA_MAX_KEY_SIZE_BITS    4096
29  #elif RSA_3072
30  #   define RSA_MAX_KEY_SIZE_BITS    3072
31  #elif RSA_2048
32  #   define RSA_MAX_KEY_SIZE_BITS    2048
33  #elif RSA_1024
34  #   define RSA_MAX_KEY_SIZE_BITS    1024
35  #else
36  #   define RSA_MAX_KEY_SIZE_BITS    0
37  #endif
38  #define MAX_RSA_KEY_BITS              RSA_MAX_KEY_SIZE_BITS
39  #define MAX_RSA_KEY_BYTES              ((RSA_MAX_KEY_SIZE_BITS + 7) / 8)

```

TCG Algorithm Registry: Table 13 - Defines for SHA1 Hash Values

```

40  #define SHA1_DIGEST_SIZE    20
41  #define SHA1_BLOCK_SIZE    64

```

TCG Algorithm Registry: Table 14 - Defines for SHA256 Hash Values

```

42  #define SHA256_DIGEST_SIZE  32
43  #define SHA256_BLOCK_SIZE   64

```

TCG Algorithm Registry: Table 15 - Defines for SHA384 Hash Values

```

44  #define SHA384_DIGEST_SIZE  48
45  #define SHA384_BLOCK_SIZE   128

```

TCG Algorithm Registry: Table 16 - Defines for SHA512 Hash Values

```

46  #define SHA512_DIGEST_SIZE  64
47  #define SHA512_BLOCK_SIZE   128

```

TCG Algorithm Registry: Table 17 - Defines for SM3_256 Hash Values

```

48  #define SM3_256_DIGEST_SIZE  32
49  #define SM3_256_BLOCK_SIZE   64

```

TCG Algorithm Registry: Table 18 - Defines for SHA3_256 Hash Values

```

50  #define SHA3_256_DIGEST_SIZE  32
51  #define SHA3_256_BLOCK_SIZE   136

```

TCG Algorithm Registry: Table 19 - Defines for SHA3_384 Hash Values

```

52  #define SHA3_384_DIGEST_SIZE  48
53  #define SHA3_384_BLOCK_SIZE   104

```

TCG Algorithm Registry: Table 20 - Defines for SHA3_512 Hash Values

```

54  #define SHA3_512_DIGEST_SIZE  64
55  #define SHA3_512_BLOCK_SIZE   72

```

TCG Algorithm Registry: Table 21 - Defines for AES Symmetric Cipher Algorithm Constants

```

56  #define AES_KEY_SIZES_BITS    \
57      (128 * AES_128), (192 * AES_192), (256 * AES_256)

```

```

58 #if AES_256
59 # define AES_MAX_KEY_SIZE_BITS 256
60 #elif AES_192
61 # define AES_MAX_KEY_SIZE_BITS 192
62 #elif AES_128
63 # define AES_MAX_KEY_SIZE_BITS 128
64 #else
65 # define AES_MAX_KEY_SIZE_BITS 0
66 #endif
67 #define MAX_AES_KEY_BITS AES_MAX_KEY_SIZE_BITS
68 #define MAX_AES_KEY_BYTES ((AES_MAX_KEY_SIZE_BITS + 7) / 8)
69 #define AES_128_BLOCK_SIZE_BYTES (AES_128 * 16)
70 #define AES_192_BLOCK_SIZE_BYTES (AES_192 * 16)
71 #define AES_256_BLOCK_SIZE_BYTES (AES_256 * 16)
72 #define AES_BLOCK_SIZES \
73     AES_128_BLOCK_SIZE_BYTES, AES_192_BLOCK_SIZE_BYTES, \
74     AES_256_BLOCK_SIZE_BYTES
75 #if ALG_AES
76 # define AES_MAX_BLOCK_SIZE 16
77 #else
78 # define AES_MAX_BLOCK_SIZE 0
79 #endif
80 #define MAX_AES_BLOCK_SIZE_BYTES AES_MAX_BLOCK_SIZE

```

TCG Algorithm Registry: Table 22 - Defines for SM4 Symmetric Cipher Algorithm Constants

```

81 #define SM4_KEY_SIZES_BITS (128 * SM4_128)
82 #if SM4_128
83 # define SM4_MAX_KEY_SIZE_BITS 128
84 #else
85 # define SM4_MAX_KEY_SIZE_BITS 0
86 #endif
87 #define MAX_SM4_KEY_BITS SM4_MAX_KEY_SIZE_BITS
88 #define MAX_SM4_KEY_BYTES ((SM4_MAX_KEY_SIZE_BITS + 7) / 8)
89 #define SM4_128_BLOCK_SIZE_BYTES (SM4_128 * 16)
90 #define SM4_BLOCK_SIZES SM4_128_BLOCK_SIZE_BYTES
91 #if ALG_SM4
92 # define SM4_MAX_BLOCK_SIZE 16
93 #else
94 # define SM4_MAX_BLOCK_SIZE 0
95 #endif
96 #define MAX_SM4_BLOCK_SIZE_BYTES SM4_MAX_BLOCK_SIZE

```

TCG Algorithm Registry: Table 23 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```

97 #define CAMELLIA_KEY_SIZES_BITS \
98     (128 * CAMELLIA_128), (192 * CAMELLIA_192), (256 * CAMELLIA_256)
99 #if CAMELLIA_256
100 # define CAMELLIA_MAX_KEY_SIZE_BITS 256
101 #elif CAMELLIA_192
102 # define CAMELLIA_MAX_KEY_SIZE_BITS 192
103 #elif CAMELLIA_128
104 # define CAMELLIA_MAX_KEY_SIZE_BITS 128
105 #else
106 # define CAMELLIA_MAX_KEY_SIZE_BITS 0
107 #endif
108 #define MAX_CAMELLIA_KEY_BITS CAMELLIA_MAX_KEY_SIZE_BITS
109 #define MAX_CAMELLIA_KEY_BYTES ((CAMELLIA_MAX_KEY_SIZE_BITS + 7) / 8)
110 #define CAMELLIA_128_BLOCK_SIZE_BYTES (CAMELLIA_128 * 16)
111 #define CAMELLIA_192_BLOCK_SIZE_BYTES (CAMELLIA_192 * 16)
112 #define CAMELLIA_256_BLOCK_SIZE_BYTES (CAMELLIA_256 * 16)
113 #define CAMELLIA_BLOCK_SIZES \
114     CAMELLIA_128_BLOCK_SIZE_BYTES, CAMELLIA_192_BLOCK_SIZE_BYTES, \
115     CAMELLIA_256_BLOCK_SIZE_BYTES
116 #if ALG_CAMELLIA

```

```

117 # define CAMELLIA_MAX_BLOCK_SIZE 16
118 #else
119 # define CAMELLIA_MAX_BLOCK_SIZE 0
120 #endif
121 #define MAX_CAMELLIA_BLOCK_SIZE_BYTES CAMELLIA_MAX_BLOCK_SIZE

```

TCG Algorithm Registry: Table 24 - Defines for TDES Symmetric Cipher Algorithm Constants

```

122 #define TDES_KEY_SIZES_BITS (128 * TDES_128), (192 * TDES_192)
123 #if TDES_192
124 # define TDES_MAX_KEY_SIZE_BITS 192
125 #elif TDES_128
126 # define TDES_MAX_KEY_SIZE_BITS 128
127 #else
128 # define TDES_MAX_KEY_SIZE_BITS 0
129 #endif
130 #define MAX_TDES_KEY_BITS TDES_MAX_KEY_SIZE_BITS
131 #define MAX_TDES_KEY_BYTES ((TDES_MAX_KEY_SIZE_BITS + 7) / 8)
132 #define TDES_128_BLOCK_SIZE_BYTES (TDES_128 * 8)
133 #define TDES_192_BLOCK_SIZE_BYTES (TDES_192 * 8)
134 #define TDES_BLOCK_SIZES \
135     TDES_128_BLOCK_SIZE_BYTES, TDES_192_BLOCK_SIZE_BYTES
136 #if ALG_TDES
137 # define TDES_MAX_BLOCK_SIZE 8
138 #else
139 # define TDES_MAX_BLOCK_SIZE 0
140 #endif
141 #define MAX_TDES_BLOCK_SIZE_BYTES TDES_MAX_BLOCK_SIZE

```

Additional values for benefit of code

```

142 #define TPM_CC_FIRST 0x0000011F
143 #define TPM_CC_LAST 0x0000019A
144
145 #if COMPRESSED_LISTS
146 #define ADD_FILL 0
147 #else
148 #define ADD_FILL 1
149 #endif

```

Size the array of library commands based on whether or not the array is packed (only defined commands) or dense (having entries for unimplemented commands)

```

150 #define LIBRARY_COMMAND_ARRAY_SIZE (0 \
151     + (ADD_FILL || CC_NV_UndefineSpaceSpecial) /* 0x0000011F */ \
152     + (ADD_FILL || CC_EvictControl) /* 0x00000120 */ \
153     + (ADD_FILL || CC_HierarchyControl) /* 0x00000121 */ \
154     + (ADD_FILL || CC_NV_UndefineSpace) /* 0x00000122 */ \
155     + ADD_FILL /* 0x00000123 */ \
156     + (ADD_FILL || CC_ChangeEPS) /* 0x00000124 */ \
157     + (ADD_FILL || CC_ChangePPS) /* 0x00000125 */ \
158     + (ADD_FILL || CC_Clear) /* 0x00000126 */ \
159     + (ADD_FILL || CC_ClearControl) /* 0x00000127 */ \
160     + (ADD_FILL || CC_ClockSet) /* 0x00000128 */ \
161     + (ADD_FILL || CC_HierarchyChangeAuth) /* 0x00000129 */ \
162     + (ADD_FILL || CC_NV_DefineSpace) /* 0x0000012A */ \
163     + (ADD_FILL || CC_PCR_Allocate) /* 0x0000012B */ \
164     + (ADD_FILL || CC_PCR_SetAuthPolicy) /* 0x0000012C */ \
165     + (ADD_FILL || CC_PP_Commands) /* 0x0000012D */ \
166     + (ADD_FILL || CC_SetPrimaryPolicy) /* 0x0000012E */ \
167     + (ADD_FILL || CC_FieldUpgradeStart) /* 0x0000012F */ \
168     + (ADD_FILL || CC_ClockRateAdjust) /* 0x00000130 */ \
169     + (ADD_FILL || CC_CreatePrimary) /* 0x00000131 */ \
170     + (ADD_FILL || CC_NV_GlobalWriteLock) /* 0x00000132 */ \

```


171	+ (ADD_FILL CC_GetCommandAuditDigest)	/* 0x00000133 */	\
172	+ (ADD_FILL CC_NV_Increment)	/* 0x00000134 */	\
173	+ (ADD_FILL CC_NV_SetBits)	/* 0x00000135 */	\
174	+ (ADD_FILL CC_NV_Extend)	/* 0x00000136 */	\
175	+ (ADD_FILL CC_NV_Write)	/* 0x00000137 */	\
176	+ (ADD_FILL CC_NV_WriteLock)	/* 0x00000138 */	\
177	+ (ADD_FILL CC_DictionaryAttackLockReset)	/* 0x00000139 */	\
178	+ (ADD_FILL CC_DictionaryAttackParameters)	/* 0x0000013A */	\
179	+ (ADD_FILL CC_NV_ChangeAuth)	/* 0x0000013B */	\
180	+ (ADD_FILL CC_PCR_Extend)	/* 0x0000013C */	\
181	+ (ADD_FILL CC_PCR_Reset)	/* 0x0000013D */	\
182	+ (ADD_FILL CC_SequenceComplete)	/* 0x0000013E */	\
183	+ (ADD_FILL CC_SetAlgorithmSet)	/* 0x0000013F */	\
184	+ (ADD_FILL CC_SetCommandCodeAuditStatus)	/* 0x00000140 */	\
185	+ (ADD_FILL CC_FieldUpgradeData)	/* 0x00000141 */	\
186	+ (ADD_FILL CC_IncrementalSelfTest)	/* 0x00000142 */	\
187	+ (ADD_FILL CC_SelfTest)	/* 0x00000143 */	\
188	+ (ADD_FILL CC_Startup)	/* 0x00000144 */	\
189	+ (ADD_FILL CC_Shutdown)	/* 0x00000145 */	\
190	+ (ADD_FILL CC_StirRandom)	/* 0x00000146 */	\
191	+ (ADD_FILL CC_ActivateCredential)	/* 0x00000147 */	\
192	+ (ADD_FILL CC_Certify)	/* 0x00000148 */	\
193	+ (ADD_FILL CC_PolicyNV)	/* 0x00000149 */	\
194	+ (ADD_FILL CC_CertifyCreation)	/* 0x0000014A */	\
195	+ (ADD_FILL CC_Duplicate)	/* 0x0000014B */	\
196	+ (ADD_FILL CC_GetTime)	/* 0x0000014C */	\
197	+ (ADD_FILL CC_GetSessionAuditDigest)	/* 0x0000014D */	\
198	+ (ADD_FILL CC_NV_Read)	/* 0x0000014E */	\
199	+ (ADD_FILL CC_NV_ReadLock)	/* 0x0000014F */	\
200	+ (ADD_FILL CC_ObjectChangeAuth)	/* 0x00000150 */	\
201	+ (ADD_FILL CC_PolicySecret)	/* 0x00000151 */	\
202	+ (ADD_FILL CC_Rewrap)	/* 0x00000152 */	\
203	+ (ADD_FILL CC_Create)	/* 0x00000153 */	\
204	+ (ADD_FILL CC_ECDH_ZGen)	/* 0x00000154 */	\
205	+ (ADD_FILL CC_HMAC CC_MAC)	/* 0x00000155 */	\
206	+ (ADD_FILL CC_Import)	/* 0x00000156 */	\
207	+ (ADD_FILL CC_Load)	/* 0x00000157 */	\
208	+ (ADD_FILL CC_Quote)	/* 0x00000158 */	\
209	+ (ADD_FILL CC_RSA_Decrypt)	/* 0x00000159 */	\
210	+ ADD_FILL	/* 0x0000015A */	\
211	+ (ADD_FILL CC_HMAC_Start CC_MAC_Start)	/* 0x0000015B */	\
212	+ (ADD_FILL CC_SequenceUpdate)	/* 0x0000015C */	\
213	+ (ADD_FILL CC_Sign)	/* 0x0000015D */	\
214	+ (ADD_FILL CC_Unseal)	/* 0x0000015E */	\
215	+ ADD_FILL	/* 0x0000015F */	\
216	+ (ADD_FILL CC_PolicySigned)	/* 0x00000160 */	\
217	+ (ADD_FILL CC_ContextLoad)	/* 0x00000161 */	\
218	+ (ADD_FILL CC_ContextSave)	/* 0x00000162 */	\
219	+ (ADD_FILL CC_ECDH_KeyGen)	/* 0x00000163 */	\
220	+ (ADD_FILL CC_EncryptDecrypt)	/* 0x00000164 */	\
221	+ (ADD_FILL CC_FlushContext)	/* 0x00000165 */	\
222	+ ADD_FILL	/* 0x00000166 */	\
223	+ (ADD_FILL CC_LoadExternal)	/* 0x00000167 */	\
224	+ (ADD_FILL CC_MakeCredential)	/* 0x00000168 */	\
225	+ (ADD_FILL CC_NV_ReadPublic)	/* 0x00000169 */	\
226	+ (ADD_FILL CC_PolicyAuthorize)	/* 0x0000016A */	\
227	+ (ADD_FILL CC_PolicyAuthValue)	/* 0x0000016B */	\
228	+ (ADD_FILL CC_PolicyCommandCode)	/* 0x0000016C */	\
229	+ (ADD_FILL CC_PolicyCounterTimer)	/* 0x0000016D */	\
230	+ (ADD_FILL CC_PolicyCpHash)	/* 0x0000016E */	\
231	+ (ADD_FILL CC_PolicyLocality)	/* 0x0000016F */	\
232	+ (ADD_FILL CC_PolicyNameHash)	/* 0x00000170 */	\
233	+ (ADD_FILL CC_PolicyOR)	/* 0x00000171 */	\
234	+ (ADD_FILL CC_PolicyTicket)	/* 0x00000172 */	\
235	+ (ADD_FILL CC_ReadPublic)	/* 0x00000173 */	\
236	+ (ADD_FILL CC_RSA_Encrypt)	/* 0x00000174 */	\


```

237 + ADD_FILL /* 0x00000175 */ \
238 + (ADD_FILL || CC_StartAuthSession) /* 0x00000176 */ \
239 + (ADD_FILL || CC_VerifySignature) /* 0x00000177 */ \
240 + (ADD_FILL || CC_ECC_Parameters) /* 0x00000178 */ \
241 + (ADD_FILL || CC_FirmwareRead) /* 0x00000179 */ \
242 + (ADD_FILL || CC_GetCapability) /* 0x0000017A */ \
243 + (ADD_FILL || CC_GetRandom) /* 0x0000017B */ \
244 + (ADD_FILL || CC_GetTestResult) /* 0x0000017C */ \
245 + (ADD_FILL || CC_Hash) /* 0x0000017D */ \
246 + (ADD_FILL || CC_PCR_Read) /* 0x0000017E */ \
247 + (ADD_FILL || CC_PolicyPCR) /* 0x0000017F */ \
248 + (ADD_FILL || CC_PolicyRestart) /* 0x00000180 */ \
249 + (ADD_FILL || CC_ReadClock) /* 0x00000181 */ \
250 + (ADD_FILL || CC_PCR_Extend) /* 0x00000182 */ \
251 + (ADD_FILL || CC_PCR_SetAuthValue) /* 0x00000183 */ \
252 + (ADD_FILL || CC_NV_Certify) /* 0x00000184 */ \
253 + (ADD_FILL || CC_EventSequenceComplete) /* 0x00000185 */ \
254 + (ADD_FILL || CC_HashSequenceStart) /* 0x00000186 */ \
255 + (ADD_FILL || CC_PolicyPhysicalPresence) /* 0x00000187 */ \
256 + (ADD_FILL || CC_PolicyDuplicationSelect) /* 0x00000188 */ \
257 + (ADD_FILL || CC_PolicyGetDigest) /* 0x00000189 */ \
258 + (ADD_FILL || CC_TestParms) /* 0x0000018A */ \
259 + (ADD_FILL || CC_Commit) /* 0x0000018B */ \
260 + (ADD_FILL || CC_PolicyPassword) /* 0x0000018C */ \
261 + (ADD_FILL || CC_ZGen_2Phase) /* 0x0000018D */ \
262 + (ADD_FILL || CC_EC_Ephemeral) /* 0x0000018E */ \
263 + (ADD_FILL || CC_PolicyNvWritten) /* 0x0000018F */ \
264 + (ADD_FILL || CC_PolicyTemplate) /* 0x00000190 */ \
265 + (ADD_FILL || CC_CreateLoaded) /* 0x00000191 */ \
266 + (ADD_FILL || CC_PolicyAuthorizeNV) /* 0x00000192 */ \
267 + (ADD_FILL || CC_EncryptDecrypt2) /* 0x00000193 */ \
268 + (ADD_FILL || CC_AC_GetCapability) /* 0x00000194 */ \
269 + (ADD_FILL || CC_AC_Send) /* 0x00000195 */ \
270 + (ADD_FILL || CC_Policy_AC_SendSelect) /* 0x00000196 */ \
271 + (ADD_FILL || CC_CertifyX509) /* 0x00000197 */ \
272 + (ADD_FILL || CC_ACT_SetTimeout) /* 0x00000198 */ \
273 + (ADD_FILL || CC_ECC_Encrypt) /* 0x00000199 */ \
274 + (ADD_FILL || CC_ECC_Decrypt) /* 0x0000019A */ \
275 ) \
276
277 #define VENDOR_COMMAND_ARRAY_SIZE (0 + CC_Vendor_TCG_Test)
278
279 #define COMMAND_COUNT (LIBRARY_COMMAND_ARRAY_SIZE + VENDOR_COMMAND_ARRAY_SIZE)
280
281 #define HASH_COUNT \
282 (ALG_SHA1 + ALG_SHA256 + ALG_SHA384 + ALG_SHA3_256 + \
283 ALG_SHA3_384 + ALG_SHA3_512 + ALG_SHA512 + ALG_SM3_256) \
284
285 #define MAX_HASH_BLOCK_SIZE \
286 (MAX(ALG_SHA1 * SHA1_BLOCK_SIZE, \
287 MAX(ALG_SHA256 * SHA256_BLOCK_SIZE, \
288 MAX(ALG_SHA384 * SHA384_BLOCK_SIZE, \
289 MAX(ALG_SHA3_256 * SHA3_256_BLOCK_SIZE, \
290 MAX(ALG_SHA3_384 * SHA3_384_BLOCK_SIZE, \
291 MAX(ALG_SHA3_512 * SHA3_512_BLOCK_SIZE, \
292 MAX(ALG_SHA512 * SHA512_BLOCK_SIZE, \
293 MAX(ALG_SM3_256 * SM3_256_BLOCK_SIZE, \
294 0)))))) \
295
296 #define MAX_DIGEST_SIZE \
297 (MAX(ALG_SHA1 * SHA1_DIGEST_SIZE, \
298 MAX(ALG_SHA256 * SHA256_DIGEST_SIZE, \
299 MAX(ALG_SHA384 * SHA384_DIGEST_SIZE, \
300 MAX(ALG_SHA3_256 * SHA3_256_DIGEST_SIZE, \
301 MAX(ALG_SHA3_384 * SHA3_384_DIGEST_SIZE, \
302 MAX(ALG_SHA3_512 * SHA3_512_DIGEST_SIZE, \

```

```

303         MAX(ALG_SHA512 * SHA512_DIGEST_SIZE, \
304         MAX(ALG_SM3_256 * SM3_256_DIGEST_SIZE, \
305         0))))))
306
307 #if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
308 #error "Hash data not valid"
309 #endif

```

Define the 2B structure that would hold any hash block

```

310 TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);

```

Following typedef is for some old code

```

311 typedef TPM2B_MAX_HASH_BLOCK TPM2B_HASH_BLOCK;
312
313 #define MAX_SYM_KEY_BITS \
314     (MAX(AES_MAX_KEY_SIZE_BITS, MAX(CAMELLIA_MAX_KEY_SIZE_BITS, \
315     MAX(SM4_MAX_KEY_SIZE_BITS, MAX(TDES_MAX_KEY_SIZE_BITS, \
316     0))))
317
318 #define MAX_SYM_KEY_BYTES ((MAX_SYM_KEY_BITS + 7) / 8)
319
320 #define MAX_SYM_BLOCK_SIZE \
321     (MAX(AES_MAX_BLOCK_SIZE, MAX(CAMELLIA_MAX_BLOCK_SIZE, \
322     MAX(SM4_MAX_BLOCK_SIZE, MAX(TDES_MAX_BLOCK_SIZE, \
323     0))))
324
325 #if MAX_SYM_KEY_BITS == 0 || MAX_SYM_BLOCK_SIZE == 0
326 # error Bad size for MAX_SYM_KEY_BITS or MAX_SYM_BLOCK
327 #endif
328
329 #endif // _TPM_ALGORITHM_DEFINES_H_

```

10.2 Source

10.2.1 AlgorithmTests.c

10.2.1.1 Introduction

This file contains the code to perform the various self-test functions.

NOTE In this implementation, large local variables are made static to minimize stack usage, which is critical for stack-constrained platforms.

10.2.1.2 Includes and Defines

```
1  #include    "Tpm.h"
2
3  #define     SELF_TEST_DATA
4
5  #if SELF_TEST
```

These includes pull in the data structures. They contain data definitions for the various tests.

```
6  #include    "SelfTest.h"
7  #include    "SymmetricTest.h"
8  #include    "RsaTestData.h"
9  #include    "EccTestData.h"
10 #include    "HashTestData.h"
11 #include    "KdfTestData.h"
12
```

```
13 #define TEST_DEFAULT_TEST_HASH(vector)           \
14     if(TEST_BIT(DEFAULT_TEST_HASH, g_toTest))    \
15         TestHash(DEFAULT_TEST_HASH, vector);
```

Make sure that the algorithm has been tested

```
16 #define CLEAR_BOTH(alg)    {    CLEAR_BIT(alg, *toTest);           \
17                             if(toTest != &g_toTest)              \
18                                 CLEAR_BIT(alg, g_toTest); }        \
19 #define SET_BOTH(alg)     {    SET_BIT(alg, *toTest);              \
20                             if(toTest != &g_toTest)              \
21                                 SET_BIT(alg, g_toTest); }          \
22 #define TEST_BOTH(alg)    ((toTest != &g_toTest)                  \
23                             ? TEST_BIT(alg, *toTest) || TEST_BIT(alg, g_toTest) \
24                             : TEST_BIT(alg, *toTest))
```

Can only cancel if doing a list.

```
25 #define CHECK_CANCELED                                         \
26     if(_plat__IsCanceled() && toTest != &g_toTest)           \
27         return TPM_RC_CANCELED;
```

10.2.1.3 Hash Tests

10.2.1.3.1 Description

The hash test does a known-value HMAC using the specified hash algorithm.

10.2.1.3.2 TestHash()

The hash test function.

```

28 static TPM_RC
29 TestHash(
30     TPM_ALG_ID      hashAlg,
31     ALGORITHM_VECTOR *toTest
32 )
33 {
34     static TPM2B_DIGEST    computed; // value computed
35     static HMAC_STATE      state;
36     UINT16                 digestSize;
37     const TPM2B             *testDigest = NULL;
38     // TPM2B_TYPE(HMAC_BLOCK, DEFAULT_TEST_HASH_BLOCK_SIZE);
39
40     pAssert(hashAlg != TPM_ALG_NULL);
41     #define HASH_CASE_FOR_TEST(HASH, hash)      case ALG_##HASH##_VALUE: \
42                                                  testDigest = &c_##HASH##_digest.b; \
43                                                  break;
44     switch(hashAlg)
45     {
46         FOR_EACH_HASH(HASH_CASE_FOR_TEST)
47
48         default:
49             FAIL(FATAL_ERROR_INTERNAL);
50     }
51     // Clear the to-test bits
52     CLEAR_BOTH(hashAlg);
53
54     // If there is an algorithm without test vectors, then assume that things are OK.
55     if(testDigest == NULL || testDigest->size == 0)
56         return TPM_RC_SUCCESS;
57
58     // Set the HMAC key to twice the digest size
59     digestSize = CryptHashGetDigestSize(hashAlg);
60     CryptHmacStart(&state, hashAlg, digestSize * 2,
61                   (BYTE *)c_hashTestKey.t.buffer);
62     CryptDigestUpdate(&state.hashState, 2 * CryptHashGetBlockSize(hashAlg),
63                      (BYTE *)c_hashTestData.t.buffer);
64     computed.t.size = digestSize;
65     CryptHmacEnd(&state, digestSize, computed.t.buffer);
66     if((testDigest->size != computed.t.size)
67        || (memcmp(testDigest->buffer, computed.t.buffer, computed.b.size) != 0))
68         SELF_TEST_FAILURE;
69     return TPM_RC_SUCCESS;
70 }

```

10.2.1.4 Symmetric Test Functions

10.2.1.4.1 MakeIv()

Internal function to make the appropriate IV depending on the mode.

```

71 static UINT32
72 MakeIv(
73     TPM_ALG_ID    mode, // IN: symmetric mode
74     UINT32        size, // IN: block size of the algorithm
75     BYTE          *iv   // OUT: IV to fill in
76 )
77 {
78     BYTE    i;
79 }

```

```

80     if(mode == TPM_ALG_ECB)
81         return 0;
82     if(mode == TPM_ALG_CTR)
83     {
84         // The test uses an IV that has 0xff in the last byte
85         for(i = 1; i <= size; i++)
86             *iv++ = 0xff - (BYTE)(size - i);
87     }
88     else
89     {
90         for(i = 0; i < size; i++)
91             *iv++ = i;
92     }
93     return size;
94 }

```

10.2.1.4.2 TestSymmetricAlgorithm()

Function to test a specific algorithm, key size, and mode.

```

95 static void
96 TestSymmetricAlgorithm(
97     const SYMMETRIC_TEST_VECTOR *test, //
98     TPM_ALG_ID mode //
99 )
100 {
101     static BYTE encrypted[MAX_SYM_BLOCK_SIZE * 2];
102     static BYTE decrypted[MAX_SYM_BLOCK_SIZE * 2];
103     static TPM2B_IV iv;
104     //
105     // Get the appropriate IV
106     iv.t.size = (UINT16)MakeIv(mode, test->ivSize, iv.t.buffer);
107
108     // Encrypt known data
109     CryptSymmetricEncrypt(encrypted, test->alg, test->keyBits, test->key, &iv,
110                          mode, test->dataInOutSize, test->dataIn);
111     // Check that it matches the expected value
112     if(!MemoryEqual(encrypted, test->dataOut[mode - TPM_ALG_CTR],
113                    test->dataInOutSize))
114         SELF_TEST_FAILURE;
115     // Reinitialize the iv for decryption
116     MakeIv(mode, test->ivSize, iv.t.buffer);
117     CryptSymmetricDecrypt(decrypted, test->alg, test->keyBits, test->key, &iv,
118                          mode, test->dataInOutSize,
119                          test->dataOut[mode - TPM_ALG_CTR]);
120     // Make sure that it matches what we started with
121     if(!MemoryEqual(decrypted, test->dataIn, test->dataInOutSize))
122         SELF_TEST_FAILURE;
123 }

```

10.2.1.4.3 AllSymsAreDone()

Checks if both symmetric algorithms have been tested. This is put here so that addition of a symmetric algorithm will be relatively easy to handle.

Return Value	Meaning
TRUE(1)	all symmetric algorithms tested
FALSE(0)	not all symmetric algorithms tested

```

124 static BOOL
125 AllSymsAreDone(

```

```

126     ALGORITHM_VECTOR      *toTest
127 )
128 {
129     return (!TEST_BOTH(TPM_ALG_AES) && !TEST_BOTH(TPM_ALG_SM4));
130 }

```

10.2.1.4.4 AllModesAreDone()

Checks if all the modes have been tested.

Return Value	Meaning
TRUE(1)	all modes tested
FALSE(0)	all modes not tested

```

131 static BOOL
132 AllModesAreDone(
133     ALGORITHM_VECTOR      *toTest
134 )
135 {
136     TPM_ALG_ID            alg;
137     for(alg = SYM_MODE_FIRST; alg <= SYM_MODE_LAST; alg++)
138         if(TEST_BOTH(alg))
139             return FALSE;
140     return TRUE;
141 }

```

10.2.1.4.5 TestSymmetric()

If *alg* is a symmetric block cipher, then all of the modes that are selected are tested. If *alg* is a mode, then all algorithms of that mode are tested.

```

142 static TPM_RC
143 TestSymmetric(
144     TPM_ALG_ID            alg,
145     ALGORITHM_VECTOR      *toTest
146 )
147 {
148     SYM_INDEX             index;
149     TPM_ALG_ID            mode;
150     //
151     if(!TEST_BIT(alg, *toTest))
152         return TPM_RC_SUCCESS;
153     if(alg == TPM_ALG_AES || alg == TPM_ALG_SM4 || alg == TPM_ALG_CAMELLIA)
154     {
155         // Will test the algorithm for all modes and key sizes
156         CLEAR_BOTH(alg);
157
158         // A test this algorithm for all modes
159         for(index = 0; index < NUM_SYMS; index++)
160         {
161             if(c_symTestValues[index].alg == alg)
162             {
163                 for(mode = SYM_MODE_FIRST;
164                     mode <= SYM_MODE_LAST;
165                     mode++)
166                 {
167                     if(TEST_BIT(mode, *toTest))
168                         TestSymmetricAlgorithm(&c_symTestValues[index], mode);
169                 }
170             }
171         }

```

```

172     // if all the symmetric tests are done
173     if(AllSymsAreDone(toTest))
174     {
175         // all symmetric algorithms tested so no modes should be set
176         for(alg = SYM_MODE_FIRST; alg <= SYM_MODE_LAST; alg++)
177             CLEAR_BOTH(alg);
178     }
179 }
180 else if(SYM_MODE_FIRST <= alg && alg <= SYM_MODE_LAST)
181 {
182     // Test this mode for all key sizes and algorithms
183     for(index = 0; index < NUM_SYMS; index++)
184     {
185         // The mode testing only comes into play when doing self tests
186         // by command. When doing self tests by command, the block ciphers are
187         // tested first. That means that all of their modes would have been
188         // tested for all key sizes. If there is no block cipher left to
189         // test, then clear this mode bit.
190         if(!TEST_BIT(TPM_ALG_AES, *toTest)
191             && !TEST_BIT(TPM_ALG_SM4, *toTest))
192         {
193             CLEAR_BOTH(alg);
194         }
195         else
196         {
197             for(index = 0; index < NUM_SYMS; index++)
198             {
199                 if(TEST_BIT(c_symTestValues[index].alg, *toTest))
200                     TestSymmetricAlgorithm(&c_symTestValues[index], alg);
201             }
202             // have tested this mode for all algorithms
203             CLEAR_BOTH(alg);
204         }
205     }
206     if(AllModesAreDone(toTest))
207     {
208         CLEAR_BOTH(TPM_ALG_AES);
209         CLEAR_BOTH(TPM_ALG_SM4);
210     }
211 }
212 else
213     pAssert(alg == 0 && alg != 0);
214 return TPM_RC_SUCCESS;
215 }

```

10.2.1.5 RSA Tests

```
216 #if ALG_RSA
```

10.2.1.5.1 Introduction

The tests are for public key only operations and for private key operations. Signature verification and encryption are public key operations. They are tested by using a KVT. For signature verification, this means that a known good signature is checked by `CryptRsaValidateSignature()`. If it fails, then the TPM enters failure mode. For encryption, the TPM encrypts known values using the selected scheme and checks that the returned value matches the expected value.

For private key operations, a full scheme check is used. For a signing key, a known key is used to sign a known message. Then that signature is verified. since the signature may involve use of random values, the signature will be different each time and we can't always check that the signature matches a known value. The same technique is used for decryption (RSADP/RSAEP).

When an operation uses the public key and the verification has not been tested, the TPM will do a KVT.

The test for the signing algorithm is built into the call for the algorithm

10.2.1.5.2 RsaKeyInitialize()

The test key is defined by a public modulus and a private prime. The TPM's RSA code computes the second prime and the private exponent.

```

217 static void
218 RsaKeyInitialize(
219     OBJECT          *testObject
220 )
221 {
222     MemoryCopy2B(&testObject->publicArea.unique.rsa.b, (P2B)&c_rsaPublicModulus,
223                 sizeof(c_rsaPublicModulus));
224     MemoryCopy2B(&testObject->sensitive.sensitive.rsa.b, (P2B)&c_rsaPrivatePrime,
225                 sizeof(testObject->sensitive.sensitive.rsa.t.buffer));
226     testObject->publicArea.parameters.rsaDetail.keyBits = RSA_TEST_KEY_SIZE * 8;
227     // Use the default exponent
228     testObject->publicArea.parameters.rsaDetail.exponent = 0;
229 }

```

10.2.1.5.3 TestRsaEncryptDecrypt()

These tests are for a public key encryption that uses a random value.

```

230 static TPM_RC
231 TestRsaEncryptDecrypt(
232     TPM_ALG_ID      scheme,           // IN: the scheme
233     ALGORITHM_VECTOR *toTest         //
234 )
235 {
236     static TPM2B_PUBLIC_KEY_RSA testInput;
237     static TPM2B_PUBLIC_KEY_RSA testOutput;
238     static OBJECT testObject;
239     const TPM2B_RSA_TEST_KEY *kvtValue = NULL;
240     TPM_RC result = TPM_RC_SUCCESS;
241     const TPM2B *testLabel = NULL;
242     TPMT_RSA_DECRYPT rsaScheme;
243     //
244     // Don't need to initialize much of the test object
245     RsaKeyInitialize(&testObject);
246     rsaScheme.scheme = scheme;
247     rsaScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
248     CLEAR_BOTH(scheme);
249     CLEAR_BOTH(TPM_ALG_NULL);
250     if(scheme == TPM_ALG_NULL)
251     {
252         // This is an encryption scheme using the private key without any encoding.
253         memcpy(testInput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue));
254         testInput.t.size = sizeof(c_RsaTestValue);
255         if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
256                                             &testObject, &rsaScheme, NULL, NULL))
257             SELF_TEST_FAILURE;
258         if(!MemoryEqual(testOutput.t.buffer, c_RsaepKvt.buffer, c_RsaepKvt.size))
259             SELF_TEST_FAILURE;
260         MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
261         if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
262                                             &testObject, &rsaScheme, NULL))
263             SELF_TEST_FAILURE;
264         if(!MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
265                         sizeof(c_RsaTestValue)))
266             SELF_TEST_FAILURE;

```

```

267     }
268     else
269     {
270         // TPM_ALG_RSAES:
271         // This is an decryption scheme using padding according to
272         // PKCS#1v2.1, 7.2. This padding uses random bits. To test a public
273         // key encryption that uses random data, encrypt a value and then
274         // decrypt the value and see that we get the encrypted data back.
275         // The hash is not used by this encryption so it can be TPM_ALG_NULL
276
277         // TPM_ALG_OAEP:
278         // This is also an decryption scheme and it also uses a
279         // pseudo-random
280         // value. However, this also uses a hash algorithm. So, we may need
281         // to test that algorithm before use.
282         if(scheme == TPM_ALG_OAEP)
283         {
284             TEST_DEFAULT_TEST_HASH(toTest);
285             kvtValue = &c_OaepKvt;
286             testLabel = OAEP_TEST_STRING;
287         }
288         else if(scheme == TPM_ALG_RSAES)
289         {
290             kvtValue = &c_RsaesKvt;
291             testLabel = NULL;
292         }
293         else
294             SELF_TEST_FAILURE;
295         // Only use a digest-size portion of the test value
296         memcpy(testInput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
297         testInput.t.size = DEFAULT_TEST_DIGEST_SIZE;
298
299         // See if the encryption works
300         if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
301                                             &testObject, &rsaScheme, testLabel,
302                                             NULL))
303             SELF_TEST_FAILURE;
304         MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
305         // see if we can decrypt this value and get the original data back
306         if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
307                                             &testObject, &rsaScheme, testLabel))
308             SELF_TEST_FAILURE;
309         // See if the results compare
310         if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
311            || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
312                            DEFAULT_TEST_DIGEST_SIZE))
313             SELF_TEST_FAILURE;
314         // Now check that the decryption works on a known value
315         MemoryCopy2B(&testInput.b, (P2B)kvtValue,
316                     sizeof(testInput.t.buffer));
317         if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
318                                             &testObject, &rsaScheme, testLabel))
319             SELF_TEST_FAILURE;
320         if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
321            || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
322                            DEFAULT_TEST_DIGEST_SIZE))
323             SELF_TEST_FAILURE;
324     }
325     return result;
326 }

```

10.2.1.5.4 TestRsaSignAndVerify()

This function does the testing of the RSA sign and verification functions. This test does a KVT.

```

327 static TPM_RC
328 TestRsaSignAndVerify(
329     TPM_ALG_ID          scheme,
330     ALGORITHM_VECTOR    *toTest
331 )
332 {
333     TPM_RC          result = TPM_RC_SUCCESS;
334     static OBJECT    testObject;
335     static TPM2B_DIGEST testDigest;
336     static TPMT_SIGNATURE testSig;
337
338     // Do a sign and signature verification.
339     // RSASSA:
340     // This is a signing scheme according to PKCS#1-v2.1 8.2. It does not
341     // use random data so there is a KVT for the signing operation. On
342     // first use of the scheme for signing, use the TPM's RSA key to
343     // sign a portion of c_RsaTestData and compare the results to c_RsassaKvt. Then
344     // decrypt the data to see that it matches the starting value. This verifies
345     // the signature with a KVT
346
347     // Clear the bits indicating that the function has not been checked. This is to
348     // prevent looping
349     CLEAR_BOTH(scheme);
350     CLEAR_BOTH(TPM_ALG_NULL);
351     CLEAR_BOTH(TPM_ALG_RSA);
352
353     RsaKeyInitialize(&testObject);
354     memcpy(testDigest.t.buffer, (BYTE *)c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
355     testDigest.t.size = DEFAULT_TEST_DIGEST_SIZE;
356     testSig.sigAlg = scheme;
357     testSig.signature.rsapss.hash = DEFAULT_TEST_HASH;
358
359     // RSAPSS:
360     // This is a signing scheme according to PKCS#1-v2.2 8.1 it uses
361     // random data in the signature so there is no KVT for the signing
362     // operation. To test signing, the TPM will use the TPM's RSA key
363     // to sign a portion of c_RsaTestValue and then it will verify the
364     // signature. For verification, c_RsapssKvt is verified before the
365     // user signature blob is verified. The worst case for testing of this
366     // algorithm is two private and one public key operation.
367
368     // The process is to sign known data. If RSASSA is being done, verify that the
369     // signature matches the precomputed value. For both, use the signed value and
370     // see that the verification says that it is a good signature. Then
371     // if testing RSAPSS, do a verify of a known good signature. This ensures that
372     // the validation function works.
373
374     if(TPM_RC_SUCCESS != CryptRsaSign(&testSig, &testObject, &testDigest, NULL))
375         SELF_TEST_FAILURE;
376     // For RSASSA, make sure the results is what we are looking for
377     if(testSig.sigAlg == TPM_ALG_RSASSA)
378     {
379         if(testSig.signature.rsassa.sig.t.size != RSA_TEST_KEY_SIZE
380            || !MemoryEqual(c_RsassaKvt.buffer,
381                           testSig.signature.rsassa.sig.t.buffer,
382                           RSA_TEST_KEY_SIZE))
383             SELF_TEST_FAILURE;
384     }
385     // See if the TPM will validate its own signatures
386     if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
387                                                    &testDigest))
388         SELF_TEST_FAILURE;
389     // If this is RSAPSS, check the verification with known signature
390     // Have to copy because CryptRsaValidateSignature() eats the signature
391     if(TPM_ALG_RSAPSS == scheme)
392     {

```

```

393     MemoryCopy2B(&testSig.signature.rsapss.sig.b, (P2B)&c_RsapssKvt,
394                 sizeof(testSig.signature.rsapss.sig.t.buffer));
395     if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
396                                                  &testDigest))
397         SELF_TEST_FAILURE;
398 }
399 return result;
400 }

```

10.2.1.5.5 TestRSA()

Function uses the provided vector to indicate which tests to run. It will clear the vector after each test is run and also clear *g_toTest*

```

401 static TPM_RC
402 TestRsa(
403     TPM_ALG_ID      alg,
404     ALGORITHM_VECTOR *toTest
405 )
406 {
407     TPM_RC      result = TPM_RC_SUCCESS;
408     //
409     switch(alg)
410     {
411         case TPM_ALG_NULL:
412             // This is the RSAEP/RSADP function. If we are processing a list, don't
413             // need to test these now because any other test will validate
414             // RSAEP/RSADP. Can tell this is list of test by checking to see if
415             // 'toTest' is pointing at g_toTest. If so, this is an isolated test
416             // an need to go ahead and do the test;
417             if((toTest == &g_toTest)
418                 || (!TEST_BIT(TPM_ALG_RSASSA, *toTest)
419                     && !TEST_BIT(TPM_ALG_RSAES, *toTest)
420                     && !TEST_BIT(TPM_ALG_RSAPSS, *toTest)
421                     && !TEST_BIT(TPM_ALG_OAEP, *toTest)))
422                 // Not running a list of tests or no other tests on the list
423                 // so run the test now
424                 result = TestRsaEncryptDecrypt(alg, toTest);
425             // if not running the test now, leave the bit on, just in case things
426             // get interrupted
427             break;
428         case TPM_ALG_OAEP:
429         case TPM_ALG_RSAES:
430             result = TestRsaEncryptDecrypt(alg, toTest);
431             break;
432         case TPM_ALG_RSAPSS:
433         case TPM_ALG_RSASSA:
434             result = TestRsaSignAndVerify(alg, toTest);
435             break;
436         default:
437             SELF_TEST_FAILURE;
438     }
439     return result;
440 }
441 #endif // ALG_RSA

```

10.2.1.6 ECC Tests

```

442 #if ALG_ECC

```

10.2.1.6.1 LoadEccParameter()

This function is mostly for readability and type checking

```

443 static void
444 LoadEccParameter(
445     TPM2B_ECC_PARAMETER      *to,          // target
446     const TPM2B_EC_TEST      *from         // source
447 )
448 {
449     MemoryCopy2B(&to->b, &from->b, sizeof(to->t.buffer));
450 }

```

10.2.1.6.2 LoadEccPoint()

```

451 static void
452 LoadEccPoint(
453     TPMS_ECC_POINT           *point,        // target
454     const TPM2B_EC_TEST      *x,           // source
455     const TPM2B_EC_TEST      *y
456 )
457 {
458     MemoryCopy2B(&point->x.b, (TPM2B *)x, sizeof(point->x.t.buffer));
459     MemoryCopy2B(&point->y.b, (TPM2B *)y, sizeof(point->y.t.buffer));
460 }

```

10.2.1.6.3 TestECDH()

This test does a KVT on a point multiply.

```

461 static TPM_RC
462 TestECDH(
463     TPM_ALG_ID               scheme,        // IN: for consistency
464     ALGORITHM_VECTOR         *toTest        // IN/OUT: modified after test is run
465 )
466 {
467     static TPMS_ECC_POINT     Z;
468     static TPMS_ECC_POINT     Qe;
469     static TPM2B_ECC_PARAMETER ds;
470     TPM_RC                    result = TPM_RC_SUCCESS;
471     //
472     NOT_REFERENCED(scheme);
473     CLEAR_BOTH(TPM_ALG_ECDH);
474     LoadEccParameter(&ds, &c_ecTestKey_ds);
475     LoadEccPoint(&Qe, &c_ecTestKey_QeX, &c_ecTestKey_QeY);
476     if(TPM_RC_SUCCESS != CryptEccPointMultiply(&Z, c_testCurve, &Qe, &ds,
477                                                NULL, NULL))
478         SELF_TEST_FAILURE;
479     if(!MemoryEqual2B(&c_ecTestEcdh_X.b, &Z.x.b)
480        || !MemoryEqual2B(&c_ecTestEcdh_Y.b, &Z.y.b))
481         SELF_TEST_FAILURE;
482     return result;
483 }

```

10.2.1.6.4 TestEccSignAndVerify()

```

484 static TPM_RC
485 TestEccSignAndVerify(
486     TPM_ALG_ID               scheme,
487     ALGORITHM_VECTOR         *toTest
488 )

```

```

489 {
490     static OBJECT                testObject;
491     static TPMT_SIGNATURE        testSig;
492     static TPMT_ECC_SCHEME       eccScheme;
493
494     testSig.sigAlg = scheme;
495     testSig.signature.ecdsa.hash = DEFAULT_TEST_HASH;
496
497     eccScheme.scheme = scheme;
498     eccScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
499
500     CLEAR_BOTH(scheme);
501     CLEAR_BOTH(TPM_ALG_ECDH);
502
503     // ECC signature verification testing uses a KVT.
504     switch(scheme)
505     {
506         case TPM_ALG_ECDSA:
507             LoadEccParameter(&testSig.signature.ecdsa.signatureR, &c_TestEcDsa_r);
508             LoadEccParameter(&testSig.signature.ecdsa.signatureS, &c_TestEcDsa_s);
509             break;
510         case TPM_ALG_ECSCHNORR:
511             LoadEccParameter(&testSig.signature.ecschnorr.signatureR,
512                             &c_TestEcSchnorr_r);
513             LoadEccParameter(&testSig.signature.ecschnorr.signatureS,
514                             &c_TestEcSchnorr_s);
515             break;
516         case TPM_ALG_SM2:
517             // don't have a test for SM2
518             return TPM_RC_SUCCESS;
519         default:
520             SELF_TEST_FAILURE;
521             break;
522     }
523     TEST_DEFAULT_TEST_HASH(toTest);
524
525     // Have to copy the key. This is because the size used in the test vectors
526     // is the size of the ECC parameter for the test key while the size of a point
527     // is TPM dependent
528     MemoryCopy2B(&testObject.sensitive.sensitive.ecc.b, &c_ecTestKey_ds.b,
529                 sizeof(testObject.sensitive.sensitive.ecc.t.buffer));
530     LoadEccPoint(&testObject.publicArea.unique.ecc, &c_ecTestKey_QsX,
531                 &c_ecTestKey_QsY);
532     testObject.publicArea.parameters.eccDetail.curveID = c_testCurve;
533
534     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
535                                                    (TPM2B_DIGEST *)&c_ecTestValue.b))
536     {
537         SELF_TEST_FAILURE;
538     }
539     CHECK_CANCELED;
540
541     // Now sign and verify some data
542     if(TPM_RC_SUCCESS != CryptEccSign(&testSig, &testObject,
543                                       (TPM2B_DIGEST *)&c_ecTestValue,
544                                       &eccScheme, NULL))
545     {
546         SELF_TEST_FAILURE;
547     }
548     CHECK_CANCELED;
549
550     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
551                                                    (TPM2B_DIGEST *)&c_ecTestValue))
552     {
553         SELF_TEST_FAILURE;
554     }
555     CHECK_CANCELED;
556

```

```

555     return TPM_RC_SUCCESS;
556 }

```

10.2.1.6.5 TestKDFa()

```

557 static TPM_RC
558 TestKDFa(
559     ALGORITHM_VECTOR    *toTest
560 )
561 {
562     static TPM2B_KDF_TEST_KEY    keyOut;
563     UINT32                        counter = 0;
564     //
565     CLEAR_BOTH(TPM_ALG_KDF1_SP800_108);
566
567     keyOut.t.size = CryptKDFa(KDF_TEST_ALG, &c_kdfTestKeyIn.b, &c_kdfTestLabel.b,
568                             &c_kdfTestContextU.b, &c_kdfTestContextV.b,
569                             TEST_KDF_KEY_SIZE * 8, keyOut.t.buffer,
570                             &counter, FALSE);
571     if ( keyOut.t.size != TEST_KDF_KEY_SIZE
572         || !MemoryEqual(keyOut.t.buffer, c_kdfTestKeyOut.t.buffer,
573                         TEST_KDF_KEY_SIZE))
574         SELF_TEST_FAILURE;
575
576     return TPM_RC_SUCCESS;
577 }

```

10.2.1.6.6 TestEcc()

```

578 static TPM_RC
579 TestEcc(
580     TPM_ALG_ID            alg,
581     ALGORITHM_VECTOR    *toTest
582 )
583 {
584     TPM_RC                result = TPM_RC_SUCCESS;
585     NOT_REFERENCED(toTest);
586     switch(alg)
587     {
588         case TPM_ALG_ECC:
589         case TPM_ALG_ECDH:
590             // If this is in a loop then see if another test is going to deal with
591             // this.
592             // If toTest is not a self-test list
593             if((toTest == &q_toTest)
594                 // or this is the only ECC test in the list
595                 || !(TEST_BIT(TPM_ALG_ECDSA, *toTest)
596                     || TEST_BIT(ALG_EC Schnorr, *toTest)
597                     || TEST_BIT(TPM_ALG_SM2, *toTest)))
598             {
599                 result = TestECDH(alg, toTest);
600             }
601             break;
602         case TPM_ALG_ECDSA:
603         case TPM_ALG_EC Schnorr:
604         case TPM_ALG_SM2:
605             result = TestEccSignAndVerify(alg, toTest);
606             break;
607         default:
608             SELF_TEST_FAILURE;
609             break;
610     }
611     return result;
612 }

```



```
613 #endif // ALG_ECC
```

10.2.1.6.7 TestAlgorithm()

Dispatches to the correct test function for the algorithm or gets a list of testable algorithms.

If *toTest* is not NULL, then the test decisions are based on the algorithm selections in *toTest*. Otherwise, *g_toTest* is used. When bits are clear in *g_toTest* they will also be cleared *toTest*.

If there doesn't happen to be a test for the algorithm, its associated bit is quietly cleared.

If *alg* is zero (TPM_ALG_ERROR), then the *toTest* vector is cleared of any bits for which there is no test (i.e. no tests are actually run but the vector is cleared).

NOTE *toTest* will only ever have bits set for implemented algorithms but *alg* can be anything.

Error Return	Meaning
TPM_RC_CANCELED	test was canceled

```
614 LIB_EXPORT
615 TPM_RC(
616     TPM_ALG_ID          alg,
617     ALGORITHM_VECTOR    *toTest
618 )
619 {
620     TPM_ALG_ID          first = (alg == TPM_ALG_ERROR) ? TPM_ALG_FIRST : alg;
621     TPM_ALG_ID          last  = (alg == TPM_ALG_ERROR) ? TPM_ALG_LAST  : alg;
622     BOOL                doTest = (alg != TPM_ALG_ERROR);
623     TPM_RC               result = TPM_RC_SUCCESS;
624
625     if(toTest == NULL)
626         toTest = &g_toTest;
627
628     // This is kind of strange. This function will either run a test of the selected
629     // algorithm or just clear a bit if there is no test for the algorithm. So,
630     // either this loop will be executed once for the selected algorithm or once for
631     // each of the possible algorithms. If it is executed more than once ('alg' ==
632     // ALG_ERROR), then no test will be run but bits will be cleared for
633     // unimplemented algorithms. This was done this way so that there is only one
634     // case statement with all of the algorithms. It was easier to have one case
635     // statement than to have multiple ones to manage whenever an algorithm ID is
636     // added.
637     for(alg = first; (alg <= last); alg++)
638     {
639         // if 'alg' was TPM_ALG_ERROR, then we will be cycling through
640         // values, some of which may not be implemented. If the bit in toTest
641         // happens to be set, then we could either generated an assert, or just
642         // silently CLEAR it. Decided to just clear.
643         if(!TEST_BIT(alg, g_implementedAlgorithms))
644         {
645             CLEAR_BIT(alg, *toTest);
646             continue;
647         }
648         // Process whatever is left.
649         // NOTE: since this switch will only be called if the algorithm is
650         // implemented, it is not necessary to modify this list except to comment
651         // out the algorithms for which there is no test
652         switch(alg)
653         {
654             // Symmetric block ciphers
655 #if ALG_AES
656             case TPM_ALG_AES:
657 #endif // ALG_AES
```

```

658 #if ALG_SM4
659     // if SM4 is implemented, its test is like other block ciphers but there
660     // aren't any test vectors for it yet
661     // case TPM_ALG_SM4:
662 #endif // ALG_SM4
663 #if ALG_CAMELLIA
664     // no test vectors for camellia
665     // case TPM_ALG_CAMELLIA:
666 #endif
667     // Symmetric modes
668 #if !ALG_CFB
669 # error CFB is required in all TPM implementations
670 #endif // !ALG_CFB
671     case TPM_ALG_CFB:
672         if(doTest)
673             result = TestSymmetric(alg, toTest);
674         break;
675 #if ALG_CTR
676     case TPM_ALG_CTR:
677 #endif // ALG_CTR
678 #if ALG_OFB
679     case TPM_ALG_OFB:
680 #endif // ALG_OFB
681 #if ALG_CBC
682     case TPM_ALG_CBC:
683 #endif // ALG_CBC
684 #if ALG_ECB
685     case TPM_ALG_ECB:
686 #endif
687         if(doTest)
688             result = TestSymmetric(alg, toTest);
689         else
690             // If doing the initialization of g_toTest vector, only need
691             // to test one of the modes for the symmetric algorithms. If
692             // initializing for a SelfTest(FULL_TEST), allow all the modes.
693             if(toTest == &g_toTest)
694                 CLEAR_BIT(alg, *toTest);
695         break;
696 #if !ALG_HMAC
697 # error HMAC is required in all TPM implementations
698 #endif
699     case TPM_ALG_HMAC:
700         // Clear the bit that indicates that HMAC is required because
701         // HMAC is used as the basic test for all hash algorithms.
702         CLEAR_BOTH(alg);
703         // Testing HMAC means test the default hash
704         if(doTest)
705             TestHash(DEFAULT_TEST_HASH, toTest);
706         else
707             // If not testing, then indicate that the hash needs to be
708             // tested because this uses HMAC
709             SET_BOTH(DEFAULT_TEST_HASH);
710         break;
711 // Have to use two arguments for the macro even though only the first is used in the
712 // expansion.
713 #define HASH_CASE_TEST(HASH, hash) \
714     case ALG_##HASH##_VALUE:
715     FOR_EACH_HASH(HASH_CASE_TEST)
716 #undef HASH_CASE_TEST
717     if(doTest)
718         result = TestHash(alg, toTest);
719     break;
720 // RSA-dependent
721 #if ALG_RSA
722     case TPM_ALG_RSA:
723         CLEAR_BOTH(alg);

```

```

724         if(doTest)
725             result = TestRsa(TPM_ALG_NULL, toTest);
726         else
727             SET_BOTH(TPM_ALG_NULL);
728             break;
729         case TPM_ALG_RSASSA:
730         case TPM_ALG_RSAES:
731         case TPM_ALG_RSAPSS:
732         case TPM_ALG_OAEP:
733         case TPM_ALG_NULL: // used or RSADP
734             if(doTest)
735                 result = TestRsa(alg, toTest);
736             break;
737     #endif // ALG_RSA
738     #if ALG_KDF1_SP800_108
739         case TPM_ALG_KDF1_SP800_108:
740             if(doTest)
741                 result = TestKDFa(toTest);
742             break;
743     #endif // ALG_KDF1_SP800_108
744     #if ALG_ECC
745         // ECC dependent but no tests
746         // case TPM_ALG_ECDSA:
747         // case TPM_ALG_ECMQV:
748         // case TPM_ALG_KDF1_SP800_56a:
749         // case TPM_ALG_KDF2:
750         // case TPM_ALG_MGF1:
751         case TPM_ALG_ECC:
752             CLEAR_BOTH(alg);
753             if(doTest)
754                 result = TestEcc(TPM_ALG_ECDH, toTest);
755             else
756                 SET_BOTH(TPM_ALG_ECDH);
757             break;
758         case TPM_ALG_ECDSA:
759         case TPM_ALG_ECDH:
760         case TPM_ALG_ECSCHNORR:
761         // case TPM_ALG_SM2:
762             if(doTest)
763                 result = TestEcc(alg, toTest);
764             break;
765     #endif // ALG_ECC
766         default:
767             CLEAR_BIT(alg, *toTest);
768             break;
769     }
770     if(result != TPM_RC_SUCCESS)
771         break;
772 }
773 return result;
774 }
775 #endif // SELF_TESTS

```

10.2.2 BnConvert.c

10.2.2.1 Introduction

This file contains the basic conversion functions that will convert TPM2B to/from the internal format. The internal format is a *bigNum*,

10.2.2.2 Includes

```
1  #include "Tpm.h"
```

10.2.2.3 Functions

10.2.2.3.1 BnFromBytes()

This function will convert a big-endian byte array to the internal number format. If bn is NULL, then the output is NULL. If bytes is null or the required size is 0, then the output is set to zero

```
2  LIB_EXPORT bigNum
3  BnFromBytes(
4      bigNum          bn,
5      const BYTE      *bytes,
6      NUMBYTES        nBytes
7  )
8  {
9      const BYTE      *pFrom; // 'p' points to the least significant bytes of source
10     BYTE             *pTo;   // points to least significant bytes of destination
11     crypt_ushort_t   size;
12     //
13
14     size = (bytes != NULL) ? BYTES_TO_CRYPT_WORDS(nBytes) : 0;
15
16     // If nothing in, nothing out
17     if(bn == NULL)
18         return NULL;
19
20     // make sure things fit
21     pAssert(BnGetAllocated(bn) >= size);
22
23     if(size > 0)
24     {
25         // Clear the topmost word in case it is not filled with data
26         bn->d[size - 1] = 0;
27         // Moving the input bytes from the end of the list (LSB) end
28         pFrom = bytes + nBytes - 1;
29         // To the LS0 of the LSW of the bigNum.
30         pTo = (BYTE *)bn->d;
31         for(; nBytes != 0; nBytes--)
32             *pTo++ = *pFrom--;
33         // For a little-endian machine, the conversion is a straight byte
34         // reversal. For a big-endian machine, we have to put the words in
35         // big-endian byte order
36     #if BIG_ENDIAN_TPM
37         {
38             crypt_ushort_t   t;
39             for(t = (crypt_ushort_t)size - 1; t >= 0; t--)
40                 bn->d[t] = SWAP_CRYPT_WORD(bn->d[t]);
41         }
42     #endif
43     }
```

```

44     BnSetTop(bn, size);
45     return bn;
46 }

```

10.2.2.3.2 BnFrom2B()

Convert an TPM2B to a BIG_NUM. If the input value does not exist, or the output does not exist, or the input will not fit into the output the function returns NULL

```

47 LIB_EXPORT bigNum
48 BnFrom2B(
49     bigNum      bn,          // OUT:
50     const TPM2B *a2B        // IN: number to convert
51 )
52 {
53     if(a2B != NULL)
54         return BnFromBytes(bn, a2B->buffer, a2B->size);
55     // Make sure that the number has an initialized value rather than whatever
56     // was there before
57     BnSetTop(bn, 0); // Function accepts NULL
58     return NULL;
59 }

```

10.2.2.3.3 BnFromHex()

Convert a hex string into a *bigNum*. This is primarily used in debugging.

```

60 LIB_EXPORT bigNum
61 BnFromHex(
62     bigNum      bn,          // OUT:
63     const char  *hex         // IN:
64 )
65 {
66     #define FromHex(a) ((a) - (((a) > 'a') ? ('a' + 10) : ((a) > 'A') ? ('A' - 10) : '0')) \
67
68     unsigned    i;
69     unsigned    wordCount;
70     const char  *p;
71     BYTE        *d = (BYTE *) &(bn->d[0]);
72 //
73     pAssert(bn && hex);
74     i = (unsigned) strlen(hex);
75     wordCount = BYTES_TO_CRYPT_WORDS((i + 1) / 2);
76     if((i == 0) || (wordCount >= BnGetAllocated(bn)))
77         BnSetWord(bn, 0);
78     else
79     {
80         bn->d[wordCount - 1] = 0;
81         p = hex + i - 1;
82         for(; i > 1; i -= 2)
83         {
84             BYTE a;
85             a = FromHex(*p);
86             p--;
87             *d++ = a + (FromHex(*p) << 4);
88             p--;
89         }
90         if(i == 1)
91             *d = FromHex(*p);
92     }
93     #if !BIG_ENDIAN_TPM
94     for(i = 0; i < wordCount; i++)
95         bn->d[i] = SWAP_CRYPT_WORD(bn->d[i]);

```

```

96 #endif // BIG_ENDIAN_TPM
97     BnSetTop(bn, wordCount);
98     return bn;
99 }

```

10.2.2.3.4 BnToBytes()

This function converts a `BIG_NUM` to a byte array. It converts the *bigNum* to a big-endian byte string and sets *size* to the normalized value. If *size* is an input 0, then the receiving buffer is guaranteed to be large enough for the result and the size will be set to the size required for *bigNum* (leading zeros suppressed). The conversion for a little-endian machine simply requires that all significant bytes of the *bigNum* be reversed. For a big-endian machine, rather than unpack each word individually, the *bigNum* is converted to little-endian words, copied, and then converted back to big-endian.

```

100 LIB_EXPORT_BOOL
101 BnToBytes(
102     bigConst      bn,
103     BYTE          *buffer,
104     NUMBYTES      *size // This the number of bytes that are
105                        // available in the buffer. The result
106                        // should be this big.
107 )
108 {
109     crypt_ushort_t    requiredSize;
110     BYTE              *pFrom;
111     BYTE              *pTo;
112     crypt_ushort_t    count;
113 //
114 // validate inputs
115 pAssert(bn && buffer && size);
116
117 requiredSize = (BnSizeInBits(bn) + 7) / 8;
118 if(requiredSize == 0)
119 {
120     // If the input value is 0, return a byte of zero
121     *size = 1;
122     *buffer = 0;
123 }
124 else
125 {
126 #if BIG_ENDIAN_TPM
127     // Copy the constant input value into a modifiable value
128     BN_VAR(bnL, LARGEST_NUMBER_BITS * 2);
129     BnCopy(bnL, bn);
130     // byte swap the words in the local value to make them little-endian
131     for(count = 0; count < bnL->size; count++)
132         bnL->d[count] = SWAP_CRYPT_WORD(bnL->d[count]);
133     bn = (bigConst)bnL;
134 #endif
135     if(*size == 0)
136         *size = (NUMBYTES)requiredSize;
137     pAssert(requiredSize <= *size);
138     // Byte swap the number (not words but the whole value)
139     count = *size;
140     // Start from the least significant word and offset to the most significant
141     // byte which is in some high word
142     pFrom = (BYTE *)(&bn->d[0]) + requiredSize - 1;
143     pTo = buffer;
144
145     // If the number of output bytes is larger than the number bytes required
146     // for the input number, pad with zeros
147     for(count = *size; count > requiredSize; count--)
148         *pTo++ = 0;

```

```

149         // Move the most significant byte at the end of the BigNum to the next most
150         // significant byte position of the 2B and repeat for all significant bytes.
151         for(; requiredSize > 0; requiredSize--)
152             *pTo++ = *pFrom--;
153     }
154     return TRUE;
155 }

```

10.2.2.3.5 BnTo2B()

Function to convert a BIG_NUM to TPM2B. The TPM2B size is set to the requested size which may require padding. If size is non-zero and less than required by the value in *bn* then an error is returned. If size is zero, then the TPM2B is assumed to be large enough for the data and *a2b*→size will be adjusted accordingly.

```

156 LIB_EXPORT BOOL
157 BnTo2B(
158     bigConst      bn,                // IN:
159     TPM2B         *a2B,              // OUT:
160     NUMBYTES      size               // IN: the desired size
161 )
162 {
163     // Set the output size
164     if(bn && a2B)
165     {
166         a2B->size = size;
167         return BnToBytes(bn, a2B->buffer, &a2B->size);
168     }
169     return FALSE;
170 }
171 #if ALG_ECC

```

10.2.2.3.6 BnPointFrom2B()

Function to create a BIG_POINT structure from a 2B point. A point is going to be two ECC values in the same buffer. The values are going to be the size of the modulus. They are in modular form.

```

172 LIB_EXPORT bn_point_t *
173 BnPointFrom2B(
174     bigPoint      ecP,                // OUT: the preallocated point structure
175     TPMS_ECC_POINT *p                 // IN: the number to convert
176 )
177 {
178     if(p == NULL)
179         return NULL;
180
181     if(NULL != ecP)
182     {
183         BnFrom2B(ecP->x, &p->x.b);
184         BnFrom2B(ecP->y, &p->y.b);
185         BnSetWord(ecP->z, 1);
186     }
187     return ecP;
188 }

```

10.2.2.3.7 BnPointTo2B()

This function converts a BIG_POINT into a TPMS_ECC_POINT. A TPMS_ECC_POINT contains two TPM2B_ECC_PARAMETER values. The maximum size of the parameters is dependent on the maximum

EC key size used in an implementation. The presumption is that the TPMS_ECC_POINT is large enough to hold 2 TPM2B values, each as large as a MAX_ECC_PARAMETER_BYTES

```

189  LIB_EXPORT BOOL
190  BnPointTo2B(
191      TPMS_ECC_POINT *p,           // OUT: the converted 2B structure
192      bigPoint       ecP,         // IN: the values to be converted
193      bigCurve        E,          // IN: curve descriptor for the point
194  )
195  {
196      UINT16          size;
197      //
198      pAssert(p && ecP && E);
199      pAssert(BnEqualWord(ecP->z, 1));
200      // BnMsb is the bit number of the MSB. This is one less than the number of bits
201      size = (UINT16)BITS_TO_BYTES(BnSizeInBits(CurveGetOrder(AccessCurveData(E))));
202      BnTo2B(ecP->x, &p->x.b, size);
203      BnTo2B(ecP->y, &p->y.b, size);
204      return TRUE;
205  }
206  #endif // ALG_ECC

```

10.2.3 BnMath.c

10.2.3.1 Introduction

The simulator code uses the canonical form whenever possible in order to make the code in Part 3 more accessible. The canonical data formats are simple and not well suited for complex big number computations. When operating on big numbers, the data format is changed for easier manipulation. The format is native words in little-endian format. As the magnitude of the number decreases, the length of the array containing the number decreases but the starting address doesn't change.

The functions in this file perform simple operations on these big numbers. Only the more complex operations are passed to the underlying support library. Although the support library would have most of these functions, the interface code to convert the format for the values is greater than the size of the code to implement the functions here. So, rather than incur the overhead of conversion, they are done here.

If an implementer would prefer, the underlying library can be used simply by making code substitutions here.

NOTE There is an intention to continue to augment these functions so that there would be no need to use an external big number library.

Many of these functions have no error returns and will always return TRUE. This is to allow them to be used in **guarded** sequences. That is: OK = OK || BnSomething(s); where the BnSomething() function should not be called if OK isn't true.

10.2.3.2 Includes

```
1 #include "Tpm.h"
```

A constant value of zero as a stand in for NULL *bigNum* values

```
2 const bignum_t BnConstZero = {1, 0, {0}};
```

10.2.3.3 Functions

10.2.3.3.1 AddSame()

Adds two values that are the same size. This function allows *result* to be the same as either of the addends. This is a nice function to put into assembly because handling the carry for multi-precision stuff is not as easy in C (unless there is a REALLY smart compiler). It would be nice if there were idioms in a language that a compiler could recognize what is going on and optimize loops like this.

Return Value	Meaning
0	no carry out
1	carry out

```
3 static BOOL
4 AddSame(
5     crypt_ushort_t    *result,
6     const crypt_ushort_t *op1,
7     const crypt_ushort_t *op2,
8     int               count
9 )
10 {
11     int    carry = 0;
12     int    i;
```

```

13
14     for(i = 0; i < count; i++)
15     {
16         crypt_ushort_t    a = op1[i];
17         crypt_ushort_t    sum = a + op2[i];
18         result[i] = sum + carry;
19         // generate a carry if the sum is less than either of the inputs
20         // propagate a carry if there was a carry and the sum + carry is zero
21         // do this using bit operations rather than logical operations so that
22         // the time is about the same.
23         //           propagate term           | generate term
24         carry = ((result[i] == 0) & carry) | (sum < a);
25     }
26     return carry;
27 }

```

10.2.3.3.2 CarryProp()

Propagate a carry

```

28 static int
29 CarryProp(
30     crypt_ushort_t    *result,
31     const crypt_ushort_t *op,
32     int               count,
33     int               carry
34 )
35 {
36     for(; count; count--)
37         carry = ((*result++ = *op++ + carry) == 0) & carry;
38     return carry;
39 }
40 static void
41 CarryResolve(
42     bigNum             result,
43     int               stop,
44     int               carry
45 )
46 {
47     if(carry)
48     {
49         pAssert((unsigned)stop < result->allocated);
50         result->d[stop++] = 1;
51     }
52     BnSetTop(result, stop);
53 }

```

10.2.3.3.3 BnAdd()

This function adds two *bigNum* values. This function always returns TRUE.

```

54 LIB_EXPORT BOOL
55 BnAdd(
56     bigNum             result,
57     bigConst           op1,
58     bigConst           op2
59 )
60 {
61     crypt_ushort_t    stop;
62     int               carry;
63     const bignum_t    *n1 = op1;
64     const bignum_t    *n2 = op2;
65 }

```

```

66  //
67  if(n2->size > n1->size)
68  {
69      n1 = op2;
70      n2 = op1;
71  }
72  pAssert(result->allocated >= n1->size);
73  stop = MIN(n1->size, n2->allocated);
74  carry = (int)AddSame(result->d, n1->d, n2->d, (int)stop);
75  if(n1->size > stop)
76      carry = CarryProp(&result->d[stop], &n1->d[stop], (int)(n1->size - stop),
carry);
77  CarryResolve(result, (int)n1->size, carry);
78  return TRUE;
79  }

```

10.2.3.3.4 BnAddWord()

This function adds a word value to a *bigNum*. This function always returns TRUE.

```

80  LIB_EXPORT BOOL
81  BnAddWord(
82      bigNum          result,
83      bigConst        op,
84      crypt_uword_t   word
85  )
86  {
87      int             carry;
88      //
89      carry = (result->d[0] = op->d[0] + word) < word;
90      carry = CarryProp(&result->d[1], &op->d[1], (int)(op->size - 1), carry);
91      CarryResolve(result, (int)op->size, carry);
92      return TRUE;
93  }

```

10.2.3.3.5 SubSame()

This function subtracts two values that have the same size.

```

94  static int
95  SubSame(
96      crypt_uword_t   *result,
97      const crypt_uword_t *op1,
98      const crypt_uword_t *op2,
99      int             count
100  )
101  {
102      int             borrow = 0;
103      int             i;
104      for(i = 0; i < count; i++)
105      {
106          crypt_uword_t a = op1[i];
107          crypt_uword_t diff = a - op2[i];
108          result[i] = diff - borrow;
109          // generate | propagate
110          borrow = (diff > a) | ((diff == 0) & borrow);
111      }
112      return borrow;
113  }

```

10.2.3.3.6 BorrowProp()

This propagates a borrow. If borrow is true when the end of the array is reached, then it means that op2 was larger than op1 and we don't handle that case so an assert is generated. This design choice was made because our only *bigNum* computations are on large positive numbers (primes) or on fields. Propagate a borrow.

```

114 static int
115 BorrowProp(
116     crypt_ushort_t      *result,
117     const crypt_ushort_t *op,
118     int                  size,
119     int                  borrow
120 )
121 {
122     for(; size > 0; size--)
123         borrow = ((*result++ = *op++ - borrow) == MAX_CRYPT_USHORT) && borrow;
124     return borrow;
125 }

```

10.2.3.3.7 BnSub()

This function does subtraction of two *bigNum* values and returns result = op1 - op2 when op1 is greater than op2. If op2 is greater than op1, then a fault is generated. This function always returns TRUE.

```

126 LIB_EXPORT BOOL
127 BnSub(
128     bigNum      result,
129     bigConst    op1,
130     bigConst    op2
131 )
132 {
133     int borrow;
134     int stop = (int)MIN(op1->size, op2->allocated);
135     //
136     // Make sure that op2 is not obviously larger than op1
137     pAssert(op1->size >= op2->size);
138     borrow = SubSame(result->d, op1->d, op2->d, stop);
139     if(op1->size > (crypt_ushort_t)stop)
140         borrow = BorrowProp(&result->d[stop], &op1->d[stop], (int)(op1->size - stop),
141                             borrow);
142     pAssert(!borrow);
143     BnSetTop(result, op1->size);
144     return TRUE;
145 }

```

10.2.3.3.8 BnSubWord()

This function subtracts a word value from a *bigNum*. This function always returns TRUE.

```

146 LIB_EXPORT BOOL
147 BnSubWord(
148     bigNum      result,
149     bigConst    op,
150     crypt_ushort_t word
151 )
152 {
153     int borrow;
154     //
155     pAssert(op->size > 1 || word <= op->d[0]);
156     borrow = word > op->d[0];

```

```

157     result->d[0] = op->d[0] - word;
158     borrow = BorrowProp(&result->d[1], &op->d[1], (int)(op->size - 1), borrow);
159     pAssert(!borrow);
160     BnSetTop(result, op->size);
161     return TRUE;
162 }

```

10.2.3.3.9 BnUnsignedCmp()

This function performs a comparison of op1 to op2. The compare is approximately constant time if the size of the values used in the compare is consistent across calls (from the same line in the calling code).

Return Value	Meaning
< 0	op1 is less than op2
0	op1 is equal to op2
> 0	op1 is greater than op2

```

163 LIB_EXPORT int
164 BnUnsignedCmp(
165     bigConst          op1,
166     bigConst          op2
167 )
168 {
169     int          retVal;
170     int          diff;
171     int          i;
172     //
173     pAssert((op1 != NULL) && (op2 != NULL));
174     retVal = (int)(op1->size - op2->size);
175     if(retVal == 0)
176     {
177         for(i = (int)(op1->size - 1); i >= 0; i--)
178         {
179             diff = (op1->d[i] < op2->d[i]) ? -1 : (op1->d[i] != op2->d[i]);
180             retVal = retVal == 0 ? diff : retVal;
181         }
182     }
183     else
184         retVal = (retVal < 0) ? -1 : 1;
185     return retVal;
186 }

```

10.2.3.3.10 BnUnsignedCmpWord()

Compare a *bigNum* to a *crypt_uword_t*.

Return Value	Meaning
-1	op1 is less that word
0	op1 is equal to word
1	op1 is greater than word

```

187 LIB_EXPORT int
188 BnUnsignedCmpWord(
189     bigConst          op1,
190     crypt_uword_t     word
191 )
192 {

```

```

193     if(op1->size > 1)
194         return 1;
195     else if(op1->size == 1)
196         return (op1->d[0] < word) ? -1 : (op1->d[0] > word);
197     else // op1 is zero
198         // equal if word is zero
199         return (word == 0) ? 0 : -1;
200 }

```

10.2.3.3.11 BnModWord()

This function does modular division of a big number when the modulus is a word value.

```

201 LIB_EXPORT crypt_word_t
202 BnModWord(
203     bigConst      numerator,
204     crypt_word_t  modulus
205 )
206 {
207     BN_MAX(remainder);
208     BN_VAR(mod, RADIX_BITS);
209     //
210     mod->d[0] = modulus;
211     mod->size = (modulus != 0);
212     BnDiv(NULL, remainder, numerator, mod);
213     return remainder->d[0];
214 }

```

10.2.3.3.12 Msb()

This function returns the bit number of the most significant bit of a `crypt_uword_t`. The number for the least significant bit of any *bigNum* value is 0. The maximum return value is `RADIX_BITS - 1`,

Return Value	Meaning
-1	the word was zero
n	the bit number of the most significant bit in the word

```

215 LIB_EXPORT int
216 Msb(
217     crypt_uword_t  word
218 )
219 {
220     int      retVal = -1;
221     //
222     #if RADIX_BITS == 64
223         if(word & 0xffffffff00000000) { retVal += 32; word >>= 32; }
224     #endif
225     if(word & 0xffff0000) { retVal += 16; word >>= 16; }
226     if(word & 0x0000ff00) { retVal += 8; word >>= 8; }
227     if(word & 0x000000f0) { retVal += 4; word >>= 4; }
228     if(word & 0x0000000c) { retVal += 2; word >>= 2; }
229     if(word & 0x00000002) { retVal += 1; word >>= 1; }
230     return retVal + (int)word;
231 }

```

10.2.3.3.13 BnMsb()

This function returns the number of the MSb of a *bigNum* value.

Return Value	Meaning
-1	the word was zero or <i>bn</i> was NULL
n	the bit number of the most significant bit in the word

```

232  LIB_EXPORT int
233  BnMsb(
234      bigConst      bn
235  )
236  {
237      // If the value is NULL, or the size is zero then treat as zero and return -1
238      if(bn != NULL && bn->size > 0)
239      {
240          int      retVal = Msb(bn->d[bn->size - 1]);
241          retVal += (int)(bn->size - 1) * RADIX_BITS;
242          return retVal;
243      }
244      else
245          return -1;
246  }

```

10.2.3.3.14 BnSizeInBits()

This function returns the number of bits required to hold a number. It is one greater than the Msb.

```

247  LIB_EXPORT unsigned
248  BnSizeInBits(
249      bigConst      n
250  )
251  {
252      int      bits = BnMsb(n) + 1;
253      //
254      return bits < 0? 0 : (unsigned)bits;
255  }

```

10.2.3.3.15 BnSetWord()

Change the value of a *bignum_t* to a word value.

```

256  LIB_EXPORT bigNum
257  BnSetWord(
258      bigNum      n,
259      crypt_ushort_t  w
260  )
261  {
262      if(n != NULL)
263      {
264          pAssert(n->allocated > 1);
265          n->d[0] = w;
266          BnSetTop(n, (w != 0) ? 1 : 0);
267      }
268      return n;
269  }

```

10.2.3.3.16 BnSetBit()

This function will SET a bit in a *bigNum*. Bit 0 is the least-significant bit in the 0th *digit_t*. The function always return TRUE

```

270  LIB_EXPORT BOOL

```

```

271 BnSetBit(
272     bigNum          bn,          // IN/OUT: big number to modify
273     unsigned int    bitNum       // IN: Bit number to SET
274 )
275 {
276     crypt_ushort_t    offset = bitNum / RADIX_BITS;
277     pAssert(bn->allocated * RADIX_BITS >= bitNum);
278     // Grow the number if necessary to set the bit.
279     while(bn->size <= offset)
280         bn->d[bn->size++] = 0;
281     bn->d[offset] |= ((crypt_ushort_t)1 << RADIX_MOD(bitNum));
282     return TRUE;
283 }

```

10.2.3.3.17 BnTestBit()

This function is used to check to see if a bit is SET in a `bigNum_t`. The 0th bit is the LSb of `d[0]`.

Return Value	Meaning
TRUE(1)	the bit is set
FALSE(0)	the bit is not set or the number is out of range

```

284 LIB_EXPORT BOOL
285 BnTestBit(
286     bigNum          bn,          // IN: number to check
287     unsigned int    bitNum       // IN: bit to test
288 )
289 {
290     crypt_ushort_t    offset = RADIX_DIV(bitNum);
291     //
292     if(bn->size > offset)
293         return ((bn->d[offset] & (((crypt_ushort_t)1) << RADIX_MOD(bitNum))) != 0);
294     else
295         return FALSE;
296 }

```

10.2.3.3.18 BnMaskBits()

This function is used to mask off high order bits of a big number. The returned value will have no more than *maskBit* bits set.

NOTE There is a requirement that unused words of a `bigNum_t` are set to zero.

Return Value	Meaning
TRUE(1)	result masked
FALSE(0)	the input was not as large as the mask

```

297 LIB_EXPORT BOOL
298 BnMaskBits(
299     bigNum          bn,          // IN/OUT: number to mask
300     crypt_ushort_t  maskBit     // IN: the bit number for the mask.
301 )
302 {
303     crypt_ushort_t    finalSize;
304     BOOL              retVal;
305
306     finalSize = BITS_TO_CRYPT_WORDS(maskBit);
307     retVal = (finalSize <= bn->allocated);

```

```

308     if(retVal && (finalSize > 0))
309     {
310         crypt_uword_t    mask;
311         mask = ~((crypt_uword_t)0) >> RADIX_MOD(maskBit);
312         bn->d[finalSize - 1] &= mask;
313     }
314     BnSetTop(bn, finalSize);
315     return retVal;
316 }

```

10.2.3.3.19 BnShiftRight()

This function will shift a *bigNum* to the right by the *shiftAmount*. This function always returns TRUE.

```

317 LIB_EXPORT BOOL
318 BnShiftRight(
319     bigNum          result,
320     bigConst        toShift,
321     uint32_t        shiftAmount
322 )
323 {
324     uint32_t        offset = (shiftAmount >> RADIX_LOG2);
325     uint32_t        i;
326     uint32_t        shiftIn;
327     crypt_uword_t    finalSize;
328     //
329     shiftAmount = shiftAmount & RADIX_MASK;
330     shiftIn = RADIX_BITS - shiftAmount;
331
332     // The end size is toShift->size - offset less one additional
333     // word if the shiftAmount would make the upper word == 0
334     if(toShift->size > offset)
335     {
336         finalSize = toShift->size - offset;
337         finalSize -= (toShift->d[toShift->size - 1] >> shiftAmount) == 0 ? 1 : 0;
338     }
339     else
340         finalSize = 0;
341
342     pAssert(finalSize <= result->allocated);
343     if(finalSize != 0)
344     {
345         for(i = 0; i < finalSize; i++)
346         {
347             result->d[i] = (toShift->d[i + offset] >> shiftAmount)
348                 | (toShift->d[i + offset + 1] << shiftIn);
349         }
350         if(offset == 0)
351             result->d[i] = toShift->d[i] >> shiftAmount;
352     }
353     BnSetTop(result, finalSize);
354     return TRUE;
355 }

```

10.2.3.3.20 BnGetRandomBits()

This function gets random bits for use in various places. To make sure that the number is generated in a portable format, it is created as a TPM2B and then converted to the internal format.

One consequence of the generation scheme is that, if the number of bits requested is not a multiple of 8, then the high-order bits are set to zero. This would come into play when generating a 521-bit ECC key. A 66-byte (528-bit) value is generated and the high order 7 bits are masked off (CLEAR).

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

356 LIB_EXPORT BOOL
357 BnGetRandomBits (
358     bigNum      n,
359     size_t      bits,
360     RAND_STATE  *rand
361 )
362 {
363     // Since this could be used for ECC key generation using the extra bits method,
364     // make sure that the value is large enough
365     TPM2B_TYPE(LARGEST, LARGEST_NUMBER + 8);
366     TPM2B_LARGEST large;
367     //
368     large.b.size = (UINT16)BITS_TO_BYTES(bits);
369     if(DRBG_Generate(rand, large.t.buffer, large.t.size) == large.t.size)
370     {
371         if(BnFrom2B(n, &large.b) != NULL)
372         {
373             if(BnMaskBits(n, (crypt_uword_t)bits))
374                 return TRUE;
375         }
376     }
377     return FALSE;
378 }

```

10.2.3.3.21 BnGenerateRandomInRange()

This function is used to generate a random number r in the range $1 \leq r < \text{limit}$. The function gets a random number of bits that is the size of limit. There is some some probability that the returned number is going to be greater than or equal to the limit. If it is, try again. There is no more than 50% chance that the next number is also greater, so try again. We keep trying until we get a value that meets the criteria. Since limit is very often a number with a LOT of high order ones, this rarely would need a second try.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure (<i>limit</i> is too small)

```

379 LIB_EXPORT BOOL
380 BnGenerateRandomInRange (
381     bigNum      dest,
382     bigConst     limit,
383     RAND_STATE  *rand
384 )
385 {
386     size_t bits = BnSizeInBits(limit);
387     //
388     if(bits < 2)
389     {
390         BnSetWord(dest, 0);
391         return FALSE;
392     }
393     else
394     {
395         while(BnGetRandomBits(dest, bits, rand)
396             && (BnEqualZero(dest) || (BnUnsignedCmp(dest, limit) >= 0)));
397     }

```

```
398     return !g_inFailureMode;  
399 }
```

DRAFT

10.2.4 BnMemory.c

10.2.4.1 Introduction

This file contains the memory setup functions used by the *bigNum* functions in *CryptoEngine()*

10.2.4.2 Includes

```
1  #include "Tpm.h"
```

10.2.4.3 Functions

10.2.4.3.1 BnSetTop()

This function is used when the size of a *bignum_t* is changed. It makes sure that the unused words are set to zero and that any significant words of zeros are eliminated from the used size indicator.

```
2  LIB_EXPORT bigNum
3  BnSetTop(
4      bigNum      bn,          // IN/OUT: number to clean
5      crypt_ushort_t  top      // IN: the new top
6  )
7  {
8      if(bn != NULL)
9      {
10         pAssert(top <= bn->allocated);
11         // If forcing the size to be decreased, make sure that the words being
12         // discarded are being set to 0
13         while(bn->size > top)
14             bn->d[--bn->size] = 0;
15         bn->size = top;
16         // Now make sure that the words that are left are 'normalized' (no high-order
17         // words of zero.
18         while((bn->size > 0) && (bn->d[bn->size - 1] == 0))
19             bn->size -= 1;
20     }
21     return bn;
22 }
```

10.2.4.3.2 BnClearTop()

This function will make sure that all unused words are zero.

```
23  LIB_EXPORT bigNum
24  BnClearTop(
25      bigNum      bn
26  )
27  {
28      crypt_ushort_t  i;
29      //
30      if(bn != NULL)
31      {
32          for(i = bn->size; i < bn->allocated; i++)
33              bn->d[i] = 0;
34          while((bn->size > 0) && (bn->d[bn->size] == 0))
35              bn->size -= 1;
36      }
37      return bn;
38 }
```

10.2.4.3.3 BnInitializeWord()

This function is used to initialize an allocated *bigNum* with a word value. The *bigNum* does not have to be allocated with a single word.

```

39  LIB_EXPORT bigNum
40  BnInitializeWord(
41      bigNum          bn,          // IN:
42      crypt_uword_t   allocated,   // IN:
43      crypt_uword_t   word        // IN:
44  )
45  {
46      bn->allocated = allocated;
47      bn->size = (word != 0);
48      bn->d[0] = word;
49      while(allocated > 1)
50          bn->d[--allocated] = 0;
51      return bn;
52  }

```

10.2.4.3.4 BnInit()

This function initializes a stack allocated *bignum_t*. It initializes *allocated* and *size* and zeros the words of *d*.

```

53  LIB_EXPORT bigNum
54  BnInit(
55      bigNum          bn,
56      crypt_uword_t   allocated
57  )
58  {
59      if(bn != NULL)
60      {
61          bn->allocated = allocated;
62          bn->size = 0;
63          while(allocated != 0)
64              bn->d[--allocated] = 0;
65      }
66      return bn;
67  }

```

10.2.4.3.5 BnCopy()

Function to copy a *bignum_t*. If the output is NULL, then nothing happens. If the input is NULL, the output is set to zero.

```

68  LIB_EXPORT BOOL
69  BnCopy(
70      bigNum          out,
71      bigConst         in
72  )
73  {
74      if(in == out)
75          BnSetTop(out, BnGetSize(out));
76      else if(out != NULL)
77      {
78          if(in != NULL)
79          {
80              unsigned int i;
81              pAssert(BnGetAllocated(out) >= BnGetSize(in));
82              for(i = 0; i < BnGetSize(in); i++)
83                  out->d[i] = in->d[i];

```



```

84         BnSetTop(out, BnGetSize(in));
85     }
86     else
87         BnSetTop(out, 0);
88 }
89 return TRUE;
90 }
91 #if ALG_ECC

```

10.2.4.3.6 BnPointCopy()

Function to copy a bn point.

```

92 LIB_EXPORT BOOL
93 BnPointCopy(
94     bigPoint          pOut,
95     pointConst        pIn
96 )
97 {
98     return BnCopy(pOut->x, pIn->x)
99         && BnCopy(pOut->y, pIn->y)
100         && BnCopy(pOut->z, pIn->z);
101 }

```

10.2.4.3.7 BnInitializePoint()

This function is used to initialize a point structure with the addresses of the coordinates.

```

102 LIB_EXPORT bn_point_t *
103 BnInitializePoint(
104     bigPoint          p,      // OUT: structure to receive pointers
105     bigNum             x,      // IN: x coordinate
106     bigNum             y,      // IN: y coordinate
107     bigNum             z,      // IN: x coordinate
108 )
109 {
110     p->x = x;
111     p->y = y;
112     p->z = z;
113     BnSetWord(z, 1);
114     return p;
115 }
116 #endif // ALG_ECC

```

10.2.5 CryptCmac.c

10.2.5.1 Introduction

This file contains the implementation of the message authentication codes based on a symmetric block cipher. These functions only use the single block encryption functions of the selected symmetric cryptographic library.

10.2.5.2 Includes, Defines, and Typedefs

```

1  #define _CRYPT_HASH_C_
2  #include "Tpm.h"
3  #include "CryptSym.h"
4
5  #if ALG_CMAC

```

10.2.5.3 Functions

10.2.5.3.1 CryptCmacStart()

This is the function to start the CMAC sequence operation. It initializes the dispatch functions for the data and end operations for CMAC and initializes the parameters that are used for the processing of data, including the key, key size and block cipher algorithm.

```

6  UINT16
7  CryptCmacStart(
8      SMAC_STATE      *state,
9      TPMU_PUBLIC_PARMS *keyParms,
10     TPM_ALG_ID       macAlg,
11     TPM2B            *key
12 )
13 {
14     tpmCmacState_t      *cState = &state->state.cmac;
15     TPMT_SYM_DEF_OBJECT *def = &keyParms->symDetail.sym;
16     //
17     if(macAlg != TPM_ALG_CMAC)
18         return 0;
19     // set up the encryption algorithm and parameters
20     cState->symAlg = def->algorithm;
21     cState->keySizeBits = def->keyBits.sym;
22     cState->iv.t.size = CryptGetSymmetricBlockSize(def->algorithm,
23                                                     def->keyBits.sym);
24     MemoryCopy2B(&cState->symKey.b, key, sizeof(cState->symKey.t.buffer));
25
26     // Set up the dispatch methods for the CMAC
27     state->smacMethods.data = CryptCmacData;
28     state->smacMethods.end = CryptCmacEnd;
29     return cState->iv.t.size;
30 }

```

10.2.5.3.2 CryptCmacData()

This function is used to add data to the CMAC sequence computation. The function will XOR new data into the IV. If the buffer is full, and there is additional input data, the data is encrypted into the IV buffer, the new data is then XOR into the IV. When the data runs out, the function returns without encrypting even if the buffer is full. The last data block of a sequence will not be encrypted until the call to CryptCmacEnd(). This is to allow the proper subkey to be computed and applied before the last block is encrypted.

```

31 void
32 CryptCmacData(
33     SMAC_STATES      *state,
34     UINT32           size,
35     const BYTE       *buffer
36 )
37 {
38     tpmCmacState_t    *cmacState = &state->cmac;
39     TPM_ALG_ID        algorithm = cmacState->symAlg;
40     BYTE              *key = cmacState->symKey.t.buffer;
41     UINT16             keySizeInBits = cmacState->keySizeBits;
42     tpmCryptKeySchedule_t keySchedule;
43     TpmCryptSetSymKeyCall_t encrypt;
44     //
45     // Set up the encryption values based on the algorithm
46     switch (algorithm)
47     {
48         FOR_EACH_SYM(ENCRYPT_CASE)
49         default:
50             FAIL(FATAL_ERROR_INTERNAL);
51     }
52     while(size > 0)
53     {
54         if(cmacState->bcount == cmacState->iv.t.size)
55         {
56             ENCRYPT(&keySchedule, cmacState->iv.t.buffer, cmacState->iv.t.buffer);
57             cmacState->bcount = 0;
58         }
59         for(; (size > 0) && (cmacState->bcount < cmacState->iv.t.size);
60             size--, cmacState->bcount++)
61         {
62             cmacState->iv.t.buffer[cmacState->bcount] ^= *buffer++;
63         }
64     }
65 }

```

10.2.5.3.3 CryptCmacEnd()

This is the completion function for the CMAC. It does padding, if needed, and selects the subkey to be applied before the last block is encrypted.

```

66 UINT16
67 CryptCmacEnd(
68     SMAC_STATES      *state,
69     UINT32           outSize,
70     BYTE             *outBuffer
71 )
72 {
73     tpmCmacState_t    *cState = &state->cmac;
74     // Need to set algorithm, key, and keySizeInBits in the local context so that
75     // the SELECT and ENCRYPT macros will work here
76     TPM_ALG_ID        algorithm = cState->symAlg;
77     BYTE              *key = cState->symKey.t.buffer;
78     UINT16             keySizeInBits = cState->keySizeBits;
79     tpmCryptKeySchedule_t keySchedule;
80     TpmCryptSetSymKeyCall_t encrypt;
81     TPM2B_IV           subkey = {{0, {0}}};
82     BOOL               xorVal;
83     UINT16             i;
84
85     subkey.t.size = cState->iv.t.size;
86     // Encrypt a block of zero
87     // Set up the encryption values based on the algorithm
88     switch (algorithm)

```

```

89     {
90         FOR_EACH_SYM(ENCRYPT_CASE)
91         default:
92             return 0;
93     }
94     ENCRYPT(&keySchedule, subkey.t.buffer, subkey.t.buffer);
95
96     // shift left by 1 and XOR with 0x0...87 if the MSb was 0
97     xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
98     ShiftLeft(&subkey.b);
99     subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
100    // this is a sanity check to make sure that the algorithm is working properly.
101    // remove this check when debug is done
102    pAssert(cState->bcount <= cState->iv.t.size);
103    // If the buffer is full then no need to compute subkey 2.
104    if(cState->bcount < cState->iv.t.size)
105    {
106        //Pad the data
107        cState->iv.t.buffer[cState->bcount++] ^= 0x80;
108        // The rest of the data is a pad of zero which would simply be XORed
109        // with the iv value so nothing to do...
110        // Now compute K2
111        xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
112        ShiftLeft(&subkey.b);
113        subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
114    }
115    // XOR the subkey into the IV
116    for(i = 0; i < subkey.t.size; i++)
117        cState->iv.t.buffer[i] ^= subkey.t.buffer[i];
118    ENCRYPT(&keySchedule, cState->iv.t.buffer, cState->iv.t.buffer);
119    i = (UINT16)MIN(cState->iv.t.size, outSize);
120    MemoryCopy(outBuffer, cState->iv.t.buffer, i);
121
122    return i;
123 }
124 #endif

```

10.2.6 CryptUtil.c

10.2.6.1 Introduction

This module contains the interfaces to the CryptoEngine() and provides miscellaneous cryptographic functions in support of the TPM.

10.2.6.2 Includes

```
1 #include "Tpm.h"
```

10.2.6.3 Hash/HMAC Functions

10.2.6.3.1 CryptHmacSign()

Sign a digest using an HMAC key. This an HMAC of a digest, not an HMAC of a message.

Error Return	Meaning
TPM_RC_HASH	not a valid hash

```
2 static TPM_RC
3 CryptHmacSign(
4     TPMT_SIGNATURE    *signature,    // OUT: signature
5     OBJECT             *signKey,     // IN: HMAC key sign the hash
6     TPM2B_DIGEST       *hashData    // IN: hash to be signed
7 )
8 {
9     HMAC_STATE          hmacState;
10    UINT32              digestSize;
11
12    digestSize = CryptHmacStart2B(&hmacState, signature->signature.any.hashAlg,
13                                &signKey->sensitive.sensitive.bits.b);
14    CryptDigestUpdate2B(&hmacState.hashState, &hashData->b);
15    CryptHmacEnd(&hmacState, digestSize,
16                (BYTE *) &signature->signature.hmac.digest);
17    return TPM_RC_SUCCESS;
18 }
```

10.2.6.3.2 CryptHMACVerifySignature()

This function will verify a signature signed by a HMAC key. Note that a caller needs to prepare *signature* with the signature algorithm (TPM_ALG_HMAC) and the hash algorithm to use. This function then builds a signature of that type.

Error Return	Meaning
TPM_RC_SCHEME	not the proper scheme for this key type
TPM_RC_SIGNATURE	if invalid input or signature is not genuine

```
19 static TPM_RC
20 CryptHMACVerifySignature(
21     OBJECT             *signKey,     // IN: HMAC key signed the hash
22     TPM2B_DIGEST       *hashData,    // IN: digest being verified
23     TPMT_SIGNATURE     *signature    // IN: signature to be verified
24 )
25 {
```

```

26     TPMT_SIGNATURE      test;
27     TPMT_KEYEDHASH_SCHEME *keyScheme =
28         &signKey->publicArea.parameters.keyedHashDetail.scheme;
29     //
30     if((signature->sigAlg != TPM_ALG_HMAC)
31         || (signature->signature.hmac.hashAlg == TPM_ALG_NULL))
32         return TPM_RC_SCHEME;
33     // This check is not really needed for verification purposes. However, it does
34     // prevent someone from trying to validate a signature using a weaker hash
35     // algorithm than otherwise allowed by the key. That is, a key with a scheme
36     // other than TPM_ALG_NULL can only be used to validate signatures that have
37     // a matching scheme.
38     if((keyScheme->scheme != TPM_ALG_NULL)
39         && ((keyScheme->scheme != signature->sigAlg)
40             || (keyScheme->details.hmac.hashAlg
41                 != signature->signature.any.hashAlg)))
42         return TPM_RC_SIGNATURE;
43     test.sigAlg = signature->sigAlg;
44     test.signature.hmac.hashAlg = signature->signature.hmac.hashAlg;
45
46     CryptHmacSign(&test, signKey, hashData);
47
48     // Compare digest
49     if(!MemoryEqual(&test.signature.hmac.digest,
50                     &signature->signature.hmac.digest,
51                     CryptHashGetDigestSize(signature->signature.any.hashAlg)))
52         return TPM_RC_SIGNATURE;
53
54     return TPM_RC_SUCCESS;
55 }

```

10.2.6.3.3 CryptGenerateKeyedHash()

This function creates a *keyedHash* object.

Error Return	Meaning
TPM_RC_NO_RESULT	cannot get values from random number generator
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme

```

56 static TPM_RC
57 CryptGenerateKeyedHash(
58     TPMT_PUBLIC      *publicArea,           // IN/OUT: the public area template
59                                     //      for the new key.
60     TPMT_SENSITIVE    *sensitive,           // OUT: sensitive area
61     TPMS_SENSITIVE_CREATE *sensitiveCreate, // IN: sensitive creation data
62     RAND_STATE        *rand,               // IN: "entropy" source
63 )
64 {
65     TPMT_KEYEDHASH_SCHEME *scheme;
66     TPM_ALG_ID             hashAlg;
67     UINT16                 digestSize;
68
69     scheme = &publicArea->parameters.keyedHashDetail.scheme;
70
71     if(publicArea->type != TPM_ALG_KEYEDHASH)
72         return TPM_RC_FAILURE;
73
74     // Pick the limiting hash algorithm
75     if(scheme->scheme == TPM_ALG_NULL)
76         hashAlg = publicArea->nameAlg;
77     else if(scheme->scheme == TPM_ALG_XOR)
78         hashAlg = scheme->details.xor.hashAlg;

```

```

79     else
80         hashAlg = scheme->details.hmac.hashAlg;
81         digestSize = CryptHashGetDigestSize(hashAlg);
82
83         // if this is a signing or a decryption key, then the limit
84         // for the data size is the block size of the hash. This limit
85         // is set because larger values have lower entropy because of the
86         // HMAC function. The lower limit is 1/2 the size of the digest
87         //
88         // If the user provided the key, check that it is a proper size
89         if(sensitiveCreate->data.t.size != 0)
90         {
91             if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
92                || IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
93             {
94                 if(sensitiveCreate->data.t.size > CryptHashGetBlockSize(hashAlg))
95                     return TPM_RC_SIZE;
96             }
97             // May make this a FIPS-mode requirement
98             if(sensitiveCreate->data.t.size < (digestSize / 2))
99                 return TPM_RC_SIZE;
100         }
101         // If this is a data blob, then anything that will get past the unmarshaling
102         // is OK
103         MemoryCopy2B(&sensitive->sensitive.bits.b, &sensitiveCreate->data.b,
104                     sizeof(sensitive->sensitive.bits.t.buffer));
105     }
106     else
107     {
108         // The TPM is going to generate the data so set the size to be the
109         // size of the digest of the algorithm
110         sensitive->sensitive.bits.t.size =
111             DRBG_Generate(rand, sensitive->sensitive.bits.t.buffer, digestSize);
112         if(sensitive->sensitive.bits.t.size == 0)
113             return (g_inFailureMode) ? TPM_RC_FAILURE : TPM_RC_NO_RESULT;
114     }
115     return TPM_RC_SUCCESS;
116 }

```

10.2.6.3.4 CryptIsSchemeAnonymous()

This function is used to test a scheme to see if it is an anonymous scheme. The only anonymous scheme is ECDSA. ECDSA can be used to do things like U-Prove.

```

117 BOOL
118 CryptIsSchemeAnonymous(
119     TPM_ALG_ID    scheme           // IN: the scheme algorithm to test
120 )
121 {
122     return scheme == TPM_ALG_ECDSA;
123 }

```

10.2.6.4 Symmetric Functions

10.2.6.4.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```

124 void
125 ParmDecryptSym(
126     TPM_ALG_ID    symAlg,           // IN: the symmetric algorithm
127     TPM_ALG_ID    hash,             // IN: hash algorithm for KDFa

```



```

128     UINT16      keySizeInBits, // IN: the key size in bits
129     TPM2B       *key,          // IN: KDF HMAC key
130     TPM2B       *nonceCaller,  // IN: nonce caller
131     TPM2B       *nonceTpm,     // IN: nonce TPM
132     UINT32      dataSize,      // IN: size of parameter buffer
133     BYTE        *data          // OUT: buffer to be decrypted
134 )
135 {
136     // KDF output buffer
137     // It contains parameters for the CFB encryption
138     // From MSB to LSB, they are the key and iv
139     BYTE        symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
140     // Symmetric key size in byte
141     UINT16      keySize = (keySizeInBits + 7) / 8;
142     TPM2B_IV    iv;
143
144     iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
145     // If there is decryption to do...
146     if(iv.t.size > 0)
147     {
148         // Generate key and iv
149         CryptKDFa(hash, key, CFB_KEY, nonceCaller, nonceTpm,
150                 keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
151         MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
152
153         CryptSymmetricDecrypt(data, symAlg, keySizeInBits, symParmString,
154                             &iv, TPM_ALG_CFB, dataSize, data);
155     }
156     return;
157 }

```

10.2.6.4.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```

158 void
159 ParmEncryptSym(
160     TPM_ALG_ID    symAlg,          // IN: symmetric algorithm
161     TPM_ALG_ID    hash,           // IN: hash algorithm for KDFa
162     UINT16        keySizeInBits,  // IN: symmetric key size in bits
163     TPM2B         *key,           // IN: KDF HMAC key
164     TPM2B         *nonceCaller,   // IN: nonce caller
165     TPM2B         *nonceTpm,     // IN: nonce TPM
166     UINT32        dataSize,      // IN: size of parameter buffer
167     BYTE          *data          // OUT: buffer to be encrypted
168 )
169 {
170     // KDF output buffer
171     // It contains parameters for the CFB encryption
172     BYTE        symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
173
174     // Symmetric key size in bytes
175     UINT16      keySize = (keySizeInBits + 7) / 8;
176
177     TPM2B_IV    iv;
178
179     iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
180     // See if there is any encryption to do
181     if(iv.t.size > 0)
182     {
183         // Generate key and iv
184         CryptKDFa(hash, key, CFB_KEY, nonceTpm, nonceCaller,
185                 keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
186         MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);

```

```

187
188     CryptSymmetricEncrypt(data, symAlg, keySizeInBits, symParmString, &iv,
189                           TPM_ALG_CFB, dataSize, data);
190 }
191 return;
192 }

```

10.2.6.4.3 CryptGenerateKeySymmetric()

This function generates a symmetric cipher key. The derivation process is determined by the type of the provided *rand*

Error Return	Meaning
TPM_RC_NO_RESULT	cannot get a random value
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area
TPM_RC_KEY	provided key value is not allowed

```

193 static TPM_RC
194 CryptGenerateKeySymmetric(
195     TPMT_PUBLIC      *publicArea,          // IN/OUT: The public area template
196                                     // for the new key.
197     TPMT_SENSITIVE   *sensitive,          // OUT: sensitive area
198     TPMS_SENSITIVE_CREATE *sensitiveCreate, // IN: sensitive creation data
199     RAND_STATE        *rand               // IN: the "entropy" source for
200 )
201 {
202     UINT16      keyBits = publicArea->parameters.symDetail.sym.keyBits.sym;
203     TPM_RC      result;
204     //
205     // only do multiples of RADIX_BITS
206     if((keyBits % RADIX_BITS) != 0)
207         return TPM_RC_KEY_SIZE;
208     // If this is not a new key, then the provided key data must be the right size
209     if(sensitiveCreate->data.t.size != 0)
210     {
211         result = CryptSymKeyValidate(&publicArea->parameters.symDetail.sym,
212                                     (TPM2B_SYM_KEY *)&sensitiveCreate->data);
213         if(result == TPM_RC_SUCCESS)
214             MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
215                         sizeof(sensitive->sensitive.sym.t.buffer));
216     }
217     #if ALG_TDES
218     else if(publicArea->parameters.symDetail.sym.algorithm == TPM_ALG_TDES)
219     {
220         result = CryptGenerateKeyDes(publicArea, sensitive, rand);
221     }
222     #endif
223     else
224     {
225         sensitive->sensitive.sym.t.size =
226             DRBG_Generate(rand, sensitive->sensitive.sym.t.buffer,
227                           BITS_TO_BYTES(keyBits));
228         if(g_inFailureMode)
229             result = TPM_RC_FAILURE;
230         else if(sensitive->sensitive.sym.t.size == 0)
231             result = TPM_RC_NO_RESULT;
232         else
233             result = TPM_RC_SUCCESS;
234     }
235     return result;
236 }

```

10.2.6.4.4 CryptXORObfuscation()

This function implements XOR obfuscation. It should not be called if the hash algorithm is not implemented. The only return value from this function is TPM_RC_SUCCESS.

```

237 void
238 CryptXORObfuscation(
239     TPM_ALG_ID    hash,           // IN: hash algorithm for KDF
240     TPM2B         *key,           // IN: KDF key
241     TPM2B         *contextU,      // IN: contextU
242     TPM2B         *contextV,      // IN: contextV
243     UINT32        dataSize,       // IN: size of data buffer
244     BYTE          *data           // IN/OUT: data to be XORed in place
245 )
246 {
247     BYTE          mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
248     BYTE          *pm;
249     UINT32        i;
250     UINT32        counter = 0;
251     UINT16        hLen = CryptHashGetDigestSize(hash);
252     UINT32        requestSize = dataSize * 8;
253     INT32         remainBytes = (INT32)dataSize;
254
255     pAssert((key != NULL) && (data != NULL) && (hLen != 0));
256
257     // Call KDFa to generate XOR mask
258     for(; remainBytes > 0; remainBytes -= hLen)
259     {
260         // Make a call to KDFa to get next iteration
261         CryptKDFa(hash, key, XOR_KEY, contextU, contextV,
262                 requestSize, mask, &counter, TRUE);
263
264         // XOR next piece of the data
265         pm = mask;
266         for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
267             *data++ ^= *pm++;
268     }
269     return;
270 }

```

10.2.6.5 Initialization and shut down

10.2.6.5.1 CryptInit()

This function is called when the TPM receives a _TPM_Init indication.

NOTE The hash algorithms do not have to be tested, they just need to be available. They have to be tested before the TPM can accept HMAC authorization or return any result that relies on a hash algorithm.

Return Value	Meaning
TRUE(1)	initializations succeeded
FALSE(0)	initialization failed and caller should place the TPM into Failure Mode

```

271 BOOL
272 CryptInit(
273     void
274 )
275 {
276     BOOL    ok;

```

```

277 // Initialize the vector of implemented algorithms
278 AlgorithmGetImplementedVector(&g_implementedAlgorithms);
279
280 // Indicate that all test are necessary
281 CryptInitializeToTest();
282
283 // Do any library initializations that are necessary. If any fails,
284 // the caller should go into failure mode;
285 ok = SupportLibInit();
286 ok = ok && CryptSymInit();
287 ok = ok && CryptRandInit();
288 ok = ok && CryptHashInit();
289 #if ALG_RSA
290 ok = ok && CryptRsaInit();
291 #endif // ALG_RSA
292 #if ALG_ECC
293 ok = ok && CryptEccInit();
294 #endif // ALG_ECC
295 return ok;
296 }

```

10.2.6.5.2 CryptStartup()

This function is called by TPM2_Startup() to initialize the functions in this cryptographic library and in the provided CryptoLibrary(). This function and CryptUtilInit() are both provided so that the implementation may move the initialization around to get the best interaction.

Return Value	Meaning
TRUE(1)	startup succeeded
FALSE(0)	startup failed and caller should place the TPM into Failure Mode

```

297 BOOL
298 CryptStartup(
299     STARTUP_TYPE    type           // IN: the startup type
300 )
301 {
302     BOOL    OK;
303     NOT_REFERENCED(type);
304
305     OK = CryptSymStartup() && CryptRandStartup() && CryptHashStartup()
306 #if ALG_RSA
307     && CryptRsaStartup()
308 #endif // ALG_RSA
309 #if ALG_ECC
310     && CryptEccStartup()
311 #endif // ALG_ECC
312     ;
313 #if ALG_ECC
314     // Don't directly check for SU_RESET because that is the default
315     if(OK && (type != SU_RESTART) && (type != SU_RESUME))
316     {
317         // If the shutdown was orderly, then the values recovered from NV will
318         // be OK to use.
319         // Get a new random commit nonce
320         gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
321         CryptRandomGenerate(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
322         // Reset the counter and commit array
323         gr.commitCounter = 0;
324         MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
325     }
326 #endif // ALG_ECC

```

```

327     return OK;
328 }

```

10.2.6.6 Algorithm-Independent Functions

10.2.6.6.1 Introduction

These functions are used generically when a function of a general type (e.g., symmetric encryption) is required. The functions will modify the parameters as required to interface to the indicated algorithms.

10.2.6.6.2 CryptIsAsymAlgorithm()

This function indicates if an algorithm is an asymmetric algorithm.

Return Value	Meaning
TRUE(1)	if it is an asymmetric algorithm
FALSE(0)	if it is not an asymmetric algorithm

```

329  BOOL
330  CryptIsAsymAlgorithm(
331      TPM_ALG_ID      algID          // IN: algorithm ID
332  )
333  {
334      switch(algID)
335      {
336      #if ALG_RSA
337          case TPM_ALG_RSA:
338      #endif
339      #if ALG_ECC
340          case TPM_ALG_ECC:
341      #endif
342          return TRUE;
343          break;
344      default:
345          break;
346      }
347      return FALSE;
348  }

```

10.2.6.6.3 CryptSecretEncrypt()

This function creates a secret value and its associated secret structure using an asymmetric algorithm.

This function is used by TPM2_Rewrap() TPM2_MakeCredential(), and TPM2_Duplicate().

Error Return	Meaning
TPM_RC_ATTRIBUTES	<i>keyHandle</i> does not reference a valid decryption key
TPM_RC_KEY	invalid ECC key (public point is not on the curve)
TPM_RC_SCHEME	RSA key with an unsupported padding scheme
TPM_RC_VALUE	numeric value of the data to be decrypted is greater than the RSA key modulus

```

349  TPM_RC
350  CryptSecretEncrypt(
351      OBJECT          *encryptKey,    // IN: encryption key object
352      const TPM2B     *label,         // IN: a null-terminated string as L

```

```

353     TPM2B_DATA          *data,           // OUT: secret value
354     TPM2B_ENCRYPTED_SECRET *secret        // OUT: secret structure
355 )
356 {
357     TPMT_RSA_DECRYPT      scheme;
358     TPM_RC               result = TPM_RC_SUCCESS;
359 //
360     if(data == NULL || secret == NULL)
361         return TPM_RC_FAILURE;
362
363     // The output secret value has the size of the digest produced by the nameAlg.
364     data->t.size = CryptHashGetDigestSize(encryptKey->publicArea.nameAlg);
365     // The encryption scheme is OAEP using the nameAlg of the encrypt key.
366     scheme.scheme = TPM_ALG_OAEP;
367     scheme.details.anySig.hashAlg = encryptKey->publicArea.nameAlg;
368
369     if(!IS_ATTRIBUTE(encryptKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
370         return TPM_RC_ATTRIBUTES;
371     switch(encryptKey->publicArea.type)
372     {
373 #if ALG_RSA
374         case TPM_ALG_RSA:
375         {
376             // Create secret data from RNG
377             CryptRandomGenerate(data->t.size, data->t.buffer);
378
379             // Encrypt the data by RSA OAEP into encrypted secret
380             result = CryptRsaEncrypt((TPM2B_PUBLIC_KEY_RSA *)secret, &data->b,
381                                     encryptKey, &scheme, label, NULL);
382         }
383         break;
384 #endif // ALG_RSA
385
386 #if ALG_ECC
387         case TPM_ALG_ECC:
388         {
389             TPMS_ECC_POINT      eccPublic;
390             TPM2B_ECC_PARAMETER eccPrivate;
391             TPMS_ECC_POINT      eccSecret;
392             BYTE                *buffer = secret->t.secret;
393
394             // Need to make sure that the public point of the key is on the
395             // curve defined by the key.
396             if(!CryptEccIsPointOnCurve(
397                 encryptKey->publicArea.parameters.eccDetail.curveID,
398                 &encryptKey->publicArea.unique.ecc))
399                 result = TPM_RC_KEY;
400             else
401             {
402                 // Call crypto engine to create an auxiliary ECC key
403                 // We assume crypt engine initialization should always success.
404                 // Otherwise, TPM should go to failure mode.
405
406                 CryptEccNewKeyPair(&eccPublic, &eccPrivate,
407                                     encryptKey->publicArea.parameters.eccDetail.curveID);
408                 // Marshal ECC public to secret structure. This will be used by the
409                 // recipient to decrypt the secret with their private key.
410                 secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
411
412                 // Compute ECDH shared secret which is R = [d]Q where d is the
413                 // private part of the ephemeral key and Q is the public part of a
414                 // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
415                 // because the auxiliary ECC key is just created according to the
416                 // parameters of input ECC encrypt key.
417                 if(CryptEccPointMultiply(&eccSecret,
418                                         encryptKey->publicArea.parameters.eccDetail.curveID,

```

```

419         &encryptKey->publicArea.unique.ecc, &eccPrivate,
420         NULL, NULL)
421     != TPM_RC_SUCCESS)
422         result = TPM_RC_KEY;
423     else
424     {
425         // The secret value is computed from Z using KDFe as:
426         // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
427         // Where:
428         // HashID the nameAlg of the decrypt key
429         // Z the x coordinate (Px) of the product (P) of the point
430         // (Q) of the secret and the private x coordinate (de,V)
431         // of the decryption key
432         // Use a null-terminated string containing "SECRET"
433         // PartyUInfo the x coordinate of the point in the secret
434         // (Qe,U )
435         // PartyVInfo the x coordinate of the public key (Qs,V )
436         // bits the number of bits in the digest of HashID
437         // Retrieve seed from KDFe
438         CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
439                 label, &eccPublic.x.b,
440                 &encryptKey->publicArea.unique.ecc.x.b,
441                 data->t.size * 8, data->t.buffer);
442     }
443 }
444 }
445 break;
446 #endif // ALG_ECC
447 default:
448     FAIL(FATAL_ERROR_INTERNAL);
449     break;
450 }
451 return result;
452 }

```

10.2.6.6.4 CryptSecretDecrypt()

Decrypt a secret value by asymmetric (or symmetric) algorithm. This function is used for ActivateCredential() and Import for asymmetric decryption, and StartAuthSession() for both asymmetric and symmetric decryption process.

Error Return	Meaning
TPM_RC_ATTRIBUTES	RSA key is not a decryption key
TPM_RC_BINDING	Invalid RSA key (public and private parts are not cryptographically bound).
TPM_RC_ECC_POINT	ECC point in the secret is not on the curve
TPM_RC_INSUFFICIENT	failed to retrieve ECC point from the secret
TPM_RC_NO_RESULT	multiplication resulted in ECC point at infinity
TPM_RC_SIZE	data to decrypt is not of the same size as RSA key
TPM_RC_VALUE	For RSA key, numeric value of the encrypted data is greater than the modulus, or the recovered data is larger than the output buffer. For keyedHash or symmetric key, the secret is larger than the size of the digest produced by the name algorithm.
TPM_RC_FAILURE	internal error

```

453 TPM_RC
454 CryptSecretDecrypt(
455     OBJECT                *decryptKey,    // IN: decrypt key

```



```

456     TPM2B_NONCE          *nonceCaller,    // IN: nonceCaller. It is needed for
457                                     // symmetric decryption. For
458                                     // asymmetric decryption, this
459                                     // parameter is NULL
460     const TPM2B          *label,          // IN: a value for L
461     TPM2B_ENCRYPTED_SECRET *secret,        // IN: input secret
462     TPM2B_DATA           *data           // OUT: decrypted secret value
463 )
464 {
465     TPM_RC      result = TPM_RC_SUCCESS;
466
467     // Decryption for secret
468     switch(decryptKey->publicArea.type)
469     {
470 #if ALG_RSA
471         case TPM_ALG_RSA:
472         {
473             TPMT_RSA_DECRYPT      scheme;
474             TPMT_RSA_SCHEME      *keyScheme
475             = &decryptKey->publicArea.parameters.rsaDetail.scheme;
476             UINT16               digestSize;
477
478             scheme = *(TPMT_RSA_DECRYPT *)keyScheme;
479             // If the key scheme is TPM_ALG_NULL, set the scheme to OAEP and
480             // set the algorithm to the name algorithm.
481             if(scheme.scheme == TPM_ALG_NULL)
482             {
483                 // Use OAEP scheme
484                 scheme.scheme = TPM_ALG_OAEP;
485                 scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
486             }
487             // use the digestSize as an indicator of whether or not the scheme
488             // is using a supported hash algorithm.
489             // Note: depending on the scheme used for encryption, a hashAlg might
490             // not be needed. However, the return value has to have some upper
491             // limit on the size. In this case, it is the size of the digest of the
492             // hash algorithm. It is checked after the decryption is done but, there
493             // is no point in doing the decryption if the size is going to be
494             // 'wrong' anyway.
495             digestSize = CryptHashGetDigestSize(scheme.details.oaep.hashAlg);
496             if(scheme.scheme != TPM_ALG_OAEP || digestSize == 0)
497                 return TPM_RC_SCHEME;
498
499             // Set the output buffer capacity
500             data->t.size = sizeof(data->t.buffer);
501
502             // Decrypt seed by RSA OAEP
503             result = CryptRsaDecrypt(&data->b, &secret->b,
504                                     decryptKey, &scheme, label);
505             if((result == TPM_RC_SUCCESS) && (data->t.size > digestSize))
506                 result = TPM_RC_VALUE;
507         }
508         break;
509 #endif // ALG_RSA
510 #if ALG_ECC
511         case TPM_ALG_ECC:
512         {
513             TPMS_ECC_POINT      eccPublic;
514             TPMS_ECC_POINT      eccSecret;
515             BYTE                *buffer = secret->t.secret;
516             INT32               size = secret->t.size;
517
518             // Retrieve ECC point from secret buffer
519             result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
520             if(result == TPM_RC_SUCCESS)
521             {

```

```

522         result = CryptEccPointMultiply(&eccSecret,
523                                         decryptKey->publicArea.parameters.eccDetail.curveID,
524                                         &eccPublic, &decryptKey->sensitive.sensitive.ecc,
525                                         NULL, NULL);
526         if(result == TPM_RC_SUCCESS)
527         {
528             // Set the size of the "recovered" secret value to be the size
529             // of the digest produced by the nameAlg.
530             data->t.size =
531                 CryptHashGetDigestSize(decryptKey->publicArea.nameAlg);
532
533             // The secret value is computed from Z using KDFe as:
534             // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
535             // Where:
536             // HashID -- the nameAlg of the decrypt key
537             // Z -- the x coordinate (Px) of the product (P) of the point
538             //      (Q) of the secret and the private x coordinate (de,V)
539             //      of the decryption key
540             // Use -- a null-terminated string containing "SECRET"
541             // PartyUInfo -- the x coordinate of the point in the secret
542             //      (Qe,U )
543             // PartyVInfo -- the x coordinate of the public key (Qs,V )
544             // bits -- the number of bits in the digest of HashID
545             // Retrieve seed from KDFe
546             CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
547                     &eccPublic.x.b,
548                     &decryptKey->publicArea.unique.ecc.x.b,
549                     data->t.size * 8, data->t.buffer);
550         }
551     }
552 }
553 break;
554 #endif // ALG_ECC
555 #if !ALG_KEYEDHASH
556 #error "KEYEDHASH support is required"
557 #endif
558 case TPM_ALG_KEYEDHASH:
559     // The seed size can not be bigger than the digest size of nameAlg
560     if(secret->t.size >
561         CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
562         result = TPM_RC_VALUE;
563     else
564     {
565         // Retrieve seed by XOR Obfuscation:
566         // seed = XOR(secret, hash, key, nonceCaller, nullNonce)
567         // where:
568         // secret the secret parameter from the TPM2_StartAuthHMAC
569         // command that contains the seed value
570         // hash nameAlg of tpmKey
571         // key the key or data value in the object referenced by
572         // entityHandle in the TPM2_StartAuthHMAC command
573         // nonceCaller the parameter from the TPM2_StartAuthHMAC command
574         // nullNonce a zero-length nonce
575         // XOR Obfuscation in place
576         CryptXORObfuscation(decryptKey->publicArea.nameAlg,
577                             &decryptKey->sensitive.sensitive.bits.b,
578                             &nonceCaller->b, NULL,
579                             secret->t.size, secret->t.secret);
580         // Copy decrypted seed
581         MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
582     }
583     break;
584 case TPM_ALG_SYMCIPHER:
585     {
586         TPM2B_IV iv = {{0}};
587         TPMT_SYM_DEF_OBJECT *symDef;

```

```

588 // The seed size can not be bigger than the digest size of nameAlg
589 if(secret->t.size >
590     CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
591     result = TPM_RC_VALUE;
592 else
593 {
594     symDef = &decryptKey->publicArea.parameters.symDetail.sym;
595     iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
596                                             symDef->keyBits.sym);
597     if(iv.t.size == 0)
598         return TPM_RC_FAILURE;
599     if(nonceCaller->t.size >= iv.t.size)
600     {
601         MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
602     }
603     else
604     {
605         if(nonceCaller->t.size > sizeof(iv.t.buffer))
606             return TPM_RC_FAILURE;
607         MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
608                   nonceCaller->t.size);
609     }
610     // make sure secret will fit
611     if(secret->t.size > data->t.size)
612         return TPM_RC_FAILURE;
613     data->t.size = secret->t.size;
614     // CFB decrypt, using nonceCaller as iv
615     CryptSymmetricDecrypt(data->t.buffer, symDef->algorithm,
616                           symDef->keyBits.sym,
617                           decryptKey->sensitive.sensitive.sym.t.buffer,
618                           &iv, TPM_ALG_CFB, secret->t.size,
619                           secret->t.secret);
620 }
621 }
622 break;
623 default:
624     FAIL(FATAL_ERROR_INTERNAL);
625     break;
626 }
627 return result;
628 }

```

10.2.6.6.5 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```

629 void
630 CryptParameterEncryption(
631     TPM_HANDLE    handle,           // IN: encrypt session handle
632     TPM2B         *nonceCaller,     // IN: nonce caller
633     UINT16        leadingSizeInByte, // IN: the size of the leading size field in
634                                     // bytes
635     TPM2B_AUTH    *extraKey,        // IN: additional key material other than
636                                     // sessionAuth
637     BYTE          *buffer           // IN/OUT: parameter buffer to be encrypted
638 )
639 {
640     SESSION        *session = SessionGet(handle); // encrypt session
641     TPM2B_TYPE(TEMP_KEY, (sizeof(extraKey->t.buffer)
642                           + sizeof(session->sessionKey.t.buffer)));
643     TPM2B_TEMP_KEY key;             // encryption key
644     UINT32          cipherSize = 0; // size of cipher text
645     //
646     // Retrieve encrypted data size.

```

```

647     if(leadingSizeInByte == 2)
648     {
649         // Extract the first two bytes as the size field as the data size
650         // encrypt
651         cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
652         // advance the buffer
653         buffer = &buffer[2];
654     }
655 #ifdef TPM4B
656     else if(leadingSizeInByte == 4)
657     {
658         // use the first four bytes to indicate the number of bytes to encrypt
659         cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
660         //advance pointer
661         buffer = &buffer[4];
662     }
663 #endif
664     else
665     {
666         FAIL(FATAL_ERROR_INTERNAL);
667     }
668     // Compute encryption key by concatenating sessionKey with extra key
669     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
670     MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
671
672     if(session->symmetric.algorithm == TPM_ALG_XOR)
673     {
674         // XOR parameter encryption formulation:
675         // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
676         CryptXORObfuscation(session->authHashAlg, &(key.b),
677                             &(session->nonceTPM.b),
678                             nonceCaller, cipherSize, buffer);
679     }
680     else
681     {
682         ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
683                         session->symmetric.keyBits.aes, &(key.b),
684                         nonceCaller, &(session->nonceTPM.b),
685                         cipherSize, buffer);
686     }
687     return;
688 }

```

10.2.6.6.6 CryptParameterDecryption()

This function does in-place decryption of a command parameter.

Error Return	Meaning
TPM_RC_SIZE	The number of bytes in the input buffer is less than the number of bytes to be decrypted.

```

686 TPM_RC
687 CryptParameterDecryption(
688     TPM_HANDLE    handle,           // IN: encrypted session handle
689     TPM2B         *nonceCaller,     // IN: nonce caller
690     UINT32        bufferSize,       // IN: size of parameter buffer
691     UINT16        leadingSizeInByte, // IN: the size of the leading size field in
692                                     // byte
693     TPM2B_AUTH    *extraKey,        // IN: the authValue
694     BYTE          *buffer            // IN/OUT: parameter buffer to be decrypted
695 )
696 {
697     SESSION        *session = SessionGet(handle); // encrypt session
698     // The HMAC key is going to be the concatenation of the session key and any
699     // additional key material (like the authValue). The size of both of these
700     // is the size of the buffer which can contain a TPMT_HA.

```

```

701     TPM2B_TYPE(HMAC_KEY, (sizeof(extraKey->t.buffer)
702                          + sizeof(session->sessionKey.t.buffer)));
703     TPM2B_HMAC_KEY      key;          // decryption key
704     UINT32               cipherSize = 0; // size of cipher text
705 //
706 // Retrieve encrypted data size.
707 if(leadingSizeInByte == 2)
708 {
709     // The first two bytes of the buffer are the size of the
710     // data to be decrypted
711     cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
712     buffer = &buffer[2]; // advance the buffer
713 }
714 #ifdef TPM4B
715 else if(leadingSizeInByte == 4)
716 {
717     // the leading size is four bytes so get the four byte size field
718     cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
719     buffer = &buffer[4]; //advance pointer
720 }
721 #endif
722 else
723 {
724     FAIL(FATAL_ERROR_INTERNAL);
725 }
726 if(cipherSize > bufferSize)
727     return TPM_RC_SIZE;
728
729 // Compute decryption key by concatenating sessionAuth with extra input key
730 MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
731 MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
732
733 if(session->symmetric.algorithm == TPM_ALG_XOR)
734     // XOR parameter decryption formulation:
735     // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
736     // Call XOR obfuscation function
737     CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
738                        &(session->nonceTPM.b), cipherSize, buffer);
739 else
740     // Assume that it is one of the symmetric block ciphers.
741     ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,
742                   session->symmetric.keyBits.sym,
743                   &key.b, nonceCaller, &session->nonceTPM.b,
744                   cipherSize, buffer);
745
746 return TPM_RC_SUCCESS;
747 }

```

10.2.6.6.7 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```

748 void
749 CryptComputeSymmetricUnique(
750     TPMT_PUBLIC *publicArea, // IN: the object's public area
751     TPMT_SENSITIVE *sensitive, // IN: the associated sensitive area
752     TPM2B_DIGEST *unique // OUT: unique buffer
753 )
754 {
755     // For parents (symmetric and derivation), use an HMAC to compute
756     // the 'unique' field
757     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
758        && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt))
759     {

```

```

760     // Unique field is HMAC(sensitive->seedValue, sensitive->sensitive)
761     HMAC_STATE    hmacState;
762     unique->b.size = CryptHmacStart2B(&hmacState, publicArea->nameAlg,
763                                     &sensitive->seedValue.b);
764     CryptDigestUpdate2B(&hmacState.hashState,
765                        &sensitive->sensitive.any.b);
766     CryptHmacEnd2B(&hmacState, &unique->b);
767 }
768 else
769 {
770     HASH_STATE    hashState;
771     // Unique := Hash(sensitive->seedValue || sensitive->sensitive)
772     unique->t.size = CryptHashStart(&hashState, publicArea->nameAlg);
773     CryptDigestUpdate2B(&hashState, &sensitive->seedValue.b);
774     CryptDigestUpdate2B(&hashState, &sensitive->sensitive.any.b);
775     CryptHashEnd2B(&hashState, &unique->b);
776 }
777 return;
778 }

```

10.2.6.6.8 CryptCreateObject()

This function creates an object. For an asymmetric key, it will create a key pair and, for a parent key, a seed value for child protections.

For an symmetric object, (TPM_ALG_SYMCIPHER or TPM_ALG_KEYEDHASH), it will create a secret key if the caller did not provide one. It will create a random secret seed value that is hashed with the secret value to create the public unique value.

publicArea, *sensitive*, and *sensitiveCreate* are the only required parameters and are the only ones that are used by TPM2_Create(). The other parameters are optional and are used when the generated Object needs to be deterministic. This is the case for both Primary Objects and Derived Objects.

When a seed value is provided, a RAND_STATE will be populated and used for all operations in the object generation that require a random number. In the simplest case, TPM2_CreatePrimary() will use *seed*, *label* and *context* with context being the hash of the template. If the Primary Object is in the Endorsement hierarchy, it will also populate *proof* with *ehProof*.

For derived keys, *seed* will be the secret value from the parent, *label* and *context* will be set according to the parameters of TPM2_CreateLoaded() and *hashAlg* will be set which causes the RAND_STATE to be a KDF generator.

Error Return	Meaning
TPM_RC_KEY	a provided key is not an allowed value
TPM_RC_KEY_SIZE	key size in the public area does not match the size in the sensitive creation area for a symmetric key
TPM_RC_NO_RESULT	unable to get random values (only in derivation)
TPM_RC_RANGE	for an RSA key, the exponent is not supported
TPM_RC_SIZE	sensitive data size is larger than allowed for the scheme for a keyed hash object
TPM_RC_VALUE	exponent is not prime or could not find a prime using the provided parameters for an RSA key; unsupported name algorithm for an ECC key

```

779 TPM_RC
780 CryptCreateObject(
781     OBJECT                *object,           // IN: new object structure pointer
782     TPMS_SENSITIVE_CREATE *sensitiveCreate,  // IN: sensitive creation
783     RAND_STATE             *rand             // IN: the random number generator

```



```

784                                     //      to use
785     )
786 {
787     TPMT_PUBLIC          *publicArea = &object->publicArea;
788     TPMT_SENSITIVE       *sensitive = &object->sensitive;
789     TPM_RC               result = TPM_RC_SUCCESS;
790 //
791 // Set the sensitive type for the object
792 sensitive->sensitiveType = publicArea->type;
793
794 // For all objects, copy the initial authorization data
795 sensitive->authValue = sensitiveCreate->userAuth;
796
797 // If the TPM is the source of the data, set the size of the provided data to
798 // zero so that there's no confusion about what to do.
799 if(IS_ATTRIBUTE(publicArea->objectAttributes,
800                TPMA_OBJECT, sensitiveDataOrigin))
801     sensitiveCreate->data.t.size = 0;
802
803 // Generate the key and unique fields for the asymmetric keys and just the
804 // sensitive value for symmetric object
805 switch(publicArea->type)
806 {
807 #if ALG_RSA
808     // Create RSA key
809     case TPM_ALG_RSA:
810         // RSA uses full object so that it has a place to put the private
811         // exponent
812         result = CryptRsaGenerateKey(publicArea, sensitive, rand);
813         break;
814 #endif // ALG_RSA
815
816 #if ALG_ECC
817     // Create ECC key
818     case TPM_ALG_ECC:
819         result = CryptEccGenerateKey(publicArea, sensitive, rand);
820         break;
821 #endif // ALG_ECC
822     case TPM_ALG_SYMCIPHER:
823         result = CryptGenerateKeySymmetric(publicArea, sensitive,
824                                           sensitiveCreate, rand);
825         break;
826     case TPM_ALG_KEYEDHASH:
827         result = CryptGenerateKeyedHash(publicArea, sensitive,
828                                         sensitiveCreate, rand);
829         break;
830     default:
831         FAIL(FATAL_ERROR_INTERNAL);
832         break;
833 }
834 if(result != TPM_RC_SUCCESS)
835     return result;
836 // Create the sensitive seed value
837 // If this is a primary key in the endorsement hierarchy, stir the DRBG state
838 // This implementation uses both shProof and ehProof to make sure that there
839 // is no leakage of either.
840 if(object->attributes.primary && object->attributes.epsHierarchy)
841 {
842     DRBG_AdditionalData((DRBG_STATE *)rand, &gp.shProof.b);
843     DRBG_AdditionalData((DRBG_STATE *)rand, &gp.ehProof.b);
844 }
845 // Generate a seedValue that is the size of the digest produced by nameAlg
846 sensitive->seedValue.t.size =
847     DRBG_Generate(rand, sensitive->seedValue.t.buffer,
848                  CryptHashGetDigestSize(publicArea->nameAlg));
849 if(g_inFailureMode)

```



```

850     return TPM_RC_FAILURE;
851 else if(sensitive->seedValue.t.size == 0)
852     return TPM_RC_NO_RESULT;
853 // For symmetric objects, need to compute the unique value for the public area
854 if(publicArea->type == TPM_ALG_SYMCIPHER
855    || publicArea->type == TPM_ALG_KEYEDHASH)
856 {
857     CryptComputeSymmetricUnique(publicArea, sensitive, &publicArea->unique.sym);
858 }
859 else
860 {
861     // if this is an asymmetric key and it isn't a parent, then
862     // get rid of the seed.
863     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
864        || !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
865         memset(&sensitive->seedValue, 0, sizeof(sensitive->seedValue));
866 }
867 // Compute the name
868 PublicMarshalAndComputeName(publicArea, &object->name);
869 return result;
870 }

```

10.2.6.6.9 CryptGetSignHashAlg()

Get the hash algorithm of signature from a TPMT_SIGNATURE structure. It assumes the signature is not NULL This is a function for easy access

```

871 TPMI_ALG_HASH
872 CryptGetSignHashAlg(
873     TPMT_SIGNATURE *auth          // IN: signature
874 )
875 {
876     if(auth->sigAlg == TPM_ALG_NULL)
877         FAIL(FATAL_ERROR_INTERNAL);
878
879     // Get authHash algorithm based on signing scheme
880     switch(auth->sigAlg)
881     {
882 #if ALG_RSA
883     // If RSA is supported, both RSASSA and RSAPSS are required
884     # if !defined TPM_ALG_RSASSA || !defined TPM_ALG_RSAPSS
885     #     error "RSASSA and RSAPSS are required for RSA"
886     # endif
887         case TPM_ALG_RSASSA:
888             return auth->signature.rsassa.hash;
889         case TPM_ALG_RSAPSS:
890             return auth->signature.rsapss.hash;
891 #endif // ALG_RSA
892
893 #if ALG_ECC
894     // If ECC is defined, ECDSA is mandatory
895     # if !ALG_ECDSA
896     #     error "ECDSA is required for ECC"
897     # endif
898         case TPM_ALG_ECDSA:
899             // SM2 and ECSCHNORR are optional
900
901     # if ALG_SM2
902         case TPM_ALG_SM2:
903     # endif
904     # if ALG_ECSCHNORR
905         case TPM_ALG_ECSCHNORR:
906     # endif
907         //all ECC signatures look the same

```

```

908         return auth->signature.ecdsa.hash;
909
910 #   if ALG_ECDA
911     // Don't know how to verify an ECDA signature
912     case TPM_ALG_ECDA:
913         break;
914 #   endif
915 #endif // ALG_ECC
916
917     case TPM_ALG_HMAC:
918         return auth->signature.hmac.hashAlg;
919
920     default:
921         break;
922 }
923 return TPM_ALG_NULL;
924 }
925

```

10.2.6.6.10 CryptIsSplitSign()

This function is used to determine if the signing operation is a split signing operation that required a TPM2_Commit().

```

926 BOOL
927 CryptIsSplitSign(
928     TPM_ALG_ID    scheme           // IN: the algorithm selector
929 )
930 {
931     switch(scheme)
932     {
933 #   if ALG_ECDA
934         case TPM_ALG_ECDA:
935             return TRUE;
936             break;
937 #   endif // ALG_ECDA
938         default:
939             return FALSE;
940             break;
941     }
942 }

```

10.2.6.6.11 CryptIsAsymSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```

943 BOOL
944 CryptIsAsymSignScheme(
945     TPMI_ALG_PUBLIC    publicKey,           // IN: Type of the object
946     TPMI_ALG_ASYNC_SCHEME    scheme        // IN: the scheme
947 )
948 {
949     BOOL    isSignScheme = TRUE;
950
951     switch(publicType)
952     {
953 #if ALG_RSA
954         case TPM_ALG_RSA:
955             switch(scheme)
956             {
957 #   if !ALG_RSASSA || !ALG_RSAPSS
958                 error "RSASSA and PSAPSS required if RSA used."
959 #   endif
960             }
961         break;
962     }
963 }

```

```

960         case TPM_ALG_RSASSA:
961         case TPM_ALG_RSAPSS:
962             break;
963         default:
964             isSignScheme = FALSE;
965             break;
966     }
967     break;
968 #endif // ALG_RSA
969
970 #if ALG_ECC
971     // If ECC is implemented ECDSA is required
972     case TPM_ALG_ECC:
973         switch (scheme)
974         {
975             // Support for ECDSA is required for ECC
976             case TPM_ALG_ECDSA:
977 #if ALG_ECDAA // ECDAA is optional
978                 case TPM_ALG_ECDAA:
979 #endif
980 #if ALG_ECSCHNORR // Schnorr is also optional
981                 case TPM_ALG_ECSCHNORR:
982 #endif
983 #if ALG_SM2 // SM2 is optional
984                 case TPM_ALG_SM2:
985 #endif
986                 break;
987             default:
988                 isSignScheme = FALSE;
989                 break;
990         }
991     break;
992 #endif // ALG_ECC
993 default:
994     isSignScheme = FALSE;
995     break;
996 }
997 return isSignScheme;
998 }

```

10.2.6.6.12 CryptIsAsymDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```

999 BOOL
1000 CryptIsAsymDecryptScheme (
1001     TPMI_ALG_PUBLIC      publicKey,      // IN: Type of the object
1002     TPMI_ALG_ASYNC_SCHEME  scheme        // IN: the scheme
1003 )
1004 {
1005     BOOL      isDecryptScheme = TRUE;
1006
1007     switch (publicType)
1008     {
1009 #if ALG_RSA
1010         case TPM_ALG_RSA:
1011             switch (scheme)
1012             {
1013                 case TPM_ALG_RSAES:
1014                 case TPM_ALG_OAEP:
1015                     break;
1016             default:
1017                 isDecryptScheme = FALSE;
1018                 break;

```

```

1019         }
1020         break;
1021 #endif // ALG_RSA
1022
1023 #if ALG_ECC
1024     // If ECC is implemented ECDH is required
1025     case TPM_ALG_ECC:
1026         switch (scheme)
1027         {
1028             #if !ALG_ECDH
1029             # error "ECDH is required for ECC"
1030             #endif
1031             case TPM_ALG_ECDH:
1032             #if ALG_SM2
1033                 case TPM_ALG_SM2:
1034             #endif
1035             #if ALG_ECMQV
1036                 case TPM_ALG_ECMQV:
1037             #endif
1038                 break;
1039             default:
1040                 isDecryptScheme = FALSE;
1041                 break;
1042         }
1043         break;
1044 #endif // ALG_ECC
1045     default:
1046         isDecryptScheme = FALSE;
1047         break;
1048     }
1049     return isDecryptScheme;
1050 }

```

10.2.6.6.13 CryptSelectSignScheme()

This function is used by the attestation and signing commands. It implements the rules for selecting the signature scheme to use in signing. This function requires that the signing key either be TPM_RH_NULL or be loaded. If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both object and input scheme has a non-NULL scheme algorithm, if the schemes are compatible, the input scheme will be chosen. This function should not be called if *signObject->publicArea.type* == ALG_SYMCIPHER.

Return Value	Meaning
TRUE(1)	scheme selected
FALSE(0)	both <i>scheme</i> and key's default scheme are empty; or <i>scheme</i> is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from <i>scheme</i>

```

1051 BOOL
1052 CryptSelectSignScheme(
1053     OBJECT      *signObject,    // IN: signing key
1054     TPMT_SIG_SCHEME *scheme      // IN/OUT: signing scheme
1055 )
1056 {
1057     TPMT_SIG_SCHEME *objectScheme;
1058     TPMT_PUBLIC      *publicArea;
1059     BOOL             OK;
1060
1061     // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
1062     // of the setting of scheme
1063     if (signObject == NULL)

```

```

1064     {
1065         OK = TRUE;
1066         scheme->scheme = TPM_ALG_NULL;
1067         scheme->details.any.hashAlg = TPM_ALG_NULL;
1068     }
1069     else
1070     {
1071         // assignment to save typing.
1072         publicArea = &signObject->publicArea;
1073
1074         // A symmetric cipher can be used to encrypt and decrypt but it can't
1075         // be used for signing
1076         if(publicArea->type == TPM_ALG_SYMCIPHER)
1077             return FALSE;
1078         // Point to the scheme object
1079         if(CryptIsAsymAlgorithm(publicArea->type))
1080             objectScheme =
1081                 (TPMT_SIG_SCHEME *)&publicArea->parameters.asymDetail.scheme;
1082         else
1083             objectScheme =
1084                 (TPMT_SIG_SCHEME *)&publicArea->parameters.keyedHashDetail.scheme;
1085
1086         // If the object doesn't have a default scheme, then use the
1087         // input scheme.
1088         if(objectScheme->scheme == TPM_ALG_NULL)
1089         {
1090             // Input and default can't both be NULL
1091             OK = (scheme->scheme != TPM_ALG_NULL);
1092             // Assume that the scheme is compatible with the key. If not,
1093             // an error will be generated in the signing operation.
1094         }
1095         else if(scheme->scheme == TPM_ALG_NULL)
1096         {
1097             // input scheme is NULL so use default
1098
1099             // First, check to see if the default requires that the caller
1100             // provided scheme data
1101             OK = !CryptIsSplitSign(objectScheme->scheme);
1102             if(OK)
1103             {
1104                 // The object has a scheme and the input is TPM_ALG_NULL so copy
1105                 // the object scheme as the final scheme. It is better to use a
1106                 // structure copy than a copy of the individual fields.
1107                 *scheme = *objectScheme;
1108             }
1109         }
1110         else
1111         {
1112             // Both input and object have scheme selectors
1113             // If the scheme and the hash are not the same then...
1114             // NOTE: the reason that there is no copy here is that the input
1115             // might contain extra data for a split signing scheme and that
1116             // data is not in the object so, it has to be preserved.
1117             OK = (objectScheme->scheme == scheme->scheme)
1118                 && (objectScheme->details.any.hashAlg
1119                     == scheme->details.any.hashAlg);
1120         }
1121     }
1122     return OK;
1123 }

```

10.2.6.6.14 CryptSign()

Sign a digest with asymmetric key or HMAC. This function is called by attestation commands and the generic TPM2_Sign command. This function checks the key scheme and digest size. It does not check if the sign operation is allowed for restricted key. It should be checked before the function is called. The function will assert if the key is not a signing key.

Error Return	Meaning
TPM_RC_SCHEME	<i>signScheme</i> is not compatible with the signing key type
TPM_RC_VALUE	<i>digest</i> value is greater than the modulus of <i>signHandle</i> or size of <i>hashData</i> does not match hash algorithm <i>in'signScheme'</i> (for an RSA key); invalid commit status or failed to generate <i>r</i> value (for an ECC key)

```

1124 TPM_RC
1125 CryptSign(
1126     OBJECT          *signKey,          // IN: signing key
1127     TPMT_SIG_SCHEME *signScheme,       // IN: sign scheme.
1128     TPM2B_DIGEST    *digest,          // IN: The digest being signed
1129     TPMT_SIGNATURE  *signature        // OUT: signature
1130 )
1131 {
1132     TPM_RC result = TPM_RC_SCHEME;
1133
1134     // Initialize signature scheme
1135     signature->sigAlg = signScheme->scheme;
1136
1137     // If the signature algorithm is TPM_ALG_NULL or the signing key is NULL,
1138     // then we are done
1139     if((signature->sigAlg == TPM_ALG_NULL) || (signKey == NULL))
1140         return TPM_RC_SUCCESS;
1141
1142     // Initialize signature hash
1143     // Note: need to do the check for TPM_ALG_NULL first because the null scheme
1144     // doesn't have a hashAlg member.
1145     signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
1146
1147     // perform sign operation based on different key type
1148     switch(signKey->publicArea.type)
1149     {
1150 #if ALG_RSA
1151         case TPM_ALG_RSA:
1152             result = CryptRsaSign(signature, signKey, digest, NULL);
1153             break;
1154 #endif // ALG_RSA
1155 #if ALG_ECC
1156         case TPM_ALG_ECC:
1157             // The reason that signScheme is passed to CryptEccSign but not to the
1158             // other signing methods is that the signing for ECC may be split and
1159             // need the 'r' value that is in the scheme but not in the signature.
1160             result = CryptEccSign(signature, signKey, digest,
1161                                   (TPMT_ECC_SCHEME *)signScheme, NULL);
1162             break;
1163 #endif // ALG_ECC
1164         case TPM_ALG_KEYEDHASH:
1165             result = CryptHmacSign(signature, signKey, digest);
1166             break;
1167         default:
1168             FAIL(FATAL_ERROR_INTERNAL);
1169             break;
1170     }
1171     return result;
1172 }

```

10.2.6.6.15 CryptValidateSignature()

This function is used to verify a signature. It is called by TPM2_VerifySignature() and TPM2_PolicySigned(). Since this operation only requires use of a public key, no consistency checks are necessary for the key to signature type because a caller can load any public key that they like with any scheme that they like. This routine simply makes sure that the signature is correct, whatever the type.

Error Return	Meaning
TPM_RC_SIGNATURE	the signature is not genuine
TPM_RC_SCHEME	the scheme is not supported
TPM_RC_HANDLE	an HMAC key was selected but the private part of the key is not loaded

```

1173 TPM_RC
1174 CryptValidateSignature(
1175     TPMI_DH_OBJECT    keyHandle,      // IN: The handle of sign key
1176     TPM2B_DIGEST      *digest,        // IN: The digest being validated
1177     TPMT_SIGNATURE    *signature      // IN: signature
1178 )
1179 {
1180     // NOTE: HandleToObject will either return a pointer to a loaded object or
1181     // will assert. It will never return a non-valid value. This makes it safe
1182     // to initialize 'publicArea' with the return value from HandleToObject()
1183     // without checking it first.
1184     OBJECT *signObject = HandleToObject(keyHandle);
1185     TPMT_PUBLIC *publicArea = &signObject->publicArea;
1186     TPM_RC result = TPM_RC_SCHEME;
1187
1188     // The input unmarshaling should prevent any input signature from being
1189     // a NULL signature, but just in case
1190     if(signature->sigAlg == TPM_ALG_NULL)
1191         return TPM_RC_SIGNATURE;
1192
1193     switch(publicArea->type)
1194     {
1195 #if ALG_RSA
1196     case TPM_ALG_RSA:
1197     {
1198         //
1199         // Call RSA code to verify signature
1200         result = CryptRsaValidateSignature(signature, signObject, digest);
1201         break;
1202     }
1203 #endif // ALG_RSA
1204
1205 #if ALG_ECC
1206     case TPM_ALG_ECC:
1207     {
1208         result = CryptEccValidateSignature(signature, signObject, digest);
1209         break;
1210     }
1211 #endif // ALG_ECC
1212
1213     case TPM_ALG_KEYEDHASH:
1214     {
1215         if(signObject->attributes.publicOnly)
1216             result = TPM_RC_HANDLE;
1217         else
1218             result = CryptHMACVerifySignature(signObject, digest, signature);
1219         break;
1220     }
1221     default:
1222     {
1223         break;
1224     }
1225 }
1226 return result;
1227 }
```


10.2.6.6.16 CryptGetTestResult()

This function returns the results of a self-test function.

NOTE the behavior in this function is NOT the correct behavior for a real TPM implementation. An artificial behavior is placed here due to the limitation of a software simulation environment. For the correct behavior, consult the part 3 specification for TPM2_GetTestResult().

```

1222 TPM_RC
1223 CryptGetTestResult(
1224     TPM2B_MAX_BUFFER *outData      // OUT: test result data
1225 )
1226 {
1227     outData->t.size = 0;
1228     return TPM_RC_SUCCESS;
1229 }

```

10.2.6.6.17 CryptValidateKeys()

This function is used to verify that the key material of an object is valid. For a *publicOnly* object, the key is verified for size and, if it is an ECC key, it is verified to be on the specified curve. For a key with a sensitive area, the binding between the public and private parts of the key are verified. If the *nameAlg* of the key is TPM_ALG_NULL, then the size of the sensitive area is verified but the public portion is not verified, unless the key is an RSA key. For an RSA key, the reason for loading the sensitive area is to use it. The only way to use a private RSA key is to compute the private exponent. To compute the private exponent, the public modulus is used.

Error Return	Meaning
TPM_RC_BINDING	the public and private parts are not cryptographically bound
TPM_RC_HASH	cannot have a <i>publicOnly</i> key with <i>nameAlg</i> of TPM_ALG_NULL
TPM_RC_KEY	the public unique is not valid
TPM_RC_KEY_SIZE	the private area key is not valid
TPM_RC_TYPE	the types of the sensitive and private parts do not match

```

1230 TPM_RC
1231 CryptValidateKeys(
1232     TPMT_PUBLIC *publicArea,
1233     TPMT_SENSITIVE *sensitive,
1234     TPM_RC blamePublic,
1235     TPM_RC blameSensitive
1236 )
1237 {
1238     TPM_RC result;
1239     UINT16 keySizeInBytes;
1240     UINT16 digestSize = CryptHashGetDigestSize(publicArea->nameAlg);
1241     TPMU_PUBLIC_PARMS *params = &publicArea->parameters;
1242     TPMU_PUBLIC_ID *unique = &publicArea->unique;
1243
1244     if(sensitive != NULL)
1245     {
1246         // Make sure that the types of the public and sensitive are compatible
1247         if(publicArea->type != sensitive->sensitiveType)
1248             return TPM_RCS_TYPE + blameSensitive;
1249         // Make sure that the authValue is not bigger than allowed
1250         // If there is no name algorithm, then the size just needs to be less than
1251         // the maximum size of the buffer used for authorization. That size check
1252         // was made during unmarshaling of the sensitive area
1253         if((sensitive->authValue.t.size) > digestSize && (digestSize > 0))
1254             return TPM_RCS_SIZE + blameSensitive;
1255     }
1256 }

```

```

1255     }
1256     switch(publicArea->type)
1257     {
1258     #if ALG_RSA
1259         case TPM_ALG_RSA:
1260             keySizeInBytes = BITS_TO_BYTES(params->rsaDetail.keyBits);
1261
1262             // Regardless of whether there is a sensitive area, the public modulus
1263             // needs to have the correct size. Otherwise, it can't be used for
1264             // any public key operation nor can it be used to compute the private
1265             // exponent.
1266             // NOTE: This implementation only supports key sizes that are multiples
1267             // of 1024 bits which means that the MSb of the 0th byte will always be
1268             // SET in any prime and in the public modulus.
1269             if((unique->rsa.t.size != keySizeInBytes)
1270                || (unique->rsa.t.buffer[0] < 0x80))
1271                 return TPM_RCS_KEY + blamePublic;
1272             if(params->rsaDetail.exponent != 0
1273                && params->rsaDetail.exponent < 7)
1274                 return TPM_RCS_VALUE + blamePublic;
1275             if(sensitive != NULL)
1276             {
1277                 // If there is a sensitive area, it has to be the correct size
1278                 // including having the correct high order bit SET.
1279                 if(((sensitive->sensitive.rsa.t.size * 2) != keySizeInBytes)
1280                    || (sensitive->sensitive.rsa.t.buffer[0] < 0x80))
1281                     return TPM_RCS_KEY_SIZE + blameSensitive;
1282             }
1283             break;
1284     #endif
1285     #if ALG_ECC
1286         case TPM_ALG_ECC:
1287             {
1288                 TPMI_ECC_CURVE curveId;
1289                 curveId = params->eccDetail.curveID;
1290                 keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(curveId));
1291                 if(sensitive == NULL)
1292                 {
1293                     // Validate the public key size
1294                     if(unique->ecc.x.t.size != keySizeInBytes
1295                        || unique->ecc.y.t.size != keySizeInBytes)
1296                         return TPM_RCS_KEY + blamePublic;
1297                     if(publicArea->nameAlg != TPM_ALG_NULL)
1298                     {
1299                         if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))
1300                             return TPM_RCS_ECC_POINT + blamePublic;
1301                     }
1302                 }
1303                 else
1304                 {
1305                     // If the nameAlg is TPM_ALG_NULL, then only verify that the
1306                     // private part of the key is OK.
1307                     if(!CryptEccIsValidPrivateKey(&sensitive->sensitive.ecc,
1308                                                    curveId))
1309                         return TPM_RCS_KEY_SIZE;
1310                     if(publicArea->nameAlg != TPM_ALG_NULL)
1311                     {
1312                         // Full key load, verify that the public point belongs to the
1313                         // private key.
1314                         TPMS_ECC_POINT toCompare;
1315                         result = CryptEccPointMultiply(&toCompare, curveId, NULL,
1316                                                         &sensitive->sensitive.ecc,
1317                                                         NULL, NULL);
1318                         if(result != TPM_RC_SUCCESS)
1319                             return TPM_RCS_BINDING;
1320                     }
1321                 }
1322             }
1323     #endif
1324     }

```

```

1321     {
1322         // Make sure that the private key generated the public key.
1323         // The input values and the values produced by the point
1324         // multiply may not be the same size so adjust the computed
1325         // value to match the size of the input value by adding or
1326         // removing zeros.
1327         AdjustNumberB(&toCompare.x.b, unique->ecc.x.t.size);
1328         AdjustNumberB(&toCompare.y.b, unique->ecc.y.t.size);
1329         if(!MemoryEqual2B(&unique->ecc.x.b, &toCompare.x.b)
1330            || !MemoryEqual2B(&unique->ecc.y.b, &toCompare.y.b))
1331             return TPM_RCS_BINDING;
1332     }
1333 }
1334 }
1335 break;
1336 }
1337 #endif
1338 default:
1339     // Checks for SYMCIPHER and KEYEDHASH are largely the same
1340     // If public area has a nameAlg, then validate the public area size
1341     // and if there is also a sensitive area, validate the binding
1342
1343     // For consistency, if the object is public-only just make sure that
1344     // the unique field is consistent with the name algorithm
1345     if(sensitive == NULL)
1346     {
1347         if(unique->sym.t.size != digestSize)
1348             return TPM_RCS_KEY + blamePublic;
1349     }
1350     else
1351     {
1352         // Make sure that the key size in the sensitive area is consistent.
1353         if(publicArea->type == TPM_ALG_SYMCIPHER)
1354         {
1355             result = CryptSymKeyValidate(&params->symDetail.sym,
1356                                         &sensitive->sensitive.sym);
1357             if(result != TPM_RC_SUCCESS)
1358                 return result + blameSensitive;
1359         }
1360         else
1361         {
1362             // For a keyed hash object, the key has to be less than the
1363             // smaller of the block size of the hash used in the scheme or
1364             // 128 bytes. The worst case value is limited by the
1365             // unmarshaling code so the only thing left to be checked is
1366             // that it does not exceed the block size of the hash.
1367             // by the hash algorithm of the scheme.
1368             TPMT_KEYEDHASH_SCHEME *scheme;
1369             UINT16 maxSize;
1370             scheme = &params->keyedHashDetail.scheme;
1371             if(scheme->scheme == TPM_ALG_XOR)
1372             {
1373                 maxSize = CryptHashGetBlockSize(scheme->details.xor.hashAlg);
1374             }
1375             else if(scheme->scheme == TPM_ALG_HMAC)
1376             {
1377                 maxSize = CryptHashGetBlockSize(scheme->details.hmac.hashAlg);
1378             }
1379             else if(scheme->scheme == TPM_ALG_NULL)
1380             {
1381                 // Not signing or xor so must be a data block
1382                 maxSize = 128;
1383             }
1384             else
1385                 return TPM_RCS_SCHEME + blamePublic;
1386             if(sensitive->sensitive.bits.t.size > maxSize)

```

```

1387         return TPM_RCS_KEY_SIZE + blameSensitive;
1388     }
1389     // If there is a nameAlg, check the binding
1390     if(publicArea->nameAlg != TPM_ALG_NULL)
1391     {
1392         TPM2B_DIGEST compare;
1393         if(sensitive->seedValue.t.size != digestSize)
1394             return TPM_RCS_KEY_SIZE + blameSensitive;
1395
1396         CryptComputeSymmetricUnique(publicArea, sensitive, &compare);
1397         if(!MemoryEqual2B(&unique->sym.b, &compare.b))
1398             return TPM_RC_BINDING;
1399     }
1400 }
1401 break;
1402 }
1403 // For a parent, need to check that the seedValue is the correct size for
1404 // protections. It should be at least half the size of the nameAlg
1405 if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
1406    && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
1407    && sensitive != NULL
1408    && publicArea->nameAlg != TPM_ALG_NULL)
1409 {
1410     if((sensitive->seedValue.t.size < (digestSize / 2))
1411        || (sensitive->seedValue.t.size > digestSize))
1412         return TPM_RCS_SIZE + blameSensitive;
1413 }
1414 return TPM_RC_SUCCESS;
1415 }

```

10.2.6.6.18 CryptSelectMac()

This function is used to set the MAC scheme based on the key parameters and the input scheme.

Error Return	Meaning
TPM_RC_SCHEME	the scheme is not a valid mac scheme
TPM_RC_TYPE	the input key is not a type that supports a mac
TPM_RC_VALUE	the input scheme and the key scheme are not compatible

```

1416 TPM_RC
1417 CryptSelectMac(
1418     TPMT_PUBLIC *publicArea,
1419     TPMT_ALG_MAC_SCHEME *inMac
1420 )
1421 {
1422     TPM_ALG_ID macAlg = TPM_ALG_NULL;
1423     switch(publicArea->type)
1424     {
1425         case TPM_ALG_KEYEDHASH:
1426         {
1427             // Local value to keep lines from getting too long
1428             TPMT_KEYEDHASH_SCHEME *scheme;
1429             scheme = &publicArea->parameters.keyedHashDetail.scheme;
1430             // Expect that the scheme is either HMAC or NULL
1431             if(scheme->scheme != TPM_ALG_NULL)
1432                 macAlg = scheme->details.hmac.hashAlg;
1433             break;
1434         }
1435         case TPM_ALG_SYMCIPHER:
1436         {
1437             TPMT_SYM_DEF_OBJECT *scheme;
1438             scheme = &publicArea->parameters.symDetail.sym;

```

```

1439         // Expect that the scheme is either valid symmetric cipher or NULL
1440         if(scheme->algorithm != TPM_ALG_NULL)
1441             macAlg = scheme->mode.sym;
1442         break;
1443     }
1444     default:
1445         return TPM_RCS_TYPE;
1446 }
1447 // If the input value is not TPM_ALG_NULL ...
1448 if(*inMac != TPM_ALG_NULL)
1449 {
1450     // ... then either the scheme in the key must be TPM_ALG_NULL or the input
1451     // value must match
1452     if((macAlg != TPM_ALG_NULL) && (*inMac != macAlg))
1453         return TPM_RCS_VALUE;
1454 }
1455 else
1456 {
1457     // Since the input value is TPM_ALG_NULL, then the key value can't be
1458     // TPM_ALG_NULL
1459     if(macAlg == TPM_ALG_NULL)
1460         return TPM_RCS_VALUE;
1461     *inMac = macAlg;
1462 }
1463 if(!CryptMacIsValidForKey(publicArea->type, *inMac, FALSE))
1464     return TPM_RCS_SCHEME;
1465 return TPM_RC_SUCCESS;
1466 }

```

10.2.6.6.19 CryptMacIsValidForKey()

Check to see if the key type is compatible with the mac type

```

1467 BOOL
1468 CryptMacIsValidForKey(
1469     TPM_ALG_ID      keyType,
1470     TPM_ALG_ID      macAlg,
1471     BOOL            flag
1472 )
1473 {
1474     switch(keyType)
1475     {
1476         case TPM_ALG_KEYEDHASH:
1477             return CryptHashIsValidAlg(macAlg, flag);
1478             break;
1479         case TPM_ALG_SYMCIPHER:
1480             return CryptSmacIsValidAlg(macAlg, flag);
1481             break;
1482         default:
1483             break;
1484     }
1485     return FALSE;
1486 }

```

10.2.6.6.20 CryptSmacIsValidAlg()

This function is used to test if an algorithm is a supported SMAC algorithm. It needs to be updated as new algorithms are added.

```

1487 BOOL
1488 CryptSmacIsValidAlg(
1489     TPM_ALG_ID      alg,
1490     BOOL            FLAG      // IN: Indicates if TPM_ALG_NULL is valid

```

```

1491 )
1492 {
1493     switch (alg)
1494     {
1495 #if ALG_CMACE
1496         case TPM_ALG_CMACE:
1497             return TRUE;
1498             break;
1499 #endif
1500         case TPM_ALG_NULL:
1501             return FLAG;
1502             break;
1503         default:
1504             return FALSE;
1505     }
1506 }

```

10.2.6.6.21 CryptSymModelsValid()

Function checks to see if an algorithm ID is a valid, symmetric block cipher mode for the TPM. If *flag* is SET, then TPM_ALG_NULL is a valid mode. not include the modes used for SMAC

```

1507 BOOL
1508 CryptSymModeIsValid(
1509     TPM_ALG_ID      mode,
1510     BOOL            flag
1511 )
1512 {
1513     switch (mode)
1514     {
1515 #if ALG_CTR
1516         case TPM_ALG_CTR:
1517             // ALG_CTR
1518 #if ALG_OFB
1519         case TPM_ALG_OFB:
1520             // ALG_OFB
1521 #if ALG_CBC
1522         case TPM_ALG_CBC:
1523             // ALG_CBC
1524 #if ALG_CFB
1525         case TPM_ALG_CFB:
1526             // ALG_CFB
1527 #if ALG_ECB
1528         case TPM_ALG_ECB:
1529             // ALG_ECB
1530             return TRUE;
1531         case TPM_ALG_NULL:
1532             return flag;
1533             break;
1534         default:
1535             break;
1536     }
1537     return FALSE;
1538 }

```

10.2.7 CryptSelfTest.c

10.2.7.1 Introduction

The functions in this file are designed to support self-test of cryptographic functions in the TPM. The TPM allows the user to decide whether to run self-test on a demand basis or to run all the self-tests before proceeding.

The self-tests are controlled by a set of bit vectors. The *g_untestedDecryptionAlgorithms* vector has a bit for each decryption algorithm that needs to be tested and *g_untestedEncryptionAlgorithms* has a bit for each encryption algorithm that needs to be tested. Before an algorithm is used, the appropriate vector is checked (indexed using the algorithm ID). If the bit is 1, then the test function should be called.

For more information, see TpmSelfTests.txt

```
1 #include "Tpm.h"
```

10.2.7.2 Functions

10.2.7.2.1 RunSelfTest()

Local function to run self-test

```
2 static TPM_RC
3 CryptRunSelfTests (
4     ALGORITHM_VECTOR    *toTest           // IN: the vector of the algorithms to test
5 )
6 {
7     TPM_ALG_ID          alg;
8
9     // For each of the algorithms that are in the toTestVecor, need to run a
10    // test
11    for(alg = TPM_ALG_FIRST; alg <= TPM_ALG_LAST; alg++)
12    {
13        if(TEST_BIT(alg, *toTest))
14        {
15            TPM_RC          result = CryptTestAlgorithm(alg, toTest);
16            if(result != TPM_RC_SUCCESS)
17                return result;
18        }
19    }
20    return TPM_RC_SUCCESS;
21 }
```

10.2.7.2.2 CryptSelfTest()

This function is called to start/complete a full self-test. If *fullTest* is NO, then only the untested algorithms will be run. If *fullTest* is YES, then *g_untestedDecryptionAlgorithms* is reinitialized and then all tests are run. This implementation of the reference design does not support processing outside the framework of a TPM command. As a consequence, this command does not complete until all tests are done. Since this can take a long time, the TPM will check after each test to see if the command is canceled. If so, then the TPM will returned TPM_RC_CANCELLED. To continue with the self-tests, call TPM2_SelfTest(*fullTest* == No) and the TPM will complete the testing.

Error Return	Meaning
TPM_RC_CANCELED	if the command is canceled

```

22 LIB_EXPORT
23 TPM_RC(
24     TPMI_YES_NO      fullTest      // IN: if full test is required
25 )
26 {
27 #if SIMULATION
28     if(g_forceFailureMode)
29         FAIL(FATAL_ERROR_FORCED);
30 #endif
31
32     // If the caller requested a full test, then reset the to test vector so that
33     // all the tests will be run
34     if(fullTest == YES)
35     {
36         MemoryCopy(g_toTest,
37                     g_implementedAlgorithms,
38                     sizeof(g_toTest));
39     }
40     return CryptRunSelfTests(&g_toTest);
41 }

```

10.2.7.2.3 CryptIncrementalSelfTest()

This function is used to perform an incremental self-test. This implementation will perform the *toTest* values before returning. That is, it assumes that the TPM cannot perform background tasks between commands.

This command may be canceled. If it is, then there is no return result. However, this command can be run again and the incremental progress will not be lost.

Error Return	Meaning
TPM_RC_CANCELED	processing of this command was canceled
TPM_RC_TESTING	if <i>toTest</i> list is not empty
TPM_RC_VALUE	an algorithm in the <i>toTest</i> list is not implemented

```

42 TPM_RC
43 CryptIncrementalSelfTest(
44     TPMI_ALG      *toTest,          // IN: list of algorithms to be tested
45     TPMI_ALG      *toDoList        // OUT: list of algorithms needing test
46 )
47 {
48     ALGORITHM_VECTOR toTestVector = {0};
49     TPM_ALG_ID      alg;
50     UINT32           i;
51
52     pAssert(toTest != NULL && toDoList != NULL);
53     if(toTest->count > 0)
54     {
55         // Transcribe the toTest list into the toTestVector
56         for(i = 0; i < toTest->count; i++)
57         {
58             alg = toTest->algorithms[i];
59
60             // make sure that the algorithm value is not out of range
61             if((alg > TPM_ALG_LAST) || !TEST_BIT(alg, g_implementedAlgorithms))
62                 return TPM_RC_VALUE;
63             SET_BIT(alg, toTestVector);
64         }
65     }
66 }

```

```

64     }
65     // Run the test
66     if(CryptRunSelfTests(&toTestVector) == TPM_RC_CANCELED)
67         return TPM_RC_CANCELED;
68 }
69 // Fill in the toDoList with the algorithms that are still untested
70 toDoList->count = 0;
71
72 for(alg = TPM_ALG_FIRST;
73     toDoList->count < MAX_ALG_LIST_SIZE && alg <= TPM_ALG_LAST;
74     alg++)
75 {
76     if(TEST_BIT(alg, g_toTest))
77         toDoList->algorithms[toDoList->count++] = alg;
78 }
79 return TPM_RC_SUCCESS;
80 }

```

10.2.7.2.4 CryptInitializeToTest()

This function will initialize the data structures for testing all the algorithms. This should not be called unless CryptAlgsSetImplemented() has been called

```

81 void
82 CryptInitializeToTest(
83     void
84 )
85 {
86     // Indicate that nothing has been tested
87     memset(&g_cryptoSelfTestState, 0, sizeof(g_cryptoSelfTestState));
88
89     // Copy the implemented algorithm vector
90     MemoryCopy(g_toTest, g_implementedAlgorithms, sizeof(g_toTest));
91
92     // Setting the algorithm to null causes the test function to just clear
93     // out any algorithms for which there is no test.
94     CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);
95
96     return;
97 }

```

10.2.7.2.5 CryptTestAlgorithm()

Only point of contact with the actual self tests. If a self-test fails, there is no return and the TPM goes into failure mode. The call to TestAlgorithm() uses an algorithm selector and a bit vector. When the test is run, the corresponding bit in *toTest* and in *g_toTest* is CLEAR. If *toTest* is NULL, then only the bit in *g_toTest* is CLEAR. There is a special case for the call to TestAlgorithm(). When *alg* is ALG_ERROR, TestAlgorithm() will CLEAR any bit in *toTest* for which it has no test. This allows the knowledge about which algorithms have test to be accessed through the interface that provides the test.

Error Return	Meaning
TPM_RC_CANCELED	test was canceled

```

98 LIB_EXPORT
99 TPM_RC(
100     TPM_ALG_ID      alg,
101     ALGORITHM_VECTOR *toTest
102 )
103 {
104     TPM_RC      result;
105     #if SELF_TEST

```

```
106     result = TestAlgorithm(alg, toTest);
107 #else
108     // If this is an attempt to determine the algorithms for which there is a
109     // self test, pretend that all of them do. We do that by not clearing any
110     // of the algorithm bits. When/if this function is called to run tests, it
111     // will over report. This can be changed so that any call to check on which
112     // algorithms have tests, 'toTest' can be cleared.
113     if(alg != TPM_ALG_ERROR)
114     {
115         CLEAR_BIT(alg, g_toTest);
116         if(toTest != NULL)
117             CLEAR_BIT(alg, *toTest);
118     }
119     result = TPM_RC_SUCCESS;
120 #endif
121     return result;
122 }
```

10.2.8 CryptEccData.c

```

1  #include "Tpm.h"
2  #include "OIDs.h"

```

This file contains the ECC curve data. The format of the data depends on the setting of USE_BN_ECC_DATA. If it is defined, then the TPM's BigNum() format is used. Otherwise, it is kept in TPM2B format. The purpose of having the data in BigNum() format is so that it does not have to be reformatted before being used by the crypto library.

```

3  #if ALG_ECC
4
5  #if USE_BN_ECC_DATA
6  #   define TO_ECC_64                                TO_CRYPT_WORD_64
7  #   define TO_ECC_56(a, b, c, d, e, f, g)            TO_ECC_64(0, a, b, c, d, e, f, g)
8  #   define TO_ECC_48(a, b, c, d, e, f)              TO_ECC_64(0, 0, a, b, c, d, e, f)
9  #   define TO_ECC_40(a, b, c, d, e)                 TO_ECC_64(0, 0, 0, a, b, c, d, e)
10 #   if RADIX_BITS > 32
11 #       define TO_ECC_32(a, b, c, d)                 TO_ECC_64(0, 0, 0, 0, a, b, c, d)
12 #       define TO_ECC_24(a, b, c)                   TO_ECC_64(0, 0, 0, 0, 0, a, b, c)
13 #       define TO_ECC_16(a, b)                      TO_ECC_64(0, 0, 0, 0, 0, 0, a, b)
14 #       define TO_ECC_8(a)                          TO_ECC_64(0, 0, 0, 0, 0, 0, 0, a)
15 #   else // RADIX_BITS == 32
16 #       define TO_ECC_32                                BIG_ENDIAN_BYTES_TO_UINT32
17 #       define TO_ECC_24(a, b, c)                   TO_ECC_32(0, a, b, c)
18 #       define TO_ECC_16(a, b)                      TO_ECC_32(0, 0, a, b)
19 #       define TO_ECC_8(a)                          TO_ECC_32(0, 0, 0, a)
20 #   endif
21 #else // TPM2B
22 #   define TO_ECC_64(a, b, c, d, e, f, g, h)          a, b, c, d, e, f, g, h
23 #   define TO_ECC_56(a, b, c, d, e, f, g)            a, b, c, d, e, f, g
24 #   define TO_ECC_48(a, b, c, d, e, f)              a, b, c, d, e, f
25 #   define TO_ECC_40(a, b, c, d, e)                 a, b, c, d, e
26 #   define TO_ECC_32(a, b, c, d)                    a, b, c, d
27 #   define TO_ECC_24(a, b, c)                       a, b, c
28 #   define TO_ECC_16(a, b)                          a, b
29 #   define TO_ECC_8(a)                              a
30 #endif
31
32 #if USE_BN_ECC_DATA
33 #define BN_MIN_ALLOC(bytes)                          \
34     (BYTES_TO_CRYPT_WORDS(bytes) == 0) ? 1 : BYTES_TO_CRYPT_WORDS(bytes) \
35 # define ECC_CONST(NAME, bytes, initializer)          \
36     const struct {                                     \
37         crypt_ushort_t  allocate, size, d[BN_MIN_ALLOC(bytes)]; \
38         } NAME = {BN_MIN_ALLOC(bytes), BYTES_TO_CRYPT_WORDS(bytes), {initializer}} \
39 ECC_CONST(ECC_ZERO, 0, 0);
40
41 #else
42 # define ECC_CONST(NAME, bytes, initializer)          \
43     const TPM2B_##bytes##_BYTE_VALUE NAME = {bytes, {initializer}} \

```

Have to special case ECC_ZERO

```

44 TPM2B_BYTE_VALUE(1);
45 TPM2B_1_BYTE_VALUE ECC_ZERO = {1, {0}};
46
47 #endif
48
49 ECC_CONST(ECC_ONE, 1, 1);
50
51 #if !USE_BN_ECC_DATA
52 TPM2B_BYTE_VALUE(24);

```

```

53 #define TO_ECC_192(a, b, c) a, b, c
54 TPM2B_BYTE_VALUE(28);
55 #define TO_ECC_224(a, b, c, d) a, b, c, d
56 TPM2B_BYTE_VALUE(32);
57 #define TO_ECC_256(a, b, c, d) a, b, c, d
58 TPM2B_BYTE_VALUE(48);
59 #define TO_ECC_384(a, b, c, d, e, f) a, b, c, d, e, f
60 TPM2B_BYTE_VALUE(66);
61 #define TO_ECC_528(a, b, c, d, e, f, g, h, i) a, b, c, d, e, f, g, h, i
62 TPM2B_BYTE_VALUE(80);
63 #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j) a, b, c, d, e, f, g, h, i, j
64 #else
65 #define TO_ECC_192(a, b, c) c, b, a
66 #define TO_ECC_224(a, b, c, d) d, c, b, a
67 #define TO_ECC_256(a, b, c, d) d, c, b, a
68 #define TO_ECC_384(a, b, c, d, e, f) f, e, d, c, b, a
69 #define TO_ECC_528(a, b, c, d, e, f, g, h, i) i, h, g, f, e, d, c, b, a
70 #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j) j, i, h, g, f, e, d, c, b, a
71 #endif // !USE_BN_ECC_DATA
72
73 #if ECC_NIST_P192
74 ECC_CONST(NIST_P192_p, 24, TO_ECC_192(
75     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
76     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
77     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
78 ECC_CONST(NIST_P192_a, 24, TO_ECC_192(
79     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
80     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
81     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
82 ECC_CONST(NIST_P192_b, 24, TO_ECC_192(
83     TO_ECC_64(0x64, 0x21, 0x05, 0x19, 0xE5, 0x9C, 0x80, 0xE7),
84     TO_ECC_64(0x0F, 0xA7, 0xE9, 0xAB, 0x72, 0x24, 0x30, 0x49),
85     TO_ECC_64(0xFE, 0xB8, 0xDE, 0xEC, 0xC1, 0x46, 0xB9, 0xB1)));
86 ECC_CONST(NIST_P192_gX, 24, TO_ECC_192(
87     TO_ECC_64(0x18, 0x8D, 0xA8, 0x0E, 0xB0, 0x30, 0x90, 0xF6),
88     TO_ECC_64(0x7C, 0xBF, 0x20, 0xEB, 0x43, 0xA1, 0x88, 0x00),
89     TO_ECC_64(0xF4, 0xFF, 0x0A, 0xFD, 0x82, 0xFF, 0x10, 0x12)));
90 ECC_CONST(NIST_P192_gY, 24, TO_ECC_192(
91     TO_ECC_64(0x07, 0x19, 0x2B, 0x95, 0xFF, 0xC8, 0xDA, 0x78),
92     TO_ECC_64(0x63, 0x10, 0x11, 0xED, 0x6B, 0x24, 0xCD, 0xD5),
93     TO_ECC_64(0x73, 0xF9, 0x77, 0xA1, 0x1E, 0x79, 0x48, 0x11)));
94 ECC_CONST(NIST_P192_n, 24, TO_ECC_192(
95     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
96     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x99, 0xDE, 0xF8, 0x36),
97     TO_ECC_64(0x14, 0x6B, 0xC9, 0xB1, 0xB4, 0xD2, 0x28, 0x31)));
98 #define NIST_P192_h ECC_ONE
99 #define NIST_P192_gZ ECC_ONE
100
101 #if USE_BN_ECC_DATA
102     const ECC_CURVE_DATA NIST_P192 = {
103         (bigNum)&NIST_P192_p, (bigNum)&NIST_P192_n, (bigNum)&NIST_P192_h,
104         (bigNum)&NIST_P192_a, (bigNum)&NIST_P192_b,
105         {(bigNum)&NIST_P192_gX, (bigNum)&NIST_P192_gY, (bigNum)&NIST_P192_gZ}};
106 #else
107     const ECC_CURVE_DATA NIST_P192 = {
108         &NIST_P192_p.b, &NIST_P192_n.b, &NIST_P192_h.b,
109         &NIST_P192_a.b, &NIST_P192_b.b,
110         {&NIST_P192_gX.b, &NIST_P192_gY.b, &NIST_P192_gZ.b}};
111 #endif // USE_BN_ECC_DATA
112
113 #endif // ECC_NIST_P192
114
115 #if ECC_NIST_P224
116 ECC_CONST(NIST_P224_p, 28, TO_ECC_224(

```

```

119     TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
120     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
121     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
122     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01));
123 ECC_CONST(NIST_P224_a, 28, TO_ECC_224(
124     TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
125     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
126     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
127     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE));
128 ECC_CONST(NIST_P224_b, 28, TO_ECC_224(
129     TO_ECC_32(0xB4, 0x05, 0x0A, 0x85),
130     TO_ECC_64(0x0C, 0x04, 0xB3, 0xAB, 0xF5, 0x41, 0x32, 0x56),
131     TO_ECC_64(0x50, 0x44, 0xB0, 0xB7, 0xD7, 0xBF, 0xD8, 0xBA),
132     TO_ECC_64(0x27, 0x0B, 0x39, 0x43, 0x23, 0x55, 0xFF, 0xB4));
133 ECC_CONST(NIST_P224_gX, 28, TO_ECC_224(
134     TO_ECC_32(0xB7, 0x0E, 0x0C, 0xBD),
135     TO_ECC_64(0x6B, 0xB4, 0xBF, 0x7F, 0x32, 0x13, 0x90, 0xB9),
136     TO_ECC_64(0x4A, 0x03, 0xC1, 0xD3, 0x56, 0xC2, 0x11, 0x22),
137     TO_ECC_64(0x34, 0x32, 0x80, 0xD6, 0x11, 0x5C, 0x1D, 0x21));
138 ECC_CONST(NIST_P224_gY, 28, TO_ECC_224(
139     TO_ECC_32(0xBD, 0x37, 0x63, 0x88),
140     TO_ECC_64(0xB5, 0xF7, 0x23, 0xFB, 0x4C, 0x22, 0xDF, 0xE6),
141     TO_ECC_64(0xCD, 0x43, 0x75, 0xA0, 0x5A, 0x07, 0x47, 0x64),
142     TO_ECC_64(0x44, 0xD5, 0x81, 0x99, 0x85, 0x00, 0x7E, 0x34));
143 ECC_CONST(NIST_P224_n, 28, TO_ECC_224(
144     TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
145     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
146     TO_ECC_64(0xFF, 0xFF, 0x16, 0xA2, 0xE0, 0xB8, 0xF0, 0x3E),
147     TO_ECC_64(0x13, 0xDD, 0x29, 0x45, 0x5C, 0x5C, 0x2A, 0x3D));
148 #define NIST_P224_h      ECC_ONE
149 #define NIST_P224_gZ      ECC_ONE
150
151 #if USE_BN_ECC_DATA
152     const ECC_CURVE_DATA NIST_P224 = {
153         (bigNum)&NIST_P224_p, (bigNum)&NIST_P224_n, (bigNum)&NIST_P224_h,
154         (bigNum)&NIST_P224_a, (bigNum)&NIST_P224_b,
155         {(bigNum)&NIST_P224_gX, (bigNum)&NIST_P224_gY, (bigNum)&NIST_P224_gZ}};
156
157 #else
158     const ECC_CURVE_DATA NIST_P224 = {
159         &NIST_P224_p.b, &NIST_P224_n.b, &NIST_P224_h.b,
160         &NIST_P224_a.b, &NIST_P224_b.b,
161         {&NIST_P224_gX.b, &NIST_P224_gY.b, &NIST_P224_gZ.b}};
162
163 #endif // USE_BN_ECC_DATA
164
165 #endif // ECC_NIST_P224
166
167 #if ECC_NIST_P256
168 ECC_CONST(NIST_P256_p, 32, TO_ECC_256(
169     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
170     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
171     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
172     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF));
173 ECC_CONST(NIST_P256_a, 32, TO_ECC_256(
174     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
175     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
176     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
177     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC));
178 ECC_CONST(NIST_P256_b, 32, TO_ECC_256(
179     TO_ECC_64(0x5A, 0xC6, 0x35, 0xD8, 0xAA, 0x3A, 0x93, 0xE7),
180     TO_ECC_64(0xB3, 0xEB, 0xBD, 0x55, 0x76, 0x98, 0x86, 0xBC),
181     TO_ECC_64(0x65, 0x1D, 0x06, 0xB0, 0xCC, 0x53, 0xB0, 0xF6),
182     TO_ECC_64(0x3B, 0xCE, 0x3C, 0x3E, 0x27, 0xD2, 0x60, 0x4B));
183 ECC_CONST(NIST_P256_gX, 32, TO_ECC_256(
184     TO_ECC_64(0x6B, 0x17, 0xD1, 0xF2, 0xE1, 0x2C, 0x42, 0x47),

```

```

185     TO_ECC_64(0xF8, 0xBC, 0xE6, 0xE5, 0x63, 0xA4, 0x40, 0xF2),
186     TO_ECC_64(0x77, 0x03, 0x7D, 0x81, 0x2D, 0xEB, 0x33, 0xA0),
187     TO_ECC_64(0xF4, 0xA1, 0x39, 0x45, 0xD8, 0x98, 0xC2, 0x96));
188 ECC_CONST(NIST_P256_gY, 32, TO_ECC_256(
189     TO_ECC_64(0x4F, 0xE3, 0x42, 0xE2, 0xFE, 0x1A, 0x7F, 0x9B),
190     TO_ECC_64(0x8E, 0xE7, 0xEB, 0x4A, 0x7C, 0x0F, 0x9E, 0x16),
191     TO_ECC_64(0x2B, 0xCE, 0x33, 0x57, 0x6B, 0x31, 0x5E, 0xCE),
192     TO_ECC_64(0xCB, 0xB6, 0x40, 0x68, 0x37, 0xBF, 0x51, 0xF5));
193 ECC_CONST(NIST_P256_n, 32, TO_ECC_256(
194     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00),
195     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
196     TO_ECC_64(0xBC, 0xE6, 0xFA, 0xAD, 0xA7, 0x17, 0x9E, 0x84),
197     TO_ECC_64(0xF3, 0xB9, 0xCA, 0xC2, 0xFC, 0x63, 0x25, 0x51));
198 #define NIST_P256_h      ECC_ONE
199 #define NIST_P256_gZ      ECC_ONE
200
201 #if USE_BN_ECC_DATA
202     const ECC_CURVE_DATA NIST_P256 = {
203         (bigNum)&NIST_P256_p, (bigNum)&NIST_P256_n, (bigNum)&NIST_P256_h,
204         (bigNum)&NIST_P256_a, (bigNum)&NIST_P256_b,
205         {(bigNum)&NIST_P256_gX, (bigNum)&NIST_P256_gY, (bigNum)&NIST_P256_gZ}};
206
207 #else
208     const ECC_CURVE_DATA NIST_P256 = {
209         &NIST_P256_p.b, &NIST_P256_n.b, &NIST_P256_h.b,
210         &NIST_P256_a.b, &NIST_P256_b.b,
211         {&NIST_P256_gX.b, &NIST_P256_gY.b, &NIST_P256_gZ.b}};
212
213 #endif // USE_BN_ECC_DATA
214
215 #endif // ECC_NIST_P256
216
217 #if ECC_NIST_P384
218 ECC_CONST(NIST_P384_p, 48, TO_ECC_384(
219     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
220     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
221     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
222     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
223     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
224     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF));
225 ECC_CONST(NIST_P384_a, 48, TO_ECC_384(
226     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
227     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
228     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
229     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
230     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
231     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFC));
232 ECC_CONST(NIST_P384_b, 48, TO_ECC_384(
233     TO_ECC_64(0xB3, 0x31, 0x2F, 0xA7, 0xE2, 0x3E, 0xE7, 0xE4),
234     TO_ECC_64(0x98, 0x8E, 0x05, 0x6B, 0xE3, 0xF8, 0x2D, 0x19),
235     TO_ECC_64(0x18, 0x1D, 0x9C, 0x6E, 0xFE, 0x81, 0x41, 0x12),
236     TO_ECC_64(0x03, 0x14, 0x08, 0x8F, 0x50, 0x13, 0x87, 0x5A),
237     TO_ECC_64(0xC6, 0x56, 0x39, 0x8D, 0x8A, 0x2E, 0xD1, 0x9D),
238     TO_ECC_64(0x2A, 0x85, 0xC8, 0xED, 0xD3, 0xEC, 0x2A, 0xEF));
239 ECC_CONST(NIST_P384_gX, 48, TO_ECC_384(
240     TO_ECC_64(0xAA, 0x87, 0xCA, 0x22, 0xBE, 0x8B, 0x05, 0x37),
241     TO_ECC_64(0x8E, 0xB1, 0xC7, 0x1E, 0xF3, 0x20, 0xAD, 0x74),
242     TO_ECC_64(0x6E, 0x1D, 0x3B, 0x62, 0x8B, 0xA7, 0x9B, 0x98),
243     TO_ECC_64(0x59, 0xF7, 0x41, 0xE0, 0x82, 0x54, 0x2A, 0x38),
244     TO_ECC_64(0x55, 0x02, 0xF2, 0x5D, 0xBF, 0x55, 0x29, 0x6C),
245     TO_ECC_64(0x3A, 0x54, 0x5E, 0x38, 0x72, 0x76, 0x0A, 0xB7));
246 ECC_CONST(NIST_P384_gY, 48, TO_ECC_384(
247     TO_ECC_64(0x36, 0x17, 0xDE, 0x4A, 0x96, 0x26, 0x2C, 0x6F),
248     TO_ECC_64(0x5D, 0x9E, 0x98, 0xBF, 0x92, 0x92, 0xDC, 0x29),
249     TO_ECC_64(0xF8, 0xF4, 0x1D, 0xBD, 0x28, 0x9A, 0x14, 0x7C),
250     TO_ECC_64(0xE9, 0xDA, 0x31, 0x13, 0xB5, 0xF0, 0xB8, 0xC0),

```



```

251     TO_ECC_64(0x0A, 0x60, 0xB1, 0xCE, 0x1D, 0x7E, 0x81, 0x9D),
252     TO_ECC_64(0x7A, 0x43, 0x1D, 0x7C, 0x90, 0xEA, 0x0E, 0x5F));
253 ECC_CONST(NIST_P384_n, 48, TO_ECC_384(
254     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
255     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
256     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
257     TO_ECC_64(0xC7, 0x63, 0x4D, 0x81, 0xF4, 0x37, 0x2D, 0xDF),
258     TO_ECC_64(0x58, 0x1A, 0x0D, 0xB2, 0x48, 0xB0, 0xA7, 0x7A),
259     TO_ECC_64(0xEC, 0xEC, 0x19, 0x6A, 0xCC, 0xC5, 0x29, 0x73)));
260 #define NIST_P384_h      ECC_ONE
261 #define NIST_P384_gZ     ECC_ONE
262
263 #if USE_BN_ECC_DATA
264     const ECC_CURVE_DATA NIST_P384 = {
265         (bigNum)&NIST_P384_p, (bigNum)&NIST_P384_n, (bigNum)&NIST_P384_h,
266         (bigNum)&NIST_P384_a, (bigNum)&NIST_P384_b,
267         {(bigNum)&NIST_P384_gX, (bigNum)&NIST_P384_gY, (bigNum)&NIST_P384_gZ}};
268
269 #else
270     const ECC_CURVE_DATA NIST_P384 = {
271         &NIST_P384_p.b, &NIST_P384_n.b, &NIST_P384_h.b,
272         &NIST_P384_a.b, &NIST_P384_b.b,
273         {&NIST_P384_gX.b, &NIST_P384_gY.b, &NIST_P384_gZ.b}};
274
275 #endif // USE_BN_ECC_DATA
276
277 #endif // ECC_NIST_P384
278
279 #if ECC_NIST_P521
280 ECC_CONST(NIST_P521_p, 66, TO_ECC_528(
281     TO_ECC_16(0x01, 0xFF),
282     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
283     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
284     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
285     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
286     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
287     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
288     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
289     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF));
290 ECC_CONST(NIST_P521_a, 66, TO_ECC_528(
291     TO_ECC_16(0x01, 0xFF),
292     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
293     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
294     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
295     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
296     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
297     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
298     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
299     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF));
300 ECC_CONST(NIST_P521_b, 66, TO_ECC_528(
301     TO_ECC_16(0x00, 0x51),
302     TO_ECC_64(0x95, 0x3E, 0xB9, 0x61, 0x8E, 0x1C, 0x9A, 0x1F),
303     TO_ECC_64(0x92, 0x9A, 0x21, 0xA0, 0xB6, 0x85, 0x40, 0xEE),
304     TO_ECC_64(0xA2, 0xDA, 0x72, 0x5B, 0x99, 0xB3, 0x15, 0xF3),
305     TO_ECC_64(0xB8, 0xB4, 0x89, 0x91, 0x8E, 0xF1, 0x09, 0xE1),
306     TO_ECC_64(0x56, 0x19, 0x39, 0x51, 0xEC, 0x7E, 0x93, 0x7B),
307     TO_ECC_64(0x16, 0x52, 0xC0, 0xBD, 0x3B, 0xB1, 0xBF, 0x07),
308     TO_ECC_64(0x35, 0x73, 0xDF, 0x88, 0x3D, 0x2C, 0x34, 0xF1),
309     TO_ECC_64(0xEF, 0x45, 0x1F, 0xD4, 0x6B, 0x50, 0x3F, 0x00));
310 ECC_CONST(NIST_P521_gX, 66, TO_ECC_528(
311     TO_ECC_16(0x00, 0xC6),
312     TO_ECC_64(0x85, 0x8E, 0x06, 0xB7, 0x04, 0x04, 0xE9, 0xCD),
313     TO_ECC_64(0x9E, 0x3E, 0xCB, 0x66, 0x23, 0x95, 0xB4, 0x42),
314     TO_ECC_64(0x9C, 0x64, 0x81, 0x39, 0x05, 0x3F, 0xB5, 0x21),
315     TO_ECC_64(0xF8, 0x28, 0xAF, 0x60, 0x6B, 0x4D, 0x3D, 0xBA),
316     TO_ECC_64(0xA1, 0x4B, 0x5E, 0x77, 0xEF, 0xE7, 0x59, 0x28),

```

```

317     TO_ECC_64(0xFE, 0x1D, 0xC1, 0x27, 0xA2, 0xFF, 0xA8, 0xDE),
318     TO_ECC_64(0x33, 0x48, 0xB3, 0xC1, 0x85, 0x6A, 0x42, 0x9B),
319     TO_ECC_64(0xF9, 0x7E, 0x7E, 0x31, 0xC2, 0xE5, 0xBD, 0x66));
320 ECC_CONST(NIST_P521_gY, 66, TO_ECC_528(
321     TO_ECC_16(0x01, 0x18),
322     TO_ECC_64(0x39, 0x29, 0x6A, 0x78, 0x9A, 0x3B, 0xC0, 0x04),
323     TO_ECC_64(0x5C, 0x8A, 0x5F, 0xB4, 0x2C, 0x7D, 0x1B, 0xD9),
324     TO_ECC_64(0x98, 0xF5, 0x44, 0x49, 0x57, 0x9B, 0x44, 0x68),
325     TO_ECC_64(0x17, 0xAF, 0xBD, 0x17, 0x27, 0x3E, 0x66, 0x2C),
326     TO_ECC_64(0x97, 0xEE, 0x72, 0x99, 0x5E, 0xF4, 0x26, 0x40),
327     TO_ECC_64(0xC5, 0x50, 0xB9, 0x01, 0x3F, 0xAD, 0x07, 0x61),
328     TO_ECC_64(0x35, 0x3C, 0x70, 0x86, 0xA2, 0x72, 0xC2, 0x40),
329     TO_ECC_64(0x88, 0xBE, 0x94, 0x76, 0x9F, 0xD1, 0x66, 0x50)));
330 ECC_CONST(NIST_P521_n, 66, TO_ECC_528(
331     TO_ECC_16(0x01, 0xFF),
332     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
333     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
334     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
335     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
336     TO_ECC_64(0x51, 0x86, 0x87, 0x83, 0xBF, 0x2F, 0x96, 0x6B),
337     TO_ECC_64(0x7F, 0xCC, 0x01, 0x48, 0xF7, 0x09, 0xA5, 0xD0),
338     TO_ECC_64(0x3B, 0xB5, 0xC9, 0xB8, 0x89, 0x9C, 0x47, 0xAE),
339     TO_ECC_64(0xBB, 0x6F, 0xB7, 0x1E, 0x91, 0x38, 0x64, 0x09)));
340 #define NIST_P521_h      ECC_ONE
341 #define NIST_P521_gZ      ECC_ONE
342
343 #if USE_BN_ECC_DATA
344     const ECC_CURVE_DATA NIST_P521 = {
345         (bigNum)&NIST_P521_p, (bigNum)&NIST_P521_n, (bigNum)&NIST_P521_h,
346         (bigNum)&NIST_P521_a, (bigNum)&NIST_P521_b,
347         {(bigNum)&NIST_P521_gX, (bigNum)&NIST_P521_gY, (bigNum)&NIST_P521_gZ}};
348
349 #else
350     const ECC_CURVE_DATA NIST_P521 = {
351         &NIST_P521_p.b, &NIST_P521_n.b, &NIST_P521_h.b,
352         &NIST_P521_a.b, &NIST_P521_b.b,
353         {&NIST_P521_gX.b, &NIST_P521_gY.b, &NIST_P521_gZ.b}};
354
355 #endif // USE_BN_ECC_DATA
356
357 #endif // ECC_NIST_P521
358
359 #if ECC_BN_P256
360 ECC_CONST(BN_P256_p, 32, TO_ECC_256(
361     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
362     TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9F),
363     TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x98, 0x0A, 0x82),
364     TO_ECC_64(0xD3, 0x29, 0x2D, 0xDB, 0xAE, 0xD3, 0x30, 0x13)));
365 #define BN_P256_a      ECC_ZERO
366 ECC_CONST(BN_P256_b, 1, TO_ECC_8(3));
367 #define BN_P256_gX      ECC_ONE
368 ECC_CONST(BN_P256_gY, 1, TO_ECC_8(2));
369 ECC_CONST(BN_P256_n, 32, TO_ECC_256(
370     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
371     TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9E),
372     TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x99, 0x92, 0x1A),
373     TO_ECC_64(0xF6, 0x2D, 0x53, 0x6C, 0xD1, 0x0B, 0x50, 0x0D)));
374 #define BN_P256_h      ECC_ONE
375 #define BN_P256_gZ      ECC_ONE
376
377 #if USE_BN_ECC_DATA
378     const ECC_CURVE_DATA BN_P256 = {
379         (bigNum)&BN_P256_p, (bigNum)&BN_P256_n, (bigNum)&BN_P256_h,
380         (bigNum)&BN_P256_a, (bigNum)&BN_P256_b,
381         {(bigNum)&BN_P256_gX, (bigNum)&BN_P256_gY, (bigNum)&BN_P256_gZ}};
382

```

```

383 #else
384     const ECC_CURVE_DATA BN_P256 = {
385         &BN_P256_p.b, &BN_P256_n.b, &BN_P256_h.b,
386         &BN_P256_a.b, &BN_P256_b.b,
387         {&BN_P256_gX.b, &BN_P256_gY.b, &BN_P256_gZ.b}};
388
389 #endif // USE_BN_ECC_DATA
390
391 #endif // ECC_BN_P256
392
393 #if ECC_BN_P638
394 ECC_CONST(BN_P638_p, 80, TO_ECC_640(
395     TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
396     TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
397     TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
398     TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
399     TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
400     TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
401     TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
402     TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
403     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
404     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x67)));
405 #define BN_P638_a      ECC_ZERO
406 ECC_CONST(BN_P638_b, 2, TO_ECC_16(0x01, 0x01));
407 ECC_CONST(BN_P638_gX, 80, TO_ECC_640(
408     TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
409     TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
410     TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
411     TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
412     TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
413     TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
414     TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
415     TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
416     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
417     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x66)));
418 ECC_CONST(BN_P638_gY, 1, TO_ECC_8(0x10));
419 ECC_CONST(BN_P638_n, 80, TO_ECC_640(
420     TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
421     TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
422     TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
423     TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
424     TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
425     TO_ECC_64(0x60, 0x00, 0x86, 0x55, 0x00, 0x21, 0xE5, 0x55),
426     TO_ECC_64(0xFF, 0xFF, 0xF5, 0x4F, 0xFF, 0xF4, 0xEA, 0xC0),
427     TO_ECC_64(0x00, 0x00, 0x00, 0x49, 0x80, 0x01, 0x54, 0xD9),
428     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xED, 0xA0),
429     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x61)));
430 #define BN_P638_h      ECC_ONE
431 #define BN_P638_gZ      ECC_ONE
432
433 #if USE_BN_ECC_DATA
434     const ECC_CURVE_DATA BN_P638 = {
435         (bigNum)&BN_P638_p, (bigNum)&BN_P638_n, (bigNum)&BN_P638_h,
436         (bigNum)&BN_P638_a, (bigNum)&BN_P638_b,
437         {(bigNum)&BN_P638_gX, (bigNum)&BN_P638_gY, (bigNum)&BN_P638_gZ}};
438
439 #else
440     const ECC_CURVE_DATA BN_P638 = {
441         &BN_P638_p.b, &BN_P638_n.b, &BN_P638_h.b,
442         &BN_P638_a.b, &BN_P638_b.b,
443         {&BN_P638_gX.b, &BN_P638_gY.b, &BN_P638_gZ.b}};
444
445 #endif // USE_BN_ECC_DATA
446
447 #endif // ECC_BN_P638
448

```

```

449 #if ECC_SM2_P256
450 ECC_CONST(SM2_P256_p, 32, TO_ECC_256(
451     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
452     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
453     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
454     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
455 ECC_CONST(SM2_P256_a, 32, TO_ECC_256(
456     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
457     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
458     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
459     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
460 ECC_CONST(SM2_P256_b, 32, TO_ECC_256(
461     TO_ECC_64(0x28, 0xE9, 0xFA, 0x9E, 0x9D, 0x9F, 0x5E, 0x34),
462     TO_ECC_64(0x4D, 0x5A, 0x9E, 0x4B, 0xCF, 0x65, 0x09, 0xA7),
463     TO_ECC_64(0xF3, 0x97, 0x89, 0xF5, 0x15, 0xAB, 0x8F, 0x92),
464     TO_ECC_64(0xDD, 0xBC, 0xBD, 0x41, 0x4D, 0x94, 0x0E, 0x93)));
465 ECC_CONST(SM2_P256_gX, 32, TO_ECC_256(
466     TO_ECC_64(0x32, 0xC4, 0xAE, 0x2C, 0x1F, 0x19, 0x81, 0x19),
467     TO_ECC_64(0x5F, 0x99, 0x04, 0x46, 0x6A, 0x39, 0xC9, 0x94),
468     TO_ECC_64(0x8F, 0xE3, 0x0B, 0xBF, 0xF2, 0x66, 0x0B, 0xE1),
469     TO_ECC_64(0x71, 0x5A, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7)));
470 ECC_CONST(SM2_P256_gY, 32, TO_ECC_256(
471     TO_ECC_64(0xBC, 0x37, 0x36, 0xA2, 0xF4, 0xF6, 0x77, 0x9C),
472     TO_ECC_64(0x59, 0xBD, 0xCE, 0xE3, 0x6B, 0x69, 0x21, 0x53),
473     TO_ECC_64(0xD0, 0xA9, 0x87, 0x7C, 0xC6, 0x2A, 0x47, 0x40),
474     TO_ECC_64(0x02, 0xDF, 0x32, 0xE5, 0x21, 0x39, 0xF0, 0xA0)));
475 ECC_CONST(SM2_P256_n, 32, TO_ECC_256(
476     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
477     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
478     TO_ECC_64(0x72, 0x03, 0xDF, 0x6B, 0x21, 0xC6, 0x05, 0x2B),
479     TO_ECC_64(0x53, 0xBB, 0xF4, 0x09, 0x39, 0xD5, 0x41, 0x23)));
480 #define SM2_P256_h ECC_ONE
481 #define SM2_P256_gZ ECC_ONE
482
483 #if USE_BN_ECC_DATA
484     const ECC_CURVE_DATA SM2_P256 = {
485         (bigNum)&SM2_P256_p, (bigNum)&SM2_P256_n, (bigNum)&SM2_P256_h,
486         (bigNum)&SM2_P256_a, (bigNum)&SM2_P256_b,
487         {(bigNum)&SM2_P256_gX, (bigNum)&SM2_P256_gY, (bigNum)&SM2_P256_gZ}};
488
489 #else
490     const ECC_CURVE_DATA SM2_P256 = {
491         &SM2_P256_p.b, &SM2_P256_n.b, &SM2_P256_h.b,
492         &SM2_P256_a.b, &SM2_P256_b.b,
493         {&SM2_P256_gX.b, &SM2_P256_gY.b, &SM2_P256_gZ.b}};
494
495 #endif // USE_BN_ECC_DATA
496
497 #endif // ECC_SM2_P256
498
499 #define comma
500 const ECC_CURVE eccCurves[] = {
501 #if ECC_NIST_P192
502     comma
503     {TPM_ECC_NIST_P192,
504     192,
505     {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA256}}},
506     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
507     &NIST_P192,
508     OID_ECC_NIST_P192
509     CURVE_NAME("NIST_P192")}
510 #   undef comma
511 #   define comma ,
512 #endif // ECC_NIST_P192
513 #if ECC_NIST_P224
514     comma

```

```

515     {TPM_ECC_NIST_P224,
516     224,
517     {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA256}}},
518     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
519     &NIST_P224,
520     OID_ECC_NIST_P224
521     CURVE_NAME("NIST_P224")}
522 #   undef comma
523 #   define comma ,
524 #endif // ECC_NIST_P224
525 #if ECC_NIST_P256
526     comma
527     {TPM_ECC_NIST_P256,
528     256,
529     {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA256}}},
530     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
531     &NIST_P256,
532     OID_ECC_NIST_P256
533     CURVE_NAME("NIST_P256")}
534 #   undef comma
535 #   define comma ,
536 #endif // ECC_NIST_P256
537 #if ECC_NIST_P384
538     comma
539     {TPM_ECC_NIST_P384,
540     384,
541     {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA384}}},
542     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
543     &NIST_P384,
544     OID_ECC_NIST_P384
545     CURVE_NAME("NIST_P384")}
546 #   undef comma
547 #   define comma ,
548 #endif // ECC_NIST_P384
549 #if ECC_NIST_P521
550     comma
551     {TPM_ECC_NIST_P521,
552     521,
553     {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SHA512}}},
554     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
555     &NIST_P521,
556     OID_ECC_NIST_P521
557     CURVE_NAME("NIST_P521")}
558 #   undef comma
559 #   define comma ,
560 #endif // ECC_NIST_P521
561 #if ECC_BN_P256
562     comma
563     {TPM_ECC_BN_P256,
564     256,
565     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
566     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
567     &BN_P256,
568     OID_ECC_BN_P256
569     CURVE_NAME("BN_P256")}
570 #   undef comma
571 #   define comma ,
572 #endif // ECC_BN_P256
573 #if ECC_BN_P638
574     comma
575     {TPM_ECC_BN_P638,
576     638,
577     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
578     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},
579     &BN_P638,
580     OID_ECC_BN_P638

```

```
581     CURVE_NAME("BN_P638")}  
582 #   undef comma  
583 #   define comma ,  
584 #endif // ECC_BN_P638  
585 #if ECC_SM2_P256  
586     comma  
587     {TPM_ECC_SM2_P256,  
588     256,  
589     {TPM_ALG_KDF1_SP800_56A, {{TPM_ALG_SM3_256}}},  
590     {TPM_ALG_NULL, {{TPM_ALG_NULL}}},  
591     &SM2_P256,  
592     OID_ECC_SM2_P256  
593     CURVE_NAME("SM2_P256")}  
594 #   undef comma  
595 #   define comma ,  
596 #endif // ECC_SM2_P256  
597 };  
598 #endif // TPM_ALG_ECC
```

10.2.9 CryptDes.c

10.2.9.1 Introduction

This file contains the extra functions required for TDES.

10.2.9.2 Includes, Defines, and Typedefs

```

1  #include "Tpm.h"
2
3  #if ALG_TDES
4
5  #define DES_NUM_WEAK 64
6  const UINT64 DesWeakKeys[DES_NUM_WEAK] = {
7      0x0101010101010101ULL, 0xFEFEFEFEFEFEFEFEUULL,
8      0xE0E0E0E0F1F1F1F1ULL, 0x1F1F1F1F0E0E0E0EUULL,
9      0x011F011F010E010EUULL, 0x1F011F010E010E01ULL,
10     0x01E001E001F101F1UULL, 0xE001E001F101F101ULL,
11     0x01FE01FE01FE01FEUULL, 0xFE01FE01FE01FE01ULL,
12     0x1FE01FE00EF10EF1UULL, 0xE01FE01FF10EF10EUULL,
13     0x1FFE1FFE0EFE0EFEUULL, 0xFE1FFE1FFE0EFE0EUULL,
14     0xE0FEE0FEF1FEF1FEUULL, 0xFEE0FEE0FEF1FEF1UULL,
15     0x01011F1F01010E0EUULL, 0x1F1F01010E0E0101ULL,
16     0xE0E01F1FF1F10E0EUULL, 0x0101E0E00101F1F1UULL,
17     0x1F1FE0E00E0EF1F1UULL, 0xE0E0FEFEF1F1FEFEUULL,
18     0x0101FEFE0101FEFEUULL, 0x1F1FEFE0E0FEFEFEUULL,
19     0xE0FE011FF1FE010EUULL, 0x011F1F01010E0E01UULL,
20     0x1FE001FE0EF101FEUULL, 0xE0FE1F01F1FE0E01UULL,
21     0x011FE0FE010EF1FEUULL, 0x1FE0E01F0EF1F10EUULL,
22     0xE0FEFEE0F1FEFEF1UULL, 0x011FFEE0010EFEF1UULL,
23     0x1FE0FE010EF1FE01UULL, 0xFE0101FEFE0101FEUULL,
24     0x01E01FFE01F10EFEUULL, 0x1FFE01E00EFE01F1UULL,
25     0xFE011FE0FE010EF1UULL, 0xFE01E01FFE01F10EUULL,
26     0x1FFE0010EFEF101UULL, 0xFE1F01E0FE0E01F1UULL,
27     0x01E0E00101F1F101UULL, 0x1FFEFE1F0EFEFE0EUULL,
28     0xFE1FE001FE0EF101UULL, 0x01E0FE1F01F1FE0EUULL,
29     0xE00101E0F10101F1UULL, 0xFE1F1FEFE0E0EFEUULL,
30     0x01FE1FE001FE0EF1UULL, 0xE0011FFE1F010EFEUULL,
31     0xFEE0011FFE1F010EUULL, 0x01FEE01F01FEF10EUULL,
32     0xE001FE1FF101FE0EUULL, 0xFEE01F01FEF10E01UULL,
33     0x01FEFE0101FEFE01UULL, 0xE01F01FEF10E01FEUULL,
34     0xFEE0E0FEFEF1F1FEUULL, 0x1F01011F0E01010EUULL,
35     0xE01F1FE0F10E0EF1UULL, 0xFEFE0101FEFE0101UULL,
36     0x1F01E0FE0E01F1FEUULL, 0xE01FFE01F10EFE01UULL,
37     0xFEFE1F1FFEFE0E0EUULL, 0x1F01FEE00E01FEF1UULL,
38     0xE0E00101F1F10101UULL, 0xFEFE0E0FEFEF1F1UULL};

```

10.2.9.2.1 CryptSetOddByteParity()

This function sets the per byte parity of a 64-bit value. The least-significant bit of each byte is replaced with the odd parity of the other 7 bits in the byte. With odd parity, no byte will ever be 0x00.

```

39  UINT64
40  CryptSetOddByteParity(
41      UINT64      k
42  )
43  {
44      #define PMASK 0x0101010101010101ULL
45      UINT64      out;
46      k |= PMASK;  // set the parity bit
47      out = k;

```



```

48     k ^= k >> 4;
49     k ^= k >> 2;
50     k ^= k >> 1;
51     k &= PMASK;      // odd parity extracted
52     out ^= k;        // out is now even parity because parity bit was already set
53     out ^= PMASK;    // out is now even parity
54     return out;
55 }

```

10.2.9.2.2 CryptDesIsWeakKey()

Check to see if a DES key is on the list of weak, semi-weak, or possibly weak keys.

Return Value	Meaning
TRUE(1)	DES key is weak
FALSE(0)	DES key is not weak

```

56 static BOOL
57 CryptDesIsWeakKey(
58     UINT64          k
59 )
60 {
61     int             i;
62     //
63     for(i = 0; i < DES_NUM_WEAK; i++)
64     {
65         if(k == DesWeakKeys[i])
66             return TRUE;
67     }
68     return FALSE;
69 }

```

10.2.9.2.3 CryptDesValidateKey()

Function to check to see if the input key is a valid DES key where the definition of valid is that none of the elements are on the list of weak, semi-weak, or possibly weak keys; and that for two keys, K1!=K2, and for three keys that K1!=K2 and K2!=K3.

```

70 BOOL
71 CryptDesValidateKey(
72     TPM2B_SYM_KEY   *desKey      // IN: key to validate
73 )
74 {
75     UINT64           k[3];
76     int              i;
77     int              keys = (desKey->t.size + 7) / 8;
78     BYTE             *pk = desKey->t.buffer;
79     BOOL             ok;
80     //
81     // Note: 'keys' is the number of keys, not the maximum index for 'k'
82     ok = ((keys == 2) || (keys == 3)) && ((desKey->t.size % 8) == 0);
83     for(i = 0; ok && i < keys; pk += 8, i++)
84     {
85         k[i] = CryptSetOddByteParity(BYTE_ARRAY_TO_UINT64(pk));
86         ok = !CryptDesIsWeakKey(k[i]);
87     }
88     ok = ok && k[0] != k[1];
89     if(keys == 3)
90         ok = ok && k[1] != k[2];
91     return ok;
92 }

```

10.2.9.2.4 CryptGenerateKeyDes()

This function is used to create a DES key of the appropriate size. The key will have odd parity in the bytes.

```

93  TPM_RC
94  CryptGenerateKeyDes (
95      TPMT_PUBLIC          *publicArea,          // IN/OUT: The public area template
96                          //          for the new key.
97      TPMT_SENSITIVE        *sensitive,          // OUT: sensitive area
98      RAND_STATE            *rand                // IN: the "entropy" source for
99  )
100 {
101
102      // Assume that the publicArea key size has been validated and is a supported
103      // number of bits.
104      sensitive->sensitive.sym.t.size =
105          BITS_TO_BYTES(publicArea->parameters.symDetail.sym.keyBits.sym);
106      do
107      {
108          BYTE              *pK = sensitive->sensitive.sym.t.buffer;
109          int                i = (sensitive->sensitive.sym.t.size + 7) / 8;
110      // Use the random number generator to generate the required number of bits
111          if(DRBG_Generate(rand, pK, sensitive->sensitive.sym.t.size) == 0)
112              return TPM_RC_NO_RESULT;
113          for(; i > 0; pK += 8, i--)
114          {
115              UINT64        k = BYTE_ARRAY_TO_UINT64(pK);
116              k = CryptSetOddByteParity(k);
117              UINT64_TO_BYTE_ARRAY(k, pK);
118          }
119      } while(!CryptDesValidateKey(&sensitive->sensitive.sym));
120      return TPM_RC_SUCCESS;
121  }
122 #endif
123

```

10.2.10 CryptEccKeyExchange.c

10.2.10.1 Introduction

This file contains the functions that are used for the two-phase, ECC, key-exchange protocols

```
1  #include "Tpm.h"
2
3  #if CC_ZGen_2Phase == YES
```

10.2.10.2 Functions

```
4  #if ALG_ECMQV
```

10.2.10.2.1 avf1()

This function does the associated value computation required by MQV key exchange. Process:

- 1) Convert x_Q to an integer x_{qi} using the convention specified in Appendix C.3.
- 2) Calculate $x_{qm} = x_{qi} \bmod 2^{\text{ceil}(f/2)}$ (where $f = \text{ceil}(\log_2(n))$).
- 3) Calculate the associate value function $\text{avf}(Q) = x_{qm} + 2^{\text{ceil}(f/2)}$ Always returns TRUE(1).

```
5  static BOOL
6  avf1(
7      bigNum          bnX,          // IN/OUT: the reduced value
8      bigNum          bnN          // IN: the order of the curve
9  )
10 {
11     // compute f = 2^(ceil(ceil(log2(n)) / 2))
12     int              f = (BnSizeInBits(bnN) + 1) / 2;
13     // x' = 2^f + (x mod 2^f)
14     BnMaskBits(bnX, f); // This is mod 2*2^f but it doesn't matter because
15                          // the next operation will SET the extra bit anyway
16     BnSetBit(bnX, f);
17     return TRUE;
18 }
```

10.2.10.2.2 C_2_2_MQV()

This function performs the key exchange defined in SP800-56A 6.1.1.4 Full MQV, C(2, 2, ECC MQV).

CAUTION: Implementation of this function may require use of essential claims in patents not owned by TCG members.

Points QsB and QeB are required to be on the curve of $inQsA$. The function will fail, possibly catastrophically, if this is not the case.

Error Return	Meaning
TPM_RC_NO_RESULT	the value for dsA does not give a valid point on the curve

```
19 static TPM_RC
20 C_2_2_MQV(
21     TPMS_ECC_POINT      *outZ,          // OUT: the computed point
22     TPM_ECC_CURVE       curveId,       // IN: the curve for the computations
23     TPM2B_ECC_PARAMETER *dsA,          // IN: static private TPM key
24     TPM2B_ECC_PARAMETER *deA,          // IN: ephemeral private TPM key
25     TPMS_ECC_POINT      *QsB,          // IN: static public party B key
```

```

26     TPMS_ECC_POINT          *QeB          // IN: ephemeral public party B key
27 )
28 {
29     CURVE_INITIALIZED(E, curveId);
30     const ECC_CURVE_DATA    *C;
31     POINT(pQeA);
32     POINT_INITIALIZED(pQeB, QeB);
33     POINT_INITIALIZED(pQsB, QsB);
34     ECC_NUM(bnTa);
35     ECC_INITIALIZED(bnDeA, deA);
36     ECC_INITIALIZED(bnDsA, dsA);
37     ECC_NUM(bnN);
38     ECC_NUM(bnXeB);
39     TPM_RC          retVal;
40 //
41 // Parameter checks
42 if(E == NULL)
43     ERROR_RETURN(TPM_RC_VALUE);
44 pAssert(outZ != NULL && pQeB != NULL && pQsB != NULL && deA != NULL
45         && dsA != NULL);
46 C = AccessCurveData(E);
47 // Process:
48 // 1. implicitSigA = (de,A + avf(Qe,A)ds,A ) mod n.
49 // 2. P = h(implicitSigA)(Qe,B + avf(Qe,B)Qs,B).
50 // 3. If P = O, output an error indicator.
51 // 4. Z=xP, where xP is the x-coordinate of P.
52
53 // Compute the public ephemeral key pQeA = [de,A]G
54 if((retVal = BnPointMult(pQeA, CurveGetG(C), bnDeA, NULL, NULL, E))
55    != TPM_RC_SUCCESS)
56     goto Exit;
57
58 // 1. implicitSigA = (de,A + avf(Qe,A)ds,A ) mod n.
59 // tA := (ds,A + de,A avf(Xe,A)) mod n (3)
60 // Compute 'tA' = ('deA' + 'dsA' avf('XeA')) mod n
61 // Ta = avf(XeA);
62 BnCopy(bnTa, pQeA->x);
63 avf1(bnTa, bnN);
64 // do Ta = ds,A * Ta mod n = dsA * avf(XeA) mod n
65 BnModMult(bnTa, bnDsA, bnTa, bnN);
66 // now Ta = deA + Ta mod n = deA + dsA * avf(XeA) mod n
67 BnAdd(bnTa, bnTa, bnDeA);
68 BnMod(bnTa, bnN);
69
70 // 2. P = h(implicitSigA)(Qe,B + avf(Qe,B)Qs,B).
71 // Put this in because almost every case of h is == 1 so skip the call when
72 // not necessary.
73 if(!BnEqualWord(CurveGetCofactor(C), 1))
74     // Cofactor is not 1 so compute Ta := Ta * h mod n
75     BnModMult(bnTa, bnTa, CurveGetCofactor(C), CurveGetOrder(C));
76
77 // Now that 'tA' is (h * 'tA' mod n)
78 // 'outZ' = (tA)(Qe,B + avf(Qe,B)Qs,B).
79
80 // first, compute XeB = avf(XeB)
81 avf1(bnXeB, bnN);
82
83 // QsB := [XeB]QsB
84 BnPointMult(pQsB, pQsB, bnXeB, NULL, NULL, E);
85 BnEccAdd(pQeB, pQeB, pQsB, E);
86
87 // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
88 // If the result is not the point at infinity, return QeB
89 BnPointMult(pQeB, pQeB, bnTa, NULL, NULL, E);
90 if(BnEqualZero(pQeB->z))
91     ERROR_RETURN(TPM_RC_NO_RESULT);

```

```

92     // Convert BIGNUM E to TPM2B E
93     BnPointTo2B(outZ, pQeB, E);
94
95 Exit:
96     CURVE_FREE(E);
97     return retVal;
98 }
99 #endif // ALG_ECMQV

```

10.2.10.2.3 C_2_2_ECDH()

This function performs the two phase key exchange defined in SP800-56A, 6.1.1.2 Full Unified Model, C(2, 2, ECC CDH).

```

100 static TPM_RC
101 C_2_2_ECDH(
102     TPMS_ECC_POINT      *outZs,           // OUT: Zs
103     TPMS_ECC_POINT      *outZe,           // OUT: Ze
104     TPM_ECC_CURVE        curveId,         // IN: the curve for the computations
105     TPM2B_ECC_PARAMETER  *dsA,            // IN: static private TPM key
106     TPM2B_ECC_PARAMETER  *deA,            // IN: ephemeral private TPM key
107     TPMS_ECC_POINT      *QsB,            // IN: static public party B key
108     TPMS_ECC_POINT      *QeB,            // IN: ephemeral public party B key
109 )
110 {
111     CURVE_INITIALIZED(E, curveId);
112     ECC_INITIALIZED(bnAs, dsA);
113     ECC_INITIALIZED(bnAe, deA);
114     POINT_INITIALIZED(ecBs, QsB);
115     POINT_INITIALIZED(ecBe, QeB);
116     POINT(ecZ);
117     TPM_RC      retVal;
118 //
119 // Parameter checks
120 if(E == NULL)
121     ERROR_RETURN(TPM_RC_CURVE);
122 pAssert(outZs != NULL && dsA != NULL && deA != NULL && QsB != NULL
123     && QeB != NULL);
124
125 // Do the point multiply for the Zs value ([dsA]QsB)
126 retVal = BnPointMult(ecZ, ecBs, bnAs, NULL, NULL, E);
127 if(retVal == TPM_RC_SUCCESS)
128 {
129     // Convert the Zs value.
130     BnPointTo2B(outZs, ecZ, E);
131     // Do the point multiply for the Ze value ([deA]QeB)
132     retVal = BnPointMult(ecZ, ecBe, bnAe, NULL, NULL, E);
133     if(retVal == TPM_RC_SUCCESS)
134         BnPointTo2B(outZe, ecZ, E);
135 }
136 Exit:
137     CURVE_FREE(E);
138     return retVal;
139 }

```

10.2.10.2.4 CryptEcc2PhaseKeyExchange()

This function is the dispatch routine for the EC key exchange functions that use two ephemeral and two static keys.

Error Return	Meaning
TPM_RC_SCHEME	scheme is not defined

```

140 LIB_EXPORT TPM_RC
141 CryptEcc2PhaseKeyExchange(
142     TPMS_ECC_POINT *outZ1,           // OUT: a computed point
143     TPMS_ECC_POINT *outZ2,           // OUT: and optional second point
144     TPM_ECC_CURVE curveId,           // IN: the curve for the computations
145     TPM_ALG_ID scheme,                // IN: the key exchange scheme
146     TPM2B_ECC_PARAMETER *dsA,        // IN: static private TPM key
147     TPM2B_ECC_PARAMETER *deA,        // IN: ephemeral private TPM key
148     TPMS_ECC_POINT *QsB,             // IN: static public party B key
149     TPMS_ECC_POINT *QeB,             // IN: ephemeral public party B key
150 )
151 {
152     pAssert(outZ1 != NULL
153             && dsA != NULL && deA != NULL
154             && QsB != NULL && QeB != NULL);
155
156     // Initialize the output points so that they are empty until one of the
157     // functions decides otherwise
158     outZ1->x.b.size = 0;
159     outZ1->y.b.size = 0;
160     if(outZ2 != NULL)
161     {
162         outZ2->x.b.size = 0;
163         outZ2->y.b.size = 0;
164     }
165     switch(scheme)
166     {
167         case TPM_ALG_ECDH:
168             return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
169         break;
170     #if ALG_ECMQV
171         case TPM_ALG_ECMQV:
172             return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
173         break;
174     #endif
175     #if ALG_SM2
176         case TPM_ALG_SM2:
177             return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
178         break;
179     #endif
180         default:
181             return TPM_RC_SCHEME;
182     }
183 }
184 #if ALG_SM2

```

10.2.10.2.5 ComputeWForSM2()

Compute the value for w used by SM2

```

185 static UINT32
186 ComputeWForSM2(
187     bigCurve E
188 )
189 {
190     // w := ceil(ceil(log2(n)) / 2) - 1
191     return (BnMsb(CurveGetOrder(AccessCurveData(E))) / 2 - 1);
192 }

```

10.2.10.2.6 avfSm2()

This function does the associated value computation required by SM2 key exchange. This is different from the avf() in the international standards because it returns a value that is half the size of the value returned by the standard avf(). For example, if n is 15, Ws (w in the standard) is 2 but the W here is 1. This means that an input value of 14 (1110b) would return a value of 110b with the standard but 10b with the scheme in SM2.

```

193 static bigNum
194 avfSm2(
195     bigNum          bn,          // IN/OUT: the reduced value
196     UINT32          w           // IN: the value of w
197 )
198 {
199     // a) set w := ceil(ceil(log2(n)) / 2) - 1
200     // b) set x' := 2^w + (x & (2^w - 1))
201     // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
202
203     BnMaskBits(bn, w); // as with avf1, this is too big by a factor of 2 but
204                        // it doesn't matter because we SET the extra bit
205                        // anyway
206     BnSetBit(bn, w);
207     return bn;
208 }

```

10.2.10.2.7 SM2KeyExchange()

This function performs the key exchange defined in SM2. The first step is to compute $tA = (dsA + deA \text{ avf}(Xe, A)) \bmod n$. Then, compute the Z value from $outZ = (h \ tA \bmod n) (QsA + [\text{avf}(QeB, x)](QeB))$. The function will compute the ephemeral public key from the ephemeral private key. All points are required to be on the curve of $inQsA$. The function will fail catastrophically if this is not the case

Error Return	Meaning
TPM_RC_NO_RESULT	the value for dsA does not give a valid point on the curve

```

209 LIB_EXPORT TPM_RC
210 SM2KeyExchange(
211     TPMS_ECC_POINT *outZ,          // OUT: the computed point
212     TPM_ECC_CURVE  curveId,        // IN: the curve for the computations
213     TPM2B_ECC_PARAMETER *dsAIn,    // IN: static private TPM key
214     TPM2B_ECC_PARAMETER *deAIn,    // IN: ephemeral private TPM key
215     TPMS_ECC_POINT *QsBIn,         // IN: static public party B key
216     TPMS_ECC_POINT *QeBIn,         // IN: ephemeral public party B key
217 )
218 {
219     CURVE_INITIALIZED(E, curveId);
220     const ECC_CURVE_DATA *C;
221     ECC_INITIALIZED(dsA, dsAIn);
222     ECC_INITIALIZED(deA, deAIn);
223     POINT_INITIALIZED(QsB, QsBIn);
224     POINT_INITIALIZED(QeB, QeBIn);
225     BN_WORD_INITIALIZED(One, 1);
226     POINT(QeA);
227     ECC_NUM(XeB);
228     POINT(Z);
229     ECC_NUM(Ta);
230     UINT32 w;
231     TPM_RC retVal = TPM_RC_NO_RESULT;
232     //
233     // Parameter checks
234     if(E == NULL)

```



```

235     ERROR_RETURN(TPM_RC_CURVE);
236     C = AccessCurveData(E);
237     pAssert(outZ != NULL && dsA != NULL && deA != NULL && QsB != NULL
238           && QeB != NULL);
239
240     // Compute the value for w
241     w = ComputeWForSM2(E);
242
243     // Compute the public ephemeral key pQeA = [de,A]G
244     if(!BnEccModMult(QeA, CurveGetG(C), deA, E))
245         goto Exit;
246
247     // tA := (ds,A + de,A avf(Xe,A)) mod n (3)
248     // Compute 'tA' = ('dsA' + 'deA' avf('XeA')) mod n
249     // Ta = avf(XeA);
250     // do Ta = de,A * Ta = deA * avf(XeA)
251     BnMult(Ta, deA, avfSm2(QeA->x, w));
252     // now Ta = dsA + Ta = dsA + deA * avf(XeA)
253     BnAdd(Ta, dsA, Ta);
254     BnMod(Ta, CurveGetOrder(C));
255
256     // outZ = [h tA mod n] (Qs,B + [avf(Xe,B)](Qe,B)) (4)
257     // Put this in because almost every case of h is == 1 so skip the call when
258     // not necessary.
259     if(!BnEqualWord(CurveGetCofactor(C), 1))
260         // Cofactor is not 1 so compute Ta := Ta * h mod n
261         BnModMult(Ta, Ta, CurveGetCofactor(C), CurveGetOrder(C));
262     // Now that 'tA' is (h * 'tA' mod n)
263     // 'outZ' = ['tA'] (QsB + [avf(QeB.x)](QeB)).
264     BnCopy(XeB, QeB->x);
265     if(!BnEccModMult2(Z, QsB, One, QeB, avfSm2(XeB, w), E))
266         goto Exit;
267     // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
268     if(!BnEccModMult(Z, Z, Ta, E))
269         goto Exit;
270     // Convert BIGNUM E to TPM2B E
271     BnPointTo2B(outZ, Z, E);
272     retVal = TPM_RC_SUCCESS;
273 Exit:
274     CURVE_FREE(E);
275     return retVal;
276 }
277 #endif
278
279 #endif // CC_ZGen_2Phase

```

10.2.11 CryptEccMain.c

10.2.11.1 Includes and Defines

```
1  #include "Tpm.h"
2
3  #if ALG_ECC
```

This version requires that the new format for ECC data be used

```
4  #if !USE_BN_ECC_DATA
5  #error "Need to SET USE_BN_ECC_DATA to YES in Implementaion.h"
6  #endif
```

10.2.11.2 Functions

```
7  #if SIMULATION
8  void
9  EccSimulationEnd(
10     void
11 )
12 {
13 #if SIMULATION
14 // put things to be printed at the end of the simulation here
15 #endif
16 }
17 #endif // SIMULATION
```

10.2.11.2.1 CryptEccInit()

This function is called at _TPM_Init

```
18  BOOL
19  CryptEccInit(
20     void
21 )
22 {
23     return TRUE;
24 }
```

10.2.11.2.2 CryptEccStartup()

This function is called at TPM2_Startup().

```
25  BOOL
26  CryptEccStartup(
27     void
28 )
29 {
30     return TRUE;
31 }
```

10.2.11.2.3 ClearPoint2B(generic)

Initialize the size values of a TPMS_ECC_POINT structure.

```
32  void
33  ClearPoint2B(
```

```

34     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 }

```

10.2.11.2.4 CryptEccGetParametersByCurveId()

This function returns a pointer to the curve data that is associated with the indicated *curveId*. If there is no curve with the indicated ID, the function returns NULL. This function is in this module so that it can be called by GetCurve() data.

Return Value	Meaning
NULL	curve with the indicated TPM_ECC_CURVE is not implemented
!= NULL	pointer to the curve data

```

43 LIB_EXPORT const ECC_CURVE *
44 CryptEccGetParametersByCurveId(
45     TPM_ECC_CURVE      curveId      // IN: the curveID
46 )
47 {
48     int      i;
49     for(i = 0; i < ECC_CURVE_COUNT; i++)
50     {
51         if(eccCurves[i].curveId == curveId)
52             return &eccCurves[i];
53     }
54     return NULL;
55 }

```

10.2.11.2.5 CryptEccGetKeySizeForCurve()

This function returns the key size in bits of the indicated curve.

```

56 LIB_EXPORT UINT16
57 CryptEccGetKeySizeForCurve(
58     TPM_ECC_CURVE      curveId      // IN: the curve
59 )
60 {
61     const ECC_CURVE *curve = CryptEccGetParametersByCurveId(curveId);
62     UINT16      keySizeInBits;
63     //
64     keySizeInBits = (curve != NULL) ? curve->keySizeBits : 0;
65     return keySizeInBits;
66 }

```

10.2.11.2.6 GetCurveData()

This function returns the a pointer for the parameter data associated with a curve.

```

67 const ECC_CURVE_DATA *
68 GetCurveData(
69     TPM_ECC_CURVE      curveId      // IN: the curveID
70 )

```

```

71 {
72     const ECC_CURVE      *curve = CryptEccGetParametersByCurveId(curveId);
73     return (curve != NULL) ? curve->curveData : NULL;
74 }

```

10.2.11.2.7 CryptEccGetOID()

```

75 const BYTE *
76 CryptEccGetOID(
77     TPM_ECC_CURVE      curveId
78 )
79 {
80     const ECC_CURVE      *curve = CryptEccGetParametersByCurveId(curveId);
81     return (curve != NULL) ? curve->OID : NULL;
82 }

```

10.2.11.2.8 CryptEccGetCurveByIndex()

This function returns the number of the i -th implemented curve. The normal use would be to call this function with i starting at 0. When the i is greater than or equal to the number of implemented curves, TPM_ECC_NONE is returned.

```

83 LIB_EXPORT TPM_ECC_CURVE
84 CryptEccGetCurveByIndex(
85     UINT16              i
86 )
87 {
88     if(i >= ECC_CURVE_COUNT)
89         return TPM_ECC_NONE;
90     return eccCurves[i].curveId;
91 }

```

10.2.11.2.9 CryptEccGetParameter()

This function returns an ECC curve parameter. The parameter is selected by a single character designator from the set of "PNABXYH".

Return Value	Meaning
TRUE(1)	curve exists and parameter returned
FALSE(0)	curve does not exist or parameter selector

```

92 LIB_EXPORT BOOL
93 CryptEccGetParameter(
94     TPM2B_ECC_PARAMETER *out,          // OUT: place to put parameter
95     char                p,             // IN: the parameter selector
96     TPM_ECC_CURVE      curveId        // IN: the curve id
97 )
98 {
99     const ECC_CURVE_DATA *curve = GetCurveData(curveId);
100     bigConst              parameter = NULL;
101
102     if(curve != NULL)
103     {
104         switch(p)
105         {
106             case 'p':
107                 parameter = CurveGetPrime(curve);
108                 break;
109             case 'n':

```

```

110         parameter = CurveGetOrder(curve);
111         break;
112     case 'a':
113         parameter = CurveGet_a(curve);
114         break;
115     case 'b':
116         parameter = CurveGet_b(curve);
117         break;
118     case 'x':
119         parameter = CurveGetGx(curve);
120         break;
121     case 'y':
122         parameter = CurveGetGy(curve);
123         break;
124     case 'h':
125         parameter = CurveGetCofactor(curve);
126         break;
127     default:
128         FAIL(FATAL_ERROR_INTERNAL);
129         break;
130     }
131 }
132 // If not debugging and we get here with parameter still NULL, had better
133 // not try to convert so just return FALSE instead.
134 return (parameter != NULL) ? BnTo2B(parameter, &out->b, 0) : 0;
135 }

```

10.2.11.2.10 CryptCapGetECCCurve()

This function returns the list of implemented ECC curves.

Return Value	Meaning
YES	if no more ECC curve is available
NO	if there are more ECC curves not reported

```

136 TPMI_YES_NO
137 CryptCapGetECCCurve(
138     TPM_ECC_CURVE    curveID,        // IN: the starting ECC curve
139     UINT32            maxCount,       // IN: count of returned curves
140     TPML_ECC_CURVE    *curveList     // OUT: ECC curve list
141 )
142 {
143     TPMI_YES_NO        more = NO;
144     UINT16             i;
145     UINT32             count = ECC_CURVE_COUNT;
146     TPM_ECC_CURVE      curve;
147
148     // Initialize output property list
149     curveList->count = 0;
150
151     // The maximum count of curves we may return is MAX_ECC_CURVES
152     if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
153
154     // Scan the eccCurveValues array
155     for(i = 0; i < count; i++)
156     {
157         curve = CryptEccGetCurveByIndex(i);
158         // If curveID is less than the starting curveID, skip it
159         if(curve < curveID)
160             continue;
161         if(curveList->count < maxCount)
162         {

```

```

163         // If we have not filled up the return list, add more curves to
164         // it
165         curveList->eccCurves[curveList->count] = curve;
166         curveList->count++;
167     }
168     else
169     {
170         // If the return list is full but we still have curves
171         // available, report this and stop iterating
172         more = YES;
173         break;
174     }
175 }
176 return more;
177 }

```

10.2.11.2.11 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```

178 const TPMT_ECC_SCHEME *
179 CryptGetCurveSignScheme(
180     TPM_ECC_CURVE    curveId        // IN: The curve selector
181 )
182 {
183     const ECC_CURVE    *curve = CryptEccGetParametersByCurveId(curveId);
184     if(curve != NULL)
185         return &(curve->sign);
186     else
187         return NULL;
188 }
189

```

10.2.11.2.12 CryptGenerateR()

This function computes the commit random value for a split signing scheme.

If *c* is NULL, it indicates that *r* is being generated for TPM2_Commit. If *c* is not NULL, the TPM will validate that the *gr.commitArray* bit associated with the input value of *c* is SET. If not, the TPM returns FALSE and no *r* value is generated.

Return Value	Meaning
TRUE(1)	r value computed
FALSE(0)	no r value computed

```

190 BOOL
191 CryptGenerateR(
192     TPM2B_ECC_PARAMETER    *r,        // OUT: the generated random value
193     UINT16                  *c,        // IN/OUT: count value.
194     TPM1_ECC_CURVE          curveID,   // IN: the curve for the value
195     TPM2B_NAME              *name      // IN: optional name of a key to
196                                     // associate with 'r'
197 )
198 {
199     // This holds the marshaled g_commitCounter.
200     TPM2B_TYPE(8B, 8);
201     TPM2B_8B                cntr = {{8,{0}}};
202     UINT32                   iterations;
203     TPM2B_ECC_PARAMETER      n;
204     UINT64                   currentCount = gr.commitCounter;
205     UINT16                   t1;

```

```

206 //
207 if(!CryptEccGetParameter(&n, 'n', curveID))
208     return FALSE;
209
210 // If this is the commit phase, use the current value of the commit counter
211 if(c != NULL)
212 {
213     // if the array bit is not set, can't use the value.
214     if(!TEST_BIT((*c & COMMIT_INDEX_MASK), gr.commitArray))
215         return FALSE;
216
217     // If it is the sign phase, figure out what the counter value was
218     // when the commitment was made.
219     //
220     // When gr.commitArray has less than 64K bits, the extra
221     // bits of 'c' are used as a check to make sure that the
222     // signing operation is not using an out of range count value
223     t1 = (UINT16)currentCount;
224
225     // If the lower bits of c are greater or equal to the lower bits of t1
226     // then the upper bits of t1 must be one more than the upper bits
227     // of c
228     if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
229         // Since the counter is behind, reduce the current count
230         currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
231
232     t1 = (UINT16)currentCount;
233     if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
234         return FALSE;
235     // set the counter to the value that was
236     // present when the commitment was made
237     currentCount = (currentCount & 0xfffffffffff0000) | *c;
238 }
239 // Marshal the count value to a TPM2B buffer for the KDF
240 cntr.t.size = sizeof(currentCount);
241 UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
242
243 // Now can do the KDF to create the random value for the signing operation
244 // During the creation process, we may generate an r that does not meet the
245 // requirements of the random value.
246 // want to generate a new r.
247 r->t.size = n.t.size;
248
249 for(iterations = 1; iterations < 1000000;)
250 {
251     int i;
252     CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, COMMIT_STRING,
253               &name->b, &cntr.b, n.t.size * 8, r->t.buffer, &iterations, FALSE);
254
255     // "random" value must be less than the prime
256     if(UnsignedCompareB(r->b.size, r->b.buffer, n.t.size, n.t.buffer) >= 0)
257         continue;
258
259     // in this implementation it is required that at least bit
260     // in the upper half of the number be set
261     for(i = n.t.size / 2; i >= 0; i--)
262         if(r->b.buffer[i] != 0)
263             return TRUE;
264 }
265 return FALSE;
266 }

```


10.2.11.2.13 CryptCommit()

This function is called when the count value is committed. The *gr.commitArray* value associated with the current count value is SET and *g_commitCounter* is incremented. The low-order 16 bits of old value of the counter is returned.

```

267  UINT16
268  CryptCommit(
269      void
270  )
271  {
272      UINT16    oldCount = (UINT16)gr.commitCounter;
273      gr.commitCounter++;
274      SET_BIT(oldCount & COMMIT_INDEX_MASK, gr.commitArray);
275      return oldCount;
276  }
```

10.2.11.2.14 CryptEndCommit()

This function is called when the signing operation using the committed value is completed. It clears the *gr.commitArray* bit associated with the count value so that it can't be used again.

```

277  void
278  CryptEndCommit(
279      UINT16    c           // IN: the counter value of the commitment
280  )
281  {
282      ClearBit((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
283  }
```

10.2.11.2.15 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	unsupported ECC curve ID

```

284  BOOL
285  CryptEccGetParameters(
286      TPM_ECC_CURVE    curveId,           // IN: ECC curve ID
287      TPMS_ALGORITHM_DETAIL_ECC *parameters // OUT: ECC parameters
288  )
289  {
290      const ECC_CURVE    *curve = CryptEccGetParametersByCurveId(curveId);
291      const ECC_CURVE_DATA *data;
292      BOOL                found = curve != NULL;
293
294      if(found)
295      {
296          data = curve->curveData;
297          parameters->curveID = curve->curveId;
298          parameters->keySize = curve->keySizeBits;
299          parameters->kdf = curve->kdf;
300          parameters->sign = curve->sign;
301          // BnTo2B(data->prime, &parameters->p.b, 0);
302          BnTo2B(data->prime, &parameters->p.b, parameters->p.t.size);
303          BnTo2B(data->a, &parameters->a.b, 0);
304          BnTo2B(data->b, &parameters->b.b, 0);
305          BnTo2B(data->base.x, &parameters->gX.b, parameters->p.t.size);
```

```

306     BnTo2B(data->base.y, &parameters->gY.b, parameters->p.t.size);
307     //     BnTo2B(data->base.x, &parameters->gX.b, 0);
308     //     BnTo2B(data->base.y, &parameters->gY.b, 0);
309     BnTo2B(data->order, &parameters->n.b, 0);
310     BnTo2B(data->h, &parameters->h.b, 0);
311 }
312 return found;
313 }

```

10.2.11.2.16 BnGetCurvePrime()

This function is used to get just the prime modulus associated with a curve.

```

314 const bignum_t *
315 BnGetCurvePrime(
316     TPM_ECC_CURVE          curveId
317 )
318 {
319     const ECC_CURVE_DATA    *C = GetCurveData(curveId);
320     return (C != NULL) ? CurveGetPrime(C) : NULL;
321 }

```

10.2.11.2.17 BnGetCurveOrder()

This function is used to get just the curve order

```

322 const bignum_t *
323 BnGetCurveOrder(
324     TPM_ECC_CURVE          curveId
325 )
326 {
327     const ECC_CURVE_DATA    *C = GetCurveData(curveId);
328     return (C != NULL) ? CurveGetOrder(C) : NULL;
329 }

```

10.2.11.2.18 BnIsOnCurve()

This function checks if a point is on the curve.

```

330 BOOL
331 BnIsOnCurve(
332     pointConst      Q,
333     const ECC_CURVE_DATA *C
334 )
335 {
336     BN_VAR(right, (MAX_ECC_KEY_BITS * 3));
337     BN_VAR(left, (MAX_ECC_KEY_BITS * 2));
338     bigConst      prime = CurveGetPrime(C);
339     //
340     // Show that point is on the curve  $y^2 = x^3 + ax + b$ ;
341     // Or  $y^2 = x(x^2 + a) + b$ 
342     //  $y^2$ 
343     BnMult(left, Q->y, Q->y);
344
345     BnMod(left, prime);
346     //  $x^2$ 
347     BnMult(right, Q->x, Q->x);
348
349     //  $x^2 + a$ 
350     BnAdd(right, right, CurveGet_a(C));
351 }

```

```

352 // BnMod(right, CurveGetPrime(C));
353 // x(x^2 + a)
354 BnMult(right, right, Q->x);
355
356 // x(x^2 + a) + b
357 BnAdd(right, right, CurveGet_b(C));
358
359 BnMod(right, prime);
360 if(BnUnsignedCmp(left, right) == 0)
361     return TRUE;
362 else
363     return FALSE;
364 }

```

10.2.11.2.19 BnIsValidPrivateEcc()

Checks that $0 < x < q$

```

365 BOOL
366 BnIsValidPrivateEcc(
367     bigConst          x,          // IN: private key to check
368     bigCurve          E          // IN: the curve to check
369 )
370 {
371     BOOL          retVal;
372     retVal = (!BnEqualZero(x)
373         && (BnUnsignedCmp(x, CurveGetOrder(AccessCurveData(E))) < 0));
374     return retVal;
375 }
376 LIB_EXPORT BOOL
377 CryptEccIsValidPrivateKey(
378     TPM2B_ECC_PARAMETER *d,
379     TPM_ECC_CURVE       curveId
380 )
381 {
382     BN_INITIALIZED(bnD, MAX_ECC_PARAMETER_BYTES * 8, d);
383     return !BnEqualZero(bnD) && (BnUnsignedCmp(bnD, BnGetCurveOrder(curveId)) < 0);
384 }

```

10.2.11.2.20 BnPointMul()

This function does a point multiply of the form $R = [d]S + [u]Q$ where the parameters are *bigNum* values. If *S* is NULL and *d* is not NULL, then it computes $R = [d]G + [u]Q$ or just $R = [d]G$ if *u* and *Q* are NULL. If *skipChecks* is TRUE, then the function will not verify that the inputs are correct for the domain. This would be the case when the values were created by the `CryptoEngine()` code. It will return `TPM_RC_NO_RESULT` if the resulting point is the point at infinity.

Error Return	Meaning
TPM_RC_NO_RESULT	result of multiplication is a point at infinity
TPM_RC_ECC_POINT	S or Q is not on the curve
TPM_RC_VALUE	d or u is not < n

```

385 TPM_RC
386 BnPointMult(
387     bigPoint          R,          // OUT: computed point
388     pointConst        S,          // IN: optional point to multiply by 'd'
389     bigConst          d,          // IN: scalar for [d]S or [d]G
390     pointConst        Q,          // IN: optional second point
391     bigConst          u,          // IN: optional second scalar
392     bigCurve          E          // IN: curve parameters

```

```

393     )
394 {
395     BOOL                                OK;
396     //
397     TEST(TPM_ALG_ECDH);
398
399     // Need one scalar
400     OK = (d != NULL || u != NULL);
401
402     // If S is present, then d has to be present. If S is not
403     // present, then d may or may not be present
404     OK = OK && ((S == NULL) == (d == NULL)) || (d != NULL);
405
406     // either both u and Q have to be provided or neither can be provided (don't
407     // know what to do if only one is provided.
408     OK = OK && ((u == NULL) == (Q == NULL));
409
410     OK = OK && (E != NULL);
411     if(!OK)
412         return TPM_RC_VALUE;
413
414     OK = (S == NULL) || BnIsOnCurve(S, AccessCurveData(E));
415     OK = OK && ((Q == NULL) || BnIsOnCurve(Q, AccessCurveData(E)));
416     if(!OK)
417         return TPM_RC_ECC_POINT;
418
419     if((d != NULL) && (S == NULL))
420         S = CurveGetG(AccessCurveData(E));
421     // If only one scalar, don't need Shamir's trick
422     if((d == NULL) || (u == NULL))
423     {
424         if(d == NULL)
425             OK = BnEccModMult(R, Q, u, E);
426         else
427             OK = BnEccModMult(R, S, d, E);
428     }
429     else
430     {
431         OK = BnEccModMult2(R, S, d, Q, u, E);
432     }
433     return (OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT);
434 }

```

10.2.11.2.21 BnEccGetPrivate()

This function gets random values that are the size of the key plus 64 bits. The value is reduced (mod ($q - 1$)) and incremented by 1 (q is the order of the curve. This produces a value (d) such that $1 \leq d < q$. This is the method of FIPS 186-4 Section B.4.1 "Key Pair Generation Using Extra Random Bits".

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure generating private key

```

435     BOOL
436     BnEccGetPrivate(
437         bigNum                dOut,          // OUT: the qualified random value
438         const ECC_CURVE_DATA *C,            // IN: curve for which the private key
439                                             // needs to be appropriate
440         RAND_STATE             *rand         // IN: state for DRBG
441     )
442 {
443     bigConst                  order = CurveGetOrder(C);

```

```

444     BOOL                                OK;
445     UINT32                              orderBits = BnSizeInBits(order);
446     UINT32                              orderBytes = BITS_TO_BYTES(orderBits);
447     BN_VAR(bnExtraBits, MAX_ECC_KEY_BITS + 64);
448     BN_VAR(nMinus1, MAX_ECC_KEY_BITS);
449     //
450     OK = BnGetRandomBits(bnExtraBits, (orderBytes * 8) + 64, rand);
451     OK = OK && BnSubWord(nMinus1, order, 1);
452     OK = OK && BnMod(bnExtraBits, nMinus1);
453     OK = OK && BnAddWord(dOut, bnExtraBits, 1);
454     return OK && !g_inFailureMode;
455 }

```

10.2.11.2.22 BnEccGenerateKeyPair()

This function gets a private scalar from the source of random bits and does the point multiply to get the public key.

```

456  BOOL
457  BnEccGenerateKeyPair(
458      bigNum          bnD,          // OUT: private scalar
459      bn_point_t      *ecQ,         // OUT: public point
460      bigCurve        E,            // IN: curve for the point
461      RAND_STATE      *rand         // IN: DRBG state to use
462  )
463  {
464      BOOL              OK = FALSE;
465      // Get a private scalar
466      OK = BnEccGetPrivate(bnD, AccessCurveData(E), rand);
467
468      // Do a point multiply
469      OK = OK && BnEccModMult(ecQ, NULL, bnD, E);
470      if(!OK)
471          BnSetWord(ecQ->z, 0);
472      else
473          BnSetWord(ecQ->z, 1);
474      return OK;
475  }

```

10.2.11.2.23 CryptEccNewKeyPair(***)

This function creates an ephemeral ECC. It is ephemeral in that is expected that the private part of the key will be discarded

```

476  LIB_EXPORT TPM_RC
477  CryptEccNewKeyPair(
478      TPMS_ECC_POINT    *Qout,      // OUT: the public point
479      TPM2B_ECC_PARAMETER *dOut,    // OUT: the private scalar
480      TPM_ECC_CURVE     curveId     // IN: the curve for the key
481  )
482  {
483      CURVE_INITIALIZED(E, curveId);
484      POINT(ecQ);
485      ECC_NUM(bnD);
486      BOOL              OK;
487
488      if(E == NULL)
489          return TPM_RC_CURVE;
490
491      TEST(TPM_ALG_ECDH);
492      OK = BnEccGenerateKeyPair(bnD, ecQ, E, NULL);
493      if(OK)

```

```

494     {
495         BnPointTo2B(Qout, ecQ, E);
496         BnTo2B(bnD, &dOut->b, Qout->x.t.size);
497     }
498     else
499     {
500         Qout->x.t.size = Qout->y.t.size = dOut->t.size = 0;
501     }
502     CURVE_FREE(E);
503     return OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
504 }

```

10.2.11.2.24 CryptEccPointMultiply()

This function computes $R := [dIn]G + [uIn]QIn$. Where dIn and uIn are scalars, G and QIn are points on the specified curve and G is the default generator of the curve.

The $xOut$ and $yOut$ parameters are optional and may be set to NULL if not used.

It is not necessary to provide uIn if QIn is specified but one of uIn and dIn must be provided. If dIn and QIn are specified but uIn is not provided, then $R = [dIn]QIn$.

If the multiply produces the point at infinity, the TPM_RC_NO_RESULT is returned.

The sizes of $xOut$ and $yOut$ will be set to be the size of the degree of the curve

It is a fatal error if dIn and uIn are both unspecified (NULL) or if QIn or $Rout$ is unspecified.

Error Return	Meaning
TPM_RC_ECC_POINT	the point Pin or Qin is not on the curve
TPM_RC_NO_RESULT	the product point is at infinity
TPM_RC_CURVE	bad curve
TPM_RC_VALUE	dIn or uIn out of range

```

505 LIB_EXPORT TPM_RC
506 CryptEccPointMultiply(
507     TPMS_ECC_POINT *Rout,           // OUT: the product point R
508     TPM_ECC_CURVE   curveId,        // IN: the curve to use
509     TPMS_ECC_POINT  *Pin,           // IN: first point (can be null)
510     TPM2B_ECC_PARAMETER *dIn,       // IN: scalar value for [dIn]Qin
511                                     // the Pin
512     TPMS_ECC_POINT  *Qin,           // IN: point Q
513     TPM2B_ECC_PARAMETER *uIn,       // IN: scalar value for the multiplier
514                                     // of Q
515 )
516 {
517     CURVE_INITIALIZED(E, curveId);
518     POINT_INITIALIZED(ecP, Pin);
519     ECC_INITIALIZED(bnD, dIn);      // If dIn is null, then bnD is null
520     ECC_INITIALIZED(bnU, uIn);
521     POINT_INITIALIZED(ecQ, Qin);
522     POINT(ecR);
523     TPM_RC retVal;
524     //
525     retVal = BnPointMult(ecR, ecP, bnD, ecQ, bnU, E);
526
527     if(retVal == TPM_RC_SUCCESS)
528         BnPointTo2B(Rout, ecR, E);
529     else
530         ClearPoint2B(Rout);
531     CURVE_FREE(E);
532     return retVal;

```

533 }

10.2.11.2.25 CryptEccIsPointOnCurve()

This function is used to test if a point is on a defined curve. It does this by checking that $y^2 \bmod p = x^3 + a \cdot x + b \bmod p$.

It is a fatal error if Q is not specified (is NULL).

Return Value	Meaning
TRUE(1)	point is on curve
FALSE(0)	point is not on curve or curve is not supported

```

534 LIB_EXPORT BOOL
535 CryptEccIsPointOnCurve(
536     TPM_ECC_CURVE      curveId,          // IN: the curve selector
537     TPMS_ECC_POINT     *Qin              // IN: the point.
538 )
539 {
540     const ECC_CURVE_DATA *C = GetCurveData(curveId);
541     POINT_INITIALIZED(ecQ, Qin);
542     BOOL OK;
543     //
544     pAssert(Qin != NULL);
545     OK = (C != NULL && (BnIsOnCurve(ecQ, C)));
546     return OK;
547 }
```

10.2.11.2.26 CryptEccGenerateKey()

This function generates an ECC key pair based on the input parameters. This routine uses KDFa to produce candidate numbers. The method is according to FIPS 186-3, section B.1.2 **Key Pair Generation by Testing Candidates**. According to the method in FIPS 186-3, the resulting private value d should be $1 \leq d < n$ where n is the order of the base point.

It is a fatal error if Qout, dOut, is not provided (is NULL).

If the curve is not supported If seed is not provided, then a random number will be used for the key

Error Return	Meaning
TPM_RC_CURVE	curve is not supported
TPM_RC_NO_RESULT	could not verify key with signature (FIPS only)

```

548 LIB_EXPORT TPM_RC
549 CryptEccGenerateKey(
550     TPMT_PUBLIC      *publicArea,          // IN/OUT: The public area template for
551                                           // the new key. The public key
552                                           // area will be replaced computed
553                                           // ECC public key
554     TPMT_SENSITIVE    *sensitive,          // OUT: the sensitive area will be
555                                           // updated to contain the private
556                                           // ECC key and the symmetric
557                                           // encryption key
558     RAND_STATE        *rand               // IN: if not NULL, the deterministic
559                                           // RNG state
560 )
561 {
562     CURVE_INITIALIZED(E, publicArea->parameters.eccDetail.curveID);
563     ECC_NUM(bnD);
```



```

564     POINT(ecQ);
565     BOOL                                     OK;
566     TPM_RC                                 retVal;
567 //
568     TEST(TPM_ALG_ECDSA); // ECDSA is used to verify each key
569
570     // Validate parameters
571     if(E == NULL)
572         ERROR_RETURN(TPM_RC_CURVE);
573
574     publicArea->unique.ecc.x.t.size = 0;
575     publicArea->unique.ecc.y.t.size = 0;
576     sensitive->sensitive.ecc.t.size = 0;
577
578     OK = BnEccGenerateKeyPair(bnD, ecQ, E, rand);
579     if(OK)
580     {
581         BnPointTo2B(&publicArea->unique.ecc, ecQ, E);
582         BnTo2B(bnD, &sensitive->sensitive.ecc.b, publicArea->unique.ecc.x.t.size);
583     }
584 #if FIPS_COMPLIANT
585     // See if PWCT is required
586     if(OK && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
587     {
588         ECC_NUM(bnT);
589         ECC_NUM(bnS);
590         TPM2B_DIGEST digest;
591 //
592         TEST(TPM_ALG_ECDSA);
593         digest.t.size = MIN(sensitive->sensitive.ecc.t.size, sizeof(digest.t.buffer));
594         // Get a random value to sign using the built in DRBG state
595         DRBG_Generate(NULL, digest.t.buffer, digest.t.size);
596         if(g_inFailureMode)
597             return TPM_RC_FAILURE;
598         BnSignEcdsa(bnT, bnS, E, bnD, &digest, NULL);
599         // and make sure that we can validate the signature
600         OK = BnValidateSignatureEcdsa(bnT, bnS, E, ecQ, &digest) == TPM_RC_SUCCESS;
601     }
602 #endif
603     retVal = (OK) ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
604 Exit:
605     CURVE_FREE(E);
606     return retVal;
607 }
608 #endif // ALG_ECC

```

10.2.12 CryptEccSignature.c

10.2.12.1 Includes and Defines

```

1  #include "Tpm.h"
2  #include "CryptEccSignature_fp.h"
3
4  #if ALG_ECC

```

10.2.12.2 Utility Functions

10.2.12.2.1 EcdsaDigest()

Function to adjust the digest so that it is no larger than the order of the curve. This is used for ECDSA sign and verification.

```

5  static bigNum
6  EcdsaDigest(
7      bigNum          bnD,          // OUT: the adjusted digest
8      const TPM2B_DIGEST *digest,  // IN: digest to adjust
9      bigConst        max          // IN: value that indicates the maximum
10                                     // number of bits in the results
11  )
12  {
13      int             bitsInMax = BnSizeInBits(max);
14      int             shift;
15      //
16      if(digest == NULL)
17          BnSetWord(bnD, 0);
18      else
19      {
20          BnFromBytes(bnD, digest->t.buffer,
21                      (NUMBYTES)MIN(digest->t.size, BITS_TO_BYTES(bitsInMax)));
22          shift = BnSizeInBits(bnD) - bitsInMax;
23          if(shift > 0)
24              BnShiftRight(bnD, bnD, shift);
25      }
26      return bnD;
27  }

```

10.2.12.2.2 BnSchnorrSign()

This contains the Schnorr signature computation. It is used by both ECDSA and Schnorr signing. The result is computed as: $[s = k + r * d \pmod{n}]$ where

- 1) s is the signature
- 2) k is a random value
- 3) r is the value to sign
- 4) d is the private EC key
- 5) n is the order of the curve

Error Return	Meaning
TPM_RC_NO_RESULT	the result of the operation was zero or $r \pmod{n}$ is zero

```

28  static TPM_RC
29  BnSchnorrSign(

```

```

30     bigNum          bnS,          // OUT: 's' component of the signature
31     bigConst        bnK,          // IN: a random value
32     bigNum          bnR,          // IN: the signature 'r' value
33     bigConst        bnD,          // IN: the private key
34     bigConst        bnN          // IN: the order of the curve
35 )
36 {
37     // Need a local temp value to store the intermediate computation because product
38     // size can be larger than will fit in bnS.
39     BN_VAR(bnT1, MAX_ECC_PARAMETER_BYTES * 2 * 8);
40     //
41     // Reduce bnR without changing the input value
42     BnDiv(NULL, bnT1, bnR, bnN);
43     if(BnEqualZero(bnT1))
44         return TPM_RC_NO_RESULT;
45     // compute s = (k + r * d) (mod n)
46     // r * d
47     BnMult(bnT1, bnT1, bnD);
48     // k * r * d
49     BnAdd(bnT1, bnT1, bnK);
50     // k + r * d (mod n)
51     BnDiv(NULL, bnS, bnT1, bnN);
52     return (BnEqualZero(bnS)) ? TPM_RC_NO_RESULT : TPM_RC_SUCCESS;
53 }

```

10.2.12.3 Signing Functions

10.2.12.3.1 BnSignEcdsa()

This function implements the ECDSA signing algorithm. The method is described in the comments below.

```

54     TPM_RC
55     BnSignEcdsa(
56         bigNum          bnR,          // OUT: 'r' component of the signature
57         bigNum          bnS,          // OUT: 's' component of the signature
58         bigCurve        E,          // IN: the curve used in the signature
59                                     // process
60         bigNum          bnD,          // IN: private signing key
61         const TPM2B_DIGEST *digest,  // IN: the digest to sign
62         RAND_STATE      *rand        // IN: used in debug of signing
63     )
64 {
65     ECC_NUM(bnK);
66     ECC_NUM(bnI);
67     BN_VAR(bnE, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE) * 8);
68     POINT(ecR);
69     bigConst        order = CurveGetOrder(AccessCurveData(E));
70     TPM_RC          retVal = TPM_RC_SUCCESS;
71     INT32           tries = 10;
72     BOOL            OK = FALSE;
73     //
74     pAssert(digest != NULL);
75     // The algorithm as described in "Suite B Implementer's Guide to FIPS
76     // 186-3(ECDSA)"
77     // 1. Use one of the routines in Appendix A.2 to generate (k, k^-1), a
78     //    per-message secret number and its inverse modulo n. Since n is prime,
79     //    the output will be invalid only if there is a failure in the RBG.
80     // 2. Compute the elliptic curve point R = [k]G = (xR, yR) using EC scalar
81     //    multiplication (see [Routines]), where G is the base point included in
82     //    the set of domain parameters.
83     // 3. Compute r = xR mod n. If r = 0, then return to Step 1. 1.
84     // 4. Use the selected hash function to compute H = Hash(M).
85     // 5. Convert the bit string H to an integer e as described in Appendix B.2.

```

```

86 // 6. Compute  $s = (k^{-1} * (e + d * r)) \bmod q$ . If  $s = 0$ , return to Step 1.2.
87 // 7. Return  $(r, s)$ .
88 // In the code below,  $q$  is  $n$  (that is, the order of the curve is  $p$ )
89
90 do // This implements the loop at step 6. If  $s$  is zero, start over.
91 {
92     for(; tries > 0; tries--)
93     {
94         // Step 1 and 2 -- generate an ephemeral key and the modular inverse
95         // of the private key.
96         if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
97             continue;
98         //  $x$  coordinate is mod  $p$ . Make it mod  $q$ 
99         BnMod(ecR->x, order);
100        // Make sure that it is not zero;
101        if(BnEqualZero(ecR->x))
102            continue;
103        // write the modular reduced version of  $r$  as part of the signature
104        BnCopy(bnR, ecR->x);
105        // Make sure that a modular inverse exists and try again if not
106        OK = (BnModInverse(bnIk, bnK, order));
107        if(OK)
108            break;
109    }
110    if(!OK)
111        goto Exit;
112
113    EcDSA_Digest(bnE, digest, order);
114
115    // now have inverse of  $K$  (bnIk),  $e$  (bnE),  $r$  (bnR),  $d$  (bnD) and
116    // CurveGetOrder(E)
117    // Compute  $s = k^{-1} (e + r*d) \bmod q$ 
118    // first do  $s = r*d \bmod q$ 
119    BnModMult(bnS, bnR, bnD, order);
120    //  $s = e + s = e + r * d$ 
121    BnAdd(bnS, bnE, bnS);
122    //  $s = k^{-1} s \bmod n = k^{-1} (e + r * d) \bmod n$ 
123    BnModMult(bnS, bnIk, bnS, order);
124
125    // If  $S$  is zero, try again
126 } while(BnEqualZero(bnS));
127 Exit:
128     return retVal;
129 }
130 #if ALG_ECDA

```

10.2.12.3.2 BnSignEcdaa()

This function performs $s = r + T * d \bmod q$ where

- 1) r is a random, or pseudo-random value created in the commit phase
- 2) $nonceK$ is a TPM-generated, random value $0 < nonceK < n$
- 3) T is mod q of $\text{Hash}(nonceK || digest)$, and
- 4) d is a private key.

The signature is the tuple $(nonceK, s)$

Regrettably, the parameters in this function kind of collide with the parameter names used in ECSCHNORR making for a lot of confusion.

Error Return	Meaning
TPM_RC_SCHEME	unsupported hash algorithm
TPM_RC_NO_RESULT	cannot get values from random number generator

```

131 static TPM_RC
132 BnSignEcdaa(
133     TPM2B_ECC_PARAMETER *nonceK,      // OUT: 'nonce' component of the signature
134     bigNum bnS,                      // OUT: 's' component of the signature
135     bigCurve E,                      // IN: the curve used in signing
136     bigNum bnD,                      // IN: the private key
137     const TPM2B_DIGEST *digest,      // IN: the value to sign (mod 'q')
138     TPMT_ECC_SCHEME *scheme,         // IN: signing scheme (contains the
139                                     // commit count value).
140     OBJECT *eccKey,                  // IN: The signing key
141     RAND_STATE *rand                 // IN: a random number state
142 )
143 {
144     TPM_RC retVal;
145     TPM2B_ECC_PARAMETER r;
146     HASH_STATE state;
147     TPM2B_DIGEST T;
148     BN_MAX(bnT);
149     //
150     NOT_REFERENCED(rand);
151     if(!CryptGenerateR(&r, &scheme->details.ecdaa.count,
152                       eccKey->publicArea.parameters.eccDetail.curveID,
153                       &eccKey->name))
154         retVal = TPM_RC_VALUE;
155     else
156     {
157         // This allocation is here because 'r' doesn't have a value until
158         // CryptGenerateR() is done.
159         ECC_INITIALIZED(bnR, &r);
160         do
161         {
162             // generate nonceK such that 0 < nonceK < n
163             // use bnT as a temp.
164             if(!BnEccGetPrivate(bnT, AccessCurveData(E), rand))
165             {
166                 retVal = TPM_RC_NO_RESULT;
167                 break;
168             }
169             BnTo2B(bnT, &nonceK->b, 0);
170
171             T.t.size = CryptHashStart(&state, scheme->details.ecdaa.hashAlg);
172             if(T.t.size == 0)
173             {
174                 retVal = TPM_RC_SCHEME;
175             }
176             else
177             {
178                 CryptDigestUpdate2B(&state, &nonceK->b);
179                 CryptDigestUpdate2B(&state, &digest->b);
180                 CryptHashEnd2B(&state, &T.b);
181                 BnFrom2B(bnT, &T.b);
182                 // Watch out for the name collisions in this call!!
183                 retVal = BnSchnorrSign(bnS, bnR, bnT, bnD,
184                                       AccessCurveData(E)->order);
185             }
186         } while(retVal == TPM_RC_NO_RESULT);
187         // Because the rule is that internal state is not modified if the command
188         // fails, only end the commit if the command succeeds.
189         // NOTE that if the result of the Schnorr computation was zero

```

```

190         // it will probably not be worthwhile to run the same command again because
191         // the result will still be zero. This means that the Commit command will
192         // need to be run again to get a new commit value for the signature.
193         if(retVal == TPM_RC_SUCCESS)
194             CryptEndCommit(scheme->details.ecdaa.count);
195     }
196     return retVal;
197 }
198 #endif // ALG_ECDA
199
200 #if ALG_ECSCHNORR

```

10.2.12.3.3 SchnorrReduce()

Function to reduce a hash result if it's magnitude is too large. The size of *number* is set so that it has no more bytes of significance than *reference* value. If the resulting number can have more bits of significance than *reference*.

```

201 static void
202 SchnorrReduce(
203     TPM2B      *number,           // IN/OUT: Value to reduce
204     bigConst    reference        // IN: the reference value
205 )
206 {
207     UINT16      maxBytes = (UINT16)BITS_TO_BYTES(BnSizeInBits(reference));
208     if(number->size > maxBytes)
209         number->size = maxBytes;
210 }

```

10.2.12.3.4 SchnorrEcc()

This function is used to perform a modified Schnorr signature.

This function will generate a random value *k* and compute

- $(xR, yR) = [k]G$
- $r = \text{Hash}(xR || P)(\text{mod } q)$
- $rT = \text{truncated } r$
- $s = k + rT * ds (\text{mod } q)$
- return the tuple rT, s

Error Return	Meaning
TPM_RC_NO_RESULT	failure in the Schnorr sign process
TPM_RC_SCHEME	<i>hashAlg</i> can't produce zero-length digest

```

211 static TPM_RC
212 BnSignEcSchnorr(
213     bigNum      bnR,           // OUT: 'r' component of the signature
214     bigNum      bnS,           // OUT: 's' component of the signature
215     bigCurve     E,           // IN: the curve used in signing
216     bigNum      bnD,           // IN: the signing key
217     const TPM2B_DIGEST *digest, // IN: the digest to sign
218     TPM_ALG_ID   hashAlg,      // IN: signing scheme (contains a hash)
219     RAND_STATE   *rand         // IN: non-NULL when testing
220 )
221 {
222     HASH_STATE   hashState;
223     UINT16       digestSize = CryptHashGetDigestSize(hashAlg);

```

```

224     TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_KEY_BYTES));
225     TPM2B_T          T2b;
226     TPM2B             *e = &T2b.b;
227     TPM_RC            retVal = TPM_RC_NO_RESULT;
228     const ECC_CURVE_DATA *C;
229     bigConst          order;
230     bigConst          prime;
231     ECC_NUM(bnK);
232     POINT(ecR);
233 //
234 // Parameter checks
235 if(E == NULL)
236     ERROR_RETURN(TPM_RC_VALUE);
237 C = AccessCurveData(E);
238 order = CurveGetOrder(C);
239 prime = CurveGetOrder(C);
240
241 // If the digest does not produce a hash, then null the signature and return
242 // a failure.
243 if(digestSize == 0)
244 {
245     BnSetWord(bnR, 0);
246     BnSetWord(bnS, 0);
247     ERROR_RETURN(TPM_RC_SCHEME);
248 }
249 do
250 {
251     // Generate a random key pair
252     if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
253         break;
254     // Convert R.x to a string
255     BnTo2B(ecR->x, e, (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(prime)));
256
257     // f) compute r = Hash(e || P) (mod n)
258     CryptHashStart(&hashState, hashAlg);
259     CryptDigestUpdate2B(&hashState, e);
260     CryptDigestUpdate2B(&hashState, &digest->b);
261     e->size = CryptHashEnd(&hashState, digestSize, e->buffer);
262     // Reduce the hash size if it is larger than the curve order
263     SchnorrReduce(e, order);
264     // Convert hash to number
265     BnFrom2B(bnR, e);
266     // Do the Schnorr computation
267     retVal = BnSchnorrSign(bnS, bnK, bnR, bnD, CurveGetOrder(C));
268 } while(retVal == TPM_RC_NO_RESULT);
269 Exit:
270     return retVal;
271 }
272 #endif // ALG_ECSCHNORR
273
274 #if ALG_SM2
275 #ifdef _SM2_SIGN_DEBUG

```

10.2.12.3.5 BnHexEqual()

This function compares a bignum value to a hex string.

Return Value	Meaning
TRUE(1)	values equal
FALSE(0)	values not equal

```
276 static BOOL
```



```

277 BnHexEqual(
278     bigNum      bn,      //IN: big number value
279     const char  *c       //IN: character string number
280 )
281 {
282     ECC_NUM(bnC);
283     BnFromHex(bnC, c);
284     return (BnUnsignedCmp(bn, bnC) == 0);
285 }
286 #endif // _SM2_SIGN_DEBUG

```

10.2.12.3.6 BnSignEcSm2()

This function signs a digest using the method defined in SM2 Part 2. The method in the standard will add a header to the message to be signed that is a hash of the values that define the key. This then hashed with the message to produce a digest (e). This function signs e.

Error Return	Meaning
TPM_RC_VALUE	bad curve

```

287 static TPM_RC
288 BnSignEcSm2(
289     bigNum      bnR,      // OUT: 'r' component of the signature
290     bigNum      bnS,      // OUT: 's' component of the signature
291     bigCurve     E,       // IN: the curve used in signing
292     bigNum      bnD,      // IN: the private key
293     const TPM2B_DIGEST *digest, // IN: the digest to sign
294     RAND_STATE  *rand     // IN: random number generator (mostly for
295                          // debug)
296 )
297 {
298     BN_MAX_INITIALIZED(bnE, digest); // Don't know how big digest might be
299     ECC_NUM(bnN);
300     ECC_NUM(bnK);
301     ECC_NUM(bnT); // temp
302     POINT(Q1);
303     bigConst      order = (E != NULL)
304     ? CurveGetOrder(AccessCurveData(E)) : NULL;
305 //
306 #ifdef _SM2_SIGN_DEBUG
307     BnFromHex(bnE, "B524F552CD82B8B028476E005C377FB1"
308               "9A87E6FC682D48BB5D42E3D9B9E7FE76");
309     BnFromHex(bnD, "128B2FA8BD433C6C068C8D803DFF7979"
310               "2A519A55171B1B650C23661D15897263");
311 #endif
312     // A3: Use random number generator to generate random number 1 <= k <= n-1;
313     // NOTE: Ax: numbers are from the SM2 standard
314 loop:
315     {
316         // Get a random number 0 < k < n
317         BnGenerateRandomInRange(bnK, order, rand);
318 #ifdef _SM2_SIGN_DEBUG
319         BnFromHex(bnK, "6CB28D99385C175C94F94E934817663F"
320               "C176D925DD72B727260DBAAE1FB2F96F");
321 #endif
322         // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
323         // to details specified in 4.2.7 in Part 1 of this document, transform the
324         // data type of x1 into an integer;
325         if(!BnEccModMult(Q1, NULL, bnK, E))
326             goto loop;
327         // A5: Figure out 'r' = ('e' + 'x1') mod 'n',
328         BnAdd(bnR, bnE, Q1->x);
329         BnMod(bnR, order);

```

```

330 #ifdef _SM2_SIGN_DEBUG
331     pAssert(BnHexEqual(bnR, "40F1EC59F793D9F49E09DCEF49130D41"
332                         "94F79FB1EED2CAA55BACDB49C4E755D1"));
333 #endif
334     // if r=0 or r+k=n, return to A3;
335     if(BnEqualZero(bnR))
336         goto loop;
337     BnAdd(bnT, bnK, bnR);
338     if(BnUnsignedCmp(bnT, bnN) == 0)
339         goto loop;
340     // A6: Figure out s = ((1 + dA)^-1 (k - r dA)) mod n,
341     // if s=0, return to A3;
342     // compute t = (1+dA)^-1
343     BnAddWord(bnT, bnD, 1);
344     BnModInverse(bnT, bnT, order);
345 #ifdef _SM2_SIGN_DEBUG
346     pAssert(BnHexEqual(bnT, "79BFCF3052C80DA7B939E0C6914A18CB"
347                         "B2D96D8555256E83122743A7D4F5F956"));
348 #endif
349     // compute s = t * (k - r * dA) mod n
350     BnModMult(bnS, bnR, bnD, order);
351     // k - r * dA mod n = k + n - ((r * dA) mod n)
352     BnSub(bnS, order, bnS);
353     BnAdd(bnS, bnK, bnS);
354     BnModMult(bnS, bnS, bnT, order);
355 #ifdef _SM2_SIGN_DEBUG
356     pAssert(BnHexEqual(bnS, "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
357                         "67A457872FB09EC56327A67EC7DEEBE7"));
358 #endif
359     if(BnEqualZero(bnS))
360         goto loop;
361 }
362 // A7: According to details specified in 4.2.1 in Part 1 of this document,
363 // transform the data type of r, s into bit strings, signature of message M
364 // is (r, s).
365 // This is handled by the common return code
366 #ifdef _SM2_SIGN_DEBUG
367     pAssert(BnHexEqual(bnR, "40F1EC59F793D9F49E09DCEF49130D41"
368                         "94F79FB1EED2CAA55BACDB49C4E755D1"));
369     pAssert(BnHexEqual(bnS, "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
370                         "67A457872FB09EC56327A67EC7DEEBE7"));
371 #endif
372     return TPM_RC_SUCCESS;
373 }
374 #endif // ALG_SM2

```

10.2.12.3.7 CryptEccSign()

This function is the dispatch function for the various ECC-based signing schemes. There is a bit of ugliness to the parameter passing. In order to test this, we sometime would like to use a deterministic RNG so that we can get the same signatures during testing. The easiest way to do this for most schemes is to pass in a deterministic RNG and let it return canned values during testing. There is a competing need for a canned parameter to use in ECDA. To accommodate both needs with minimal fuss, a special type of RAND_STATE is defined to carry the address of the commit value. The setup and handling of this is not very different for the caller than what was in previous versions of the code.

Error Return	Meaning
TPM_RC_SCHEME	<i>scheme</i> is not supported

```

375 LIB_EXPORT TPM_RC
376 CryptEccSign(
377     TPMT_SIGNATURE *signature,    // OUT: signature

```

```

378     OBJECT                *signKey,           // IN: ECC key to sign the hash
379     const TPM2B_DIGEST    *digest,           // IN: digest to sign
380     TPMT_ECC_SCHEME        *scheme,           // IN: signing scheme
381     RAND_STATE             *rand
382 )
383 {
384     CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
385     ECC_INITIALIZED(bnD, &signKey->sensitive.sensitive.ecc.b);
386     ECC_NUM(bnR);
387     ECC_NUM(bnS);
388     const ECC_CURVE_DATA *C;
389     TPM_RC                retVal = TPM_RC_SCHEME;
390 //
391     NOT_REFERENCED(scheme);
392     if(E == NULL)
393         ERROR_RETURN(TPM_RC_VALUE);
394     C = AccessCurveData(E);
395     signature->signature.ecdaa.signatureR.t.size
396         = sizeof(signature->signature.ecdaa.signatureR.t.buffer);
397     signature->signature.ecdaa.signatureS.t.size
398         = sizeof(signature->signature.ecdaa.signatureS.t.buffer);
399     TEST(signature->sigAlg);
400     switch(signature->sigAlg)
401     {
402         case TPM_ALG_ECDSA:
403             retVal = BnSignEcdsa(bnR, bnS, E, bnD, digest, rand);
404             break;
405 #if ALG_ECDA
406         case TPM_ALG_ECDA:
407             retVal = BnSignEcdaa(&signature->signature.ecdaa.signatureR, bnS, E,
408                                 bnD, digest, scheme, signKey, rand);
409             bnR = NULL;
410             break;
411 #endif
412 #if ALG_ECSCHNORR
413         case TPM_ALG_ECSCHNORR:
414             retVal = BnSignEcSchnorr(bnR, bnS, E, bnD, digest,
415                                     signature->signature.ecschnorr.hash,
416                                     rand);
417             break;
418 #endif
419 #if ALG_SM2
420         case TPM_ALG_SM2:
421             retVal = BnSignEcSm2(bnR, bnS, E, bnD, digest, rand);
422             break;
423 #endif
424         default:
425             break;
426     }
427     // If signature generation worked, convert the results.
428     if(retVal == TPM_RC_SUCCESS)
429     {
430         NUMBYTES        orderBytes =
431             (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(CurveGetOrder(C)));
432         if(bnR != NULL)
433             BnTo2B(bnR, &signature->signature.ecdaa.signatureR.b, orderBytes);
434         if(bnS != NULL)
435             BnTo2B(bnS, &signature->signature.ecdaa.signatureS.b, orderBytes);
436     }
437 Exit:
438     CURVE_FREE(E);
439     return retVal;
440 }
441 #if ALG_ECDSA

```

10.2.12.3.8 BnValidateSignatureEcdsa()

This function validates an ECDSA signature. *r/n* and *s/n* should have been checked to make sure that they are in the range $0 < v < n$

Error Return	Meaning
TPM_RC_SIGNATURE	signature not valid

```

442 TPM_RC
443 BnValidateSignatureEcdsa (
444     bigNum          bnR,          // IN: 'r' component of the signature
445     bigNum          bnS,          // IN: 's' component of the signature
446     bigCurve        E,           // IN: the curve used in the signature
447                               // process
448     bn_point_t      *ecQ,         // IN: the public point of the key
449     const TPM2B_DIGEST *digest    // IN: the digest that was signed
450 )
451 {
452     // Make sure that the allocation for the digest is big enough for a maximum
453     // digest
454     BN_VAR(bnE, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE) * 8);
455     POINT(ecR);
456     ECC_NUM(bnU1);
457     ECC_NUM(bnU2);
458     ECC_NUM(bnW);
459     bigConst          order = CurveGetOrder(AccessCurveData(E));
460     TPM_RC             retVal = TPM_RC_SIGNATURE;
461 //
462     // Get adjusted digest
463     EcdsaDigest(bnE, digest, order);
464     // 1. If r and s are not both integers in the interval [1, n - 1], output
465     // INVALID.
466     // bnR and bnS were validated by the caller
467     // 2. Use the selected hash function to compute H0 = Hash(M0).
468     // This is an input parameter
469     // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
470     // Done at entry
471     // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
472     if(!BnModInverse(bnW, bnS, order))
473         goto Exit;
474     // 5. Compute u1 = (e' * w) mod n, and compute u2 = (r' * w) mod n.
475     BnModMult(bnU1, bnE, bnW, order);
476     BnModMult(bnU2, bnR, bnW, order);
477     // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
478     // scalar multiplication and EC addition (see [Routines]). If R is equal to
479     // the point at infinity O, output INVALID.
480     if(BnPointMult(ecR, CurveGetG(AccessCurveData(E)), bnU1, ecQ, bnU2, E)
481        != TPM_RC_SUCCESS)
482         goto Exit;
483     // 7. Compute v = Rx mod n.
484     BnMod(ecR->x, order);
485     // 8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
486     if(BnUnsignedCmp(ecR->x, bnR) != 0)
487         goto Exit;
488
489     retVal = TPM_RC_SUCCESS;
490 Exit:
491     return retVal;
492 }
493 #endif // ALG_ECDSA
494
495 #if ALG_SM2

```

10.2.12.3.9 BnValidateSignatureEcSm2()

This function is used to validate an SM2 signature.

Error Return	Meaning
TPM_RC_SIGNATURE	signature not valid

```

496 static TPM_RC
497 BnValidateSignatureEcSm2(
498     bigNum      bnR,      // IN: 'r' component of the signature
499     bigNum      bnS,      // IN: 's' component of the signature
500     bigCurve     E,        // IN: the curve used in the signature
501                        // process
502     bigPoint     ecQ,      // IN: the public point of the key
503     const TPM2B_DIGEST *digest // IN: the digest that was signed
504 )
505 {
506     POINT(P);
507     ECC_NUM(bnRp);
508     ECC_NUM(bnT);
509     BN_MAX_INITIALIZED(bnE, digest);
510     BOOL OK;
511     bigConst order = CurveGetOrder(AccessCurveData(E));
512
513     #ifdef _SM2_SIGN_DEBUG
514     // Make sure that the input signature is the test signature
515     pAssert(BnHexEqual(bnR,
516         "40F1EC59F793D9F49E09DCEF49130D41"
517         "94F79FB1EED2CAA55BACDB49C4E755D1"));
518     pAssert(BnHexEqual(bnS,
519         "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
520         "67A457872FB09EC56327A67EC7DEEBE7"));
521     #endif
522     // b) compute t := (r + s) mod n
523     BnAdd(bnT, bnR, bnS);
524     BnMod(bnT, order);
525     #ifdef _SM2_SIGN_DEBUG
526     pAssert(BnHexEqual(bnT,
527         "2B75F07ED7ECE7CCC1C8986B991F441A"
528         "D324D6D619FE06DD63ED32E0C997C801"));
529     #endif
530     // c) verify that t > 0
531     OK = !BnEqualZero(bnT);
532     if(!OK)
533         // set T to a value that should allow rest of the computations to run
534         // without trouble
535         BnCopy(bnT, bnS);
536     // d) compute (x, y) := [s]G + [t]Q
537     OK = BnEccModMult2(P, NULL, bnS, ecQ, bnT, E);
538     #ifdef _SM2_SIGN_DEBUG
539     pAssert(OK && BnHexEqual(P->x,
540         "110FCDA57615705D5E7B9324AC4B856D"
541         "23E6D9188B2AE47759514657CE25D112"));
542     #endif
543     // e) compute r' := (e + x) mod n (the x coordinate is in bnT)
544     OK = OK && BnAdd(bnRp, bnE, P->x);
545     OK = OK && BnMod(bnRp, order);
546
547     // f) verify that r' = r
548     OK = OK && (BnUnsignedCmp(bnR, bnRp) == 0);
549
550     if(!OK)
551         return TPM_RC_SIGNATURE;
552     else

```

```

553         return TPM_RC_SUCCESS;
554     }
555 #endif // ALG_SM2
556
557 #if ALG_ECSCHNORR

```

10.2.12.3.10 BnValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature.

Error Return	Meaning
TPM_RC_SIGNATURE	signature not valid

```

558 static TPM_RC
559 BnValidateSignatureEcSchnorr(
560     bigNum      bnR,          // IN: 'r' component of the signature
561     bigNum      bnS,          // IN: 's' component of the signature
562     TPM_ALG_ID  hashAlg,      // IN: hash algorithm of the signature
563     bigCurve     E,           // IN: the curve used in the signature
564                                     // process
565     bigPoint     ecQ,         // IN: the public point of the key
566     const TPM2B_DIGEST *digest // IN: the digest that was signed
567 )
568 {
569     BN_MAX(bnRn);
570     POINT(ecE);
571     BN_MAX(bnEx);
572     const ECC_CURVE_DATA *C = AccessCurveData(E);
573     bigConst      order = CurveGetOrder(C);
574     UINT16        digestSize = CryptHashGetDigestSize(hashAlg);
575     HASH_STATE     hashState;
576     TPM2B_TYPE(BUFFER, MAX(MAX_ECC_PARAMETER_BYTES, MAX_DIGEST_SIZE));
577     TPM2B_BUFFER   Ex2 = {{sizeof(Ex2.t.buffer), { 0 }}};
578     BOOL           OK;
579     //
580     // E = [s]G - [r]Q
581     BnMod(bnR, order);
582     // Make -r = n - r
583     BnSub(bnRn, order, bnR);
584     // E = [s]G + [-r]Q
585     OK = BnPointMult(ecE, CurveGetG(C), bnS, ecQ, bnRn, E) == TPM_RC_SUCCESS;
586     // // reduce the x portion of E mod q
587     // OK = OK && BnMod(ecE->x, order);
588     // Convert to byte string
589     OK = OK && BnTo2B(ecE->x, &Ex2.b,
590         (NUMBYTES) (BITS_TO_BYTES(BnSizeInBits(order))));
591     if(OK)
592     {
593     // Ex = h(pE.x || digest)
594         CryptHashStart(&hashState, hashAlg);
595         CryptDigestUpdate(&hashState, Ex2.t.size, Ex2.t.buffer);
596         CryptDigestUpdate(&hashState, digest->t.size, digest->t.buffer);
597         Ex2.t.size = CryptHashEnd(&hashState, digestSize, Ex2.t.buffer);
598         SchnorrReduce(&Ex2.b, order);
599         BnFrom2B(bnEx, &Ex2.b);
600         // see if Ex matches R
601         OK = BnUnsignedCmp(bnEx, bnR) == 0;
602     }
603     return (OK) ? TPM_RC_SUCCESS : TPM_RC_SIGNATURE;
604 }
605 #endif // ALG_ECSCHNORR

```


10.2.12.3.11 CryptEccValidateSignature()

This function validates an EcDsa() or EcSchnorr() signature. The point *Qin* needs to have been validated to be on the curve of *curveId*.

Error Return	Meaning
TPM_RC_SIGNATURE	not a valid signature

```

606 LIB_EXPORT TPM_RC
607 CryptEccValidateSignature(
608     TPMT_SIGNATURE *signature,    // IN: signature to be verified
609     OBJECT *signKey,             // IN: ECC key signed the hash
610     const TPM2B_DIGEST *digest    // IN: digest that was signed
611 )
612 {
613     CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
614     ECC_NUM(bnR);
615     ECC_NUM(bnS);
616     POINT_INITIALIZED(ecQ, &signKey->publicArea.unique.ecc);
617     bigConst          order;
618     TPM_RC            retVal;
619
620     if(E == NULL)
621         ERROR_RETURN(TPM_RC_VALUE);
622
623     order = CurveGetOrder(AccessCurveData(E));
624
625     // Make sure that the scheme is valid
626     switch(signature->sigAlg)
627     {
628         case TPM_ALG_ECDSA:
629             #if ALG_EC Schnorr
630                 case TPM_ALG_EC Schnorr:
631             #endif
632             #if ALG_SM2
633                 case TPM_ALG_SM2:
634             #endif
635                 break;
636             default:
637                 ERROR_RETURN(TPM_RC_SCHEME);
638                 break;
639     }
640     // Can convert r and s after determining that the scheme is an ECC scheme. If
641     // this conversion doesn't work, it means that the unmarshaling code for
642     // an ECC signature is broken.
643     BnFrom2B(bnR, &signature->signature.ecdsa.signatureR.b);
644     BnFrom2B(bnS, &signature->signature.ecdsa.signatureS.b);
645
646     // r and s have to be greater than 0 but less than the curve order
647     if(BnEqualZero(bnR) || BnEqualZero(bnS))
648         ERROR_RETURN(TPM_RC_SIGNATURE);
649     if((BnUnsignedCmp(bnS, order) >= 0)
650        || (BnUnsignedCmp(bnR, order) >= 0))
651         ERROR_RETURN(TPM_RC_SIGNATURE);
652
653     switch(signature->sigAlg)
654     {
655         case TPM_ALG_ECDSA:
656             retVal = BnValidateSignatureEcDsa(bnR, bnS, E, ecQ, digest);
657             break;
658
659             #if ALG_EC Schnorr
660             case TPM_ALG_EC Schnorr:
661                 retVal = BnValidateSignatureEcSchnorr(bnR, bnS,

```



```

662                                     signature->signature.any.hashAlg,
663                                     E, ecQ, digest);
664                                     break;
665 #endif
666 #if ALG_SM2
667     case TPM_ALG_SM2:
668         retVal = BnValidateSignatureEcSm2(bnR, bnS, E, ecQ, digest);
669         break;
670 #endif
671     default:
672         FAIL(FATAL_ERROR_INTERNAL);
673 }
674 Exit:
675     CURVE_FREE(E);
676     return retVal;
677 }

```

10.2.12.3.12 CryptEccCommitCompute()

This function performs the point multiply operations required by TPM2_Commit.

If *B* or *M* is provided, they must be on the curve defined by *curveId*. This routine does not check that they are on the curve and results are unpredictable if they are not.

It is a fatal error if *r* is NULL. If *B* is not NULL, then it is a fatal error if *d* is NULL or if *K* and *L* are both NULL. If *M* is not NULL, then it is a fatal error if *E* is NULL.

Error Return	Meaning
TPM_RC_NO_RESULT	if <i>K</i> , <i>L</i> or <i>E</i> was computed to be the point at infinity
TPM_RC_CANCELED	a cancel indication was asserted during this function

```

678 LIB_EXPORT TPM_RC
679 CryptEccCommitCompute(
680     TPMS_ECC_POINT *K,           // OUT: [d]B or [r]Q
681     TPMS_ECC_POINT *L,           // OUT: [r]B
682     TPMS_ECC_POINT *E,           // OUT: [r]M
683     TPM_ECC_CURVE   curveId,     // IN: the curve for the computations
684     TPMS_ECC_POINT *M,           // IN: M (optional)
685     TPMS_ECC_POINT *B,           // IN: B (optional)
686     TPM2B_ECC_PARAMETER *d,      // IN: d (optional)
687     TPM2B_ECC_PARAMETER *r,      // IN: the computed r value (required)
688 )
689 {
690     CURVE_INITIALIZED(curve, curveId); // Normally initialize E as the curve, but
691                                         // E means something else in this function
692     ECC_INITIALIZED(bnR, r);
693     TPM_RC retVal = TPM_RC_SUCCESS;
694 //
695     // Validate that the required parameters are provided.
696     // Note: E has to be provided if computing E := [r]Q or E := [r]M. Will do
697     // E := [r]Q if both M and B are NULL.
698     pAssert(r != NULL && E != NULL);
699
700     // Initialize the output points in case they are not computed
701     ClearPoint2B(K);
702     ClearPoint2B(L);
703     ClearPoint2B(E);
704
705     // Sizes of the r parameter may not be zero
706     pAssert(r->t.size > 0);
707
708     // If B is provided, compute K=[d]B and L=[r]B
709     if(B != NULL)

```

```

710     {
711         ECC_INITIALIZED(bnD, d);
712         POINT_INITIALIZED(pB, B);
713         POINT(pK);
714         POINT(pL);
715     //
716     pAssert(d != NULL && K != NULL && L != NULL);
717
718     if(!BnIsOnCurve(pB, AccessCurveData(curve)))
719         ERROR_RETURN(TPM_RC_VALUE);
720     // do the math for K = [d]B
721     if((retVal = BnPointMult(pK, pB, bnD, NULL, NULL, curve)) != TPM_RC_SUCCESS)
722         goto Exit;
723     // Convert BN K to TPM2B K
724     BnPointTo2B(K, pK, curve);
725     // compute L= [r]B after checking for cancel
726     if(_plat_IsCanceled())
727         ERROR_RETURN(TPM_RC_CANCELED);
728     // compute L = [r]B
729     if(!BnIsValidPrivateEcc(bnR, curve))
730         ERROR_RETURN(TPM_RC_VALUE);
731     if((retVal = BnPointMult(pL, pB, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
732         goto Exit;
733     // Convert BN L to TPM2B L
734     BnPointTo2B(L, pL, curve);
735 }
736 if((M != NULL) || (B == NULL))
737 {
738     POINT_INITIALIZED(pM, M);
739     POINT(pE);
740 //
741     // Make sure that a place was provided for the result
742     pAssert(E != NULL);
743
744     // if this is the third point multiply, check for cancel first
745     if((B != NULL) && _plat_IsCanceled())
746         ERROR_RETURN(TPM_RC_CANCELED);
747
748     // If M provided, then pM will not be NULL and will compute E = [r]M.
749     // However, if M was not provided, then pM will be NULL and E = [r]G
750     // will be computed
751     if((retVal = BnPointMult(pE, pM, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
752         goto Exit;
753     // Convert E to 2B format
754     BnPointTo2B(E, pE, curve);
755 }
756 Exit:
757     CURVE_FREE(curve);
758     return retVal;
759 }
760 #endif // ALG_ECC

```

10.2.13 CryptHash.c

10.2.13.1 Description

This file contains implementation of cryptographic functions for hashing.

10.2.13.2 Includes, Defines, and Types

```
1  #define _CRYPT_HASH_C_
2  #include "Tpm.h"
3  #include "CryptHash_fp.h"
4  #include "CryptHash.h"
5  #include "OIDs.h"
```

Instance each of the hash descriptors based on the implemented algorithms

```
6  FOR_EACH_HASH(HASH_DEF_TEMPLATE)
```

Instance a *null* def.

```
7  HASH_DEF NULL_Def = {{0}};
```

Create a table of pointers to the defined hash definitions

```
8  #define HASH_DEF_ENTRY(HASH, Hash)      &Hash##_Def,
9  PHASH_DEF HashDefArray[] = {
10     // for each implemented HASH, expands to: &HASH_Def,
11     FOR_EACH_HASH(HASH_DEF_ENTRY)
12     &NULL_Def
13 };
14 #undef HASH_DEF_ENTRY
```

10.2.13.3 Obligatory Initialization Functions

10.2.13.3.1 CryptHashInit()

This function is called by `_TPM_Init` to perform the initialization operations for the library.

```
15  BOOL
16  CryptHashInit(
17      void
18  )
19  {
20      LibHashInit();
21      return TRUE;
22  }
```

10.2.13.3.2 CryptHashStartup()

This function is called by `TPM2_Startup()`. It checks that the size of the `HashDefArray()` is consistent with the `HASH_COUNT`.

```
23  BOOL
24  CryptHashStartup(
25      void
26  )
27  {
```

```

28     int            i = sizeof(HashDefArray) / sizeof(PHASH_DEF) - 1;
29     return (i == HASH_COUNT);
30 }

```

10.2.13.4 Hash Information Access Functions

10.2.13.4.1 Introduction

These functions provide access to the hash algorithm description information.

10.2.13.4.2 CryptGetHashDef()

This function accesses the hash descriptor associated with a hash algorithm. The function returns a pointer to a *null* descriptor if *hashAlg* is TPM_ALG_NULL or not a defined algorithm.

```

31 PHASH_DEF
32 CryptGetHashDef(
33     TPM_ALG_ID      hashAlg
34 )
35 {
36 #define GET_DEF(HASH, Hash) case ALG_##HASH##_VALUE: return &Hash##_Def;
37     switch(hashAlg)
38     {
39         FOR_EACH_HASH(GET_DEF)
40     default:
41         return &NULL_Def;
42     }
43 #undef GET_DEF
44 }

```

10.2.13.4.3 CryptHashIsValidAlg()

This function tests to see if an algorithm ID is a valid hash algorithm. If flag is true, then TPM_ALG_NULL is a valid hash.

Return Value	Meaning
TRUE(1)	<i>hashAlg</i> is a valid, implemented hash on this TPM
FALSE(0)	<i>hashAlg</i> is not valid for this TPM

```

45 BOOL
46 CryptHashIsValidAlg(
47     TPM_ALG_ID      hashAlg,           // IN: the algorithm to check
48     BOOL            flag               // IN: TRUE if TPM_ALG_NULL is to be treated
49                                     // as a valid hash
50 )
51 {
52     if(hashAlg == TPM_ALG_NULL)
53         return flag;
54     return CryptGetHashDef(hashAlg) != &NULL_Def;
55 }

```

10.2.13.4.4 CryptHashGetAlgByIndex()

This function is used to iterate through the hashes. TPM_ALG_NULL is returned for all indexes that are not valid hashes. If the TPM implements 3 hashes, then an *index* value of 0 will return the first implemented hash and an *index* of 2 will return the last. All other index values will return TPM_ALG_NULL.

Return Value	Meaning
TPM_ALG_XXX	a hash algorithm
TPM_ALG_NULL	this can be used as a stop value

```

56 LIB_EXPORT TPM_ALG_ID
57 CryptHashGetAlgByIndex(
58     UINT32      index          // IN: the index
59 )
60 {
61     TPM_ALG_ID    hashAlg;
62     if(index >= HASH_COUNT)
63         hashAlg = TPM_ALG_NULL;
64     else
65         hashAlg = HashDefArray[index]->hashAlg;
66     return hashAlg;
67 }

```

10.2.13.4.5 CryptHashGetDigestSize()

Returns the size of the digest produced by the hash. If *hashAlg* is not a hash algorithm, the TPM will FAIL.

Return Value	Meaning
0	TPM_ALG_NULL
> 0	the digest size

```

68 LIB_EXPORT UINT16
69 CryptHashGetDigestSize(
70     TPM_ALG_ID    hashAlg          // IN: hash algorithm to look up
71 )
72 {
73     return CryptGetHashDef(hashAlg)->digestSize;
74 }

```

10.2.13.4.6 CryptHashGetBlockSize()

Returns the size of the block used by the hash. If *hashAlg* is not a hash algorithm, the TPM will FAIL.

Return Value	Meaning
0	TPM_ALG_NULL
> 0	the digest size

```

75 LIB_EXPORT UINT16
76 CryptHashGetBlockSize(
77     TPM_ALG_ID    hashAlg          // IN: hash algorithm to look up
78 )
79 {
80     return CryptGetHashDef(hashAlg)->blockSize;
81 }

```

10.2.13.4.7 CryptHashGetOid()

This function returns a pointer to DER-encoded OID for a hash algorithm. All OIDs are full OID values including the Tag (0x06) and length byte.

```

82  LIB_EXPORT const BYTE *
83  CryptHashGetOid(
84      TPM_ALG_ID      hashAlg
85  )
86  {
87      return CryptGetHashDef(hashAlg)->OID;
88  }

```

10.2.13.4.8 CryptHashGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```

89  TPM_ALG_ID
90  CryptHashGetContextAlg(
91      PHASH_STATE      state          // IN: the context to check
92  )
93  {
94      return state->hashAlg;
95  }

```

10.2.13.5 State Import and Export

10.2.13.5.1 CryptHashCopyState()

This function is used to clone a HASH_STATE.

```

96  LIB_EXPORT void
97  CryptHashCopyState(
98      HASH_STATE      *out,          // OUT: destination of the state
99      const HASH_STATE *in          // IN: source of the state
100  )
101  {
102      pAssert(out->type == in->type);
103      out->hashAlg = in->hashAlg;
104      out->def = in->def;
105      if(in->hashAlg != TPM_ALG_NULL)
106      {
107          HASH_STATE_COPY(out, in);
108      }
109      if(in->type == HASH_STATE_HMAC)
110      {
111          const HMAC_STATE *hIn = (HMAC_STATE *)in;
112          HMAC_STATE *hOut = (HMAC_STATE *)out;
113          hOut->hmacKey = hIn->hmacKey;
114      }
115      return;
116  }

```

10.2.13.5.2 CryptHashExportState()

This function is used to export a hash or HMAC hash state. This function would be called when preparing to context save a sequence object.

```

117  void
118  CryptHashExportState(
119      PCHASH_STATE      internalFmt,  // IN: the hash state formatted for use by
120                                  // library
121      PEXPORT_HASH_STATE externalFmt // OUT: the exported hash state
122  )
123  {

```

```

124     BYTE                                *outBuf = (BYTE *)externalFmt;
125 //
126     cAssert(sizeof(HASH_STATE) <= sizeof(EXPORT_HASH_STATE));
127     // the following #define is used to move data from an aligned internal data
128     // structure to a byte buffer (external format data.
129 #define CopyToOffset(value)                                \
130     memcpy(&outBuf[offsetof(HASH_STATE,value)], &internalFmt->value, \
131           sizeof(internalFmt->value))
132     // Copy the hashAlg
133     CopyToOffset(hashAlg);
134     CopyToOffset(type);
135 #ifndef HASH_STATE_SMAC
136     if(internalFmt->type == HASH_STATE_SMAC)
137     {
138         memcpy(outBuf, internalFmt, sizeof(HASH_STATE));
139         return;
140     }
141 #endif
142     if(internalFmt->type == HASH_STATE_HMAC)
143     {
144         HMAC_STATE *from = (HMAC_STATE *)internalFmt;
145         memcpy(&outBuf[offsetof(HMAC_STATE, hmacKey)], &from->hmacKey,
146               sizeof(from->hmacKey));
147     }
148     if(internalFmt->hashAlg != TPM_ALG_NULL)
149         HASH_STATE_EXPORT(externalFmt, internalFmt);
150 }
151

```

10.2.13.5.3 CryptHashImportState()

This function is used to import the hash state. This function would be called to import a hash state when the context of a sequence object was being loaded.

```

152 void
153 CryptHashImportState(
154     PHASH_STATE      internalFmt,    // OUT: the hash state formatted for use by
155                                     // the library
156     PCEXPORT_HASH_STATE externalFmt // IN: the exported hash state
157 )
158 {
159     BYTE *inBuf = (BYTE *)externalFmt;
160 //
161 #define CopyFromOffset(value)                                \
162     memcpy(&internalFmt->value, &inBuf[offsetof(HASH_STATE,value)], \
163           sizeof(internalFmt->value))
164
165     // Copy the hashAlg of the byte-aligned input structure to the structure-aligned
166     // internal structure.
167     CopyFromOffset(hashAlg);
168     CopyFromOffset(type);
169     if(internalFmt->hashAlg != TPM_ALG_NULL)
170     {
171 #ifndef HASH_STATE_SMAC
172         if(internalFmt->type == HASH_STATE_SMAC)
173         {
174             memcpy(internalFmt, inBuf, sizeof(HASH_STATE));
175             return;
176         }
177 #endif
178         internalFmt->def = CryptGetHashDef(internalFmt->hashAlg);
179         HASH_STATE_IMPORT(internalFmt, inBuf);
180         if(internalFmt->type == HASH_STATE_HMAC)
181         {

```



```

182         HMAC_STATE          *to = (HMAC_STATE *)internalFmt;
183         memcpy(&to->hmacKey, &inBuf[offsetof(HMAC_STATE, hmacKey)],
184               sizeof(to->hmacKey));
185     }
186 }
187 }

```

10.2.13.6 State Modification Functions

10.2.13.6.1 HashEnd()

Local function to complete a hash that uses the *hashDef* instead of an algorithm ID. This function is used to complete the hash and only return a partial digest. The return value is the size of the data copied.

```

188 static UINT16
189 HashEnd(
190     PHASH_STATE    hashState,      // IN: the hash state
191     UINT32          dOutSize,      // IN: the size of receive buffer
192     PBYTE           dOut           // OUT: the receive buffer
193 )
194 {
195     BYTE            temp[MAX_DIGEST_SIZE];
196     if((hashState->hashAlg == TPM_ALG_NULL)
197         || (hashState->type != HASH_STATE_HASH))
198         dOutSize = 0;
199     if(dOutSize > 0)
200     {
201         hashState->def = CryptGetHashDef(hashState->hashAlg);
202         // Set the final size
203         dOutSize = MIN(dOutSize, hashState->def->digestSize);
204         // Complete into the temp buffer and then copy
205         HASH_END(hashState, temp);
206         // Don't want any other functions calling the HASH_END method
207         // directly.
208 #undef HASH_END
209         memcpy(dOut, &temp, dOutSize);
210     }
211     hashState->type = HASH_STATE_EMPTY;
212     return (UINT16)dOutSize;
213 }

```

10.2.13.6.2 CryptHashStart()

Functions starts a hash stack Start a hash stack and returns the digest size. As a side effect, the value of *stateSize* in *hashState* is updated to indicate the number of bytes of state that were saved. This function calls *GetHashServer()* and that function will put the TPM into failure mode if the hash algorithm is not supported.

This function does not use the sequence parameter. If it is necessary to import or export context, this will start the sequence in a local state and export the state to the input buffer. Will need to add a flag to the state structure to indicate that it needs to be imported before it can be used. (BLEH).

Return Value	Meaning
0	hash is TPM_ALG_NULL
>0	digest size

```

214 LIB_EXPORT UINT16
215 CryptHashStart(
216     PHASH_STATE    hashState,      // OUT: the running hash state

```

```

217     TPM_ALG_ID      hashAlg      // IN: hash algorithm
218 )
219 {
220     UINT16           retVal;
221
222     TEST(hashAlg);
223
224     hashState->hashAlg = hashAlg;
225     if(hashAlg == TPM_ALG_NULL)
226     {
227         retVal = 0;
228     }
229     else
230     {
231         hashState->def = CryptGetHashDef(hashAlg);
232         HASH_START(hashState);
233         retVal = hashState->def->digestSize;
234     }
235 #undef HASH_START
236     hashState->type = HASH_STATE_HASH;
237     return retVal;
238 }

```

10.2.13.6.3 CryptDigestUpdate()

Add data to a hash or HMAC, SMAC stack.

```

239 void
240 CryptDigestUpdate(
241     PHASH_STATE      hashState,      // IN: the hash context information
242     UINT32           dataSize,      // IN: the size of data to be added
243     const BYTE       *data          // IN: data to be hashed
244 )
245 {
246     if(hashState->hashAlg != TPM_ALG_NULL)
247     {
248         if((hashState->type == HASH_STATE_HASH)
249            || (hashState->type == HASH_STATE_HMAC))
250             HASH_DATA(hashState, dataSize, (BYTE *)data);
251 #if SMAC_IMPLEMENTED
252         else if(hashState->type == HASH_STATE_SMAC)
253             (hashState->state.smac.smacMethods.data)(&hashState->state.smac.state,
254                                                     dataSize, data);
255 #endif // SMAC_IMPLEMENTED
256         else
257             FAIL(FATAL_ERROR_INTERNAL);
258     }
259     return;
260 }

```

10.2.13.6.4 CryptHashEnd()

Complete a hash or HMAC computation. This function will place the smaller of *digestSize* or the size of the digest in *dOut*. The number of bytes in the placed in the buffer is returned. If there is a failure, the returned value is ≤ 0 .

Return Value	Meaning
0	no data returned
> 0	the number of bytes in the digest or <i>dOutSize</i> , whichever is smaller

```

261 LIB_EXPORT UINT16
262 CryptHashEnd(
263     PHASH_STATE    hashState,    // IN: the state of hash stack
264     UINT32          dOutSize,    // IN: size of digest buffer
265     BYTE            *dOut        // OUT: hash digest
266 )
267 {
268     pAssert(hashState->type == HASH_STATE_HASH);
269     return HashEnd(hashState, dOutSize, dOut);
270 }

```

10.2.13.6.5 CryptHashBlock()

Start a hash, hash a single block, update *digest* and return the size of the results.

The *digestSize* parameter can be smaller than the digest. If so, only the more significant bytes are returned.

Return Value	Meaning
>= 0	number of bytes placed in <i>dOut</i>

```

271 LIB_EXPORT UINT16
272 CryptHashBlock(
273     TPM_ALG_ID      hashAlg,    // IN: The hash algorithm
274     UINT32           dataSize,   // IN: size of buffer to hash
275     const BYTE       *data,     // IN: the buffer to hash
276     UINT32           dOutSize,   // IN: size of the digest buffer
277     BYTE             *dOut      // OUT: digest buffer
278 )
279 {
280     HASH_STATE       state;
281     CryptHashStart(&state, hashAlg);
282     CryptDigestUpdate(&state, dataSize, data);
283     return HashEnd(&state, dOutSize, dOut);
284 }

```

10.2.13.6.6 CryptDigestUpdate2B()

This function updates a digest (hash or HMAC) with a TPM2B.

This function can be used for both HMAC and hash functions so the *digestState* is void so that either state type can be passed.

```

285 LIB_EXPORT void
286 CryptDigestUpdate2B(
287     PHASH_STATE      state,    // IN: the digest state
288     const TPM2B       *bIn     // IN: 2B containing the data
289 )
290 {
291     // Only compute the digest if a pointer to the 2B is provided.
292     // In CryptDigestUpdate(), if size is zero or buffer is NULL, then no change
293     // to the digest occurs. This function should not provide a buffer if bIn is
294     // not provided.
295     pAssert(bIn != NULL);

```

```

296     CryptDigestUpdate(state, bIn->size, bIn->buffer);
297     return;
298 }

```

10.2.13.6.7 CryptHashEnd2B()

This function is the same as CryptCompleteHash() but the digest is placed in a TPM2B. This is the most common use and this is provided for specification clarity. *digest.size* should be set to indicate the number of bytes to place in the buffer

Return Value	Meaning
≥ 0	the number of bytes placed in <i>digest.buffer</i>

```

299 LIB_EXPORT UINT16
300 CryptHashEnd2B(
301     PHASH_STATE    state,          // IN: the hash state
302     P2B            digest          // IN: the size of the buffer Out: requested
303                                     // number of bytes
304 )
305 {
306     return CryptHashEnd(state, digest->size, digest->buffer);
307 }

```

10.2.13.6.8 CryptDigestUpdateInt()

This function is used to include an integer value to a hash stack. The function marshals the integer into its canonical form before calling CryptDigestUpdate().

```

308 LIB_EXPORT void
309 CryptDigestUpdateInt(
310     void            *state,          // IN: the state of hash stack
311     UINT32          intSize,         // IN: the size of 'intValue' in bytes
312     UINT64          intValue         // IN: integer value to be hashed
313 )
314 {
315     #if LITTLE_ENDIAN_TPM
316         intValue = REVERSE_ENDIAN_64(intValue);
317     #endif
318     CryptDigestUpdate(state, intSize, &((BYTE *)&intValue)[8 - intSize]);
319 }

```

10.2.13.7 HMAC Functions

10.2.13.7.1 CryptHmacStart()

This function is used to start an HMAC using a temp hash context. The function does the initialization of the hash with the HMAC key XOR *iPad* and updates the HMAC key XOR *oPad*.

The function returns the number of bytes in a digest produced by *hashAlg*.

Return Value	Meaning
≥ 0	number of bytes in digest produced by <i>hashAlg</i> (may be zero)

```

320 LIB_EXPORT UINT16
321 CryptHmacStart(
322     PHMAC_STATE    state,          // IN/OUT: the state buffer

```

```

323     TPM_ALG_ID      hashAlg,          // IN: the algorithm to use
324     UINT16          keySize,          // IN: the size of the HMAC key
325     const BYTE      *key              // IN: the HMAC key
326 )
327 {
328     PHASH_DEF        hashDef;
329     BYTE *           pb;
330     UINT32           i;
331 //
332     hashDef = CryptGetHashDef(hashAlg);
333     if(hashDef->digestSize != 0)
334     {
335         // If the HMAC key is larger than the hash block size, it has to be reduced
336         // to fit. The reduction is a digest of the hashKey.
337         if(keySize > hashDef->blockSize)
338         {
339             // if the key is too big, reduce it to a digest of itself
340             state->hmacKey.t.size = CryptHashBlock(hashAlg, keySize, key,
341                                                     hashDef->digestSize,
342                                                     state->hmacKey.t.buffer);
343         }
344         else
345         {
346             memcpy(state->hmacKey.t.buffer, key, keySize);
347             state->hmacKey.t.size = keySize;
348         }
349         // XOR the key with iPad (0x36)
350         pb = state->hmacKey.t.buffer;
351         for(i = state->hmacKey.t.size; i > 0; i--)
352             *pb++ ^= 0x36;
353
354         // if the keySize is smaller than a block, fill the rest with 0x36
355         for(i = hashDef->blockSize - state->hmacKey.t.size; i > 0; i--)
356             *pb++ = 0x36;
357
358         // Increase the oPadSize to a full block
359         state->hmacKey.t.size = hashDef->blockSize;
360
361         // Start a new hash with the HMAC key
362         // This will go in the caller's state structure and may be a sequence or not
363         CryptHashStart((PHASH_STATE)state, hashAlg);
364         CryptDigestUpdate((PHASH_STATE)state, state->hmacKey.t.size,
365                           state->hmacKey.t.buffer);
366         // XOR the key block with 0x5c ^ 0x36
367         for(pb = state->hmacKey.t.buffer, i = hashDef->blockSize; i > 0; i--)
368             *pb++ ^= (0x5c ^ 0x36);
369     }
370     // Set the hash algorithm
371     state->hashState.hashAlg = hashAlg;
372     // Set the hash state type
373     state->hashState.type = HASH_STATE_HMAC;
374
375     return hashDef->digestSize;
376 }

```

10.2.13.7.2 CryptHmacEnd()

This function is called to complete an HMAC. It will finish the current digest, and start a new digest. It will then add the *oPadKey* and the completed digest and return the results in *dOut*. It will not return more than *dOutSize* bytes.

Return Value	Meaning
≥ 0	number of bytes in <i>dOut</i> (may be zero)

```

377 LIB_EXPORT UINT16
378 CryptHmacEnd(
379     PHMAC_STATE    state,           // IN: the hash state buffer
380     UINT32          dOutSize,       // IN: size of digest buffer
381     BYTE            *dOut           // OUT: hash digest
382 )
383 {
384     BYTE            temp[MAX_DIGEST_SIZE];
385     PHASH_STATE     hState = (PHASH_STATE)&state->hashState;
386
387     #if SMAC_IMPLEMENTED
388     if(hState->type == HASH_STATE_SMAC)
389         return (state->hashState.state.smac.smacMethods.end)
390             (&state->hashState.state.smac.state,
391              dOutSize,
392              dOut);
393     #endif
394     pAssert(hState->type == HASH_STATE_HMAC);
395     hState->def = CryptGetHashDef(hState->hashAlg);
396     // Change the state type for completion processing
397     hState->type = HASH_STATE_HASH;
398     if(hState->hashAlg == TPM_ALG_NULL)
399         dOutSize = 0;
400     else
401     {
402         // Complete the current hash
403         HashEnd(hState, hState->def->digestSize, temp);
404         // Do another hash starting with the oPad
405         CryptHashStart(hState, hState->hashAlg);
406         CryptDigestUpdate(hState, state->hmacKey.t.size, state->hmacKey.t.buffer);
407         CryptDigestUpdate(hState, hState->def->digestSize, temp);
408     }
409     return HashEnd(hState, dOutSize, dOut);
410 }

```

10.2.13.7.3 CryptHmacStart2B()

This function starts an HMAC and returns the size of the digest that will be produced.

This function is provided to support the most common use of starting an HMAC with a TPM2B key.

The caller must provide a block of memory in which the hash sequence state is kept. The caller should not alter the contents of this buffer until the hash sequence is completed or abandoned.

Return Value	Meaning
> 0	the digest size of the algorithm
$= 0$	the <i>hashAlg</i> was TPM_ALG_NULL

```

411 LIB_EXPORT UINT16
412 CryptHmacStart2B(
413     PHMAC_STATE     hmacState,      // OUT: the state of HMAC stack. It will be used
414                                     // in HMAC update and completion
415     TPMI_ALG_HASH    hashAlg,       // IN: hash algorithm
416     P2B              key            // IN: HMAC key
417 )
418 {
419     return CryptHmacStart(hmacState, hashAlg, key->size, key->buffer);
420 }

```

10.2.13.7.4 CryptHmacEnd2B()

This function is the same as CryptHmacEnd() but the HMAC result is returned in a TPM2B which is the most common use.

Return Value	Meaning
≥ 0	the number of bytes placed in <i>digest</i>

```

421 LIB_EXPORT UINT16
422 CryptHmacEnd2B(
423     PHMAC_STATE    hmacState,    // IN: the state of HMAC stack
424     P2B            digest        // OUT: HMAC
425 )
426 {
427     return CryptHmacEnd(hmacState, digest->size, digest->buffer);
428 }
```

10.2.13.8 Mask and Key Generation Functions

10.2.13.8.1 CryptMGF_KDF()

This function performs MGF1/KDF1 or KDF2 using the selected hash. KDF1 and KDF2 are $T(n) = T(n-1) \parallel H(\text{seed} \parallel \text{counter})$ with the difference being that, with KDF1, *counter* starts at 0 but with KDF2, *counter* starts at 1. The caller determines which version by setting the initial value of counter to either 0 or 1.

NOTE Any value that is not 0 is considered to be 1.

This function returns the length of the mask produced which could be zero if the digest algorithm is not supported

Return Value	Meaning
0	hash algorithm was TPM_ALG_NULL
> 0	should be the same as <i>mSize</i>

```

429 LIB_EXPORT UINT16
430 CryptMGF_KDF(
431     UINT32    mSize,    // IN: length of the mask to be produced
432     BYTE      *mask,    // OUT: buffer to receive the mask
433     TPM_ALG_ID hashAlg, // IN: hash to use
434     UINT32    seedSize, // IN: size of the seed
435     BYTE      *seed,    // IN: seed size
436     UINT32    counter   // IN: counter initial value
437 )
438 {
439     HASH_STATE hashState;
440     PHASH_DEF  hDef = CryptGetHashDef(hashAlg);
441     UINT32     hLen;
442     UINT32     bytes;
443     //
444     // If there is no digest to compute return
445     if((hDef->digestSize == 0) || (mSize == 0))
446         return 0;
447     if(counter != 0)
448         counter = 1;
449     hLen = hDef->digestSize;
450     for(bytes = 0; bytes < mSize; bytes += hLen)
451     {
452         // Start the hash and include the seed and counter
```



```

453     CryptHashStart(&hashState, hashAlg);
454     CryptDigestUpdate(&hashState, seedSize, seed);
455     CryptDigestUpdateInt(&hashState, 4, counter);
456     // Get as much as will fit.
457     CryptHashEnd(&hashState, MIN((mSize - bytes), hLen),
458                 &mask[bytes]);
459     counter++;
460 }
461 return (UINT16)mSize;
462 }

```

10.2.13.8.2 CryptKDFa()

This function performs the key generation according to Part 1 of the TPM specification.

This function returns the number of bytes generated which may be zero.

The *key* and *keyStream* pointers are not allowed to be NULL. The other pointer values may be NULL. The value of *sizeInBits* must be no larger than $(2^{18})-1 = 256\text{K bits}$ (32385 bytes).

The *once* parameter is set to allow incremental generation of a large value. If this flag is TRUE, *sizeInBits* will be used in the HMAC computation but only one iteration of the KDF is performed. This would be used for XOR obfuscation so that the mask value can be generated in digest-sized chunks rather than having to be generated all at once in an arbitrarily large buffer and then XORed into the result. If *once* is TRUE, then *sizeInBits* must be a multiple of 8.

Any error in the processing of this command is considered fatal.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the <i>keyStream</i> buffer

```

463 LIB_EXPORT UINT16
464 CryptKDFa(
465     TPM_ALG_ID      hashAlg,           // IN: hash algorithm used in HMAC
466     const TPM2B      *key,             // IN: HMAC key
467     const TPM2B      *label,           // IN: a label for the KDF
468     const TPM2B      *contextU,        // IN: context U
469     const TPM2B      *contextV,        // IN: context V
470     UINT32           sizeInBits,        // IN: size of generated key in bits
471     BYTE             *keyStream,        // OUT: key buffer
472     UINT32           *counterInOut,     // IN/OUT: caller may provide the iteration
473                                     // counter for incremental operations to
474                                     // avoid large intermediate buffers.
475     UINT16           blocks,           // IN: If non-zero, this is the maximum number
476                                     // of blocks to be returned, regardless
477                                     // of sizeInBits
478 )
479 {
480     UINT32           counter = 0;       // counter value
481     INT16            bytes;             // number of bytes to produce
482     UINT16           generated;         // number of bytes generated
483     BYTE             *stream = keyStream;
484     HMAC_STATE       hState;
485     UINT16           digestSize = CryptHashGetDigestSize(hashAlg);
486
487     pAssert(key != NULL && keyStream != NULL);
488
489     TEST(TPM_ALG_KDF1_SP800_108);
490
491     if(digestSize == 0)
492         return 0;

```

```

493
494     if(counterInOut != NULL)
495         counter = *counterInOut;
496
497     // If the size of the request is larger than the numbers will handle,
498     // it is a fatal error.
499     pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);
500
501     // The number of bytes to be generated is the smaller of the sizeInBits bytes or
502     // the number of requested blocks. The number of blocks is the smaller of the
503     // number requested or the number allowed by sizeInBits. A partial block is
504     // a full block.
505     bytes = (blocks > 0) ? blocks * digestSize : (UINT16)BITS_TO_BYTES(sizeInBits);
506     generated = bytes;
507
508     // Generate required bytes
509     for(; bytes > 0; bytes -= digestSize)
510     {
511         counter++;
512         // Start HMAC
513         if(CryptHmacStart(&hState, hashAlg, key->size, key->buffer) == 0)
514             return 0;
515         // Adding counter
516         CryptDigestUpdateInt(&hState.hashState, 4, counter);
517
518         // Adding label
519         if(label != NULL)
520             HASH_DATA(&hState.hashState, label->size, (BYTE *)label->buffer);
521         // Add a null. SP108 is not very clear about when the 0 is needed but to
522         // make this like the previous version that did not add an 0x00 after
523         // a null-terminated string, this version will only add a null byte
524         // if the label parameter did not end in a null byte, or if no label
525         // is present.
526         if((label == NULL)
527            || (label->size == 0)
528            || (label->buffer[label->size - 1] != 0))
529             CryptDigestUpdateInt(&hState.hashState, 1, 0);
530         // Adding contextU
531         if(contextU != NULL)
532             HASH_DATA(&hState.hashState, contextU->size, contextU->buffer);
533         // Adding contextV
534         if(contextV != NULL)
535             HASH_DATA(&hState.hashState, contextV->size, contextV->buffer);
536         // Adding size in bits
537         CryptDigestUpdateInt(&hState.hashState, 4, sizeInBits);
538
539         // Complete and put the data in the buffer
540         CryptHmacEnd(&hState, bytes, stream);
541         stream = &stream[digestSize];
542     }
543     // Masking in the KDF is disabled. If the calling function wants something
544     // less than even number of bytes, then the caller should do the masking
545     // because there is no universal way to do it here
546     if(counterInOut != NULL)
547         *counterInOut = counter;
548     return generated;
549 }

```

10.2.13.8.3 CryptKDFe()

This function implements KDFe() as defined in TPM specification part 1.

This function returns the number of bytes generated which may be zero.

The *Z* and *keyStream* pointers are not allowed to be NULL. The other pointer values may be NULL. The value of *sizeInBits* must be no larger than $(2^{18})-1 = 256\text{K bits}$ (32385 bytes). Any error in the processing of this command is considered fatal.

Return Value	Meaning
0	hash algorithm is not supported or is TPM_ALG_NULL
> 0	the number of bytes in the <i>keyStream</i> buffer

```

550 LIB_EXPORT UINT16
551 CryptKDFe(
552     TPM_ALG_ID    hashAlg,          // IN: hash algorithm used in HMAC
553     TPM2B         *Z,               // IN: Z
554     const TPM2B    *label,          // IN: a label value for the KDF
555     TPM2B         *partyUInfo,      // IN: PartyUInfo
556     TPM2B         *partyVInfo,      // IN: PartyVInfo
557     UINT32        sizeInBits,       // IN: size of generated key in bits
558     BYTE          *keyStream        // OUT: key buffer
559 )
560 {
561     HASH_STATE     hashState;
562     PHASH_DEF      hashDef = CryptGetHashDef(hashAlg);
563
564     UINT32          counter = 0;     // counter value
565     UINT16          hLen;
566     BYTE            *stream = keyStream;
567     INT16           bytes;          // number of bytes to generate
568
569     pAssert(keyStream != NULL && Z != NULL && ((sizeInBits + 7) / 8) < INT16_MAX);
570     //
571     hLen = hashDef->digestSize;
572     bytes = (INT16)((sizeInBits + 7) / 8);
573     if(hashAlg == TPM_ALG_NULL || bytes == 0)
574         return 0;
575
576     // Generate required bytes
577     //The inner loop of that KDF uses:
578     // Hash[i] := H(counter | Z | OtherInfo) (5)
579     // Where:
580     // Hash[i]    the hash generated on the i-th iteration of the loop.
581     // H()        an approved hash function
582     // counter    a 32-bit counter that is initialized to 1 and incremented
583     //             on each iteration
584     // Z          the X coordinate of the product of a public ECC key and a
585     //             different private ECC key.
586     // OtherInfo  a collection of qualifying data for the KDF defined below.
587     // In this specification, OtherInfo will be constructed by:
588     // OtherInfo := Use | PartyUInfo | PartyVInfo
589     for(; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
590     {
591         if(bytes < hLen)
592             hLen = bytes;
593         counter++;
594         // Do the hash
595         CryptHashStart(&hashState, hashAlg);
596         // Add counter
597         CryptDigestUpdateInt(&hashState, 4, counter);
598
599         // Add Z
600         if(Z != NULL)
601             CryptDigestUpdate2B(&hashState, Z);
602         // Add label
603         if(label != NULL)
604             CryptDigestUpdate2B(&hashState, label);

```

```
605 // Add a null. SP108 is not very clear about when the 0 is needed but to
606 // make this like the previous version that did not add an 0x00 after
607 // a null-terminated string, this version will only add a null byte
608 // if the label parameter did not end in a null byte, or if no label
609 // is present.
610 if((label == NULL)
611    || (label->size == 0)
612    || (label->buffer[label->size - 1] != 0))
613     CryptDigestUpdateInt(&hashState, 1, 0);
614 // Add PartyUInfo
615 if(partyUInfo != NULL)
616     CryptDigestUpdate2B(&hashState, partyUInfo);
617
618 // Add PartyVInfo
619 if(partyVInfo != NULL)
620     CryptDigestUpdate2B(&hashState, partyVInfo);
621
622 // Compute Hash. hLen was changed to be the smaller of bytes or hLen
623 // at the start of each iteration.
624 CryptHashEnd(&hashState, hLen, stream);
625 }
626 // Mask off bits if the required bits is not a multiple of byte size
627 if((sizeInBits % 8) != 0)
628     keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
629
630 return (UINT16)((sizeInBits + 7) / 8);
631 }
```

10.2.14 CryptPrime.c

10.2.14.1 Introduction

This file contains the code for prime validation.

```

1  #include "Tpm.h"
2  #include "CryptPrime_fp.h"
3
4  //#define CPRI_PRIME
5  //#include "PrimeTable.h"
6
7  #include "CryptPrimeSieve_fp.h"
8
9  extern const uint32_t      s_LastPrimeInTable;
10 extern const uint32_t      s_PrimeTableSize;
11 extern const uint32_t      s_PrimesInTable;
12 extern const unsigned char s_PrimeTable[];
13 extern bigConst            s_CompositeOfSmallPrimes;

```

10.2.14.2 Functions

10.2.14.2.1 Root2()

This finds $\text{ceil}(\sqrt{n})$ to use as a stopping point for searching the prime table.

```

14 static uint32_t
15 Root2(
16     uint32_t      n
17 )
18 {
19     int32_t      last = (int32_t) (n >> 2);
20     int32_t      next = (int32_t) (n >> 1);
21     int32_t      diff;
22     int32_t      stop = 10;
23
24     // get a starting point
25     for(; next != 0; last >>= 1, next >>= 2);
26     last++;
27     do
28     {
29         next = (last + (n / last)) >> 1;
30         diff = next - last;
31         last = next;
32         if(stop-- == 0)
33             FAIL(FATAL_ERROR_INTERNAL);
34     } while(diff < -1 || diff > 1);
35     if((n / next) > (unsigned)next)
36         next++;
37     pAssert(next != 0);
38     pAssert(((n / next) <= (unsigned)next) && (n / (next + 1) < (unsigned)next));
39     return next;
40 }

```

10.2.14.2.2 IsPrimeInt()

This will do a test of a word of up to 32-bits in size.

```

41 BOOL
42 IsPrimeInt(

```

```

43     uint32_t          n
44     )
45 {
46     uint32_t          i;
47     uint32_t          stop;
48     if(n < 3 || ((n & 1) == 0))
49         return (n == 2);
50     if(n <= s_LastPrimeInTable)
51     {
52         n >>= 1;
53         return ((s_PrimeTable[n >> 3] >> (n & 7)) & 1);
54     }
55     // Need to search
56     stop = Root2(n) >> 1;
57     // starting at 1 is equivalent to staring at (1 << 1) + 1 = 3
58     for(i = 1; i < stop; i++)
59     {
60         if((s_PrimeTable[i >> 3] >> (i & 7)) & 1)
61             // see if this prime evenly divides the number
62             if((n % ((i << 1) + 1)) == 0)
63                 return FALSE;
64     }
65     return TRUE;
66 }

```

10.2.14.2.3 BnIsProbablyPrime()

This function is used when the key sieve is not implemented. This function Will try to eliminate some of the obvious things before going on to perform MillerRabin() as a final verification of primeness.

```

67  BOOL
68  BnIsProbablyPrime(
69      bigNum          prime,          // IN:
70      RAND_STATE      *rand           // IN: the random state just
71                                      // in case Miller-Rabin is required
72  )
73  {
74      #if RADIX BITS > 32
75          if(BnUnsignedCmpWord(prime, UINT32_MAX) <= 0)
76      #else
77          if(BnGetSize(prime) == 1)
78      #endif
79          return IsPrimeInt((uint32_t)prime->d[0]);
80
81          if(BnIsEven(prime))
82              return FALSE;
83          if(BnUnsignedCmpWord(prime, s_LastPrimeInTable) <= 0)
84          {
85              crypt_uword_t    temp = prime->d[0] >> 1;
86              return ((s_PrimeTable[temp >> 3] >> (temp & 7)) & 1);
87          }
88          {
89              BN_VAR(n, LARGEST_NUMBER_BITS);
90              BnGcd(n, prime, s_CompositeOfSmallPrimes);
91              if(!BnEqualWord(n, 1))
92                  return FALSE;
93          }
94          return MillerRabin(prime, rand);
95  }

```

10.2.14.2.4 MillerRabinRounds()

Function returns the number of Miller-Rabin rounds necessary to give an error probability equal to the security strength of the prime. These values are from FIPS 186-3.

```

96  UINT32
97  MillerRabinRounds(
98      UINT32          bits          // IN: Number of bits in the RSA prime
99      )
100 {
101     if(bits < 511) return 8;        // don't really expect this
102     if(bits < 1536) return 5;      // for 512 and 1K primes
103     return 4;                      // for 3K public modulus and greater
104 }
```

10.2.14.2.5 MillerRabin()

This function performs a Miller-Rabin test from FIPS 186-3. It does *iterations* trials on the number. In all likelihood, if the number is not prime, the first test fails.

Return Value	Meaning
TRUE(1)	probably prime
FALSE(0)	composite

```

105 BOOL
106 MillerRabin(
107     bigNum          bnW,
108     RAND_STATE      *rand
109 )
110 {
111     BN_MAX(bnWml);
112     BN_PRIME(bnM);
113     BN_PRIME(bnB);
114     BN_PRIME(bnZ);
115     BOOL          ret = FALSE;    // Assumed composite for easy exit
116     unsigned int  a;
117     unsigned int  j;
118     int            wLen;
119     int            i;
120     int            iterations = MillerRabinRounds(BnSizeInBits(bnW));
121     //
122     INSTRUMENT_INC(MillerRabinTrials[PrimeIndex]);
123
124     pAssert(bnW->size > 1);
125     // Let a be the largest integer such that 2^a divides w1.
126     BnSubWord(bnWml, bnW, 1);
127     pAssert(bnWml->size != 0);
128
129     // Since w is odd (w-1) is even so start at bit number 1 rather than 0
130     // Get the number of bits in bnWml so that it doesn't have to be recomputed
131     // on each iteration.
132     i = (int)(bnWml->size * RADIX_BITS);
133     // Now find the largest power of 2 that divides w1
134     for(a = 1;
135         (a < (bnWml->size * RADIX_BITS)) &&
136         (BnTestBit(bnWml, a) == 0);
137         a++);
138     // 2. m = (w1) / 2^a
139     BnShiftRight(bnM, bnWml, a);
140     // 3. wlen = len(w).
141     wLen = BnSizeInBits(bnW);
```



```

142 // 4. For i = 1 to iterations do
143   for(i = 0; i < iterations; i++)
144   {
145       // 4.1 Obtain a string b of wlen bits from an RBG.
146       // Ensure that 1 < b < w1.
147       // 4.2 If ((b <= 1) or (b >= w1)), then go to step 4.1.
148       while(BnGetRandomBits(bnB, wlen, rand) && ((BnUnsignedCmpWord(bnB, 1) <= 0)
149           || (BnUnsignedCmp(bnB, bnWm1) >= 0)));
150       if(g_inFailureMode)
151           return FALSE;
152
153       // 4.3 z = b^m mod w.
154       // if ModExp fails, then say this is not
155       // prime and bail out.
156       BnModExp(bnZ, bnB, bnM, bnW);
157
158       // 4.4 If ((z == 1) or (z = w == 1)), then go to step 4.7.
159       if((BnUnsignedCmpWord(bnZ, 1) == 0)
160         || (BnUnsignedCmp(bnZ, bnWm1) == 0))
161           goto step4point7;
162       // 4.5 For j = 1 to a - 1 do.
163       for(j = 1; j < a; j++)
164       {
165           // 4.5.1 z = z^2 mod w.
166           BnModMult(bnZ, bnZ, bnZ, bnW);
167           // 4.5.2 If (z = w1), then go to step 4.7.
168           if(BnUnsignedCmp(bnZ, bnWm1) == 0)
169               goto step4point7;
170           // 4.5.3 If (z = 1), then go to step 4.6.
171           if(BnEqualWord(bnZ, 1))
172               goto step4point6;
173       }
174       // 4.6 Return COMPOSITE.
175   step4point6:
176       INSTRUMENT_INC(failedAtIteration[i]);
177       goto end;
178       // 4.7 Continue. Comment: Increment i for the do-loop in step 4.
179   step4point7:
180       continue;
181   }
182 // 5. Return PROBABLY PRIME
183 ret = TRUE;
184 end:
185 return ret;
186 }
187 #if ALG_RSA

```

10.2.14.2.6 RsaCheckPrime()

This will check to see if a number is prime and appropriate for an RSA prime.

This has different functionality based on whether we are using key sieving or not. If not, the number checked to see if it is divisible by the public exponent, then the number is adjusted either up or down in order to make it a better candidate. It is then checked for being probably prime.

If sieving is used, the number is used to root a sieving process.

```

188 TPM_RC
189 RsaCheckPrime(
190     bigNum          prime,
191     UINT32          exponent,
192     RAND_STATE      *rand
193 )
194 {

```

```

195  #if !RSA_KEY_SIEVE
196      TPM_RC          retVal = TPM_RC_SUCCESS;
197      UINT32          modE = BnModWord(prime, exponent);
198
199      NOT_REFERENCED(rand);
200
201      if(modE == 0)
202          // evenly divisible so add two keeping the number odd
203          BnAddWord(prime, prime, 2);
204      // want 0 != (p - 1) mod e
205      // which is 1 != p mod e
206      else if(modE == 1)
207          // subtract 2 keeping number odd and insuring that
208          // 0 != (p - 1) mod e
209          BnSubWord(prime, prime, 2);
210
211      if(BnIsProbablyPrime(prime, rand) == 0)
212          ERROR_RETURN(g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_VALUE);
213  Exit:
214      return retVal;
215  #else
216      return PrimeSelectWithSieve(prime, exponent, rand);
217  #endif
218  }

```

10.2.14.2.7 RsaAdjustPrimeCandidate()

For this math, we assume that the RSA numbers are fixed-point numbers with the decimal point to the **left** of the most significant bit. This approach helps make it clear what is happening with the MSb of the values. The two RSA primes have to be large enough so that their product will be a number with the necessary number of significant bits. For example, we want to be able to multiply two 1024-bit numbers to produce a number with 2028 significant bits. If we accept any 1024-bit prime that has its MSb set, then it is possible to produce a product that does not have the MSb SET. For example, if we use tiny keys of 16 bits and have two 8-bit *primes* of 0x80, then the public key would be 0x4000 which is only 15-bits. So, what we need to do is make sure that each of the primes is large enough so that the product of the primes is twice as large as each prime. A little arithmetic will show that the only way to do this is to make sure that each of the primes is no less than $\sqrt{2}/2$. That's what this function does. This function adjusts the candidate prime so that it is odd and $\geq \sqrt{2}/2$. This allows the product of these two numbers to be .5, which, in fixed point notation means that the most significant bit is 1. For this routine, the $\sqrt{2}/2$ (0.7071067811865475) approximated with 0xB505 which is, in fixed point, 0.7071075439453125 or an error of 0.000108%. Just setting the upper two bits would give a value > 0.75 which is an error of $> 6\%$. Given the amount of time all the other computations take, reducing the error is not much of a cost, but it isn't totally required either.

This function can be replaced with a function that just sets the two most significant bits of each prime candidate without introducing any computational issues.

```

219  LIB_EXPORT void
220  RsaAdjustPrimeCandidate(
221      bigNum          prime
222  )
223  {
224      UINT32          msw;
225      UINT32          adjusted;
226
227      // If the radix is 32, the compiler should turn this into a simple assignment
228      msw = prime->d[prime->size - 1] >> ((RADIX_BITS == 64) ? 32 : 0);
229      // Multiplying 0xff...f by 0x4AFB gives 0xff...f - 0xB5050...0
230      adjusted = (msw >> 16) * 0x4AFB;
231      adjusted += ((msw & 0xFFFF) * 0x4AFB) >> 16;
232      adjusted += 0xB5050000UL;
233  #if RADIX_BITS == 64

```

```

234     // Save the low-order 32 bits
235     prime->d[prime->size - 1] &= 0xFFFFFFFFUL;
236     // replace the upper 32-bits
237     prime->d[prime->size - 1] |= ((crypt_uword_t)adjusted << 32);
238 #else
239     prime->d[prime->size - 1] = (crypt_uword_t)adjusted;
240 #endif
241     // make sure the number is odd
242     prime->d[0] |= 1;
243 }

```

10.2.14.2.8 BnGeneratePrimeForRSA()

Function to generate a prime of the desired size with the proper attributes for an RSA prime.

```

244 TPM_RC
245 BnGeneratePrimeForRSA(
246     bigNum          prime,           // IN/OUT: points to the BN that will get the
247                                     // random value
248     UINT32          bits,           // IN: number of bits to get
249     UINT32          exponent,       // IN: the exponent
250     RAND_STATE      *rand,         // IN: the random state
251 )
252 {
253     BOOL            found = FALSE;
254     //
255     // Make sure that the prime is large enough
256     pAssert(prime->allocated >= BITS_TO_CRYPT_WORDS(bits));
257     // Only try to handle specific sizes of keys in order to save overhead
258     pAssert((bits % 32) == 0);
259
260     prime->size = BITS_TO_CRYPT_WORDS(bits);
261
262     while(!found)
263     {
264         // The change below is to make sure that all keys that are generated from the same
265         // seed value will be the same regardless of the endianness or word size of the CPU.
266         // DRBG_Generate(rand, (BYTE *)prime->d, (UINT16)BITS_TO_BYTES(bits)); // old
267         // if(g_inFailureMode) // old
268         if(!BnGetRandomBits(prime, bits, rand)) // new
269             return TPM_RC_FAILURE;
270         RsaAdjustPrimeCandidate(prime);
271         found = RsaCheckPrime(prime, exponent, rand) == TPM_RC_SUCCESS;
272     }
273     return TPM_RC_SUCCESS;
274 }
275 #endif // ALG_RSA

```

10.2.15 CryptPrimeSieve.c

10.2.15.1 Includes and defines

```

1  #include "Tpm.h"
2
3  #if RSA_KEY_SIEVE
4
5  #include "CryptPrimeSieve_fp.h"

```

This determines the number of bits in the largest sieve field.

```

6  #define MAX_FIELD_SIZE 2048
7
8  extern const uint32_t    s_LastPrimeInTable;
9  extern const uint32_t    s_PrimeTableSize;
10 extern const uint32_t    s_PrimesInTable;
11 extern const unsigned char s_PrimeTable[];

```

This table is set of prime markers. Each entry is the prime value for the $((n + 1) * 1024)$ prime. That is, the entry in `s_PrimeMarkers[1]` is the value for the 2,048th prime. This is used in the `PrimeSieve()` to adjust the limit for the prime search. When processing smaller prime candidates, fewer primes are checked directly before going to Miller-Rabin. As the prime grows, it is worth spending more time eliminating primes as, a) the density is lower, and b) the cost of Miller-Rabin is higher.

```

12 const uint32_t    s_PrimeMarkersCount = 6;
13 const uint32_t    s_PrimeMarkers[] = {
14     8167, 17881, 28183, 38891, 49871, 60961 };
15 uint32_t    primeLimit;

```

10.2.15.2 Functions

10.2.15.2.1 RsaAdjustPrimeLimit()

This used during the sieve process. The iterator for getting the next prime (`RsaNextPrime()`) will return primes until it hits the limit (*primeLimit*) set up by this function. This causes the sieve process to stop when an appropriate number of primes have been sieved.

```

16 LIB_EXPORT void
17 RsaAdjustPrimeLimit(
18     uint32_t    requestedPrimes
19 )
20 {
21     if(requestedPrimes == 0 || requestedPrimes > s_PrimesInTable)
22         requestedPrimes = s_PrimesInTable;
23     requestedPrimes = (requestedPrimes - 1) / 1024;
24     if(requestedPrimes < s_PrimeMarkersCount)
25         primeLimit = s_PrimeMarkers[requestedPrimes];
26     else
27         primeLimit = s_LastPrimeInTable;
28     primeLimit >>= 1;
29
30 }

```

10.2.15.2.2 RsaNextPrime()

This the iterator used during the sieve process. The input is the last prime returned (or any starting point) and the output is the next higher prime. The function returns 0 when the *primeLimit* is reached.

```

31  LIB_EXPORT uint32_t
32  RsaNextPrime(
33      uint32_t    lastPrime
34  )
35  {
36      if(lastPrime == 0)
37          return 0;
38      lastPrime >>= 1;
39      for(lastPrime += 1; lastPrime <= primeLimit; lastPrime++)
40      {
41          if(((s_PrimeTable[lastPrime >> 3] >> (lastPrime & 0x7)) & 1) == 1)
42              return ((lastPrime << 1) + 1);
43      }
44      return 0;
45  }

```

This table contains a previously sieved table. It has the bits for 3, 5, and 7 removed. Because of the factors, it needs to be aligned to 105 and has a repeat of 105.

```

46  const BYTE    seedValues[] = {
47      0x16, 0x29, 0xcb, 0xa4, 0x65, 0xda, 0x30, 0x6c,
48      0x99, 0x96, 0x4c, 0x53, 0xa2, 0x2d, 0x52, 0x96,
49      0x49, 0xcb, 0xb4, 0x61, 0xd8, 0x32, 0x2d, 0x99,
50      0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
51      0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
52      0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
53      0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
54      0x5a, 0xa6, 0x0d, 0xc3, 0x96, 0x69, 0xc9, 0x34,
55      0x25, 0xda, 0x22, 0x65, 0x99, 0xb4, 0x4c, 0x1b,
56      0x86, 0x2d, 0xd3, 0x92, 0x69, 0x4a, 0xb4, 0x45,
57      0xca, 0x32, 0x69, 0x99, 0x36, 0x0c, 0x5b, 0xa6,
58      0x25, 0xd3, 0x94, 0x68, 0x8b, 0x94, 0x65, 0xd2,
59      0x32, 0x6d, 0x18, 0xb6, 0x4c, 0x4b, 0xa6, 0x29,
60      0xd1};
61
62  #define USE_NIBBLE
63
64  #ifndef USE_NIBBLE
65  static const BYTE bitsInByte[256] = {
66      0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
67      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
68      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
69      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
70      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
71      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
72      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
73      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
74      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
75      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
76      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
77      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
78      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
79      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
80      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
81      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
82      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
83      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
84      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
85      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
86      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
87      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
88      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
89      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
90      0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
91      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,

```

```

92      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
93      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
94      0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
95      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
96      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
97      0x05, 0x06, 0x06, 0x07, 0x06, 0x07, 0x07, 0x08
98  };
99  #define BitsInByte(x)    bitsInByte[(unsigned char)x]
100 #else
101  const BYTE bitsInNibble[16] = {
102      0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
103      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04};
104  #define BitsInByte(x)
105      (bitsInNibble[(unsigned char)(x) & 0xf]
106      + bitsInNibble[((unsigned char)(x) >> 4) & 0xf])
107  #endif

```

10.2.15.2.3 BitsInArray()

This function counts the number of bits set in an array of bytes.

```

108  static int
109  BitsInArray(
110      const unsigned char    *a,           // IN: A pointer to an array of bytes
111      unsigned int           aSize         // IN: the number of bytes to sum
112  )
113  {
114      int    j = 0;
115      for(; aSize; a++, aSize--)
116          j += BitsInByte(*a);
117      return j;
118  }

```

10.2.15.2.4 FindNthSetBit()

This function finds the *n*th SET bit in a bit array. The *n* parameter is between 1 and the number of bits in the array (always a multiple of 8). If called when the array does not have *n* bits set, it will return -1

Return Value	Meaning
<0	no bit is set or no bit with the requested number is set
>=0	the number of the bit in the array that is the <i>n</i> th set

```

119  LIB_EXPORT int
120  FindNthSetBit(
121      const UINT16    aSize,           // IN: the size of the array to check
122      const BYTE      *a,             // IN: the array to check
123      const UINT32    n               // IN, the number of the SET bit
124  )
125  {
126      UINT16    i;
127      int       retValue;
128      UINT32    sum = 0;
129      BYTE      sel;
130
131      //find the bit
132      for(i = 0; (i < (int)aSize) && (sum < n); i++)
133          sum += BitsInByte(a[i]);
134      i--;
135      // The chosen bit is in the byte that was just accessed
136      // Compute the offset to the start of that byte
137      retValue = i * 8 - 1;

```

```

138     sel = a[i];
139     // Subtract the bits in the last byte added.
140     sum -= BitsInByte(sel);
141     // Now process the byte, one bit at a time.
142     for(; (sel != 0) && (sum != n); retValue++, sel = sel >> 1)
143         sum += (sel & 1) != 0;
144     return (sum == n) ? retValue : -1;
145 }
146 typedef struct
147 {
148     UINT16    prime;
149     UINT16    count;
150 } SIEVE_MARKS;
151
152 const SIEVE_MARKS sieveMarks[5] = {
153     {31, 7}, {73, 5}, {241, 4}, {1621, 3}, {UINT16_MAX, 2}};

```

10.2.15.2.5 PrimeSieve()

This function does a prime sieve over the input *field* which has as its starting address the value in *bnN*. Since this initializes the Sieve using a precomputed field with the bits associated with 3, 5 and 7 already turned off, the value of *pnN* may need to be adjusted by a few counts to allow the precomputed field to be used without modification.

To get better performance, one could address the issue of developing the composite numbers. When the size of the prime gets large, the time for doing the divisions goes up, noticeably. It could be better to develop larger composite numbers even if they need to be *bigNum*'s themselves. The object would be to reduce the number of times that the large prime is divided into a few large divides and then use smaller divides to get to the final 16 bit (or smaller) remainders.

```

154 LIB_EXPORT UINT32
155 PrimeSieve(
156     bigNum    bnN,           // IN/OUT: number to sieve
157     UINT32    fieldSize,    // IN: size of the field area in bytes
158     BYTE      *field        // IN: field
159 )
160 {
161     UINT32    i;
162     UINT32    j;
163     UINT32    fieldBits = fieldSize * 8;
164     UINT32    r;
165     BYTE      *pField;
166     INT32     iter;
167     adjust;
168     UINT32    mark = 0;
169     UINT32    count = sieveMarks[0].count;
170     UINT32    stop = sieveMarks[0].prime;
171     UINT32    composite;
172     UINT32    pList[8];
173     UINT32    next;
174
175     pAssert(field != NULL && bnN != NULL);
176
177     // If the remainder is odd, then subtracting the value will give an even number,
178     // but we want an odd number, so subtract the 105+rem. Otherwise, just subtract
179     // the even remainder.
180     adjust = (UINT32)BnModWord(bnN, 105);
181     if(adjust & 1)
182         adjust += 105;
183
184     // Adjust the input number so that it points to the first number in a
185     // aligned field.
186     BnSubWord(bnN, bnN, adjust);

```



```

187 //    pAssert(BnModWord(bnN, 105) == 0);
188 pField = field;
189 for(i = fieldSize; i >= sizeof(seedValues);
190     pField += sizeof(seedValues), i -= sizeof(seedValues))
191 {
192     memcpy(pField, seedValues, sizeof(seedValues));
193 }
194 if(i != 0)
195     memcpy(pField, seedValues, i);
196
197 // Cycle through the primes, clearing bits
198 // Have already done 3, 5, and 7
199 iter = 7;
200
201 #define NEXT_PRIME(iter)    (iter = RsaNextPrime(iter))
202 // Get the next N primes where N is determined by the mark in the sieveMarks
203 while((composite = NEXT_PRIME(iter)) != 0)
204 {
205     next = 0;
206     i = count;
207     pList[i--] = composite;
208     for(; i > 0; i--)
209     {
210         next = NEXT_PRIME(iter);
211         pList[i] = next;
212         if(next != 0)
213             composite *= next;
214     }
215     // Get the remainder when dividing the base field address
216     // by the composite
217     composite = (UINT32)BnModWord(bnN, composite);
218     // 'composite' is divisible by the composite components. for each of the
219     // composite components, divide 'composite'. That remainder (r) is used to
220     // pick a starting point for clearing the array. The stride is equal to the
221     // composite component. Note, the field only contains odd numbers. If the
222     // field were expanded to contain all numbers, then half of the bits would
223     // have already been cleared. We can save the trouble of clearing them a
224     // second time by having a stride of 2*next. Or we can take all of the even
225     // numbers out of the field and use a stride of 'next'
226     for(i = count; i > 0; i--)
227     {
228         next = pList[i];
229         if(next == 0)
230             goto done;
231         r = composite % next;
232         // these computations deal with the fact that we have picked a field-sized
233         // range that is aligned to a 105 count boundary. The problem is, this field
234         // only contains odd numbers. If we take our prime guess and walk through all
235         // the numbers using that prime as the 'stride', then every other 'stride' is
236         // going to be an even number. So, we are actually counting by 2 * the stride
237         // We want the count to start on an odd number at the start of our field. That
238         // is, we want to assume that we have counted up to the edge of the field by
239         // the 'stride' and now we are going to start flipping bits in the field as we
240         // continue to count up by 'stride'. If we take the base of our field and
241         // divide by the stride, we find out how much we find out how short the last
242         // count was from reaching the edge of the bit field. Say we get a quotient of
243         // 3 and remainder of 1. This means that after 3 strides, we are 1 short of
244         // the start of the field and the next stride will either land within the
245         // field or step completely over it. The confounding factor is that our field
246         // only contains odd numbers and our stride is actually 2 * stride. If the
247         // quotient is even, then that means that when we add 2 * stride, we are going
248         // to hit another even number. So, we have to know if we need to back off
249         // by 1 stride before we start counting by 2 * stride.
250         // We can tell from the remainder whether we are on an even or odd
251         // stride when we hit the beginning of the table. If we are on an odd stride
252         // (r & 1), we would start half a stride in (next - r)/2. If we are on an

```

```

253     // even stride, we need 0.5 strides (next - r/2) because the table only has
254     // odd numbers. If the remainder happens to be zero, then the start of the
255     // table is on stride so no adjustment is necessary.
256     if(r & 1)          j = (next - r) / 2;
257     else if(r == 0)    j = 0;
258     else               j = next - (r / 2);
259     for(; j < fieldBits; j += next)
260         ClearBit(j, field, fieldSize);
261 }
262 if(next >= stop)
263 {
264     mark++;
265     count = sieveMarks[mark].count;
266     stop = sieveMarks[mark].prime;
267 }
268 }
269 done:
270 INSTRUMENT_INC(totalFieldsSieved[PrimeIndex]);
271 i = BitsInArray(field, fieldSize);
272 INSTRUMENT_ADD(bitsInFieldAfterSieve[PrimeIndex], i);
273 INSTRUMENT_ADD(emptyFieldsSieved[PrimeIndex], (i == 0));
274 return i;
275 }
276 #ifdef SIEVE_DEBUG
277 static uint32_t fieldSize = 210;

```

10.2.15.2.6 SetFieldSize()

Function to set the field size used for prime generation. Used for tuning.

```

278 LIB_EXPORT uint32_t
279 SetFieldSize(
280     uint32_t      newFieldSize
281 )
282 {
283     if(newFieldSize == 0 || newFieldSize > MAX_FIELD_SIZE)
284         fieldSize = MAX_FIELD_SIZE;
285     else
286         fieldSize = newFieldSize;
287     return fieldSize;
288 }
289 #endif // SIEVE_DEBUG

```

10.2.15.2.7 PrimeSelectWithSieve()

This function will sieve the field around the input prime candidate. If the sieve field is not empty, one of the one bits in the field is chosen for testing with Miller-Rabin. If the value is prime, *pnP* is updated with this value and the function returns success. If this value is not prime, another pseudo-random candidate is chosen and tested. This process repeats until all values in the field have been checked. If all bits in the field have been checked and none is prime, the function returns FALSE and a new random value needs to be chosen.

Error Return	Meaning
TPM_RC_FAILURE	TPM in failure mode, probably due to entropy source
TPM_RC_NO_RESULT	candidate is not prime and couldn't find an alternative in the field

```

290 LIB_EXPORT TPM_RC
291 PrimeSelectWithSieve(
292     bigNum      candidate,      // IN/OUT: The candidate to filter

```

```

293     UINT32      e,                // IN: the exponent
294     RAND_STATE  *rand            // IN: the random number generator state
295 )
296 {
297     BYTE        field[MAX_FIELD_SIZE];
298     UINT32      first;
299     UINT32      ones;
300     INT32       chosen;
301     BN_PRIME(test);
302     UINT32      modE;
303 #ifndef SIEVE_DEBUG
304     UINT32      fieldSize = MAX_FIELD_SIZE;
305 #endif
306     UINT32      primeSize;
307 //
308 // Adjust the field size and prime table list to fit the size of the prime
309 // being tested. This is done to try to optimize the trade-off between the
310 // dividing done for sieving and the time for Miller-Rabin. When the size
311 // of the prime is large, the cost of Miller-Rabin is fairly high, as is the
312 // cost of the sieving. However, the time for Miller-Rabin goes up considerably
313 // faster than the cost of dividing by a number of primes.
314 primeSize = BnSizeInBits(candidate);
315
316 if(primeSize <= 512)
317 {
318     RsaAdjustPrimeLimit(1024); // Use just the first 1024 primes
319 }
320 else if(primeSize <= 1024)
321 {
322     RsaAdjustPrimeLimit(4096); // Use just the first 4K primes
323 }
324 else
325 {
326     RsaAdjustPrimeLimit(0);    // Use all available
327 }
328 // Save the low-order word to use as a search generator and make sure that
329 // it has some interesting range to it
330 first = (UINT32)(candidate->d[0] | 0x80000000);
331
332 // Sieve the field
333 ones = PrimeSieve(candidate, fieldSize, field);
334 pAssert(ones > 0 && ones < (fieldSize * 8));
335 for(; ones > 0; ones--)
336 {
337     // Decide which bit to look at and find its offset
338     chosen = FindNthSetBit((UINT16)fieldSize, field, ((first % ones) + 1));
339
340     if((chosen < 0) || (chosen >= (INT32)(fieldSize * 8)))
341         FAIL(FATAL_ERROR_INTERNAL);
342
343     // Set this as the trial prime
344     BnAddWord(test, candidate, (crypt_uword_t)(chosen * 2));
345
346     // The exponent might not have been one of the tested primes so
347     // make sure that it isn't divisible and make sure that 0 != (p-1) mod e
348     // Note: This is the same as 1 != p mod e
349     modE = (UINT32)BnModWord(test, e);
350     if((modE != 0) && (modE != 1) && MillerRabin(test, rand))
351     {
352         BnCopy(candidate, test);
353         return TPM_RC_SUCCESS;
354     }
355     // Clear the bit just tested
356     ClearBit(chosen, field, fieldSize);
357 }
358 // Ran out of bits and couldn't find a prime in this field

```

```

359     INSTRUMENT_INC(noPrimeFields[PrimeIndex]);
360     return (g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_NO_RESULT);
361 }
362 #if RSA_INSTRUMENT
363 static char      a[256];

```

10.2.15.2.8 PrintTuple()

```

364 char *
365 PrintTuple(
366     UINT32      *i
367 )
368 {
369     sprintf(a, "{%d, %d, %d}", i[0], i[1], i[2]);
370     return a;
371 }
372 #define CLEAR_VALUE(x)      memset(x, 0, sizeof(x))

```

10.2.15.2.9 RsaSimulationEnd()

```

373 void
374 RsaSimulationEnd(
375     void
376 )
377 {
378     int      i;
379     UINT32    averages[3];
380     UINT32    nonFirst = 0;
381     if((PrimeCounts[0] + PrimeCounts[1] + PrimeCounts[2]) != 0)
382     {
383         printf("Primes generated = %s\n", PrintTuple(PrimeCounts));
384         printf("Fields sieved = %s\n", PrintTuple(totalFieldsSieved));
385         printf("Fields with no primes = %s\n", PrintTuple(noPrimeFields));
386         printf("Primes checked with Miller-Rabin = %s\n",
387             PrintTuple(MillerRabinTrials));
388         for(i = 0; i < 3; i++)
389             averages[i] = (totalFieldsSieved[i]
390                 != 0 ? bitsInFieldAfterSieve[i] / totalFieldsSieved[i]
391                 : 0);
392         printf("Average candidates in field %s\n", PrintTuple(averages));
393         for(i = 1; i < (sizeof(failedAtIteration) / sizeof(failedAtIteration[0]));
394             i++)
395             nonFirst += failedAtIteration[i];
396         printf("Miller-Rabin failures not in first round = %d\n", nonFirst);
397     }
398     CLEAR_VALUE(PrimeCounts);
399     CLEAR_VALUE(totalFieldsSieved);
400     CLEAR_VALUE(noPrimeFields);
401     CLEAR_VALUE(MillerRabinTrials);
402     CLEAR_VALUE(bitsInFieldAfterSieve);
403 }
404

```

10.2.15.2.10 GetSieveStats()

```

405 LIB_EXPORT void
406 GetSieveStats(
407     uint32_t      *trials,
408     uint32_t      *emptyFields,
409     uint32_t      *averageBits
410 )
411 {

```

```

412     uint32_t      totalBits;
413     uint32_t      fields;
414     *trials = MillerRabinTrials[0] + MillerRabinTrials[1] + MillerRabinTrials[2];
415     *emptyFields = noPrimeFields[0] + noPrimeFields[1] + noPrimeFields[2];
416     fields = totalFieldsSieved[0] + totalFieldsSieved[1]
417             + totalFieldsSieved[2];
418     totalBits = bitsInFieldAfterSieve[0] + bitsInFieldAfterSieve[1]
419             + bitsInFieldAfterSieve[2];
420     if(fields != 0)
421         *averageBits = totalBits / fields;
422     else
423         *averageBits = 0;
424     CLEAR_VALUE(PrimeCounts);
425     CLEAR_VALUE(totalFieldsSieved);
426     CLEAR_VALUE(noPrimeFields);
427     CLEAR_VALUE(MillerRabinTrials);
428     CLEAR_VALUE(bitsInFieldAfterSieve);
429 }
430 #endif
431
432 #endif // RSA_KEY_SIEVE
433
434 #if !RSA_INSTRUMENT
435

```

10.2.15.2.11 RsaSimulationEnd()

Stub for call when not doing instrumentation.

```

436 void
437 RsaSimulationEnd(
438     void
439 )
440 {
441     return;
442 }
443 #endif

```

10.2.16 CryptRand.c

10.2.16.1 Introduction

This file implements a DRBG with a behavior according to SP800-90A using a block cypher. This is also compliant to ISO/IEC 18031:2011(E) C.3.2.

A state structure is created for use by TPM.lib and functions within the CryptoEngine() may use their own state structures when they need to have deterministic values.

A debug mode is available that allows the random numbers generated for TPM.lib to be repeated during runs of the simulator. The switch for it is in TpmBuildSwitches.h. It is USE_DEBUG_RNG.

This is the implementation layer of CTR DRBG mechanism as defined in SP800-90A and the functions are organized as closely as practical to the organization in SP800-90A. It is intended to be compiled as a separate module that is linked with a secure application so that both reside inside the same boundary [SP 800-90A 8.5]. The secure application in particular manages the accesses protected storage for the state of the DRBG instantiations, and supplies the implementation functions here with a valid pointer to the working state of the given instantiations (as a DRBG_STATE structure).

This DRBG mechanism implementation does not support prediction resistance. Thus *prediction_resistance_flag* is omitted from *Instantiate_function()*, *Reseed_function()*, *Generate_function()* argument lists [SP 800-90A 9.1, 9.2, 9.3], as well as from the working state data structure DRBG_STATE [SP 800-90A 9.1].

This DRBG mechanism implementation always uses the highest security strength of available in the block ciphers. Thus *requested_security_strength* parameter is omitted from *Instantiate_function()* and *Generate_function()* argument lists [SP 800-90A 9.1, 9.2, 9.3], as well as from the working state data structure DRBG_STATE [SP 800-90A 9.1].

Internal functions (ones without Crypt prefix) expect validated arguments and therefore use assertions instead of runtime parameter checks and mostly return void instead of a status value.

```

1  #include "Tpm.h"

Pull in the test vector definitions and define the space

2  #include "PRNG_TestVectors.h"
3
4  const BYTE DRBG_NistTestVector_Entropy[] = {DRBG_TEST_INITIATE_ENTROPY};
5  const BYTE DRBG_NistTestVector_GeneratedInterm[] =
6      {DRBG_TEST_GENERATED_INTERM};
7
8  const BYTE DRBG_NistTestVector_EntropyReseed[] =
9      {DRBG_TEST_RESEED_ENTROPY};
10 const BYTE DRBG_NistTestVector_Generated[] = {DRBG_TEST_GENERATED};

```

10.2.16.2 Derivation Functions

10.2.16.2.1 Description

The functions in this section are used to reduce the personalization input values to make them usable as input for reseeding and instantiation. The overall behavior is intended to produce the same results as described in SP800-90A, section 10.4.2 **Derivation Function Using a Block Cipher Algorithm (Block_Cipher_df)**. The code is broken into several subroutines to deal with the fact that the data used for personalization may come in several separate blocks such as a Template hash and a proof value and a primary seed.

10.2.16.2.2 Derivation Function Defines and Structures

```

11 #define      DF_COUNT (DRBG_KEY_SIZE_WORDS / DRBG_IV_SIZE_WORDS + 1)
12 #if DRBG_KEY_SIZE_BITS != 128 && DRBG_KEY_SIZE_BITS != 256
13 #   error "CryptRand.c only written for AES with 128- or 256-bit keys."
14 #endif
15
16 typedef struct
17 {
18     DRBG_KEY_SCHEDULE    keySchedule;
19     DRBG_IV              iv[DF_COUNT];
20     DRBG_IV              out1;
21     DRBG_IV              buf;
22     int                  contents;
23 } DF_STATE, *PDF_STATE;

```

10.2.16.2.3 DfCompute()

This function does the incremental update of the derivation function state. It encrypts the *iv* value and XOR's the results into each of the blocks of the output. This is equivalent to processing all of input data for each output block.

```

24 static void
25 DfCompute(
26     PDF_STATE      dfState
27 )
28 {
29     int            i;
30     int            iv;
31     crypt_ushort_t *pIv;
32     crypt_ushort_t temp[DRBG_IV_SIZE_WORDS] = {0};
33     //
34     for(iv = 0; iv < DF_COUNT; iv++)
35     {
36         pIv = (crypt_ushort_t *)&dfState->iv[iv].words[0];
37         for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
38         {
39             temp[i] ^= pIv[i] ^ dfState->buf.words[i];
40         }
41         DRBG_ENCRYPT(&dfState->keySchedule, &temp, pIv);
42     }
43     for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
44         dfState->buf.words[i] = 0;
45     dfState->contents = 0;
46 }

```

10.2.16.2.4 DfStart()

This initializes the output blocks with an encrypted counter value and initializes the key schedule.

```

47 static void
48 DfStart(
49     PDF_STATE      dfState,
50     uint32_t       inputLength
51 )
52 {
53     BYTE           init[8];
54     int            i;
55     UINT32         drbgSeedSize = sizeof(DRBG_SEED);
56
57     const BYTE dfKey[DRBG_KEY_SIZE_BYTES] = {
58         0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,

```



```

59     0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
60     #if DRBG_KEY_SIZE_BYTES > 16
61         ,0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
62         0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
63     #endif
64 };
65 memset(dfState, 0, sizeof(DF_STATE));
66 DRBG_ENCRYPT_SETUP(&dfKey[0], DRBG_KEY_SIZE_BITS, &dfState->keySchedule);
67 // Create the first chaining values
68 for(i = 0; i < DF_COUNT; i++)
69     ((BYTE *)&dfState->iv[i])[3] = (BYTE)i;
70 DfCompute(dfState);
71 // initialize the first 64 bits of the IV in a way that doesn't depend
72 // on the size of the words used.
73 UINT32_TO_BYTE_ARRAY(inputLength, init);
74 UINT32_TO_BYTE_ARRAY(drbgSeedSize, &init[4]);
75 memcpy(&dfState->iv[0], init, 8);
76 dfState->contents = 4;
77 }

```

10.2.16.2.5 DfUpdate()

This updates the state with the input data. A byte at a time is moved into the state buffer until it is full and then that block is encrypted by DfCompute().

```

78 static void
79 DfUpdate(
80     PDF_STATE      dfState,
81     int             size,
82     const BYTE      *data
83 )
84 {
85     while(size > 0)
86     {
87         int toFill = DRBG_IV_SIZE_BYTES - dfState->contents;
88         if(size < toFill)
89             toFill = size;
90         // Copy as many bytes as there are or until the state buffer is full
91         memcpy(&dfState->buf.bytes[dfState->contents], data, toFill);
92         // Reduce the size left by the amount copied
93         size -= toFill;
94         // Advance the data pointer by the amount copied
95         data += toFill;
96         // increase the buffer contents count by the amount copied
97         dfState->contents += toFill;
98         pAssert(dfState->contents <= DRBG_IV_SIZE_BYTES);
99         // If we have a full buffer, do a computation pass.
100         if(dfState->contents == DRBG_IV_SIZE_BYTES)
101             DfCompute(dfState);
102     }
103 }

```

10.2.16.2.6 DfEnd()

This function is called to get the result of the derivation function computation. If the buffer is not full, it is padded with zeros. The output buffer is structured to be the same as a DRBG_SEED value so that the function can return a pointer to the DRBG_SEED value in the DF_STATE structure.

```

104 static DRBG_SEED *
105 DfEnd(
106     PDF_STATE      dfState
107 )

```

```

108 {
109     // Since DfCompute is always called when a buffer is full, there is always
110     // space in the buffer for the terminator
111     dfState->buf.bytes[dfState->contents++] = 0x80;
112     // If the buffer is not full, pad with zeros
113     while(dfState->contents < DRBG_IV_SIZE_BYTES)
114         dfState->buf.bytes[dfState->contents++] = 0;
115     // Do a final state update
116     DfCompute(dfState);
117     return (DRBG_SEED *) &dfState->iv;
118 }

```

10.2.16.2.7 DfBuffer()

Function to take an input buffer and do the derivation function to produce a DRBG_SEED value that can be used in DRBG_Reseed();

```

119 static DRBG_SEED *
120 DfBuffer(
121     DRBG_SEED      *output,          // OUT: receives the result
122     int             size,             // IN: size of the buffer to add
123     BYTE            *buf,             // IN: address of the buffer
124 )
125 {
126     DF_STATE        dfState;
127     if(size == 0 || buf == NULL)
128         return NULL;
129     // Initialize the derivation function
130     DfStart(&dfState, size);
131     DfUpdate(&dfState, size, buf);
132     DfEnd(&dfState);
133     memcpy(output, &dfState.iv[0], sizeof(DRBG_SEED));
134     return output;
135 }

```

10.2.16.2.8 DRBG_GetEntropy()

Even though this implementation never fails, it may get blocked indefinitely long in the call to get entropy from the platform (DRBG_GetEntropy32()). This function is only used during instantiation of the DRBG for manufacturing and on each start-up after a non-orderly shutdown.

Return Value	Meaning
TRUE(1)	requested entropy returned
FALSE(0)	entropy Failure

```

136 BOOL
137 DRBG_GetEntropy(
138     UINT32          requiredEntropy,  // IN: requested number of bytes of full
139                                     // entropy
140     BYTE            *entropy,        // OUT: buffer to return collected entropy
141 )
142 {
143     #if !USE_DEBUG_RNG
144         UINT32      obtainedEntropy;
145         INT32       returnedEntropy;
146
147         // If in debug mode, always use the self-test values for initialization
148         if(IsSelfTest())
149         {
150

```

```

151 #endif
152 // If doing simulated DRBG, then check to see if the
153 // entropyFailure condition is being tested
154 if(!IsEntropyBad())
155 {
156     // In self-test, the caller should be asking for exactly the seed
157     // size of entropy.
158     pAssert(requiredEntropy == sizeof(DRBG_NistTestVector_Entropy));
159     memcpy(entropy, DRBG_NistTestVector_Entropy,
160           sizeof(DRBG_NistTestVector_Entropy));
161 }
162 #if !USE_DEBUG_RNG
163 }
164 else if(!IsEntropyBad())
165 {
166     // Collect entropy
167     // Note: In debug mode, the only "entropy" value ever returned
168     // is the value of the self-test vector.
169     for(returnedEntropy = 1, obtainedEntropy = 0;
170         obtainedEntropy < requiredEntropy && !IsEntropyBad();
171         obtainedEntropy += returnedEntropy)
172     {
173         returnedEntropy = _plat__GetEntropy(&entropy[obtainedEntropy],
174                                           requiredEntropy - obtainedEntropy);
175         if(returnedEntropy <= 0)
176             SetEntropyBad();
177     }
178 }
179 #endif
180 return !IsEntropyBad();
181 }

```

10.2.16.2.9 IncrementIv()

This function increments the IV value by 1. It is used by EncryptDRBG().

```

182 void
183 IncrementIv(
184     DRBG_IV *iv
185 )
186 {
187     BYTE *ivP = ((BYTE *)iv) + DRBG_IV_SIZE_BYTES;
188     while((--ivP >= (BYTE *)iv) && ((*ivP = ((*ivP + 1) & 0xFF)) == 0));
189 }

```

10.2.16.2.10 EncryptDRBG()

This does the encryption operation for the DRBG. It will encrypt the input state counter (IV) using the state key. Into the output buffer for as many times as it takes to generate the required number of bytes.

```

190 static BOOL
191 EncryptDRBG(
192     BYTE *dOut,
193     UINT32 dOutBytes,
194     DRBG_KEY_SCHEDULE *keySchedule,
195     DRBG_IV *iv,
196     UINT32 *lastValue // Points to the last output value
197 )
198 {
199     #if FIPS_COMPLIANT
200     // For FIPS compliance, the DRBG has to do a continuous self-test to make sure that
201     // no two consecutive values are the same. This overhead is not incurred if the TPM
202     // is not required to be FIPS compliant

```

```

203 //
204     UINT32          temp[DRBG_IV_SIZE_BYTES / sizeof(UINT32)];
205     int             i;
206     BYTE            *p;
207
208     for(; dOutBytes > 0;)
209     {
210         // Increment the IV before each encryption (this is what makes this
211         // different from normal counter-mode encryption
212         IncrementIv(iv);
213         DRBG_ENCRYPT(keySchedule, iv, temp);
214         // Expect a 16 byte block
215         #if DRBG_IV_SIZE_BITS != 128
216         #error "Unsupported IV size in DRBG"
217         #endif
218         if((lastValue[0] == temp[0])
219             && (lastValue[1] == temp[1])
220             && (lastValue[2] == temp[2])
221             && (lastValue[3] == temp[3])
222             )
223         {
224             LOG_FAILURE(FATAL_ERROR_ENTROPY);
225             return FALSE;
226         }
227         lastValue[0] = temp[0];
228         lastValue[1] = temp[1];
229         lastValue[2] = temp[2];
230         lastValue[3] = temp[3];
231         i = MIN(dOutBytes, DRBG_IV_SIZE_BYTES);
232         dOutBytes -= i;
233         for(p = (BYTE *)temp; i > 0; i--)
234             *dOut++ = *p++;
235     }
236     #else // version without continuous self-test
237     NOT_REFERENCED(lastValue);
238     for(; dOutBytes >= DRBG_IV_SIZE_BYTES;
239         dOut = &dOut[DRBG_IV_SIZE_BYTES], dOutBytes -= DRBG_IV_SIZE_BYTES)
240     {
241         // Increment the IV
242         IncrementIv(iv);
243         DRBG_ENCRYPT(keySchedule, iv, dOut);
244     }
245     // If there is a partial, generate into a block-sized
246     // temp buffer and copy to the output.
247     if(dOutBytes != 0)
248     {
249         BYTE          temp[DRBG_IV_SIZE_BYTES];
250         // Increment the IV
251         IncrementIv(iv);
252         DRBG_ENCRYPT(keySchedule, iv, temp);
253         memcpy(dOut, temp, dOutBytes);
254     }
255     #endif
256     return TRUE;
257 }

```

10.2.16.2.11 DRBG_Update()

This function performs the state update function. According to SP800-90A, a temp value is created by doing CTR mode encryption of *providedData* and replacing the key and IV with these values. The one difference is that, with counter mode, the IV is incremented after each block is encrypted and in this operation, the counter is incremented before each block is encrypted. This function implements an *optimized* version of the algorithm in that it does the update of the *drbgState*→*seed* in place and then

providedData is XORed into *drbgState*→*seed* to complete the encryption of *providedData*. This works because the IV is the last thing that gets encrypted.

```

258 static BOOL
259 DRBG_Update(
260     DRBG_STATE          *drbgState,      // IN:OUT state to update
261     DRBG_KEY_SCHEDULE   *keySchedule,    // IN: the key schedule (optional)
262     DRBG_SEED           *providedData    // IN: additional data
263 )
264 {
265     UINT32               i;
266     BYTE                 *temp = (BYTE *)&drbgState->seed;
267     DRBG_KEY             *key = pDRBG_KEY(&drbgState->seed);
268     DRBG_IV              *iv = pDRBG_IV(&drbgState->seed);
269     DRBG_KEY_SCHEDULE    localKeySchedule;
270 //
271     pAssert(drbgState->magic == DRBG_MAGIC);
272
273     // If an key schedule was not provided, make one
274     if(keySchedule == NULL)
275     {
276         if(DRBG_ENCRYPT_SETUP((BYTE *)key,
277             DRBG_KEY_SIZE_BITS, &localKeySchedule) != 0)
278         {
279             LOG_FAILURE(FATAL_ERROR_INTERNAL);
280             return FALSE;
281         }
282         keySchedule = &localKeySchedule;
283     }
284     // Encrypt the temp value
285
286     EncryptDRBG(temp, sizeof(DRBG_SEED), keySchedule, iv,
287                 drbgState->lastValue);
288     if(providedData != NULL)
289     {
290         BYTE *pP = (BYTE *)providedData;
291         for(i = DRBG_SEED_SIZE_BYTES; i != 0; i--)
292             *temp++ ^= *pP++;
293     }
294     // Since temp points to the input key and IV, we are done and
295     // don't need to copy the resulting 'temp' to drbgState->seed
296     return TRUE;
297 }

```

10.2.16.2.12 DRBG_Reseed()

This function is used when reseeding of the DRBG is required. If entropy is provided, it is used in lieu of using hardware entropy.

NOTE the provided entropy must be the required size.

Return Value	Meaning
TRUE(1)	reseed succeeded
FALSE(0)	reseed failed, probably due to the entropy generation

```

298 BOOL
299 DRBG_Reseed(
300     DRBG_STATE          *drbgState,      // IN: the state to update
301     DRBG_SEED           *providedEntropy, // IN: entropy
302     DRBG_SEED           *additionalData  // IN:
303 )

```

```

304 {
305     DRBG_SEED        seed;
306
307     pAssert((drbgState != NULL) && (drbgState->magic == DRBG_MAGIC));
308
309     if(providedEntropy == NULL)
310     {
311         providedEntropy = &seed;
312         if(!DRBG_GetEntropy(sizeof(DRBG_SEED), (BYTE *)providedEntropy))
313             return FALSE;
314     }
315     if(additionalData != NULL)
316     {
317         unsigned int    i;
318
319         // XOR the provided data into the provided entropy
320         for(i = 0; i < sizeof(DRBG_SEED); i++)
321             ((BYTE *)providedEntropy)[i] ^= ((BYTE *)additionalData)[i];
322     }
323     DRBG_Update(drbgState, NULL, providedEntropy);
324
325     drbgState->reseedCounter = 1;
326
327     return TRUE;
328 }

```

10.2.16.2.13 DRBG_SelfTest()

This is run when the DRBG is instantiated and at startup.

Return Value	Meaning
TRUE(1)	test OK
FALSE(0)	test failed

```

329 BOOL
330 DRBG_SelfTest(
331     void
332 )
333 {
334     BYTE        buf[sizeof(DRBG_NistTestVector_Generated)];
335     DRBG_SEED    seed;
336     UINT32      i;
337     BYTE        *p;
338     DRBG_STATE   testState;
339     //
340     pAssert(!IsSelfTest());
341
342     SetSelfTest();
343     SetDrbgTested();
344     // Do an instantiate
345     if(!DRBG_Instantiate(&testState, 0, NULL))
346         return FALSE;
347     #if DRBG_DEBUG_PRINT
348         dbgDumpMemBlock(pDRBG_KEY(&testState), DRBG_KEY_SIZE_BYTES,
349             "Key after Instantiate");
350         dbgDumpMemBlock(pDRBG_IV(&testState), DRBG_IV_SIZE_BYTES,
351             "Value after Instantiate");
352     #endif
353     if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
354         return FALSE;
355     #if DRBG_DEBUG_PRINT
356         dbgDumpMemBlock(pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,

```

```

357         "Key after 1st Generate");
358     dbgDumpMemBlock(pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
359         "Value after 1st Generate");
360 #endif
361     if(memcmp(buf, DRBG_NistTestVector_GeneratedInterm, sizeof(buf)) != 0)
362         return FALSE;
363     memcpy(seed.bytes, DRBG_NistTestVector_EntropyReSeed, sizeof(seed));
364     DRBG_Reseed(&testState, &seed, NULL);
365 #if DRBG_DEBUG_PRINT
366     dbgDumpMemBlock((BYTE *)pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
367         "Key after 2nd Generate");
368     dbgDumpMemBlock((BYTE *)pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
369         "Value after 2nd Generate");
370     dbgDumpMemBlock(buf, sizeof(buf), "2nd Generated");
371 #endif
372     if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
373         return FALSE;
374     if(memcmp(buf, DRBG_NistTestVector_Generated, sizeof(buf)) != 0)
375         return FALSE;
376     ClearSelfTest();
377
378     DRBG_Uninstantiate(&testState);
379     for(p = (BYTE *)&testState, i = 0; i < sizeof(DRBG_STATE); i++)
380     {
381         if(*p++)
382             return FALSE;
383     }
384     // Simulate hardware failure to make sure that we get an error when
385     // trying to instantiate
386     SetEntropyBad();
387     if(DRBG_Instantiate(&testState, 0, NULL))
388         return FALSE;
389     ClearEntropyBad();
390
391     return TRUE;
392 }

```

10.2.16.3 Public Interface

10.2.16.3.1 Description

The functions in this section are the interface to the RNG. These are the functions that are used by TPM.lib.

10.2.16.3.2 CryptRandomStir()

This function is used to cause a reseed. A DRBG_SEED amount of entropy is collected from the hardware and then additional data is added.

Error Return	Meaning
TPM_RC_NO_RESULT	failure of the entropy generator

```

393 LIB_EXPORT TPM_RC
394 CryptRandomStir(
395     UINT16        additionalDataSize,
396     BYTE          *additionalData
397 )
398 {
399 #if !USE_DEBUG_RNG
400     DRBG_SEED     tmpBuf;
401     DRBG_SEED     dfResult;

```



```

402 //
403 // All reseed with outside data starts with a buffer full of entropy
404 if(!DRBG_GetEntropy(sizeof(tmpBuf), (BYTE *)&tmpBuf))
405     return TPM_RC_NO_RESULT;
406
407 DRBG_Reseed(&drbgDefault, &tmpBuf,
408             DfBuffer(&dfResult, additionalDataSize, additionalData));
409 drbgDefault.reseedCounter = 1;
410
411 return TPM_RC_SUCCESS;
412
413 #else
414 // If doing debug, use the input data as the initial setting for the RNG state
415 // so that the test can be reset at any time.
416 // Note: If this is called with a data size of 0 or less, nothing happens. The
417 // presumption is that, in a debug environment, the caller will have specific
418 // values for initialization, so this check is just a simple way to prevent
419 // inadvertent programming errors from screwing things up. This doesn't use an
420 // pAssert() because the non-debug version of this function will accept these
421 // parameters as meaning that there is no additionalData and only hardware
422 // entropy is used.
423 if((additionalDataSize > 0) && (additionalData != NULL))
424 {
425     memset(drbgDefault.seed.bytes, 0, sizeof(drbgDefault.seed.bytes));
426     memcpy(drbgDefault.seed.bytes, additionalData,
427           MIN(additionalDataSize, sizeof(drbgDefault.seed.bytes)));
428 }
429 drbgDefault.reseedCounter = 1;
430
431 return TPM_RC_SUCCESS;
432 #endif
433 }

```

10.2.16.3.3 CryptRandomGenerate()

Generate a *randomSize* number of random bytes.

```

434 LIB_EXPORT UINT16
435 CryptRandomGenerate(
436     UINT16    randomSize,
437     BYTE      *buffer
438 )
439 {
440     return DRBG_Generate((RAND_STATE *)&drbgDefault, buffer, randomSize);
441 }

```

10.2.16.3.4 DRBG_InstantiateSeededKdf()

This function is used to instantiate a KDF-based RNG. This is used for derivations. This function always returns TRUE.

```

442 LIB_EXPORT BOOL
443 DRBG_InstantiateSeededKdf(
444     KDF_STATE    *state,           // OUT: buffer to hold the state
445     TPM_ALG_ID   hashAlg,         // IN: hash algorithm
446     TPM_ALG_ID   kdf,             // IN: the KDF to use
447     TPM2B        *seed,           // IN: the seed to use
448     const TPM2B   *label,         // IN: a label for the generation process.
449     TPM2B        *context,        // IN: the context value
450     UINT32       limit,           // IN: Maximum number of bits from the KDF
451 )
452 {
453     state->magic = KDF_MAGIC;

```

```

454     state->limit = limit;
455     state->seed = seed;
456     state->hash = hashAlg;
457     state->kdf = kdf;
458     state->label = label;
459     state->context = context;
460     state->digestSize = CryptHashGetDigestSize(hashAlg);
461     state->counter = 0;
462     state->residual.t.size = 0;
463     return TRUE;
464 }

```

10.2.16.3.5 DRBG_AdditionalData()

Function to reseed the DRBG with additional entropy. This is normally called before computing the protection value of a primary key in the Endorsement hierarchy.

```

465 LIB_EXPORT void
466 DRBG_AdditionalData(
467     DRBG_STATE *drbgState,    // IN/OUT: state to update
468     TPM2B *additionalData // IN: value to incorporate
469 )
470 {
471     DRBG_SEED dfResult;
472     if(drbgState->magic == DRBG_MAGIC)
473     {
474         DfBuffer(&dfResult, additionalData->size, additionalData->buffer);
475         DRBG_Reseed(drbgState, &dfResult, NULL);
476     }
477 }

```

10.2.16.3.6 DRBG_InstantiateSeeded()

This function is used to instantiate a random number generator from seed values. The nominal use of this generator is to create sequences of pseudo-random numbers from a seed value.

Error Return	Meaning
TPM_RC_FAILURE	DRBG self-test failure

```

478 LIB_EXPORT TPM_RC
479 DRBG_InstantiateSeeded(
480     DRBG_STATE *drbgState,    // IN/OUT: buffer to hold the state
481     const TPM2B *seed,        // IN: the seed to use
482     const TPM2B *purpose,     // IN: a label for the generation process.
483     const TPM2B *name,        // IN: name of the object
484     const TPM2B *additional  // IN: additional data
485 )
486 {
487     DF_STATE dfState;
488     int totalInputSize;
489     // DRBG should have been tested, but...
490     if(!IsDrbgTested() && !DRBG_SelfTest())
491     {
492         LOG_FAILURE(FATAL_ERROR_SELF_TEST);
493         return TPM_RC_FAILURE;
494     }
495     // Initialize the DRBG state
496     memset(drbgState, 0, sizeof(DRBG_STATE));
497     drbgState->magic = DRBG_MAGIC;
498
499     // Size all of the values

```

```

500     totalInputSize = (seed != NULL) ? seed->size : 0;
501     totalInputSize += (purpose != NULL) ? purpose->size : 0;
502     totalInputSize += (name != NULL) ? name->size : 0;
503     totalInputSize += (additional != NULL) ? additional->size : 0;
504
505     // Initialize the derivation
506     DfStart(&dfState, totalInputSize);
507
508     // Run all the input strings through the derivation function
509     if(seed != NULL)
510         DfUpdate(&dfState, seed->size, seed->buffer);
511     if(purpose != NULL)
512         DfUpdate(&dfState, purpose->size, purpose->buffer);
513     if(name != NULL)
514         DfUpdate(&dfState, name->size, name->buffer);
515     if(additional != NULL)
516         DfUpdate(&dfState, additional->size, additional->buffer);
517
518     // Used the derivation function output as the "entropy" input. This is not
519     // how it is described in SP800-90A but this is the equivalent function
520     DRBG_Reseed((DRBG_STATE *)drbgState, DfEnd(&dfState), NULL);
521
522     return TPM_RC_SUCCESS;
523 }

```

10.2.16.3.7 CryptRandStartup()

This function is called when TPM_Startup is executed. This function always returns TRUE.

```

524 LIB_EXPORT BOOL
525 CryptRandStartup(
526     void
527 )
528 {
529     #if !_DRBG_STATE_SAVE
530         // If not saved in NV, re-instantiate on each startup
531         return DRBG_Instantiate(&drbgDefault, 0, NULL);
532     #else
533         // If the running state is saved in NV, NV has to be loaded before it can
534         // be updated
535         if(go.drbgState.magic == DRBG_MAGIC)
536             return DRBG_Reseed(&go.drbgState, NULL, NULL);
537         else
538             return DRBG_Instantiate(&go.drbgState, 0, NULL);
539     #endif
540 }

```

10.2.16.3.8 CryptRandInit()

This function is called when _TPM_Init is being processed.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

541 LIB_EXPORT BOOL
542 CryptRandInit(
543     void
544 )
545 {
546     #if !USE_DEBUG_RNG

```

```

547     _plat__GetEntropy(NULL, 0);
548 #endif
549     return DRBG_SelfTest();
550 }

```

10.2.16.3.9 DRBG_Generate()

This function generates a random sequence according SP800-90A. If *random* is not NULL, then *randomSize* bytes of random values are generated. If *random* is NULL or *randomSize* is zero, then the function returns zero without generating any bits or updating the reseed counter. This function returns the number of bytes produced which could be less than the number requested if the request is too large (**too large** is implementation dependent.)

```

551 LIB_EXPORT UINT16
552 DRBG_Generate(
553     RAND_STATE      *state,
554     BYTE            *random,          // OUT: buffer to receive the random values
555     UINT16          randomSize       // IN: the number of bytes to generate
556 )
557 {
558     if(state == NULL)
559         state = (RAND_STATE *)&drbgDefault;
560     if(random == NULL)
561         return 0;
562
563     // If the caller used a KDF state, generate a sequence from the KDF not to
564     // exceed the limit.
565     if(state->kdf.magic == KDF_MAGIC)
566     {
567         KDF_STATE      *kdf = (KDF_STATE *)state;
568         UINT32          counter = (UINT32)kdf->counter;
569         INT32           bytesLeft = randomSize;
570
571         // If the number of bytes to be returned would put the generator
572         // over the limit, then return 0
573         if(((kdf->counter * kdf->digestSize) + randomSize) * 8 > kdf->limit)
574             return 0;
575         // Process partial and full blocks until all requested bytes provided
576         while(bytesLeft > 0)
577         {
578             // If there is any residual data in the buffer, copy it to the output
579             // buffer
580             if(kdf->residual.t.size > 0)
581             {
582                 INT32      size;
583
584                 // Don't use more of the residual than will fit or more than are
585                 // available
586                 size = MIN(kdf->residual.t.size, bytesLeft);
587
588                 // Copy some or all of the residual to the output. The residual is
589                 // at the end of the buffer. The residual might be a full buffer.
590                 MemoryCopy(random,
591                             &kdf->residual.t.buffer
592                             [kdf->digestSize - kdf->residual.t.size], size);
593
594                 // Advance the buffer pointer
595                 random += size;
596
597                 // Reduce the number of bytes left to get
598                 bytesLeft -= size;
599
600                 // And reduce the residual size appropriately
601                 kdf->residual.t.size -= (UINT16)size;

```

```

602     }
603     else
604     {
605         UINT16        blocks = (UINT16)(bytesLeft / kdf->digestSize);
606     //
607         // Get the number of required full blocks
608         if(blocks > 0)
609         {
610             UINT16        size = blocks * kdf->digestSize;
611         // Get some number of full blocks and put them in the return buffer
612             CryptKDFa(kdf->hash, kdf->seed, kdf->label, kdf->context, NULL,
613                     kdf->limit, random, &counter, blocks);
614
615             // reduce the size remaining to be moved and advance the pointer
616             bytesLeft -= size;
617             random += size;
618         }
619     else
620     {
621         // Fill the residual buffer with a full block and then loop to
622         // top to get part of it copied to the output.
623         kdf->residual.t.size = CryptKDFa(kdf->hash, kdf->seed,
624                                         kdf->label, kdf->context, NULL,
625                                         kdf->limit,
626                                         kdf->residual.t.buffer,
627                                         &counter, 1);
628     }
629 }
630 }
631 kdf->counter = counter;
632 return randomSize;
633 }
634 else if(state->drbg.magic == DRBG_MAGIC)
635 {
636     DRBG_STATE        *drbgState = (DRBG_STATE *)state;
637     DRBG_KEY_SCHEDULE    keySchedule;
638     DRBG_SEED            *seed = &drbgState->seed;
639
640     if(drbgState->reseedCounter >= CTR_DRBG_MAX_REQUESTS_PER_RESEED)
641     {
642         if(drbgState == &drbgDefault)
643         {
644             DRBG_Reseed(drbgState, NULL, NULL);
645             if(IsEntropyBad() && !IsSelfTest())
646                 return 0;
647         }
648         else
649         {
650             // If this is a PRNG then the only way to get
651             // here is if the SW has run away.
652             LOG_FAILURE(FATAL_ERROR_INTERNAL);
653             return 0;
654         }
655     }
656     // if the allowed number of bytes in a request is larger than the
657     // less than the number of bytes that can be requested, then check
658 #if UINT16_MAX >= CTR_DRBG_MAX_BYTES_PER_REQUEST
659     if(randomSize > CTR_DRBG_MAX_BYTES_PER_REQUEST)
660         randomSize = CTR_DRBG_MAX_BYTES_PER_REQUEST;
661 #endif
662     // Create encryption schedule
663     if(DRBG_ENCRYPT_SETUP((BYTE *)pDRBG_KEY(seed),
664                         DRBG_KEY_SIZE_BITS, &keySchedule) != 0)
665     {
666         LOG_FAILURE(FATAL_ERROR_INTERNAL);
667         return 0;

```

```

668     }
669     // Generate the random data
670     EncryptDRBG(random, randomSize, &keySchedule, pDRBG_IV(seed),
671               drbgState->lastValue);
672     // Do a key update
673     DRBG_Update(drbgState, &keySchedule, NULL);
674
675     // Increment the reseed counter
676     drbgState->reseedCounter += 1;
677 }
678 else
679 {
680     LOG_FAILURE(FATAL_ERROR_INTERNAL);
681     return FALSE;
682 }
683 return randomSize;
684 }

```

10.2.16.3.10 DRBG_Instantiate()

This is CTR_DRBG_Instantiate_algorithm() from [SP 800-90A 10.2.1.3.1]. This is called when a the TPM DRBG is to be instantiated. This is called to instantiate a DRBG used by the TPM for normal operations.

Return Value	Meaning
TRUE(1)	instantiation succeeded
FALSE(0)	instantiation failed

```

685 LIB_EXPORT BOOL
686 DRBG_Instantiate(
687     DRBG_STATE *drbgState,           // OUT: the instantiated value
688     UINT16 pSize,                    // IN: Size of personalization string
689     BYTE *personalization,           // IN: The personalization string
690 )
691 {
692     DRBG_SEED seed;
693     DRBG_SEED dfResult;
694 //
695     pAssert((pSize == 0) || (pSize <= sizeof(seed)) || (personalization != NULL));
696     // If the DRBG has not been tested, test when doing an instantiation. Since
697     // Instantiation is called during self test, make sure we don't get stuck in a
698     // loop.
699     if(!IsDrbgTested() && !IsSelfTest() && !DRBG_SelfTest())
700         return FALSE;
701     // If doing a self test, DRBG_GetEntropy will return the NIST
702     // test vector value.
703     if(!DRBG_GetEntropy(sizeof(seed), (BYTE *)&seed))
704         return FALSE;
705     // set everything to zero
706     memset(drbgState, 0, sizeof(DRBG_STATE));
707     drbgState->magic = DRBG_MAGIC;
708
709     // Steps 1, 2, 3, 6, 7 of SP 800-90A 10.2.1.3.1 are exactly what
710     // reseeding does. So, do a reduction on the personalization value (if any)
711     // and do a reseed.
712     DRBG_Reseed(drbgState, &seed, DfBuffer(&dfResult, pSize, personalization));
713
714     return TRUE;
715 }


```

10.2.16.3.11 DRBG_Uninstantiate()

This is Uninstantiate_function() from [SP 800-90A 9.4].

Error Return	Meaning
TPM_RC_VALUE	not a valid state

```
716 LIB_EXPORT TPM_RC
717 DRBG_Uninstantiate(
718     DRBG_STATE *drbgState    // IN/OUT: working state to erase
719 )
720 {
721     if((drbgState == NULL) || (drbgState->magic != DRBG_MAGIC))
722         return TPM_RC_VALUE;
723     memset(drbgState, 0, sizeof(DRBG_STATE));
724     return TPM_RC_SUCCESS;
725 }
```



10.2.17 CryptRsa.c

10.2.17.1 Introduction

This file contains implementation of cryptographic primitives for RSA. Vendors may replace the implementation in this file with their own library functions.

10.2.17.2 Includes

Need this define to get the *private* defines for this function

```
1  #define CRYPT_RSA_C
2  #include "Tpm.h"
3
4  #if ALG_RSA
```

10.2.17.3 Obligatory Initialization Functions

10.2.17.3.1 CryptRsaInit()

Function called at _TPM_Init().

```
5  BOOL
6  CryptRsaInit(
7      void
8  )
9  {
10     return TRUE;
11 }
```

10.2.17.3.2 CryptRsaStartup()

Function called at TPM2_Startup()

```
12  BOOL
13  CryptRsaStartup(
14      void
15  )
16  {
17     return TRUE;
18 }
```

10.2.17.4 Internal Functions

10.2.17.4.1 RsaInitializeExponent()

This function initializes the bignum data structure that holds the private exponent. This function returns the pointer to the private exponent value so that it can be used in an initializer for a data declaration.

```
19  static privateExponent *
20  RsaInitializeExponent(
21      privateExponent      *Z
22  )
23  {
24      bigNum      *bn = (bigNum *) &Z->P;
25      int          i;
```

```

26  //
27  for(i = 0; i < 5; i++)
28  {
29      bn[i] = (bigNum)&Z->entries[i];
30      BnInit(bn[i], BYTES_TO_CRYPT_WORDS(sizeof(Z->entries[0].d)));
31  }
32  return Z;
33  }

```

10.2.17.4.2 MakePgreaterThanQ()

This function swaps the pointers for P and Q if Q happens to be larger than Q.

```

34  static void
35  MakePgreaterThanQ(
36      privateExponent      *Z
37  )
38  {
39      if(BnUnsignedCmp(Z->P, Z->Q) < 0)
40      {
41          bigNum      bnT = Z->P;
42          Z->P = Z->Q;
43          Z->Q = bnT;
44      }
45  }

```

10.2.17.4.3 PackExponent()

This function takes the bignum private exponent and converts it into TPM2B form. In this form, the size field contains the overall size of the packed data. The buffer contains 5, equal sized values in P, Q, dP , dQ , q/nv order. For example, if a key has a 2Kb public key, then the packed private key will contain 5, 1Kb values. This form makes it relatively easy to load and save the values without changing the normal unmarshaling to do anything more than allow a larger TPM2B for the private key. Also, when exporting the value, all that is needed is to change the size field of the private key in order to save just the P value.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure -- The data is too big to fit

```

46  static BOOL
47  PackExponent(
48      TPM2B_PRIVATE_KEY_RSA      *packed,
49      privateExponent      *Z
50  )
51  {
52      int      i;
53      UINT16      primeSize = (UINT16)BITS_TO_BYTES(BnMsb(Z->P));
54      UINT16      pS = primeSize;
55  //
56  pAssert((primeSize * 5) <= sizeof(packed->t.buffer));
57  packed->t.size = (primeSize * 5) + RSA_prime_flag;
58  for(i = 0; i < 5; i++)
59      if(!BnToBytes((bigNum)&Z->entries[i], &packed->t.buffer[primeSize * i], &pS))
60          return FALSE;
61  if(pS != primeSize)
62      return FALSE;
63  return TRUE;
64  }

```

10.2.17.4.4 UnpackExponent()

This function unpacks the private exponent from its TPM2B form into its bignum form.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	TPM2B is not the correct size

```

65  static BOOL
66  UnpackExponent(
67      TPM2B_PRIVATE_KEY_RSA    *b,
68      privateExponent          *Z
69  )
70  {
71      UINT16                    primeSize = b->t.size & ~RSA_prime_flag;
72      int                       i;
73      bigNum                    *bn = &Z->P;
74  //
75      VERIFY(b->t.size & RSA_prime_flag);
76      RsaInitializeExponent(Z);
77      VERIFY((primeSize % 5) == 0);
78      primeSize /= 5;
79      for(i = 0; i < 5; i++)
80          VERIFY(BnFromBytes(bn[i], &b->t.buffer[primeSize * i], primeSize)
81              != NULL);
82      MakePgreaterThanQ(Z);
83      return TRUE;
84  Error:
85      return FALSE;
86  }

```

10.2.17.4.5 ComputePrivateExponent()

This function computes the private exponent from the primes.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

87  static BOOL
88  ComputePrivateExponent(
89      bigNum                    pubExp,          // IN: the public exponent
90      privateExponent          *Z               // IN/OUT: on input, has primes P and Q. On
91                                              // output, has P, Q, dP, dQ, and pInv
92  )
93  {
94      BOOL                      pOK;
95      BOOL                      qOK;
96      BN_PRIME(pT);
97  //
98      // make p the larger value so that m2 is always less than p
99      MakePgreaterThanQ(Z);
100
101      //dP = (1/e) mod (p-1)
102      pOK = BnSubWord(pT, Z->P, 1);
103      pOK = pOK && BnModInverse(Z->dP, pubExp, pT);
104      //dQ = (1/e) mod (q-1)
105      qOK = BnSubWord(pT, Z->Q, 1);
106      qOK = qOK && BnModInverse(Z->dQ, pubExp, pT);
107      // qInv = (1/q) mod p

```

```

108     if(pOK && qOK)
109         pOK = qOK = BnModInverse(Z->qInv, Z->Q, Z->P);
110     if(!pOK)
111         BnSetWord(Z->P, 0);
112     if(!qOK)
113         BnSetWord(Z->Q, 0);
114     return pOK && qOK;
115 }

```

10.2.17.4.6 RsaPrivateKeyOp()

This function is called to do the exponentiation with the private key. Compile options allow use of the simple (but slow) private exponent, or the more complex but faster CRT method.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

116 static BOOL
117 RsaPrivateKeyOp(
118     bigNum          inOut, // IN/OUT: number to be exponentiated
119     privateExponent *Z
120 )
121 {
122     BN_RSA(M1);
123     BN_RSA(M2);
124     BN_RSA(M);
125     BN_RSA(H);
126     //
127     MakePgreaterThanQ(Z);
128     // m1 = cdP mod p
129     VERIFY(BnModExp(M1, inOut, Z->dP, Z->P));
130     // m2 = cdQ mod q
131     VERIFY(BnModExp(M2, inOut, Z->dQ, Z->Q));
132     // h = qInv * (m1 - m2) mod p = qInv * (m1 + P - m2) mod P because Q < P
133     // so m2 < P
134     VERIFY(BnSub(H, Z->P, M2));
135     VERIFY(BnAdd(H, H, M1));
136     VERIFY(BnModMult(H, H, Z->qInv, Z->P));
137     // m = m2 + h * q
138     VERIFY(BnMult(M, H, Z->Q));
139     VERIFY(BnAdd(inOut, M2, M));
140     return TRUE;
141 Error:
142     return FALSE;
143 }

```

10.2.17.4.7 RSAEP()

This function performs the RSAEP operation defined in PKCS#1v2.1. It is an exponentiation of a value (*m*) with the public exponent (*e*), modulo the public (*n*).

Error Return	Meaning
TPM_RC_VALUE	number to exponentiate is larger than the modulus

```

144 static TPM_RC
145 RSAEP(
146     TPM2B          *dInOut, // IN: size of the encrypted block and the size of
147                             // the encrypted value. It must be the size of

```

```

148                                     // the modulus.
149                                     // OUT: the encrypted data. Will receive the
150                                     // decrypted value
151     OBJECT      *key                 // IN: the key to use
152 )
153 {
154     TPM2B_TYPE(4BYTES, 4);
155     TPM2B_4BYTES e2B;
156     UINT32      e = key->publicArea.parameters.rsaDetail.exponent;
157 //
158     if(e == 0)
159         e = RSA_DEFAULT_PUBLIC_EXPONENT;
160     UINT32_TO_BYTE_ARRAY(e, e2B.t.buffer);
161     e2B.t.size = 4;
162     return ModExpB(dInOut->size, dInOut->buffer, dInOut->size, dInOut->buffer,
163                   e2B.t.size, e2B.t.buffer, key->publicArea.unique.rsa.t.size,
164                   key->publicArea.unique.rsa.t.buffer);
165 }

```

10.2.17.4.8 RSADP()

This function performs the RSADP operation defined in PKCS#1v2.1. It is an exponentiation of a value (c) with the private exponent (d), modulo the public modulus (n). The decryption is in place.

This function also checks the size of the private key. If the size indicates that only a prime value is present, the key is converted to being a private exponent.

Error Return	Meaning
TPM_RC_SIZE	the value to decrypt is larger than the modulus

```

166 static TPM_RC
167 RSADP(
168     TPM2B      *inOut,           // IN/OUT: the value to encrypt
169     OBJECT      *key             // IN: the key
170 )
171 {
172     BN_RSA_INITIALIZED(bnM, inOut);
173     NEW_PRIVATE_EXPONENT(Z);
174     if(UndersignedCompareB(inOut->size, inOut->buffer,
175                           key->publicArea.unique.rsa.t.size,
176                           key->publicArea.unique.rsa.t.buffer) >= 0)
177         return TPM_RC_SIZE;
178     // private key operation requires that private exponent be loaded
179     // During self-test, this might not be the case so load it up if it hasn't
180     // already done
181     // been done
182     if((key->sensitive.sensitive.rsa.t.size & RSA_prime_flag) == 0)
183     {
184         if(CryptRsaLoadPrivateExponent(&key->publicArea, &key->sensitive)
185           != TPM_RC_SUCCESS)
186             return TPM_RC_BINDING;
187     }
188     VERIFY(UnpackExponent(&key->sensitive.sensitive.rsa, Z));
189     VERIFY(RsaPrivateKeyOp(bnM, Z));
190     VERIFY(BnTo2B(bnM, inOut, inOut->size));
191     return TPM_RC_SUCCESS;
192 Error:
193     return TPM_RC_FAILURE;
194 }

```

10.2.17.4.9 OaepEncode()

This function performs OAEP padding. The size of the buffer to receive the OAEP padded data must equal the size of the modulus

Error Return	Meaning
TPM_RC_VALUE	<i>hashAlg</i> is not valid or message size is too large

```

195 static TPM_RC
196 OaepEncode(
197     TPM2B      *padded,          // OUT: the pad data
198     TPM_ALG_ID  hashAlg,         // IN: algorithm to use for padding
199     const TPM2B *label,          // IN: null-terminated string (may be NULL)
200     TPM2B      *message,         // IN: the message being padded
201     RAND_STATE  *rand            // IN: the random number generator to use
202 )
203 {
204     INT32      padLen;
205     INT32      dbSize;
206     INT32      i;
207     BYTE       mySeed[MAX_DIGEST_SIZE];
208     BYTE       *seed = mySeed;
209     UINT16     hLen = CryptHashGetDigestSize(hashAlg);
210     BYTE       mask[MAX_RSA_KEY_BYTES];
211     BYTE       *pp;
212     BYTE       *pm;
213     TPM_RC     retVal = TPM_RC_SUCCESS;
214
215     pAssert(padded != NULL && message != NULL);
216
217     // A value of zero is not allowed because the KDF can't produce a result
218     // if the digest size is zero.
219     if(hLen == 0)
220         return TPM_RC_VALUE;
221
222     // Basic size checks
223     // make sure digest isn't too big for key size
224     if(padded->size < (2 * hLen) + 2)
225         ERROR_RETURN(TPM_RC_HASH);
226
227     // and that message will fit messageSize <= k - 2hLen - 2
228     if(message->size > (padded->size - (2 * hLen) - 2))
229         ERROR_RETURN(TPM_RC_VALUE);
230
231     // Hash L even if it is null
232     // Offset into padded leaving room for masked seed and byte of zero
233     pp = &padded->buffer[hLen + 1];
234     if(CryptHashBlock(hashAlg, label->buffer, (BYTE *)label->buffer,
235         hLen, pp) != hLen)
236         ERROR_RETURN(TPM_RC_FAILURE);
237
238     // concatenate PS of k mLen 2hLen 2
239     padLen = padded->size - message->size - (2 * hLen) - 2;
240     MemorySet(&pp[hLen], 0, padLen);
241     pp[hLen + padLen] = 0x01;
242     padLen += 1;
243     memcpy(&pp[hLen + padLen], message->buffer, message->size);
244
245     // The total size of db = hLen + pad + mSize;
246     dbSize = hLen + padLen + message->size;
247
248     // If testing, then use the provided seed. Otherwise, use values
249     // from the RNG
250     CryptRandomGenerate(hLen, mySeed);

```

```

251     DRBG_Generate(rand, mySeed, (UINT16)hLen);
252     if(g_inFailureMode)
253         ERROR_RETURN(TPM_RC_FAILURE);
254     // mask = MGF1 (seed, nSize hLen 1)
255     CryptMGF_KDF(dbSize, mask, hashAlg, hLen, seed, 0);
256
257     // Create the masked db
258     pm = mask;
259     for(i = dbSize; i > 0; i--)
260         *pp++ ^= *pm++;
261     pp = &padded->buffer[hLen + 1];
262
263     // Run the masked data through MGF1
264     if(CryptMGF_KDF(hLen, &padded->buffer[1], hashAlg, dbSize, pp, 0) !=
(unsigned)hLen)
265         ERROR_RETURN(TPM_RC_VALUE);
266 // Now XOR the seed to create masked seed
267 pp = &padded->buffer[1];
268 pm = seed;
269 for(i = hLen; i > 0; i--)
270     *pp++ ^= *pm++;
271 // Set the first byte to zero
272 padded->buffer[0] = 0x00;
273 Exit:
274     return retVal;
275 }

```

10.2.17.4.10 OaepDecode()

This function performs OAEP padding checking. The size of the buffer to receive the recovered data. If the padding is not valid, the *dSize* size is set to zero and the function returns TPM_RC_VALUE.

The *dSize* parameter is used as an input to indicate the size available in the buffer.

If insufficient space is available, the size is not changed and the return code is TPM_RC_VALUE.

Error Return	Meaning
TPM_RC_VALUE	the value to decode was larger than the modulus, or the padding is wrong or the buffer to receive the results is too small

```

276 static TPM_RC
277 OaepDecode(
278     TPM2B          *dataOut,           // OUT: the recovered data
279     TPM_ALG_ID     hashAlg,           // IN: algorithm to use for padding
280     const TPM2B    *label,            // IN: null-terminated string (may be NULL)
281     TPM2B          *padded,           // IN: the padded data
282 )
283 {
284     UINT32          i;
285     BYTE            seedMask[MAX_DIGEST_SIZE];
286     UINT32          hLen = CryptHashGetDigestSize(hashAlg);
287
288     BYTE            mask[MAX_RSA_KEY_BYTES];
289     BYTE            *pp;
290     BYTE            *pm;
291     TPM_RC          retVal = TPM_RC_SUCCESS;
292
293     // Strange size (anything smaller can't be an OAEP padded block)
294     // Also check for no leading 0
295     if((padded->size < (unsigned)((2 * hLen) + 2)) || (padded->buffer[0] != 0))
296         ERROR_RETURN(TPM_RC_VALUE);
297 // Use the hash size to determine what to put through MGF1 in order
298 // to recover the seedMask

```



```

299     CryptMGF_KDF(hLen, seedMask, hashAlg, padded->size - hLen - 1,
300                 &padded->buffer[hLen + 1], 0);
301
302     // Recover the seed into seedMask
303     pAssert(hLen <= sizeof(seedMask));
304     pp = &padded->buffer[1];
305     pm = seedMask;
306     for(i = hLen; i > 0; i--)
307         *pm++ ^= *pp++;
308
309     // Use the seed to generate the data mask
310     CryptMGF_KDF(padded->size - hLen - 1, mask, hashAlg, hLen, seedMask, 0);
311
312     // Use the mask generated from seed to recover the padded data
313     pp = &padded->buffer[hLen + 1];
314     pm = mask;
315     for(i = (padded->size - hLen - 1); i > 0; i--)
316         *pm++ ^= *pp++;
317
318     // Make sure that the recovered data has the hash of the label
319     // Put trial value in the seed mask
320     if((CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
321                       hLen, seedMask)) != hLen)
322         FAIL(FATAL_ERROR_INTERNAL);
323     if(memcmp(seedMask, mask, hLen) != 0)
324         ERROR_RETURN(TPM_RC_VALUE);
325
326     // find the start of the data
327     pm = &mask[hLen];
328     for(i = (UINT32)padded->size - (2 * hLen) - 1; i > 0; i--)
329     {
330         if(*pm++ != 0)
331             break;
332     }
333     // If we ran out of data or didn't end with 0x01, then return an error
334     if(i == 0 || pm[-1] != 0x01)
335         ERROR_RETURN(TPM_RC_VALUE);
336
337     // pm should be pointing at the first part of the data
338     // and i is one greater than the number of bytes to move
339     i--;
340     if(i > dataOut->size)
341         // Special exit to preserve the size of the output buffer
342         return TPM_RC_VALUE;
343     memcpy(dataOut->buffer, pm, i);
344     dataOut->size = (UINT16)i;
345 Exit:
346     if(retVal != TPM_RC_SUCCESS)
347         dataOut->size = 0;
348     return retVal;
349 }

```

10.2.17.4.11 PKCS1v1_5Encode()

This function performs the encoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

Error Return	Meaning
TPM_RC_VALUE	message size is too large

```

350 static TPM_RC
351 RSAES_PKCS1v1_5Encode(
352     TPM2B      *padded,           // OUT: the pad data
353     TPM2B      *message,         // IN: the message being padded

```

```

354     RAND_STATE *rand
355 )
356 {
357     UINT32      ps = padded->size - message->size - 3;
358     //
359     if(message->size > padded->size - 11)
360         return TPM_RC_VALUE;
361     // move the message to the end of the buffer
362     memcpy(&padded->buffer[padded->size - message->size], message->buffer,
363         message->size);
364     // Set the first byte to 0x00 and the second to 0x02
365     padded->buffer[0] = 0;
366     padded->buffer[1] = 2;
367
368     // Fill with random bytes
369     DRBG_Generate(rand, &padded->buffer[2], (UINT16)ps);
370     if(g_inFailureMode)
371         return TPM_RC_FAILURE;
372
373     // Set the delimiter for the random field to 0
374     padded->buffer[2 + ps] = 0;
375
376     // Now, the only messy part. Make sure that all the 'ps' bytes are non-zero
377     // In this implementation, use the value of the current index
378     for(ps++; ps > 1; ps--)
379     {
380         if(padded->buffer[ps] == 0)
381             padded->buffer[ps] = 0x55; // In the < 0.5% of the cases that the
382                                         // random value is 0, just pick a value to
383                                         // put into the spot.
384     }
385     return TPM_RC_SUCCESS;
386 }

```

10.2.17.4.12 RSAES_Decode()

This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

Error Return	Meaning
TPM_RC_FAIL	decoding error or results would no fit into provided buffer

```

387 static TPM_RC
388 RSAES_Decode(
389     TPM2B      *message, // OUT: the recovered message
390     TPM2B      *coded    // IN: the encoded message
391 )
392 {
393     BOOL      fail = FALSE;
394     UINT16     pSize;
395
396     fail = (coded->size < 11);
397     fail = (coded->buffer[0] != 0x00) | fail;
398     fail = (coded->buffer[1] != 0x02) | fail;
399     for(pSize = 2; pSize < coded->size; pSize++)
400     {
401         if(coded->buffer[pSize] == 0)
402             break;
403     }
404     pSize++;
405
406     // Make sure that pSize has not gone over the end and that there are at least 8
407     // bytes of pad data.
408     fail = (pSize > coded->size) | fail;

```

```

409     fail = ((pSize - 2) <= 8) | fail;
410     if((message->size < (UINT16)(coded->size - pSize)) || fail)
411         return TPM_RC_VALUE;
412     message->size = coded->size - pSize;
413     memcpy(message->buffer, &coded->buffer[pSize], coded->size - pSize);
414     return TPM_RC_SUCCESS;
415 }

```

10.2.17.4.13 CryptRsaPssSaltSize()

This function computes the salt size used in PSS. It is broken out so that the X509 code can get the same value that is used by the encoding function in this module.

```

416 INT16
417 CryptRsaPssSaltSize(
418     INT16      hashSize,
419     INT16      outSize
420 )
421 {
422     INT16      saltSize;
423     //
424     // (Mask Length) = (outSize - hashSize - 1);
425     // Max saltSize is (Mask Length) - 1
426     saltSize = (outSize - hashSize - 1) - 1;
427     // Use the maximum salt size allowed by FIPS 186-4
428     if(saltSize > hashSize)
429         saltSize = hashSize;
430     else if(saltSize < 0)
431         saltSize = 0;
432     return saltSize;
433 }

```

10.2.17.4.14 PssEncode()

This function creates an encoded block of data that is the size of modulus. The function uses the maximum salt size that will fit in the encoded block.

Returns TPM_RC_SUCCESS or goes into failure mode.

```

434 static TPM_RC
435 PssEncode(
436     TPM2B      *out,           // OUT: the encoded buffer
437     TPM_ALG_ID hashAlg,       // IN: hash algorithm for the encoding
438     TPM2B      *digest,       // IN: the digest
439     RAND_STATE *rand,         // IN: random number source
440 )
441 {
442     UINT32      hLen = CryptHashGetDigestSize(hashAlg);
443     BYTE        salt[MAX_RSA_KEY_BYTES - 1];
444     UINT16      saltSize;
445     BYTE        *ps = salt;
446     BYTE        *pOut;
447     UINT16      mLen;
448     HASH_STATE  hashState;
449
450     // These are fatal errors indicating bad TPM firmware
451     pAssert(out != NULL && hLen > 0 && digest != NULL);
452
453     // Get the size of the mask
454     mLen = (UINT16)(out->size - hLen - 1);
455
456     // Set the salt size
457     saltSize = CryptRsaPssSaltSize((INT16)hLen, (INT16)out->size);

```

```

458
459 //using eOut for scratch space
460 // Set the first 8 bytes to zero
461 pOut = out->buffer;
462 memset(pOut, 0, 8);
463
464 // Get set the salt
465 DRBG_Generate(rand, salt, saltSize);
466 if(g_inFailureMode)
467     return TPM_RC_FAILURE;
468
469 // Create the hash of the pad || input hash || salt
470 CryptHashStart(&hashState, hashAlg);
471 CryptDigestUpdate(&hashState, 8, pOut);
472 CryptDigestUpdate2B(&hashState, digest);
473 CryptDigestUpdate(&hashState, saltSize, salt);
474 CryptHashEnd(&hashState, hLen, &pOut[out->size - hLen - 1]);
475
476 // Create a mask
477 if(CryptMGF_KDF(mLen, pOut, hashAlg, hLen, &pOut[mLen], 0) != mLen)
478     FAIL(FATAL_ERROR_INTERNAL);
479
480 // Since this implementation uses key sizes that are all even multiples of
481 // 8, just need to make sure that the most significant bit is CLEAR
482 *pOut &= 0x7f;
483
484 // Before we mess up the pOut value, set the last byte to 0xbc
485 pOut[out->size - 1] = 0xbc;
486
487 // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
488 pOut = &pOut[mLen - saltSize - 1];
489 *pOut++ ^= 0x01;
490
491 // XOR the salt data into the buffer
492 for(; saltSize > 0; saltSize--)
493     *pOut++ ^= *ps++;
494
495 // and we are done
496 return TPM_RC_SUCCESS;
497 }

```

10.2.17.4.15 PssDecode()

This function checks that the PSS encoded block was built from the provided digest. If the check is successful, TPM_RC_SUCCESS is returned. Any other value indicates an error.

This implementation of PSS decoding is intended for the reference TPM implementation and is not at all generalized. It is used to check signatures over hashes and assumptions are made about the sizes of values. Those assumptions are enforced by this implementation. This implementation does allow for a variable size salt value to have been used by the creator of the signature.

Error Return	Meaning
TPM_RC_SCHEME	<i>hashAlg</i> is not a supported hash algorithm
TPM_RC_VALUE	decode operation failed

```

498 static TPM_RC
499 PssDecode(
500     TPM_ALG_ID    hashAlg,           // IN: hash algorithm to use for the encoding
501     TPM2B         *dIn,              // IN: the digest to compare
502     TPM2B         *eIn,              // IN: the encoded data
503 )
504 {

```

```

505     UINT32          hLen = CryptHashGetDigestSize(hashAlg);
506     BYTE            mask[MAX_RSA_KEY_BYTES];
507     BYTE            *pm = mask;
508     BYTE            *pe;
509     BYTE            pad[8] = {0};
510     UINT32          i;
511     UINT32          mLen;
512     BYTE            fail;
513     TPM_RC          retVal = TPM_RC_SUCCESS;
514     HASH_STATE      hashState;
515
516     // These errors are indicative of failures due to programmer error
517     pAssert(dIn != NULL && eIn != NULL);
518     pe = eIn->buffer;
519
520     // check the hash scheme
521     if(hLen == 0)
522         ERROR_RETURN(TPM_RC_SCHEME);
523
524     // most significant bit must be zero
525     fail = pe[0] & 0x80;
526
527     // last byte must be 0xbc
528     fail |= pe[eIn->size - 1] ^ 0xbc;
529
530     // Use the hLen bytes at the end of the buffer to generate a mask
531     // Doesn't start at the end which is a flag byte
532     mLen = eIn->size - hLen - 1;
533     CryptMGF_KDF(mLen, mask, hashAlg, hLen, &pe[mLen], 0);
534
535     // Clear the MS0 of the mask to make it consistent with the encoding.
536     mask[0] &= 0x7F;
537
538     pAssert(mLen <= sizeof(mask));
539     // XOR the data into the mask to recover the salt. This sequence
540     // advances eIn so that it will end up pointing to the seed data
541     // which is the hash of the signature data
542     for(i = mLen; i > 0; i--)
543         *pm++ ^= *pe++;
544
545     // Find the first byte of 0x01 after a string of all 0x00
546     for(pm = mask, i = mLen; i > 0; i--)
547     {
548         if(*pm == 0x01)
549             break;
550         else
551             fail |= *pm++;
552     }
553     // i should not be zero
554     fail |= (i == 0);
555
556     // if we have failed, will continue using the entire mask as the salt value so
557     // that the timing attacks will not disclose anything (I don't think that this
558     // is a problem for TPM applications but, usually, we don't fail so this
559     // doesn't cost anything).
560     if(fail)
561     {
562         i = mLen;
563         pm = mask;
564     }
565     else
566     {
567         pm++;
568         i--;
569     }
570     // i contains the salt size and pm points to the salt. Going to use the input

```

```

571 // hash and the seed to recreate the hash in the lower portion of eIn.
572 CryptHashStart(&hashState, hashAlg);
573
574 // add the pad of 8 zeros
575 CryptDigestUpdate(&hashState, 8, pad);
576
577 // add the provided digest value
578 CryptDigestUpdate(&hashState, dIn->size, dIn->buffer);
579
580 // and the salt
581 CryptDigestUpdate(&hashState, i, pm);
582
583 // get the result
584 fail |= (CryptHashEnd(&hashState, hLen, mask) != hLen);
585
586 // Compare all bytes
587 for(pm = mask; hLen > 0; hLen--)
588     // don't use fail = because that could skip the increment and compare
589     // operations after the first failure and that gives away timing
590     // information.
591     fail |= *pm++ ^ *pe++;
592
593 retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
594 Exit:
595     return retVal;
596 }

```

10.2.17.4.16 MakeDerTag()

Construct the DER value that is used in RSASSA

Return Value	Meaning
> 0	size of value
<= 0	no hash exists

```

597 INT16
598 MakeDerTag(
599     TPM_ALG_ID hashAlg,
600     INT16      sizeofBuffer,
601     BYTE       *buffer
602 )
603 {
604     // 0x30, 0x31, // SEQUENCE (2 elements) 1st
605     // 0x30, 0x0D, // SEQUENCE (2 elements)
606     // 0x06, 0x09, // HASH OID
607     // 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01,
608     // 0x05, 0x00, // NULL
609     // 0x04, 0x20 // OCTET STRING
610     HASH_DEF *info = CryptGetHashDef(hashAlg);
611     INT16 oidSize;
612     // If no OID, can't do encode
613     VERIFY(info != NULL);
614     oidSize = 2 + (info->OID)[1];
615     // make sure this fits in the buffer
616     VERIFY(sizeofBuffer >= (oidSize + 8));
617     *buffer++ = 0x30; // 1st SEQUENCE
618     // Size of the 1st SEQUENCE is 6 bytes + size of the hash OID + size of the
619     // digest size
620     *buffer++ = (BYTE)(6 + oidSize + info->digestSize); //
621     *buffer++ = 0x30; // 2nd SEQUENCE
622     // size is 4 bytes of overhead plus the side of the OID
623     *buffer++ = (BYTE)(2 + oidSize);

```

```

624     MemoryCopy(buffer, info->OID, oidSize);
625     buffer += oidSize;
626     *buffer++ = 0x05;    // Add a NULL
627     *buffer++ = 0x00;
628
629     *buffer++ = 0x04;
630     *buffer++ = (BYTE)(info->digestSize);
631     return oidSize + 8;
632 Error:
633     return 0;
634
635 }

```

10.2.17.4.17 RSASSA_Encode()

Encode a message using PKCS1v1.5 method.

Error Return	Meaning
TPM_RC_SCHEME	<i>hashAlg</i> is not a supported hash algorithm
TPM_RC_SIZE	<i>eOutSize</i> is not large enough
TPM_RC_VALUE	<i>hInSize</i> does not match the digest size of <i>hashAlg</i>

```

636 static TPM_RC
637 RSASSA_Encode(
638     TPM2B          *pOut,          // IN:OUT on in, the size of the public key
639                                     // on out, the encoded area
640     TPM_ALG_ID     hashAlg,       // IN: hash algorithm for PKCS1v1_5
641     TPM2B          *hIn           // IN: digest value to encode
642 )
643 {
644     BYTE            DER[20];
645     BYTE            *der = DER;
646     INT32           derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
647     BYTE            *eOut;
648     INT32           fillSize;
649     TPM_RC          retVal = TPM_RC_SUCCESS;
650
651     // Can't use this scheme if the algorithm doesn't have a DER string defined.
652     if(derSize == 0)
653         ERROR_RETURN(TPM_RC_SCHEME);
654
655     // If the digest size of 'hashAlg' doesn't match the input digest size, then
656     // the DER will misidentify the digest so return an error
657     if(CryptHashGetDigestSize(hashAlg) != hIn->size)
658         ERROR_RETURN(TPM_RC_VALUE);
659     fillSize = pOut->size - derSize - hIn->size - 3;
660     eOut = pOut->buffer;
661
662     // Make sure that this combination will fit in the provided space
663     if(fillSize < 8)
664         ERROR_RETURN(TPM_RC_SIZE);
665
666     // Start filling
667     *eOut++ = 0; // initial byte of zero
668     *eOut++ = 1; // byte of 0x01
669     for(; fillSize > 0; fillSize--)
670         *eOut++ = 0xff; // bunch of 0xff
671     *eOut++ = 0; // another 0
672     for(; derSize > 0; derSize--)
673         *eOut++ = *der++; // copy the DER
674     der = hIn->buffer;
675     for(fillSize = hIn->size; fillSize > 0; fillSize--)

```



```

676         *eOut++ = *der++;    // copy the hash
677 Exit:
678     return retVal;
679 }

```

10.2.17.4.18 RSASSA_Decode()

This function performs the RSASSA decoding of a signature.

Error Return	Meaning
TPM_RC_VALUE	decode unsuccessful
TPM_RC_SCHEME	hasAlg is not supported

```

680 static TPM_RC
681 RSASSA_Decode(
682     TPM_ALG_ID      hashAlg,          // IN: hash algorithm to use for the encoding
683     TPM2B           *hIn,             // IN: the digest to compare
684     TPM2B           *eIn              // IN: the encoded data
685 )
686 {
687     BYTE            fail;
688     BYTE            DER[20];
689     BYTE            *der = DER;
690     INT32           derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
691     BYTE            *pe;
692     INT32           hashSize = CryptHashGetDigestSize(hashAlg);
693     INT32           fillSize;
694     TPM_RC          retVal;
695     BYTE            *digest;
696     UINT16          digestSize;
697
698     pAssert(hIn != NULL && eIn != NULL);
699     pe = eIn->buffer;
700
701     // Can't use this scheme if the algorithm doesn't have a DER string
702     // defined or if the provided hash isn't the right size
703     if(derSize == 0 || (unsigned)hashSize != hIn->size)
704         ERROR_RETURN(TPM_RC_SCHEME);
705
706     // Make sure that this combination will fit in the provided space
707     // Since no data movement takes place, can just walk through this
708     // and accept nearly random values. This can only be called from
709     // CryptValidateSignature() so eInSize is known to be in range.
710     fillSize = eIn->size - derSize - hashSize - 3;
711
712     // Start checking (fail will become non-zero if any of the bytes do not have
713     // the expected value.
714     fail = *pe++;                      // initial byte of zero
715     fail |= *pe++ ^ 1;                 // byte of 0x01
716     for(; fillSize > 0; fillSize--)
717         fail |= *pe++ ^ 0xff;          // bunch of 0xff
718     fail |= *pe++;                     // another 0
719     for(; derSize > 0; derSize--)
720         fail |= *pe++ ^ *der++;        // match the DER
721     digestSize = hIn->size;
722     digest = hIn->buffer;
723     for(; digestSize > 0; digestSize--)
724         fail |= *pe++ ^ *digest++;     // match the hash
725     retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
726 Exit:
727     return retVal;
728 }

```

10.2.17.5 Externally Accessible Functions

10.2.17.5.1 CryptRsaSelectScheme()

This function is used by TPM2_RSA_Decrypt and TPM2_RSA_Encrypt. It sets up the rules to select a scheme between input and object default. This function assume the RSA object is loaded. If a default scheme is defined in object, the default scheme should be chosen, otherwise, the input scheme should be chosen. In the case that both the object and *scheme* are not TPM_ALG_NULL, then if the schemes are the same, the input scheme will be chosen. if the scheme are not compatible, a NULL pointer will be returned. The return pointer may point to a TPM_ALG_NULL scheme.

```

729 TPMT_RSA_DECRYPT*
730 CryptRsaSelectScheme(
731     TPMI_DH_OBJECT    rsaHandle,    // IN: handle of an RSA key
732     TPMT_RSA_DECRYPT   *scheme      // IN: a sign or decrypt scheme
733 )
734 {
735     OBJECT             *rsaObject;
736     TPMT_ASYM_SCHEME   *keyScheme;
737     TPMT_RSA_DECRYPT   *retVal = NULL;
738
739     // Get sign object pointer
740     rsaObject = HandleToObject(rsaHandle);
741     keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
742
743     // if the default scheme of the object is TPM_ALG_NULL, then select the
744     // input scheme
745     if(keyScheme->scheme == TPM_ALG_NULL)
746     {
747         retVal = scheme;
748     }
749     // if the object scheme is not TPM_ALG_NULL and the input scheme is
750     // TPM_ALG_NULL, then select the default scheme of the object.
751     else if(scheme->scheme == TPM_ALG_NULL)
752     {
753         // if input scheme is NULL
754         retVal = (TPMT_RSA_DECRYPT *)keyScheme;
755     }
756     // get here if both the object scheme and the input scheme are
757     // not TPM_ALG_NULL. Need to insure that they are the same.
758     // IMPLEMENTATION NOTE: This could cause problems if future versions have
759     // schemes that have more values than just a hash algorithm. A new function
760     // (IsSchemeSame()) might be needed then.
761     else if(keyScheme->scheme == scheme->scheme
762             && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
763     {
764         retVal = scheme;
765     }
766     // two different, incompatible schemes specified will return NULL
767     return retVal;
768 }

```

10.2.17.5.2 CryptRsaLoadPrivateExponent()

This function is called to generate the private exponent of an RSA key.

Error Return	Meaning
TPM_RC_BINDING	public and private parts of <i>rsaKey</i> are not matched

```

769 TPM_RC
770 CryptRsaLoadPrivateExponent(

```

```

771     TPMT_PUBLIC                *publicArea,
772     TPMT_SENSITIVE             *sensitive
773 )
774 {
775     //
776     if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) == 0)
777     {
778         if((sensitive->sensitive.rsa.t.size * 2) == publicArea->unique.rsa.t.size)
779         {
780             NEW_PRIVATE_EXPONENT(Z);
781             BN_RSA_INITIALIZED(bnN, &publicArea->unique.rsa);
782             BN_RSA(bnQr);
783             BN_VAR(bnE, RADIX_BITS);
784
785             TEST(TPM_ALG_NULL);
786
787             VERIFY((sensitive->sensitive.rsa.t.size * 2)
788                 == publicArea->unique.rsa.t.size);
789             // Initialize the exponent
790             BnSetWord(bnE, publicArea->parameters.rsaDetail.exponent);
791             if(BnEqualZero(bnE))
792                 BnSetWord(bnE, RSA_DEFAULT_PUBLIC_EXPONENT);
793             // Convert first prime to 2B
794             VERIFY(BnFrom2B(Z->P, &sensitive->sensitive.rsa.b) != NULL);
795
796             // Find the second prime by division. This uses 'bQ' rather than Z->Q
797             // because the division could make the quotient larger than a prime during
798             // some intermediate step.
799             VERIFY(BnDiv(Z->Q, bnQr, bnN, Z->P));
800             VERIFY(BnEqualZero(bnQr));
801             // Compute the private exponent and return it if found
802             VERIFY(ComputePrivateExponent(bnE, Z));
803             VERIFY(PackExponent(&sensitive->sensitive.rsa, Z));
804         }
805         else
806             VERIFY(((sensitive->sensitive.rsa.t.size / 5) * 2)
807                 == publicArea->unique.rsa.t.size);
808         sensitive->sensitive.rsa.t.size |= RSA_prime_flag;
809     }
810     return TPM_RC_SUCCESS;
811 Error:
812     return TPM_RC_BINDING;
813 }

```

10.2.17.5.3 CryptRsaEncrypt()

This is the entry point for encryption using RSA. Encryption is use of the public exponent. The padding parameter determines what padding will be used.

The *cOutSize* parameter must be at least as large as the size of the key.

If the padding is RSA_PAD_NONE, *dIn* is treated as a number. It must be lower in value than the key modulus.

NOTE

If *dIn* has fewer bytes than *cOut*, then we don't add low-order zeros to *dIn* to make it the size of the RSA key for the call to RSAEP. This is because the high order bytes of *dIn* might have a numeric value that is greater than the value of the key modulus. If this had low-order zeros added, it would have a numeric value larger than the modulus even though it started out with a lower numeric value.

Error Return	Meaning
TPM_RC_VALUE	<i>cOutSize</i> is too small (must be the size of the modulus)
TPM_RC_SCHEME	<i>padType</i> is not a supported scheme

```

814 LIB_EXPORT TPM_RC
815 CryptRsaEncrypt(
816     TPM2B_PUBLIC_KEY_RSA    *cOut,           // OUT: the encrypted data
817     TPM2B                    *dIn,           // IN: the data to encrypt
818     OBJECT                   *key,           // IN: the key used for encryption
819     TPMT_RSA_DECRYPT          *scheme,        // IN: the type of padding and hash
820                                     // if needed
821     const TPM2B              *label,         // IN: in case it is needed
822     RAND_STATE               *rand           // IN: random number generator
823                                     // state (mostly for testing)
824 )
825 {
826     TPM_RC retVal = TPM_RC_SUCCESS;
827     TPM2B_PUBLIC_KEY_RSA dataIn;
828     //
829     // if the input and output buffers are the same, copy the input to a scratch
830     // buffer so that things don't get messed up.
831     if(dIn == &cOut->b)
832     {
833         MemoryCopy2B(&dataIn.b, dIn, sizeof(dataIn.t.buffer));
834         dIn = &dataIn.b;
835     }
836     // All encryption schemes return the same size of data
837     cOut->t.size = key->publicArea.unique.rsa.t.size;
838     TEST(scheme->scheme);
839
840     switch(scheme->scheme)
841     {
842     case TPM_ALG_NULL: // 'raw' encryption
843     {
844         INT32 i;
845         INT32 dSize = dIn->size;
846         // dIn can have more bytes than cOut as long as the extra bytes
847         // are zero. Note: the more significant bytes of a number in a byte
848         // buffer are the bytes at the start of the array.
849         for(i = 0; (i < dSize) && (dIn->buffer[i] == 0); i++);
850         dSize -= i;
851         if(dSize > cOut->t.size)
852             ERROR_RETURN(TPM_RC_VALUE);
853         // Pad cOut with zeros if dIn is smaller
854         memset(cOut->t.buffer, 0, cOut->t.size - dSize);
855         // And copy the rest of the value
856         memcpy(&cOut->t.buffer[cOut->t.size - dSize], &dIn->buffer[i], dSize);
857
858         // If the size of dIn is the same as cOut dIn could be larger than
859         // the modulus. If it is, then RSAEP() will catch it.
860     }
861     break;
862     case TPM_ALG_RSAES:
863         retVal = RSAES_PKCS1v1_5Encode(&cOut->b, dIn, rand);
864         break;
865     case TPM_ALG_OAEP:
866         retVal = OaepEncode(&cOut->b, scheme->details.oaep.hashAlg, label, dIn,
867                             rand);

```

```

868         break;
869     default:
870         ERROR_RETURN(TPM_RC_SCHEME);
871         break;
872     }
873     // All the schemes that do padding will come here for the encryption step
874     // Check that the Encoding worked
875     if(retVal == TPM_RC_SUCCESS)
876         // Padding OK so do the encryption
877         retVal = RSAEP(&cOut->b, key);
878 Exit:
879     return retVal;
880 }

```

10.2.17.5.4 CryptRsaDecrypt()

This is the entry point for decryption using RSA. Decryption is use of the private exponent. The *padType* parameter determines what padding was used.

Error Return	Meaning
TPM_RC_SIZE	<i>cInSize</i> is not the same as the size of the public modulus of <i>key</i> , or numeric value of the encrypted data is greater than the modulus
TPM_RC_VALUE	<i>dOutSize</i> is not large enough for the result
TPM_RC_SCHEME	<i>padType</i> is not supported

```

881 LIB_EXPORT TPM_RC
882 CryptRsaDecrypt(
883     TPM2B          *dOut,           // OUT: the decrypted data
884     TPM2B          *cIn,           // IN: the data to decrypt
885     OBJECT         *key,           // IN: the key to use for decryption
886     TPMT_RSA_DECRYPT *scheme,       // IN: the padding scheme
887     const TPM2B    *label          // IN: in case it is needed for the scheme
888 )
889 {
890     TPM_RC      retVal;
891
892     // Make sure that the necessary parameters are provided
893     pAssert(cIn != NULL && dOut != NULL && key != NULL);
894
895     // Size is checked to make sure that the encrypted value is the right size
896     if(cIn->size != key->publicArea.unique.rsa.t.size)
897         ERROR_RETURN(TPM_RC_SIZE);
898
899     TEST(scheme->scheme);
900
901     // For others that do padding, do the decryption in place and then
902     // go handle the decoding.
903     retVal = RSADP(cIn, key);
904     if(retVal == TPM_RC_SUCCESS)
905     {
906         // Remove padding
907         switch(scheme->scheme)
908         {
909             case TPM_ALG_NULL:
910                 if(dOut->size < cIn->size)
911                     return TPM_RC_VALUE;
912                 MemoryCopy2B(dOut, cIn, dOut->size);
913                 break;
914             case TPM_ALG_RSAES:
915                 retVal = RSAES_Decode(dOut, cIn);
916                 break;

```

```

917         case TPM_ALG_OAEP:
918             retVal = OaepDecode(dOut, scheme->details.oaep.hashAlg, label, cIn);
919             break;
920         default:
921             retVal = TPM_RC_SCHEME;
922             break;
923     }
924 }
925 Exit:
926     return retVal;
927 }

```

10.2.17.5.5 CryptRsaSign()

This function is used to generate an RSA signature of the type indicated in *scheme*.

Error Return	Meaning
TPM_RC_SCHEME	<i>scheme</i> or <i>hashAlg</i> are not supported
TPM_RC_VALUE	<i>hInSize</i> does not match <i>hashAlg</i> (for RSASSA)

```

928 LIB_EXPORT TPM_RC
929 CryptRsaSign(
930     TPMT_SIGNATURE    *sigOut,
931     OBJECT             *key,           // IN: key to use
932     TPM2B_DIGEST       *hIn,          // IN: the digest to sign
933     RAND_STATE         *rand,         // IN: the random number generator
934                                     // to use (mostly for testing)
935 )
936 {
937     TPM_RC             retVal = TPM_RC_SUCCESS;
938     UINT16             modSize;
939
940     // parameter checks
941     pAssert(sigOut != NULL && key != NULL && hIn != NULL);
942
943     modSize = key->publicArea.unique.rsa.t.size;
944
945     // for all non-null signatures, the size is the size of the key modulus
946     sigOut->signature.rsapss.sig.t.size = modSize;
947
948     TEST(sigOut->sigAlg);
949
950     switch(sigOut->sigAlg)
951     {
952         case TPM_ALG_NULL:
953             sigOut->signature.rsapss.sig.t.size = 0;
954             return TPM_RC_SUCCESS;
955         case TPM_ALG_RSAPSS:
956             retVal = PssEncode(&sigOut->signature.rsapss.sig.b,
957                               sigOut->signature.rsapss.hash, &hIn->b, rand);
958             break;
959         case TPM_ALG_RSASSA:
960             retVal = RSASSA_Encode(&sigOut->signature.rsassa.sig.b,
961                                    sigOut->signature.rsassa.hash, &hIn->b);
962             break;
963         default:
964             retVal = TPM_RC_SCHEME;
965     }
966     if(retVal == TPM_RC_SUCCESS)
967     {
968         // Do the encryption using the private key
969         retVal = RSADP(&sigOut->signature.rsapss.sig.b, key);

```

```

970     }
971     return retVal;
972 }

```

10.2.17.5.6 CryptRsaValidateSignature()

This function is used to validate an RSA signature. If the signature is valid TPM_RC_SUCCESS is returned. If the signature is not valid, TPM_RC_SIGNATURE is returned. Other return codes indicate either parameter problems or fatal errors.

Error Return	Meaning
TPM_RC_SIGNATURE	the signature does not check
TPM_RC_SCHEME	unsupported scheme or hash algorithm

```

973 LIB_EXPORT TPM_RC
974 CryptRsaValidateSignature(
975     TPMT_SIGNATURE *sig,           // IN: signature
976     OBJECT *key,                  // IN: public modulus
977     TPM2B_DIGEST *digest          // IN: The digest being validated
978 )
979 {
980     TPM_RC      retVal;
981     //
982     // Fatal programming errors
983     pAssert(key != NULL && sig != NULL && digest != NULL);
984     switch(sig->sigAlg)
985     {
986         case TPM_ALG_RSAPSS:
987         case TPM_ALG_RSASSA:
988             break;
989         default:
990             return TPM_RC_SCHEME;
991     }
992     // Errors that might be caused by calling parameters
993     if(sig->signature.rsassa.sig.t.size != key->publicArea.unique.rsa.t.size)
994         ERROR_RETURN(TPM_RC_SIGNATURE);
995
996     TEST(sig->sigAlg);
997
998     // Decrypt the block
999     retVal = RSAEP(&sig->signature.rsassa.sig.b, key);
1000     if(retVal == TPM_RC_SUCCESS)
1001     {
1002         switch(sig->sigAlg)
1003         {
1004             case TPM_ALG_RSAPSS:
1005                 retVal = PssDecode(sig->signature.any.hashAlg, &digest->b,
1006                                     &sig->signature.rsassa.sig.b);
1007                 break;
1008             case TPM_ALG_RSASSA:
1009                 retVal = RSASSA_Decode(sig->signature.any.hashAlg, &digest->b,
1010                                         &sig->signature.rsassa.sig.b);
1011                 break;
1012             default:
1013                 return TPM_RC_SCHEME;
1014         }
1015     }
1016     Exit:
1017     return (retVal != TPM_RC_SUCCESS) ? TPM_RC_SIGNATURE : TPM_RC_SUCCESS;
1018 }
1019 #if SIMULATION && USE_RSA_KEY_CACHE
1020 extern int s_rsaKeyCacheEnabled;

```



```

1021 int GetCachedRsaKey(TPMT_PUBLIC *publicArea, TPMT_SENSITIVE *sensitive,
1022                    RAND_STATE *rand);
1023 #define GET_CACHED_KEY(publicArea, sensitive, rand) \
1024     (s_rsaKeyCacheEnabled && GetCachedRsaKey(publicArea, sensitive, rand))
1025 #else
1026 #define GET_CACHED_KEY(key, rand)
1027 #endif

```

10.2.17.5.7 CryptRsaGenerateKey()

Generate an RSA key from a provided seed

Error Return	Meaning
TPM_RC_CANCELED	operation was canceled
TPM_RC_RANGE	public exponent is not supported
TPM_RC_VALUE	could not find a prime using the provided parameters

```

1028 LIB_EXPORT TPM_RC
1029 CryptRsaGenerateKey(
1030     TPMT_PUBLIC      *publicArea,
1031     TPMT_SENSITIVE   *sensitive,
1032     RAND_STATE       *rand           // IN: if not NULL, the deterministic
1033                                     // RNG state
1034 )
1035 {
1036     UINT32 i;
1037     BN_RSA(bnD);
1038     BN_RSA(bnN);
1039     BN_WORD(bnPubExp);
1040     UINT32 e = publicArea->parameters.rsaDetail.exponent;
1041     int keySizeInBits;
1042     TPM_RC retVal = TPM_RC_NO_RESULT;
1043     NEW_PRIVATE_EXPONENT(Z);
1044     //
1045     // Need to make sure that the caller did not specify an exponent that is
1046     // not supported
1047     e = publicArea->parameters.rsaDetail.exponent;
1048     if(e == 0)
1049         e = RSA_DEFAULT_PUBLIC_EXPONENT;
1050     else
1051     {
1052         if(e < 65537)
1053             ERROR_RETURN(TPM_RC_RANGE);
1054         // Check that e is prime
1055         if(!IsPrimeInt(e))
1056             ERROR_RETURN(TPM_RC_RANGE);
1057     }
1058     BnSetWord(bnPubExp, e);
1059     // check for supported key size.
1060     keySizeInBits = publicArea->parameters.rsaDetail.keyBits;
1061     if(((keySizeInBits % 1024) != 0)
1062        || (keySizeInBits > MAX_RSA_KEY_BITS) // this might be redundant, but...
1063        || (keySizeInBits == 0))
1064         ERROR_RETURN(TPM_RC_VALUE);
1065     // Set the prime size for instrumentation purposes
1066     INSTRUMENT_SET(PrimeIndex, PRIME_INDEX(keySizeInBits / 2));
1067     #if SIMULATION && USE_RSA_KEY_CACHE
1068         if(GET_CACHED_KEY(publicArea, sensitive, rand))

```

```

1073     return TPM_RC_SUCCESS;
1074 #endif
1075
1076     // Make sure that key generation has been tested
1077     TEST(TPM_ALG_NULL);
1078
1079     // The prime is computed in P. When a new prime is found, Q is checked to
1080     // see if it is zero. If so, P is copied to Q and a new P is found.
1081     // When both P and Q are non-zero, the modulus and
1082     // private exponent are computed and a trial encryption/decryption is
1083     // performed. If the encrypt/decrypt fails, assume that at least one of the
1084     // primes is composite. Since we don't know which one, set Q to zero and start
1085     // over and find a new pair of primes.
1086
1087     for(i = 1; (retVal == TPM_RC_NO_RESULT) && (i != 100); i++)
1088     {
1089         if(_plat_IsCanceled())
1090             ERROR_RETURN(TPM_RC_CANCELED);
1091
1092         if(BnGeneratePrimeForRSA(Z->P, keySizeInBits / 2, e, rand) == TPM_RC_FAILURE)
1093         {
1094             retVal = TPM_RC_FAILURE;
1095             goto Exit;
1096         }
1097         INSTRUMENT_INC(PrimeCounts[PrimeIndex]);
1098
1099         // If this is the second prime, make sure that it differs from the
1100         // first prime by at least 2^100
1101         if(BnEqualZero(Z->Q))
1102         {
1103             // copy p to q and compute another prime in p
1104             BnCopy(Z->Q, Z->P);
1105             continue;
1106         }
1107         // Make sure that the difference is at least 100 bits. Need to do it this
1108         // way because the big numbers are only positive values
1109         if(BnUnsignedCmp(Z->P, Z->Q) < 0)
1110             BnSub(bnD, Z->Q, Z->P);
1111         else
1112             BnSub(bnD, Z->P, Z->Q);
1113         if(BnMsb(bnD) < 100)
1114             continue;
1115
1116         //Form the public modulus and set the unique value
1117         BnMult(bnN, Z->P, Z->Q);
1118         BnTo2B(bnN, &publicArea->unique.rsa.b,
1119             (NUMBYTES)BITS_TO_BYTES(keySizeInBits));
1120         // Make sure everything came out right. The MSb of the values must be one
1121         if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1122             || (publicArea->unique.rsa.t.size
1123                 != (NUMBYTES)BITS_TO_BYTES(keySizeInBits)))
1124             FAIL(FATAL_ERROR_INTERNAL);
1125
1126         // Make sure that we can form the private exponent values
1127         if(ComputePrivateExponent(bnPubExp, Z) != TRUE)
1128         {
1129             // If ComputePrivateExponent could not find an inverse for
1130             // Q, then copy P and recompute P. This might
1131             // cause both to be recomputed if P is also zero
1132             if(BnEqualZero(Z->Q))
1133                 BnCopy(Z->Q, Z->P);
1134             continue;
1135         }
1136         // Pack the private exponent into the sensitive area
1137         PackExponent(&sensitive->sensitive.rsa, Z);
1138         // Make sure everything came out right. The MSb of the values must be one

```

```
1139     if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1140         || ((sensitive->sensitive.rsa.t.buffer[0] & 0x80) == 0))
1141         FAIL(FATAL_ERROR_INTERNAL);
1142
1143     retVal = TPM_RC_SUCCESS;
1144     // Do a trial encryption decryption if this is a signing key
1145     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
1146     {
1147         BN_RSA(temp1);
1148         BN_RSA(temp2);
1149         BnGenerateRandomInRange(temp1, bnN, rand);
1150
1151         // Encrypt with public exponent...
1152         BnModExp(temp2, temp1, bnPubExp, bnN);
1153         // ... then decrypt with private exponent
1154         RsaPrivateKeyOp(temp2, Z);
1155
1156         // If the starting and ending values are not the same,
1157         // start over -;
1158         if(BnUnsignedCmp(temp2, temp1) != 0)
1159         {
1160             BnSetWord(Z->Q, 0);
1161             retVal = TPM_RC_NO_RESULT;
1162         }
1163     }
1164 }
1165 Exit:
1166     return retVal;
1167 }
1168 #endif // ALG_RSA
```

10.2.18 CryptSmac.c

10.2.18.1 Introduction

This file contains the implementation of the message authentication codes based on a symmetric block cipher. These functions only use the single block encryption functions of the selected symmetric cryptographic library.

10.2.18.2 Includes, Defines, and Typedefs

```
1  #define _CRYPT_HASH_C_
2  #include "Tpm.h"
3
4  #if SMAC_IMPLEMENTED
```

10.2.18.2.1 CryptSmacStart()

Function to start an SMAC.

```
5  UINT16
6  CryptSmacStart(
7      HASH_STATE          *state,
8      TPMU_PUBLIC_PARMS  *keyParameters,
9      TPM_ALG_ID          macAlg,      // IN: the type of MAC
10     TPM2B                *key
11 )
12 {
13     UINT16      retVal = 0;
14     //
15     // Make sure that the key size is correct. This should have been checked
16     // at key load, but...
17     if(BITS_TO_BYTES(keyParameters->symDetail.sym.keyBits.sym) == key->size)
18     {
19         switch(macAlg)
20         {
21             #if ALG_CMIC
22             case TPM_ALG_CMIC:
23                 retVal = CryptCmicStart(&state->state.smac, keyParameters,
24                                         macAlg, key);
25                 break;
26             #endif
27             default:
28                 break;
29         }
30     }
31     state->type = (retVal != 0) ? HASH_STATE_SMAC : HASH_STATE_EMPTY;
32     return retVal;
33 }
```

10.2.18.2.2 CryptMacStart()

Function to start either an HMAC or an SMAC. Cannot reuse the CryptHmacStart() function because of the difference in number of parameters.

```
34  UINT16
35  CryptMacStart(
36      HMAC_STATE          *state,
37      TPMU_PUBLIC_PARMS  *keyParameters,
38      TPM_ALG_ID          macAlg,      // IN: the type of MAC
```

```

39     TPM2B                *key
40 )
41 {
42     MemorySet(state, 0, sizeof(HMAC_STATE));
43     if(CryptHashIsValidAlg(macAlg, FALSE))
44     {
45         return CryptHmacStart(state, macAlg, key->size, key->buffer);
46     }
47     else if(CryptSmacIsValidAlg(macAlg, FALSE))
48     {
49         return CryptSmacStart(&state->hashState, keyParameters, macAlg, key);
50     }
51     else
52         return 0;
53 }

```

10.2.18.2.3 CryptMacEnd()

Dispatch to the MAC end function using a size and buffer pointer.

```

54  UINT16
55  CryptMacEnd(
56      HMAC_STATE      *state,
57      UINT32          size,
58      BYTE            *buffer
59  )
60  {
61      UINT16          retVal = 0;
62      if(state->hashState.type == HASH_STATE_SMAC)
63          retVal = (state->hashState.state.smac.smacMethods.end) (
64              &state->hashState.state.smac.state, size, buffer);
65      else if(state->hashState.type == HASH_STATE_HMAC)
66          retVal = CryptHmacEnd(state, size, buffer);
67      state->hashState.type = HASH_STATE_EMPTY;
68      return retVal;
69  }

```

10.2.18.2.4 CryptMacEnd2B()

Dispatch to the MAC end function using a 2B.

```

70  UINT16
71  CryptMacEnd2B (
72      HMAC_STATE      *state,
73      TPM2B           *data
74  )
75  {
76      return CryptMacEnd(state, data->size, data->buffer);
77  }
78  #endif // SMAC_IMPLEMENTED

```

10.2.19 CryptSym.c

10.2.19.1 Introduction

This file contains the implementation of the symmetric block cipher modes allowed for a TPM. These functions only use the single block encryption functions of the selected symmetric crypto library.

10.2.19.2 Includes, Defines, and Typedefs

```

1  #include "Tpm.h"
2
3  #include "CryptSym.h"
4
5  #define      KEY_BLOCK_SIZES(ALG, alg)
6  static const INT16      alg##_KeyBlockSizes[] = {
7                          ALG##_KEY_SIZES_BITS, -1, ALG##_BLOCK_SIZES };
8
9  FOR_EACH_SYM(KEY_BLOCK_SIZES)

```

10.2.19.3 Initialization and Data Access Functions

10.2.19.3.1 CryptSymInit()

This function is called to do _TPM_Init processing

```

10  BOOL
11  CryptSymInit(
12      void
13  )
14  {
15      return TRUE;
16  }

```

10.2.19.3.2 CryptSymStartup()

This function is called to do TPM2_Startup() processing

```

17  BOOL
18  CryptSymStartup(
19      void
20  )
21  {
22      return TRUE;
23  }

```

10.2.19.3.3 CryptGetSymmetricBlockSize()

This function returns the block size of the algorithm. The table of bit sizes has an entry for each allowed key size. The entry for a key size is 0 if the TPM does not implement that key size. The key size table is delimited with a negative number (-1). After the delimiter is a list of block sizes with each entry corresponding to the key bit size. For most symmetric algorithms, the block size is the same regardless of the key size but this arrangement allows them to be different.

Return Value	Meaning
≤ 0	cipher not supported
> 0	the cipher block size in bytes

```

24  LIB_EXPORT INT16
25  CryptGetSymmetricBlockSize(
26      TPM_ALG_ID      symmetricAlg,    // IN: the symmetric algorithm
27      UINT16          keySizeInBits    // IN: the key size
28  )
29  {
30      const INT16      *sizes;
31      INT16            i;
32      #define ALG_CASE(SYM, sym) case TPM_ALG_##SYM: sizes = sym##KeyBlockSizes; break
33      switch(symmetricAlg)
34      {
35          #define GET_KEY_BLOCK_POINTER(SYM, sym)
36          case TPM_ALG_##SYM:
37              sizes = sym##KeyBlockSizes;
38              break;
39              // Get the pointer to the block size array
40              FOR_EACH_SYM(GET_KEY_BLOCK_POINTER);
41
42          default:
43              return 0;
44      }
45      // Find the index of the indicated keySizeInBits
46      for(i = 0; *sizes >= 0; i++, sizes++)
47      {
48          if(*sizes == keySizeInBits)
49              break;
50      }
51      // If sizes is pointing at the end of the list of key sizes, then the desired
52      // key size was not found so set the block size to zero.
53      if(*sizes++ < 0)
54          return 0;
55      // Advance until the end of the list is found
56      while(*sizes++ >= 0);
57      // sizes is pointing to the first entry in the list of block sizes. Use the
58      // ith index to find the block size for the corresponding key size.
59      return sizes[i];
60  }

```

10.2.19.4 Symmetric Encryption

This function performs symmetric encryption based on the mode.

Error Return	Meaning
TPM_RC_SIZE	dSize is not a multiple of the block size for an algorithm that requires it
TPM_RC_FAILURE	Fatal error

```

61  LIB_EXPORT TPM_RC
62  CryptSymmetricEncrypt(
63      BYTE             *dOut,           // OUT:
64      TPM_ALG_ID      algorithm,       // IN: the symmetric algorithm
65      UINT16          keySizeInBits,   // IN: key size in bits
66      const BYTE      *key,           // IN: key buffer. The size of this buffer
67                                      // in bytes is (keySizeInBits + 7) / 8
68      TPM2B_IV        *ivInOut,       // IN/OUT: IV for decryption.
69      TPM_ALG_ID      mode,           // IN: Mode to use

```



```

70     INT32                dSize,                // IN: data size (may need to be a
71                                     //      multiple of the blockSize)
72     const BYTE           *dIn                  // IN: data buffer
73 )
74 {
75     BYTE                 *pIv;
76     int                  i;
77     BYTE                 tmp[MAX_SYM_BLOCK_SIZE];
78     BYTE                 *pT;
79     tpmCryptKeySchedule_t keySchedule;
80     INT16                 blockSize;
81     TpmCryptSetSymKeyCall_t encrypt;
82     BYTE                 *iv;
83     BYTE                 defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
84 //
85     pAssert(dOut != NULL && key != NULL && dIn != NULL);
86     if(dSize == 0)
87         return TPM_RC_SUCCESS;
88
89     TEST(algorithm);
90     blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
91     if(blockSize == 0)
92         return TPM_RC_FAILURE;
93     // If the iv is provided, then it is expected to be block sized. In some cases,
94     // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
95     // with no knowledge of the actual block size. This function will set it.
96     if((ivInOut != NULL) && (mode != TPM_ALG_ECB))
97     {
98         ivInOut->t.size = blockSize;
99         iv = ivInOut->t.buffer;
100     }
101     else
102         iv = defaultIv;
103     pIv = iv;
104
105     // Create encrypt key schedule and set the encryption function pointer.
106     switch (algorithm)
107     {
108         FOR_EACH_SYM(ENCRYPT_CASE)
109
110         default:
111             return TPM_RC_SYMMETRIC;
112     }
113     switch(mode)
114     {
115 #if ALG_CTR
116         case TPM_ALG_CTR:
117             for(; dSize > 0; dSize -= blockSize)
118             {
119                 // Encrypt the current value of the IV(counter)
120                 ENCRYPT(&keySchedule, iv, tmp);
121
122                 //increment the counter (counter is big-endian so start at end)
123                 for(i = blockSize - 1; i >= 0; i--)
124                     if((iv[i] += 1) != 0)
125                         break;
126
127                 // XOR the encrypted counter value with input and put into output
128                 pT = tmp;
129                 for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
130                     *dOut++ = *dIn++ ^ *pT++;
131             }
132             break;
133 #endif
134 #if ALG_OFB
135         case TPM_ALG_OFB:
136             // This is written so that dIn and dOut may be the same

```

```

136         for(; dSize > 0; dSize -= blockSize)
137         {
138             // Encrypt the current value of the "IV"
139             ENCRYPT(&keySchedule, iv, iv);
140
141             // XOR the encrypted IV into dIn to create the cipher text (dOut)
142             pIv = iv;
143             for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
144                 *dOut++ = (*pIv++ ^ *dIn++);
145         }
146         break;
147     #endif
148     #if ALG_CBC
149     case TPM_ALG_CBC:
150         // For CBC the data size must be an even multiple of the
151         // cipher block size
152         if((dSize % blockSize) != 0)
153             return TPM_RC_SIZE;
154         // XOR the data block into the IV, encrypt the IV into the IV
155         // and then copy the IV to the output
156         for(; dSize > 0; dSize -= blockSize)
157         {
158             pIv = iv;
159             for(i = blockSize; i > 0; i--)
160                 *pIv++ ^= *dIn++;
161             ENCRYPT(&keySchedule, iv, iv);
162             pIv = iv;
163             for(i = blockSize; i > 0; i--)
164                 *dOut++ = *pIv++;
165         }
166         break;
167     #endif
168     // CFB is not optional
169     case TPM_ALG_CFB:
170         // Encrypt the IV into the IV, XOR in the data, and copy to output
171         for(; dSize > 0; dSize -= blockSize)
172         {
173             // Encrypt the current value of the IV
174             ENCRYPT(&keySchedule, iv, iv);
175             pIv = iv;
176             for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)
177                 // XOR the data into the IV to create the cipher text
178                 // and put into the output
179                 *dOut++ = *pIv++ ^= *dIn++;
180         }
181         // If the inner loop (i loop) was smaller than blockSize, then dSize
182         // would have been smaller than blockSize and it is now negative. If
183         // it is negative, then it indicates how many bytes are needed to pad
184         // out the IV for the next round.
185         for(; dSize < 0; dSize++)
186             *pIv++ = 0;
187         break;
188     #if ALG_ECB
189     case TPM_ALG_ECB:
190         // For ECB the data size must be an even multiple of the
191         // cipher block size
192         if((dSize % blockSize) != 0)
193             return TPM_RC_SIZE;
194         // Encrypt the input block to the output block
195         for(; dSize > 0; dSize -= blockSize)
196         {
197             ENCRYPT(&keySchedule, dIn, dOut);
198             dIn = &dIn[blockSize];
199             dOut = &dOut[blockSize];
200         }
201         break;

```

```

202 #endif
203     default:
204         return TPM_RC_FAILURE;
205     }
206     return TPM_RC_SUCCESS;
207 }

```

10.2.19.4.1 CryptSymmetricDecrypt()

This function performs symmetric decryption based on the mode.

Error Return	Meaning
TPM_RC_FAILURE	A fatal error
TPM_RCS_SIZE	dSize is not a multiple of the block size for an algorithm that requires it

```

208 LIB_EXPORT TPM_RC
209 CryptSymmetricDecrypt(
210     BYTE                *dOut,           // OUT: decrypted data
211     TPM_ALG_ID          algorithm,       // IN: the symmetric algorithm
212     UINT16              keySizeInBits,   // IN: key size in bits
213     const BYTE          *key,           // IN: key buffer. The size of this buffer
214                                     // in bytes is (keySizeInBits + 7) / 8
215     TPM2B_IV            *ivInOut,       // IN/OUT: IV for decryption.
216     TPM_ALG_ID          mode,           // IN: Mode to use
217     INT32               dSize,          // IN: data size (may need to be a
218                                     // multiple of the blockSize)
219     const BYTE          *dIn            // IN: data buffer
220 )
221 {
222     BYTE                *pIv;
223     int                 i;
224     BYTE                tmp[MAX_SYM_BLOCK_SIZE];
225     BYTE                *pT;
226     tpmCryptKeySchedule_t keySchedule;
227     INT16               blockSize;
228     BYTE                *iv;
229     TpmCryptSetSymKeyCall_t encrypt;
230     TpmCryptSetSymKeyCall_t decrypt;
231     BYTE                defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
232
233     // These are used but the compiler can't tell because they are initialized
234     // in case statements and it can't tell if they are always initialized
235     // when needed, so... Comment these out if the compiler can tell or doesn't
236     // care that these are initialized before use.
237     encrypt = NULL;
238     decrypt = NULL;
239
240     pAssert(dOut != NULL && key != NULL && dIn != NULL);
241     if(dSize == 0)
242         return TPM_RC_SUCCESS;
243
244     TEST(algorithm);
245     blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
246     if(blockSize == 0)
247         return TPM_RC_FAILURE;
248     // If the iv is provided, then it is expected to be block sized. In some cases,
249     // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
250     // with no knowledge of the actual block size. This function will set it.
251     if((ivInOut != NULL) && (mode != TPM_ALG_ECB))
252     {
253         ivInOut->t.size = blockSize;

```

```

254     iv = ivInOut->t.buffer;
255 }
256 else
257     iv = defaultIv;
258
259 pIv = iv;
260 // Use the mode to select the key schedule to create. Encrypt always uses the
261 // encryption schedule. Depending on the mode, decryption might use either
262 // the decryption or encryption schedule.
263 switch(mode)
264 {
265 #if ALG_CBC || ALG_ECB
266     case TPM_ALG_CBC: // decrypt = decrypt
267     case TPM_ALG_ECB:
268         // For ECB and CBC, the data size must be an even multiple of the
269         // cipher block size
270         if((dSize % blockSize) != 0)
271             return TPM_RC_SIZE;
272         switch (algorithm)
273         {
274             FOR_EACH_SYM(DECRYPT_CASE)
275             default:
276                 return TPM_RC_SYMMETRIC;
277         }
278         break;
279 #endif
280     default:
281         // For the remaining stream ciphers, use encryption to decrypt
282         switch (algorithm)
283         {
284             FOR_EACH_SYM(ENCRYPT_CASE)
285             default:
286                 return TPM_RC_SYMMETRIC;
287         }
288     }
289     // Now do the mode-dependent decryption
290     switch(mode)
291     {
292     #if ALG_CBC
293     case TPM_ALG_CBC:
294         // Copy the input data to a temp buffer, decrypt the buffer into the
295         // output, XOR in the IV, and copy the temp buffer to the IV and repeat.
296         for(; dSize > 0; dSize -= blockSize)
297         {
298             pT = tmp;
299             for(i = blockSize; i > 0; i--)
300                 *pT++ = *dIn++;
301             DECRYPT(&keySchedule, tmp, dOut);
302             pIv = iv;
303             pT = tmp;
304             for(i = blockSize; i > 0; i--)
305             {
306                 *dOut++ ^= *pIv;
307                 *pIv++ = *pT++;
308             }
309         }
310         break;
311     #endif
312     case TPM_ALG_CFB:
313         for(; dSize > 0; dSize -= blockSize)
314         {
315             // Encrypt the IV into the temp buffer
316             ENCRYPT(&keySchedule, iv, tmp);
317             pT = tmp;
318             pIv = iv;
319             for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)

```

```

320         // Copy the current cipher text to IV, XOR
321         // with the temp buffer and put into the output
322         *dOut++ = *pT++ ^ (*pIv++ = *dIn++);
323     }
324     // If the inner loop (i loop) was smaller than blockSize, then dSize
325     // would have been smaller than blockSize and it is now negative
326     // If it is negative, then it indicates how many fill bytes
327     // are needed to pad out the IV for the next round.
328     for(; dSize < 0; dSize++)
329         *pIv++ = 0;
330
331     break;
332 #if ALG_CTR
333     case TPM_ALG_CTR:
334         for(; dSize > 0; dSize -= blockSize)
335         {
336             // Encrypt the current value of the IV(counter)
337             ENCRYPT(&keySchedule, iv, tmp);
338
339             //increment the counter (counter is big-endian so start at end)
340             for(i = blockSize - 1; i >= 0; i--)
341                 if((iv[i] += 1) != 0)
342                     break;
343             // XOR the encrypted counter value with input and put into output
344             pT = tmp;
345             for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
346                 *dOut++ = *dIn++ ^ *pT++;
347         }
348         break;
349 #endif
350 #if ALG_ECB
351     case TPM_ALG_ECB:
352         for(; dSize > 0; dSize -= blockSize)
353         {
354             DECRYPT(&keySchedule, dIn, dOut);
355             dIn = &dIn[blockSize];
356             dOut = &dOut[blockSize];
357         }
358         break;
359 #endif
360 #if ALG_OFB
361     case TPM_ALG_OFB:
362         // This is written so that dIn and dOut may be the same
363         for(; dSize > 0; dSize -= blockSize)
364         {
365             // Encrypt the current value of the "IV"
366             ENCRYPT(&keySchedule, iv, iv);
367
368             // XOR the encrypted IV into dIn to create the cipher text (dOut)
369             pIv = iv;
370             for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
371                 *dOut++ = (*pIv++ ^ *dIn++);
372         }
373         break;
374 #endif
375     default:
376         return TPM_RC_FAILURE;
377 }
378 return TPM_RC_SUCCESS;
379 }

```

10.2.19.4.2 CryptSymKeyValidate()

Validate that a provided symmetric key meets the requirements of the TPM

Error Return	Meaning
TPM_RC_KEY_SIZE	Key size specifiers do not match
TPM_RC_KEY	Key is not allowed

```

380  TPM_RC
381  CryptSymKeyValidate(
382      TPMT_SYM_DEF_OBJECT *symDef,
383      TPM2B_SYM_KEY      *key
384  )
385  {
386      if(key->t.size != BITS_TO_BYTES(symDef->keyBits.sym))
387          return TPM_RCS_KEY_SIZE;
388  #if ALG_TDES
389      if(symDef->algorithm == TPM_ALG_TDES && !CryptDesValidateKey(key))
390          return TPM_RCS_KEY;
391  #endif // ALG_TDES
392      return TPM_RC_SUCCESS;
393  }

```

10.2.20 PrimeData.c

```
1  #include "Tpm.h"
```

This table is the product of all of the primes up to 1000. Checking to see if there is a GCD between a prime candidate and this number will eliminate many prime candidates from consideration before running Miller-Rabin on the result.

```
2  const BN_STRUCT(43 * RADIX_BITS) s_CompositeOfSmallPrimes_ =
3  {44, 44,
4  { 0x2ED42696, 0x2BBFA177, 0x4820594F, 0xF73F4841,
5  0xBFAC313A, 0xCAC3EB81, 0xF6F26BF8, 0x7FAB5061,
6  0x59746FB7, 0xF71377F6, 0x3B19855B, 0xCBD03132,
7  0xBB92EF1B, 0x3AC3152C, 0xE87C8273, 0xC0AE0E69,
8  0x74A9E295, 0x448CCE86, 0x63CA1907, 0x8A0BF944,
9  0xF8CC3BE0, 0xC26F0AF5, 0xC501C02F, 0x6579441A,
10 0xD1099CDA, 0x6BC76A00, 0xC81A3228, 0xBFB1AB25,
11 0x70FA3841, 0x51B3D076, 0xCC2359ED, 0xD9EE0769,
12 0x75E47AF0, 0xD45FF31E, 0x52CCE4F6, 0x04DBC891,
13 0x96658ED2, 0x1753EFE5, 0x3AE4A5A6, 0x8FD4A97F,
14 0x8B15E7EB, 0x0243C3E1, 0xE0F0C31D, 0x0000000B }
15 };
16
17 bigConst      s_CompositeOfSmallPrimes = (const bigNum)&s_CompositeOfSmallPrimes_;
```

This table contains a bit for each of the odd values between 1 and $2^{16} + 1$. This table allows fast checking of the primes in that range. Don't change the size of this table unless you are prepared to redo IsPrimeInt().

```
18 const uint32_t s_LastPrimeInTable = 65537;
19 const uint32_t s_PrimeTableSize = 4097;
20 const uint32_t s_PrimesInTable = 6542;
21 const unsigned char s_PrimeTable[] = {
22 0x6e, 0xcb, 0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x81, 0x32, 0x4c, 0x4a, 0x86,
23 0x0d, 0x82, 0x96, 0x21, 0xc9, 0x34, 0x04, 0x5a, 0x20, 0x61, 0x89, 0xa4,
24 0x44, 0x11, 0x86, 0x29, 0xd1, 0x82, 0x28, 0x4a, 0x30, 0x40, 0x42, 0x32,
25 0x21, 0x99, 0x34, 0x08, 0x4b, 0x06, 0x25, 0x42, 0x84, 0x48, 0x8a, 0x14,
26 0x05, 0x42, 0x30, 0x6c, 0x08, 0xb4, 0x40, 0x0b, 0xa0, 0x08, 0x51, 0x12,
27 0x28, 0x89, 0x04, 0x65, 0x98, 0x30, 0x4c, 0x80, 0x96, 0x44, 0x12, 0x80,
28 0x21, 0x42, 0x12, 0x41, 0xc9, 0x04, 0x21, 0xc0, 0x32, 0x2d, 0x98, 0x00,
29 0x00, 0x49, 0x04, 0x08, 0x81, 0x96, 0x68, 0x82, 0xb0, 0x25, 0x08, 0x22,
30 0x48, 0x89, 0xa2, 0x40, 0x59, 0x26, 0x04, 0x90, 0x06, 0x40, 0x43, 0x30,
31 0x44, 0x92, 0x00, 0x69, 0x10, 0x82, 0x08, 0x08, 0xa4, 0x0d, 0x41, 0x12,
32 0x60, 0xc0, 0x00, 0x24, 0xd2, 0x22, 0x61, 0x08, 0x84, 0x04, 0x1b, 0x82,
33 0x01, 0xd3, 0x10, 0x01, 0x02, 0xa0, 0x44, 0xc0, 0x22, 0x60, 0x91, 0x14,
34 0x0c, 0x40, 0xa6, 0x04, 0xd2, 0x94, 0x20, 0x09, 0x94, 0x20, 0x00,
35 0x08, 0x10, 0xa2, 0x4c, 0x00, 0x82, 0x01, 0x51, 0x10, 0x08, 0x8b, 0xa4,
36 0x25, 0x9a, 0x30, 0x44, 0x81, 0x10, 0x4c, 0x03, 0x02, 0x25, 0x52, 0x80,
37 0x08, 0x49, 0x84, 0x20, 0x50, 0x32, 0x00, 0x18, 0xa2, 0x40, 0x11, 0x24,
38 0x28, 0x01, 0x84, 0x01, 0x01, 0xa0, 0x41, 0x0a, 0x12, 0x45, 0x00, 0x36,
39 0x08, 0x00, 0x26, 0x29, 0x83, 0x82, 0x61, 0xc0, 0x80, 0x04, 0x10, 0x10,
40 0x6d, 0x00, 0x22, 0x48, 0x58, 0x26, 0x0c, 0xc2, 0x10, 0x48, 0x89, 0x24,
41 0x20, 0x58, 0x20, 0x45, 0x88, 0x24, 0x00, 0x19, 0x02, 0x25, 0xc0, 0x10,
42 0x68, 0x08, 0x14, 0x01, 0xca, 0x32, 0x28, 0x80, 0x00, 0x04, 0x4b, 0x26,
43 0x00, 0x13, 0x90, 0x60, 0x82, 0x80, 0x25, 0xd0, 0x00, 0x01, 0x10, 0x32,
44 0x0c, 0x43, 0x86, 0x21, 0x11, 0x00, 0x08, 0x43, 0x24, 0x04, 0x48, 0x10,
45 0x0c, 0x90, 0x92, 0x00, 0x43, 0x20, 0x2d, 0x00, 0x06, 0x09, 0x88, 0x24,
46 0x40, 0xc0, 0x32, 0x09, 0x09, 0x82, 0x00, 0x53, 0x80, 0x08, 0x80, 0x96,
47 0x41, 0x81, 0x00, 0x40, 0x48, 0x10, 0x48, 0x08, 0x96, 0x48, 0x58, 0x20,
48 0x29, 0xc3, 0x80, 0x20, 0x02, 0x94, 0x60, 0x92, 0x00, 0x20, 0x81, 0x22,
49 0x44, 0x10, 0xa0, 0x05, 0x40, 0x90, 0x01, 0x49, 0x20, 0x04, 0x0a, 0x00,
50 0x24, 0x89, 0x34, 0x48, 0x13, 0x80, 0x2c, 0xc0, 0x82, 0x29, 0x00, 0x24,
51 0x45, 0x08, 0x00, 0x08, 0x98, 0x36, 0x04, 0x52, 0x84, 0x04, 0xd0, 0x04,
```


52 0x00, 0x8a, 0x90, 0x44, 0x82, 0x32, 0x65, 0x18, 0x90, 0x00, 0x0a, 0x02,
53 0x01, 0x40, 0x02, 0x28, 0x40, 0xa4, 0x04, 0x92, 0x30, 0x04, 0x11, 0x86,
54 0x08, 0x42, 0x00, 0x2c, 0x52, 0x04, 0x08, 0xc9, 0x84, 0x60, 0x48, 0x12,
55 0x09, 0x99, 0x24, 0x44, 0x00, 0x24, 0x00, 0x03, 0x14, 0x21, 0x00, 0x10,
56 0x01, 0x1a, 0x32, 0x05, 0x88, 0x20, 0x40, 0x40, 0x06, 0x09, 0xc3, 0x84,
57 0x40, 0x01, 0x30, 0x60, 0x18, 0x02, 0x68, 0x11, 0x90, 0x0c, 0x02, 0xa2,
58 0x04, 0x00, 0x86, 0x29, 0x89, 0x14, 0x24, 0x82, 0x02, 0x41, 0x08, 0x80,
59 0x04, 0x19, 0x80, 0x08, 0x10, 0x12, 0x68, 0x42, 0xa4, 0x04, 0x00, 0x02,
60 0x61, 0x10, 0x06, 0x0c, 0x10, 0x00, 0x01, 0x12, 0x10, 0x20, 0x03, 0x94,
61 0x21, 0x42, 0x12, 0x65, 0x18, 0x94, 0x0c, 0x0a, 0x04, 0x28, 0x01, 0x14,
62 0x29, 0x0a, 0xa4, 0x40, 0xd0, 0x00, 0x40, 0x01, 0x90, 0x04, 0x41, 0x20,
63 0x2d, 0x40, 0x82, 0x48, 0xc1, 0x20, 0x00, 0x10, 0x30, 0x01, 0x08, 0x24,
64 0x04, 0x59, 0x84, 0x24, 0x00, 0x02, 0x29, 0x82, 0x00, 0x61, 0x58, 0x02,
65 0x48, 0x81, 0x16, 0x48, 0x10, 0x00, 0x21, 0x11, 0x06, 0x00, 0xca, 0xa0,
66 0x40, 0x02, 0x00, 0x04, 0x91, 0xb0, 0x00, 0x42, 0x04, 0x0c, 0x81, 0x06,
67 0x09, 0x48, 0x14, 0x25, 0x92, 0x20, 0x25, 0x11, 0xa0, 0x00, 0x0a, 0x86,
68 0x0c, 0xc1, 0x02, 0x48, 0x00, 0x20, 0x45, 0x08, 0x32, 0x00, 0x98, 0x06,
69 0x04, 0x13, 0x22, 0x00, 0x82, 0x04, 0x48, 0x81, 0x14, 0x44, 0x82, 0x12,
70 0x24, 0x18, 0x10, 0x40, 0x43, 0x80, 0x28, 0xd0, 0x04, 0x20, 0x81, 0x24,
71 0x64, 0xd8, 0x00, 0x2c, 0x09, 0x12, 0x08, 0x41, 0xa2, 0x00, 0x00, 0x02,
72 0x41, 0xca, 0x20, 0x41, 0xc0, 0x10, 0x01, 0x18, 0xa4, 0x04, 0x18, 0xa4,
73 0x20, 0x12, 0x94, 0x20, 0x83, 0xa0, 0x40, 0x02, 0x32, 0x44, 0x80, 0x04,
74 0x00, 0x18, 0x00, 0x0c, 0x40, 0x86, 0x60, 0x8a, 0x00, 0x64, 0x88, 0x12,
75 0x05, 0x01, 0x82, 0x00, 0x4a, 0xa2, 0x01, 0xc1, 0x10, 0x61, 0x09, 0x04,
76 0x01, 0x88, 0x00, 0x60, 0x01, 0xb4, 0x40, 0x08, 0x06, 0x01, 0x03, 0x80,
77 0x08, 0x40, 0x94, 0x04, 0x8a, 0x20, 0x29, 0x80, 0x02, 0x0c, 0x52, 0x02,
78 0x01, 0x42, 0x84, 0x00, 0x80, 0x84, 0x64, 0x02, 0x32, 0x48, 0x00, 0x30,
79 0x44, 0x40, 0x22, 0x21, 0x00, 0x02, 0x08, 0xc3, 0xa0, 0x04, 0xd0, 0x20,
80 0x40, 0x18, 0x16, 0x40, 0x40, 0x00, 0x28, 0x52, 0x90, 0x08, 0x82, 0x14,
81 0x01, 0x18, 0x10, 0x08, 0x09, 0x82, 0x40, 0x0a, 0xa0, 0x20, 0x93, 0x80,
82 0x08, 0xc0, 0x00, 0x20, 0x52, 0x00, 0x05, 0x01, 0x10, 0x40, 0x11, 0x06,
83 0x0c, 0x82, 0x00, 0x00, 0x4b, 0x90, 0x44, 0x9a, 0x00, 0x28, 0x80, 0x90,
84 0x04, 0x4a, 0x06, 0x09, 0x43, 0x02, 0x28, 0x00, 0x34, 0x01, 0x18, 0x00,
85 0x65, 0x09, 0x80, 0x44, 0x03, 0x00, 0x24, 0x02, 0x82, 0x61, 0x48, 0x14,
86 0x41, 0x00, 0x12, 0x28, 0x00, 0x34, 0x08, 0x51, 0x04, 0x05, 0x12, 0x90,
87 0x28, 0x89, 0x84, 0x60, 0x12, 0x10, 0x49, 0x10, 0x26, 0x40, 0x49, 0x82,
88 0x00, 0x91, 0x10, 0x01, 0x0a, 0x24, 0x40, 0x88, 0x10, 0x4c, 0x10, 0x04,
89 0x00, 0x50, 0xa2, 0x2c, 0x40, 0x90, 0x48, 0x0a, 0xb0, 0x01, 0x50, 0x12,
90 0x08, 0x00, 0xa4, 0x04, 0x09, 0xa0, 0x28, 0x92, 0x02, 0x00, 0x43, 0x10,
91 0x21, 0x02, 0x20, 0x41, 0x81, 0x32, 0x00, 0x08, 0x04, 0x0c, 0x52, 0x00,
92 0x21, 0x49, 0x84, 0x20, 0x10, 0x02, 0x01, 0x81, 0x10, 0x48, 0x40, 0x22,
93 0x01, 0x01, 0x84, 0x69, 0xc1, 0x30, 0x01, 0xc8, 0x02, 0x44, 0x88, 0x00,
94 0x0c, 0x01, 0x02, 0x2d, 0xc0, 0x12, 0x61, 0x00, 0xa0, 0x00, 0xc0, 0x30,
95 0x40, 0x01, 0x12, 0x08, 0x0b, 0x20, 0x00, 0x80, 0x94, 0x40, 0x01, 0x84,
96 0x40, 0x00, 0x32, 0x00, 0x10, 0x84, 0x00, 0x0b, 0x24, 0x00, 0x01, 0x06,
97 0x29, 0x8a, 0x84, 0x41, 0x80, 0x10, 0x08, 0x08, 0x94, 0x4c, 0x03, 0x80,
98 0x01, 0x40, 0x96, 0x40, 0x41, 0x20, 0x20, 0x50, 0x22, 0x25, 0x89, 0xa2,
99 0x40, 0x40, 0xa4, 0x20, 0x02, 0x86, 0x28, 0x01, 0x20, 0x21, 0x4a, 0x10,
100 0x08, 0x00, 0x14, 0x08, 0x40, 0x04, 0x25, 0x42, 0x02, 0x21, 0x43, 0x10,
101 0x04, 0x92, 0x00, 0x21, 0x11, 0xa0, 0x4c, 0x18, 0x22, 0x09, 0x03, 0x84,
102 0x41, 0x89, 0x10, 0x04, 0x82, 0x22, 0x24, 0x01, 0x14, 0x08, 0x08, 0x84,
103 0x08, 0xc1, 0x00, 0x09, 0x42, 0xb0, 0x41, 0x8a, 0x02, 0x00, 0x80, 0x36,
104 0x04, 0x49, 0xa0, 0x24, 0x91, 0x00, 0x00, 0x02, 0x94, 0x41, 0x92, 0x02,
105 0x01, 0x08, 0x06, 0x08, 0x09, 0x00, 0x01, 0xd0, 0x16, 0x28, 0x89, 0x80,
106 0x60, 0x00, 0x00, 0x68, 0x01, 0x90, 0x0c, 0x50, 0x20, 0x01, 0x40, 0x80,
107 0x40, 0x42, 0x30, 0x41, 0x00, 0x20, 0x25, 0x81, 0x06, 0x40, 0x49, 0x00,
108 0x08, 0x01, 0x12, 0x49, 0x00, 0xa0, 0x20, 0x18, 0x30, 0x05, 0x01, 0xa6,
109 0x00, 0x10, 0x24, 0x28, 0x00, 0x02, 0x20, 0xc8, 0x20, 0x00, 0x88, 0x12,
110 0x0c, 0x90, 0x92, 0x00, 0x02, 0x26, 0x01, 0x42, 0x16, 0x49, 0x00, 0x04,
111 0x24, 0x42, 0x02, 0x01, 0x88, 0x80, 0x0c, 0x1a, 0x80, 0x08, 0x10, 0x00,
112 0x60, 0x02, 0x94, 0x44, 0x88, 0x00, 0x69, 0x11, 0x30, 0x08, 0x12, 0xa0,
113 0x24, 0x13, 0x84, 0x00, 0x82, 0x00, 0x65, 0xc0, 0x10, 0x28, 0x00, 0x30,
114 0x04, 0x03, 0x20, 0x01, 0x11, 0x06, 0x01, 0xc8, 0x80, 0x00, 0xc2, 0x20,
115 0x08, 0x10, 0x82, 0x0c, 0x13, 0x02, 0x0c, 0x52, 0x06, 0x40, 0x00, 0xb0,
116 0x61, 0x40, 0x10, 0x01, 0x98, 0x86, 0x04, 0x10, 0x84, 0x08, 0x92, 0x14,
117 0x60, 0x41, 0x80, 0x41, 0x1a, 0x10, 0x04, 0x81, 0x22, 0x40, 0x41, 0x20,

```

118 0x29, 0x52, 0x00, 0x41, 0x08, 0x34, 0x60, 0x10, 0x00, 0x28, 0x01, 0x10,
119 0x40, 0x00, 0x84, 0x08, 0x42, 0x90, 0x20, 0x48, 0x04, 0x04, 0x52, 0x02,
120 0x00, 0x08, 0x20, 0x04, 0x00, 0x82, 0x0d, 0x00, 0x82, 0x40, 0x02, 0x10,
121 0x05, 0x48, 0x20, 0x40, 0x99, 0x00, 0x00, 0x01, 0x06, 0x24, 0xc0, 0x00,
122 0x68, 0x82, 0x04, 0x21, 0x12, 0x10, 0x44, 0x08, 0x04, 0x00, 0x40, 0xa6,
123 0x20, 0xd0, 0x16, 0x09, 0xc9, 0x24, 0x41, 0x02, 0x20, 0x0c, 0x09, 0x92,
124 0x40, 0x12, 0x00, 0x00, 0x40, 0x00, 0x09, 0x43, 0x84, 0x20, 0x98, 0x02,
125 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,
127 0x41, 0x09, 0x10, 0x20, 0x18, 0x10, 0x44, 0x80, 0x10, 0x00, 0x4a, 0x24,
128 0x0d, 0x01, 0x94, 0x28, 0x80, 0x30, 0x00, 0xc0, 0x02, 0x60, 0x10, 0x84,
129 0x0c, 0x02, 0x00, 0x09, 0x02, 0x82, 0x01, 0x08, 0x10, 0x04, 0xc2, 0x20,
130 0x68, 0x09, 0x06, 0x04, 0x18, 0x00, 0x00, 0x11, 0x90, 0x08, 0x0b, 0x10,
131 0x21, 0x82, 0x02, 0x0c, 0x10, 0xb6, 0x08, 0x00, 0x26, 0x00, 0x41, 0x02,
132 0x01, 0x4a, 0x24, 0x21, 0x1a, 0x20, 0x24, 0x80, 0x00, 0x44, 0x02, 0x00,
133 0x2d, 0x40, 0x02, 0x00, 0x8b, 0x94, 0x20, 0x10, 0x00, 0x20, 0x90, 0xa6,
134 0x40, 0x13, 0x00, 0x2c, 0x11, 0x86, 0x61, 0x01, 0x80, 0x41, 0x10, 0x02,
135 0x04, 0x81, 0x30, 0x48, 0x48, 0x20, 0x28, 0x50, 0x80, 0x21, 0x8a, 0x10,
136 0x04, 0x08, 0x10, 0x09, 0x10, 0x10, 0x48, 0x42, 0xa0, 0x0c, 0x82, 0x92,
137 0x60, 0xc0, 0x20, 0x05, 0xd2, 0x20, 0x40, 0x01, 0x00, 0x04, 0x08, 0x82,
138 0x2d, 0x82, 0x02, 0x00, 0x48, 0x80, 0x41, 0x48, 0x10, 0x00, 0x91, 0x04,
139 0x04, 0x03, 0x84, 0x00, 0xc2, 0x04, 0x68, 0x00, 0x00, 0x64, 0xc0, 0x22,
140 0x40, 0x08, 0x32, 0x44, 0x09, 0x86, 0x00, 0x91, 0x02, 0x28, 0x01, 0x00,
141 0x64, 0x48, 0x00, 0x24, 0x10, 0x90, 0x00, 0x43, 0x00, 0x21, 0x52, 0x86,
142 0x41, 0x8b, 0x90, 0x20, 0x40, 0x20, 0x08, 0x88, 0x04, 0x44, 0x13, 0x20,
143 0x00, 0x02, 0x84, 0x60, 0x81, 0x90, 0x24, 0x40, 0x30, 0x00, 0x08, 0x10,
144 0x08, 0x08, 0x02, 0x01, 0x10, 0x04, 0x20, 0x43, 0xb4, 0x40, 0x90, 0x12,
145 0x68, 0x01, 0x80, 0x4c, 0x18, 0x00, 0x08, 0xc0, 0x12, 0x49, 0x40, 0x10,
146 0x24, 0x1a, 0x00, 0x41, 0x89, 0x24, 0x4c, 0x10, 0x00, 0x04, 0x52, 0x10,
147 0x09, 0x4a, 0x20, 0x41, 0x48, 0x22, 0x69, 0x11, 0x14, 0x08, 0x10, 0x06,
148 0x24, 0x80, 0x84, 0x28, 0x00, 0x10, 0x00, 0x40, 0x10, 0x01, 0x08, 0x26,
149 0x08, 0x48, 0x06, 0x28, 0x00, 0x14, 0x01, 0x42, 0x84, 0x04, 0x0a, 0x20,
150 0x00, 0x01, 0x82, 0x08, 0x00, 0x82, 0x24, 0x12, 0x04, 0x40, 0x40, 0xa0,
151 0x40, 0x90, 0x10, 0x04, 0x90, 0x22, 0x40, 0x10, 0x20, 0x2c, 0x80, 0x10,
152 0x28, 0x43, 0x00, 0x04, 0x58, 0x00, 0x01, 0x81, 0x10, 0x48, 0x09, 0x20,
153 0x21, 0x83, 0x04, 0x00, 0x42, 0xa4, 0x44, 0x00, 0x00, 0x6c, 0x10, 0xa0,
154 0x44, 0x48, 0x80, 0x00, 0x83, 0x80, 0x48, 0xc9, 0x00, 0x00, 0x02,
155 0x05, 0x10, 0xb0, 0x04, 0x13, 0x04, 0x29, 0x10, 0x92, 0x40, 0x08, 0x04,
156 0x44, 0x82, 0x22, 0x00, 0x19, 0x20, 0x00, 0x19, 0x20, 0x01, 0x81, 0x90,
157 0x60, 0x8a, 0x00, 0x41, 0xc0, 0x02, 0x45, 0x10, 0x04, 0x00, 0x02, 0xa2,
158 0x09, 0x40, 0x10, 0x21, 0x49, 0x20, 0x01, 0x42, 0x30, 0x2c, 0x00, 0x14,
159 0x44, 0x01, 0x22, 0x04, 0x02, 0x92, 0x08, 0x89, 0x04, 0x21, 0x80, 0x10,
160 0x05, 0x01, 0x20, 0x40, 0x41, 0x80, 0x04, 0x00, 0x12, 0x09, 0x40, 0xb0,
161 0x64, 0x58, 0x32, 0x01, 0x08, 0x90, 0x00, 0x41, 0x04, 0x09, 0xc1, 0x80,
162 0x61, 0x08, 0x90, 0x00, 0x9a, 0x00, 0x24, 0x01, 0x12, 0x08, 0x02, 0x26,
163 0x05, 0x82, 0x06, 0x08, 0x08, 0x00, 0x20, 0x48, 0x20, 0x00, 0x18, 0x24,
164 0x48, 0x03, 0x02, 0x00, 0x11, 0x00, 0x09, 0x00, 0x84, 0x01, 0x4a, 0x10,
165 0x01, 0x98, 0x00, 0x04, 0x18, 0x86, 0x00, 0xc0, 0x00, 0x20, 0x81, 0x80,
166 0x04, 0x10, 0x30, 0x05, 0x00, 0xb4, 0x0c, 0x4a, 0x82, 0x29, 0x91, 0x02,
167 0x28, 0x00, 0x20, 0x44, 0xc0, 0x00, 0x2c, 0x91, 0x80, 0x40, 0x01, 0xa2,
168 0x00, 0x12, 0x04, 0x09, 0xc3, 0x20, 0x00, 0x08, 0x02, 0x0c, 0x10, 0x22,
169 0x04, 0x00, 0x00, 0x2c, 0x11, 0x86, 0x00, 0xc0, 0x00, 0x00, 0x12, 0x32,
170 0x40, 0x89, 0x80, 0x40, 0x40, 0x02, 0x05, 0x50, 0x86, 0x60, 0x82, 0xa4,
171 0x60, 0x0a, 0x12, 0x4d, 0x80, 0x90, 0x08, 0x12, 0x80, 0x09, 0x02, 0x14,
172 0x48, 0x01, 0x24, 0x20, 0x8a, 0x00, 0x44, 0x90, 0x04, 0x04, 0x01, 0x02,
173 0x00, 0xd1, 0x12, 0x00, 0x0a, 0x04, 0x40, 0x00, 0x32, 0x21, 0x81, 0x24,
174 0x08, 0x19, 0x84, 0x20, 0x02, 0x04, 0x08, 0x89, 0x80, 0x24, 0x02, 0x02,
175 0x68, 0x18, 0x82, 0x44, 0x42, 0x00, 0x21, 0x40, 0x00, 0x28, 0x01, 0x80,
176 0x45, 0x82, 0x20, 0x40, 0x11, 0x80, 0x0c, 0x02, 0x00, 0x24, 0x40, 0x90,
177 0x01, 0x40, 0x20, 0x20, 0x50, 0x20, 0x28, 0x19, 0x00, 0x40, 0x09, 0x20,
178 0x08, 0x80, 0x04, 0x60, 0x40, 0x80, 0x20, 0x08, 0x30, 0x49, 0x09, 0x34,
179 0x00, 0x11, 0x24, 0x24, 0x82, 0x00, 0x41, 0xc2, 0x00, 0x04, 0x92, 0x02,
180 0x24, 0x80, 0x00, 0x0c, 0x02, 0xa0, 0x00, 0x01, 0x06, 0x60, 0x41, 0x04,
181 0x21, 0xd0, 0x00, 0x01, 0x01, 0x00, 0x48, 0x12, 0x84, 0x04, 0x91, 0x12,
182 0x08, 0x00, 0x24, 0x44, 0x00, 0x12, 0x41, 0x18, 0x26, 0x0c, 0x41, 0x80,
183 0x00, 0x52, 0x04, 0x20, 0x09, 0x00, 0x24, 0x90, 0x20, 0x48, 0x18, 0x02,

```

184 0x00, 0x03, 0xa2, 0x09, 0xd0, 0x14, 0x00, 0x8a, 0x84, 0x25, 0x4a, 0x00,
185 0x20, 0x98, 0x14, 0x40, 0x00, 0xa2, 0x05, 0x00, 0x00, 0x00, 0x40, 0x14,
186 0x01, 0x58, 0x20, 0x2c, 0x80, 0x84, 0x00, 0x09, 0x20, 0x20, 0x91, 0x02,
187 0x08, 0x02, 0xb0, 0x41, 0x08, 0x30, 0x00, 0x09, 0x10, 0x00, 0x18, 0x02,
188 0x21, 0x02, 0x02, 0x00, 0x00, 0x24, 0x44, 0x08, 0x12, 0x60, 0x00, 0xb2,
189 0x44, 0x12, 0x02, 0x0c, 0xc0, 0x80, 0x40, 0xc8, 0x20, 0x04, 0x50, 0x20,
190 0x05, 0x00, 0xb0, 0x04, 0x0b, 0x04, 0x29, 0x53, 0x00, 0x61, 0x48, 0x30,
191 0x00, 0x82, 0x20, 0x29, 0x00, 0x16, 0x00, 0x53, 0x22, 0x20, 0x43, 0x10,
192 0x48, 0x00, 0x80, 0x04, 0xd2, 0x00, 0x40, 0x00, 0xa2, 0x44, 0x03, 0x80,
193 0x29, 0x00, 0x04, 0x08, 0xc0, 0x04, 0x64, 0x40, 0x30, 0x28, 0x09, 0x84,
194 0x44, 0x50, 0x80, 0x21, 0x02, 0x92, 0x00, 0xc0, 0x10, 0x60, 0x88, 0x22,
195 0x08, 0x80, 0x00, 0x00, 0x18, 0x84, 0x04, 0x83, 0x96, 0x00, 0x81, 0x20,
196 0x05, 0x02, 0x00, 0x45, 0x88, 0x84, 0x00, 0x51, 0x20, 0x20, 0x51, 0x86,
197 0x41, 0x4b, 0x94, 0x00, 0x80, 0x00, 0x08, 0x11, 0x20, 0x4c, 0x58, 0x80,
198 0x04, 0x03, 0x06, 0x20, 0x89, 0x00, 0x05, 0x08, 0x22, 0x05, 0x90, 0x00,
199 0x40, 0x00, 0x82, 0x09, 0x50, 0x00, 0x00, 0x00, 0xa0, 0x41, 0xc2, 0x20,
200 0x08, 0x00, 0x16, 0x08, 0x40, 0x26, 0x21, 0xd0, 0x90, 0x08, 0x81, 0x90,
201 0x41, 0x00, 0x02, 0x44, 0x08, 0x10, 0x0c, 0x0a, 0x86, 0x09, 0x90, 0x04,
202 0x00, 0xc8, 0xa0, 0x04, 0x08, 0x30, 0x20, 0x89, 0x84, 0x00, 0x11, 0x22,
203 0x2c, 0x40, 0x00, 0x08, 0x02, 0xb0, 0x01, 0x48, 0x02, 0x01, 0x09, 0x20,
204 0x04, 0x03, 0x04, 0x00, 0x80, 0x02, 0x60, 0x42, 0x30, 0x21, 0x4a, 0x10,
205 0x44, 0x09, 0x02, 0x00, 0x01, 0x24, 0x00, 0x12, 0x82, 0x21, 0x80, 0xa4,
206 0x20, 0x10, 0x02, 0x04, 0x91, 0xa0, 0x40, 0x18, 0x04, 0x00, 0x02, 0x06,
207 0x69, 0x09, 0x00, 0x05, 0x58, 0x02, 0x01, 0x00, 0x00, 0x48, 0x00, 0x00,
208 0x00, 0x03, 0x92, 0x20, 0x00, 0x34, 0x01, 0xc8, 0x20, 0x48, 0x08, 0x30,
209 0x08, 0x42, 0x80, 0x20, 0x91, 0x90, 0x68, 0x01, 0x04, 0x40, 0x12, 0x02,
210 0x61, 0x00, 0x12, 0x08, 0x01, 0xa0, 0x00, 0x11, 0x04, 0x21, 0x48, 0x04,
211 0x24, 0x92, 0x00, 0x0c, 0x01, 0x84, 0x04, 0x00, 0x00, 0x01, 0x12, 0x96,
212 0x40, 0x01, 0xa0, 0x41, 0x88, 0x22, 0x28, 0x88, 0x00, 0x44, 0x42, 0x80,
213 0x24, 0x12, 0x14, 0x01, 0x42, 0x90, 0x60, 0x1a, 0x10, 0x04, 0x81, 0x10,
214 0x48, 0x08, 0x06, 0x29, 0x83, 0x02, 0x40, 0x02, 0x24, 0x64, 0x80, 0x10,
215 0x05, 0x80, 0x10, 0x40, 0x02, 0x02, 0x08, 0x42, 0x84, 0x01, 0x09, 0x20,
216 0x04, 0x50, 0x00, 0x60, 0x11, 0x30, 0x40, 0x13, 0x02, 0x04, 0x81, 0x00,
217 0x09, 0x08, 0x20, 0x45, 0x4a, 0x10, 0x61, 0x90, 0x26, 0x0c, 0x08, 0x02,
218 0x21, 0x91, 0x00, 0x60, 0x02, 0x04, 0x00, 0x02, 0x00, 0x0c, 0x08, 0x06,
219 0x08, 0x48, 0x84, 0x08, 0x11, 0x02, 0x00, 0x80, 0xa4, 0x00, 0x5a, 0x20,
220 0x00, 0x88, 0x04, 0x04, 0x02, 0x00, 0x09, 0x00, 0x14, 0x08, 0x49, 0x14,
221 0x20, 0xc8, 0x00, 0x04, 0x91, 0xa0, 0x40, 0x59, 0x80, 0x00, 0x12, 0x10,
222 0x00, 0x80, 0x80, 0x65, 0x00, 0x00, 0x04, 0x00, 0x80, 0x40, 0x19, 0x00,
223 0x21, 0x03, 0x84, 0x60, 0xc0, 0x04, 0x24, 0x1a, 0x12, 0x61, 0x80, 0x80,
224 0x08, 0x02, 0x04, 0x09, 0x42, 0x12, 0x20, 0x08, 0x34, 0x04, 0x90, 0x20,
225 0x01, 0x01, 0xa0, 0x00, 0x0b, 0x00, 0x08, 0x91, 0x92, 0x40, 0x02, 0x34,
226 0x40, 0x88, 0x10, 0x61, 0x19, 0x02, 0x00, 0x40, 0x04, 0x25, 0xc0, 0x80,
227 0x68, 0x08, 0x04, 0x21, 0x80, 0x22, 0x04, 0x00, 0xa0, 0x0c, 0x01, 0x84,
228 0x20, 0x41, 0x00, 0x08, 0x8a, 0x00, 0x20, 0x8a, 0x00, 0x48, 0x88, 0x04,
229 0x04, 0x11, 0x82, 0x08, 0x40, 0x86, 0x09, 0x49, 0xa4, 0x40, 0x00, 0x10,
230 0x01, 0x01, 0xa2, 0x04, 0x50, 0x80, 0x0c, 0x80, 0x00, 0x48, 0x82, 0xa0,
231 0x01, 0x18, 0x12, 0x41, 0x01, 0x04, 0x48, 0x41, 0x00, 0x24, 0x01, 0x00,
232 0x00, 0x88, 0x14, 0x00, 0x02, 0x00, 0x68, 0x01, 0x20, 0x08, 0x4a, 0x22,
233 0x08, 0x83, 0x80, 0x00, 0x89, 0x04, 0x01, 0xc2, 0x00, 0x00, 0x00, 0x34,
234 0x04, 0x00, 0x82, 0x28, 0x02, 0x02, 0x41, 0x4a, 0x90, 0x05, 0x82, 0x02,
235 0x09, 0x80, 0x24, 0x04, 0x41, 0x00, 0x01, 0x92, 0x80, 0x28, 0x01, 0x14,
236 0x00, 0x50, 0x20, 0x4c, 0x10, 0xb0, 0x04, 0x43, 0xa4, 0x21, 0x90, 0x04,
237 0x01, 0x02, 0x00, 0x44, 0x48, 0x00, 0x64, 0x08, 0x06, 0x00, 0x42, 0x20,
238 0x08, 0x02, 0x92, 0x01, 0x4a, 0x00, 0x20, 0x50, 0x32, 0x25, 0x90, 0x22,
239 0x04, 0x09, 0x00, 0x08, 0x11, 0x80, 0x21, 0x01, 0x10, 0x05, 0x00, 0x32,
240 0x08, 0x88, 0x94, 0x08, 0x08, 0x24, 0x0d, 0xc1, 0x80, 0x40, 0x0b, 0x20,
241 0x40, 0x18, 0x12, 0x04, 0x00, 0x22, 0x40, 0x10, 0x26, 0x05, 0xc1, 0x82,
242 0x00, 0x01, 0x30, 0x24, 0x02, 0x22, 0x41, 0x08, 0x24, 0x48, 0x1a, 0x00,
243 0x25, 0xd2, 0x12, 0x28, 0x42, 0x00, 0x04, 0x40, 0x30, 0x41, 0x00, 0x02,
244 0x00, 0x13, 0x20, 0x24, 0xd1, 0x84, 0x08, 0x89, 0x80, 0x04, 0x52, 0x00,
245 0x44, 0x18, 0xa4, 0x00, 0x00, 0x06, 0x20, 0x91, 0x10, 0x09, 0x42, 0x20,
246 0x24, 0x40, 0x30, 0x28, 0x00, 0x84, 0x40, 0x40, 0x80, 0x08, 0x10, 0x04,
247 0x09, 0x08, 0x04, 0x40, 0x08, 0x22, 0x00, 0x19, 0x02, 0x00, 0x00, 0x80,
248 0x2c, 0x02, 0x02, 0x21, 0x01, 0x90, 0x20, 0x40, 0x00, 0x0c, 0x00, 0x34,
249 0x48, 0x58, 0x20, 0x01, 0x43, 0x04, 0x20, 0x80, 0x14, 0x00, 0x90, 0x00,

```

250 0x6d, 0x11, 0x00, 0x00, 0x40, 0x20, 0x00, 0x03, 0x10, 0x40, 0x88, 0x30,
251 0x05, 0x4a, 0x00, 0x65, 0x10, 0x24, 0x08, 0x18, 0x84, 0x28, 0x03, 0x80,
252 0x20, 0x42, 0xb0, 0x40, 0x00, 0x10, 0x69, 0x19, 0x04, 0x00, 0x00, 0x80,
253 0x04, 0xc2, 0x04, 0x00, 0x01, 0x00, 0x05, 0x00, 0x22, 0x25, 0x08, 0x96,
254 0x04, 0x02, 0x22, 0x00, 0xd0, 0x10, 0x29, 0x01, 0xa0, 0x60, 0x08, 0x10,
255 0x04, 0x01, 0x16, 0x44, 0x10, 0x02, 0x28, 0x02, 0x82, 0x48, 0x40, 0x84,
256 0x20, 0x90, 0x22, 0x28, 0x80, 0x04, 0x00, 0x40, 0x04, 0x24, 0x00, 0x80,
257 0x29, 0x03, 0x10, 0x60, 0x48, 0x00, 0x00, 0x81, 0xa0, 0x00, 0x51, 0x20,
258 0x0c, 0xd1, 0x00, 0x01, 0x41, 0x20, 0x04, 0x92, 0x00, 0x00, 0x10, 0x92,
259 0x00, 0x42, 0x04, 0x05, 0x01, 0x86, 0x40, 0x80, 0x10, 0x20, 0x52, 0x20,
260 0x21, 0x00, 0x10, 0x48, 0x0a, 0x02, 0x00, 0xd0, 0x12, 0x41, 0x48, 0x80,
261 0x04, 0x00, 0x00, 0x48, 0x09, 0x22, 0x04, 0x00, 0x24, 0x00, 0x43, 0x10,
262 0x60, 0x0a, 0x00, 0x44, 0x12, 0x20, 0x2c, 0x08, 0x20, 0x44, 0x00, 0x84,
263 0x09, 0x40, 0x06, 0x08, 0xc1, 0x00, 0x40, 0x80, 0x20, 0x00, 0x98, 0x12,
264 0x48, 0x10, 0xa2, 0x20, 0x00, 0x84, 0x48, 0xc0, 0x10, 0x20, 0x90, 0x12,
265 0x08, 0x98, 0x82, 0x00, 0x0a, 0xa0, 0x04, 0x03, 0x00, 0x28, 0xc3, 0x00,
266 0x44, 0x42, 0x10, 0x04, 0x08, 0x04, 0x40, 0x00, 0x00, 0x05, 0x10, 0x00,
267 0x21, 0x03, 0x80, 0x04, 0x88, 0x12, 0x69, 0x10, 0x00, 0x04, 0x08, 0x04,
268 0x04, 0x02, 0x84, 0x48, 0x49, 0x04, 0x20, 0x18, 0x02, 0x64, 0x80, 0x30,
269 0x08, 0x01, 0x02, 0x00, 0x52, 0x12, 0x49, 0x08, 0x20, 0x41, 0x88, 0x10,
270 0x48, 0x08, 0x34, 0x00, 0x01, 0x86, 0x05, 0xd0, 0x00, 0x00, 0x83, 0x84,
271 0x21, 0x40, 0x02, 0x41, 0x10, 0x80, 0x48, 0x40, 0xa2, 0x20, 0x51, 0x00,
272 0x00, 0x49, 0x00, 0x01, 0x90, 0x20, 0x40, 0x18, 0x02, 0x40, 0x02, 0x22,
273 0x05, 0x40, 0x80, 0x08, 0x82, 0x10, 0x20, 0x18, 0x00, 0x05, 0x01, 0x82,
274 0x40, 0x58, 0x00, 0x04, 0x81, 0x90, 0x29, 0x01, 0xa0, 0x64, 0x00, 0x22,
275 0x40, 0x01, 0xa2, 0x00, 0x18, 0x04, 0x0d, 0x00, 0x00, 0x60, 0x80, 0x94,
276 0x60, 0x82, 0x10, 0x0d, 0x80, 0x30, 0x0c, 0x12, 0x20, 0x00, 0x00, 0x12,
277 0x40, 0xc0, 0x20, 0x21, 0x58, 0x02, 0x41, 0x10, 0x80, 0x44, 0x03, 0x02,
278 0x04, 0x13, 0x90, 0x29, 0x08, 0x00, 0x44, 0xc0, 0x00, 0x21, 0x00, 0x26,
279 0x00, 0x1a, 0x80, 0x01, 0x13, 0x14, 0x20, 0x0a, 0x14, 0x20, 0x00, 0x32,
280 0x61, 0x08, 0x00, 0x40, 0x42, 0x20, 0x09, 0x80, 0x06, 0x01, 0x81, 0x80,
281 0x60, 0x42, 0x00, 0x68, 0x90, 0x82, 0x08, 0x42, 0x80, 0x04, 0x02, 0x80,
282 0x09, 0x0b, 0x04, 0x00, 0x98, 0x00, 0x0c, 0x81, 0x06, 0x44, 0x48, 0x84,
283 0x28, 0x03, 0x92, 0x00, 0x01, 0x80, 0x40, 0x0a, 0x00, 0x0c, 0x81, 0x02,
284 0x08, 0x51, 0x04, 0x28, 0x90, 0x02, 0x20, 0x09, 0x10, 0x60, 0x00, 0x00,
285 0x09, 0x81, 0xa0, 0x0c, 0x00, 0xa4, 0x09, 0x00, 0x02, 0x28, 0x80, 0x20,
286 0x00, 0x02, 0x02, 0x04, 0x81, 0x14, 0x04, 0x00, 0x04, 0x09, 0x11, 0x12,
287 0x60, 0x40, 0x20, 0x01, 0x48, 0x30, 0x40, 0x11, 0x00, 0x08, 0x0a, 0x86,
288 0x00, 0x00, 0x04, 0x60, 0x81, 0x04, 0x01, 0xd0, 0x02, 0x41, 0x18, 0x90,
289 0x00, 0x0a, 0x20, 0x00, 0xc1, 0x06, 0x01, 0x08, 0x80, 0x64, 0xca, 0x10,
290 0x04, 0x99, 0x80, 0x48, 0x01, 0x82, 0x20, 0x50, 0x90, 0x48, 0x80, 0x84,
291 0x20, 0x90, 0x22, 0x00, 0x19, 0x00, 0x04, 0x18, 0x20, 0x24, 0x10, 0x86,
292 0x40, 0xc2, 0x00, 0x24, 0x12, 0x10, 0x44, 0x00, 0x16, 0x08, 0x10, 0x24,
293 0x00, 0x12, 0x06, 0x01, 0x08, 0x90, 0x00, 0x12, 0x02, 0x4d, 0x10, 0x80,
294 0x40, 0x50, 0x22, 0x00, 0x43, 0x10, 0x01, 0x00, 0x30, 0x21, 0x0a, 0x00,
295 0x00, 0x01, 0x14, 0x00, 0x10, 0x84, 0x04, 0xc1, 0x10, 0x29, 0x0a, 0x00,
296 0x01, 0x8a, 0x00, 0x20, 0x01, 0x12, 0x0c, 0x49, 0x20, 0x04, 0x81, 0x00,
297 0x48, 0x01, 0x04, 0x60, 0x80, 0x12, 0x0c, 0x08, 0x10, 0x48, 0x4a, 0x04,
298 0x28, 0x10, 0x00, 0x28, 0x40, 0x84, 0x45, 0x50, 0x10, 0x60, 0x10, 0x06,
299 0x44, 0x01, 0x80, 0x09, 0x00, 0x86, 0x01, 0x42, 0xa0, 0x00, 0x90, 0x00,
300 0x05, 0x90, 0x22, 0x40, 0x41, 0x00, 0x08, 0x80, 0x02, 0x08, 0xc0, 0x00,
301 0x01, 0x58, 0x30, 0x49, 0x09, 0x14, 0x00, 0x41, 0x02, 0x0c, 0x02, 0x80,
302 0x40, 0x89, 0x00, 0x24, 0x08, 0x10, 0x05, 0x90, 0x32, 0x40, 0x0a, 0x82,
303 0x08, 0x00, 0x12, 0x61, 0x00, 0x04, 0x21, 0x00, 0x22, 0x04, 0x10, 0x24,
304 0x08, 0x0a, 0x04, 0x01, 0x10, 0x00, 0x20, 0x40, 0x84, 0x04, 0x88, 0x22,
305 0x20, 0x90, 0x12, 0x00, 0x53, 0x06, 0x24, 0x01, 0x04, 0x40, 0x0b, 0x14,
306 0x60, 0x82, 0x02, 0x0d, 0x10, 0x90, 0x0c, 0x08, 0x20, 0x09, 0x00, 0x14,
307 0x09, 0x80, 0x80, 0x24, 0x82, 0x00, 0x40, 0x01, 0x02, 0x44, 0x01, 0x20,
308 0x0c, 0x40, 0x84, 0x40, 0x0a, 0x10, 0x41, 0x00, 0x30, 0x05, 0x09, 0x80,
309 0x44, 0x08, 0x20, 0x20, 0x02, 0x00, 0x49, 0x43, 0x20, 0x21, 0x00, 0x20,
310 0x00, 0x01, 0xb6, 0x08, 0x40, 0x04, 0x08, 0x02, 0x80, 0x01, 0x41, 0x80,
311 0x40, 0x08, 0x10, 0x24, 0x00, 0x20, 0x04, 0x12, 0x86, 0x09, 0xc0, 0x12,
312 0x21, 0x81, 0x14, 0x04, 0x00, 0x02, 0x20, 0x89, 0xb4, 0x44, 0x12, 0x80,
313 0x00, 0xd1, 0x00, 0x69, 0x40, 0x80, 0x00, 0x42, 0x12, 0x00, 0x18, 0x04,
314 0x00, 0x49, 0x06, 0x21, 0x02, 0x04, 0x28, 0x02, 0x84, 0x01, 0xc0, 0x10,
315 0x68, 0x00, 0x20, 0x08, 0x40, 0x00, 0x08, 0x91, 0x10, 0x01, 0x81, 0x24,

```

```

316    0x04, 0xd2, 0x10, 0x4c, 0x88, 0x86, 0x00, 0x10, 0x80, 0x0c, 0x02, 0x14,
317    0x00, 0x8a, 0x90, 0x40, 0x18, 0x20, 0x21, 0x80, 0xa4, 0x00, 0x58, 0x24,
318    0x20, 0x10, 0x10, 0x60, 0xc1, 0x30, 0x41, 0x48, 0x02, 0x48, 0x09, 0x00,
319    0x40, 0x09, 0x02, 0x05, 0x11, 0x82, 0x20, 0x4a, 0x20, 0x24, 0x18, 0x02,
320    0x0c, 0x10, 0x22, 0x0c, 0x0a, 0x04, 0x00, 0x03, 0x06, 0x48, 0x48, 0x04,
321    0x04, 0x02, 0x00, 0x21, 0x80, 0x84, 0x00, 0x18, 0x00, 0x0c, 0x02, 0x12,
322    0x01, 0x00, 0x14, 0x05, 0x82, 0x10, 0x41, 0x89, 0x12, 0x08, 0x40, 0xa4,
323    0x21, 0x01, 0x84, 0x48, 0x02, 0x10, 0x60, 0x40, 0x02, 0x28, 0x00, 0x14,
324    0x08, 0x40, 0xa0, 0x20, 0x51, 0x12, 0x00, 0xc2, 0x00, 0x01, 0x1a, 0x30,
325    0x40, 0x89, 0x12, 0x4c, 0x02, 0x80, 0x00, 0x00, 0x14, 0x01, 0x01, 0xa0,
326    0x21, 0x18, 0x22, 0x21, 0x18, 0x06, 0x40, 0x01, 0x80, 0x00, 0x90, 0x04,
327    0x48, 0x02, 0x30, 0x04, 0x08, 0x00, 0x05, 0x88, 0x24, 0x08, 0x48, 0x04,
328    0x24, 0x02, 0x06, 0x00, 0x80, 0x00, 0x00, 0x00, 0x10, 0x65, 0x11, 0x90,
329    0x00, 0x0a, 0x82, 0x04, 0xc3, 0x04, 0x60, 0x48, 0x24, 0x04, 0x92, 0x02,
330    0x44, 0x88, 0x80, 0x40, 0x18, 0x06, 0x29, 0x80, 0x10, 0x01, 0x00, 0x00,
331    0x44, 0xc8, 0x10, 0x21, 0x89, 0x30, 0x00, 0x4b, 0xa0, 0x01, 0x10, 0x14,
332    0x00, 0x02, 0x94, 0x40, 0x00, 0x20, 0x65, 0x00, 0xa2, 0x0c, 0x40, 0x22,
333    0x20, 0x81, 0x12, 0x20, 0x82, 0x04, 0x01, 0x10, 0x00, 0x08, 0x88, 0x00,
334    0x00, 0x11, 0x80, 0x04, 0x42, 0x80, 0x40, 0x41, 0x14, 0x00, 0x40, 0x32,
335    0x2c, 0x80, 0x24, 0x04, 0x19, 0x00, 0x00, 0x91, 0x00, 0x20, 0x83, 0x00,
336    0x05, 0x40, 0x20, 0x09, 0x01, 0x84, 0x40, 0x40, 0x20, 0x20, 0x11, 0x00,
337    0x40, 0x41, 0x90, 0x20, 0x00, 0x00, 0x40, 0x90, 0x92, 0x48, 0x18, 0x06,
338    0x08, 0x81, 0x80, 0x48, 0x01, 0x34, 0x24, 0x10, 0x20, 0x04, 0x00, 0x20,
339    0x04, 0x18, 0x06, 0x2d, 0x90, 0x10, 0x01, 0x00, 0x90, 0x00, 0x0a, 0x22,
340    0x01, 0x00, 0x22, 0x00, 0x11, 0x84, 0x01, 0x01, 0x00, 0x20, 0x88, 0x00,
341    0x44, 0x00, 0x22, 0x01, 0x00, 0xa6, 0x40, 0x02, 0x06, 0x20, 0x11, 0x00,
342    0x01, 0xc8, 0xa0, 0x04, 0x8a, 0x00, 0x28, 0x19, 0x80, 0x00, 0x52, 0xa0,
343    0x24, 0x12, 0x12, 0x09, 0x08, 0x24, 0x01, 0x48, 0x00, 0x04, 0x00, 0x24,
344    0x40, 0x02, 0x84, 0x08, 0x00, 0x04, 0x48, 0x40, 0x90, 0x60, 0x0a, 0x22,
345    0x01, 0x88, 0x14, 0x08, 0x01, 0x02, 0x08, 0xd3, 0x00, 0x20, 0xc0, 0x90,
346    0x24, 0x10, 0x00, 0x00, 0x01, 0xb0, 0x08, 0x0a, 0xa0, 0x00, 0x80, 0x00,
347    0x01, 0x09, 0x00, 0x20, 0x52, 0x02, 0x25, 0x00, 0x24, 0x04, 0x02, 0x84,
348    0x24, 0x10, 0x92, 0x40, 0x02, 0xa0, 0x40, 0x00, 0x22, 0x08, 0x11, 0x04,
349    0x08, 0x01, 0x22, 0x00, 0x42, 0x14, 0x00, 0x09, 0x90, 0x21, 0x00, 0x30,
350    0x6c, 0x00, 0x00, 0x0c, 0x00, 0x22, 0x09, 0x90, 0x10, 0x28, 0x40, 0x00,
351    0x20, 0xc0, 0x20, 0x00, 0x90, 0x00, 0x40, 0x01, 0x82, 0x05, 0x12, 0x12,
352    0x09, 0xc1, 0x04, 0x61, 0x80, 0x02, 0x28, 0x81, 0x24, 0x00, 0x49, 0x04,
353    0x08, 0x10, 0x86, 0x29, 0x41, 0x80, 0x21, 0x0a, 0x30, 0x49, 0x88, 0x90,
354    0x00, 0x41, 0x04, 0x29, 0x81, 0x80, 0x41, 0x09, 0x00, 0x40, 0x12, 0x10,
355    0x40, 0x00, 0x10, 0x40, 0x48, 0x02, 0x05, 0x80, 0x02, 0x21, 0x40, 0x20,
356    0x00, 0x58, 0x20, 0x60, 0x00, 0x90, 0x48, 0x00, 0x80, 0x28, 0xc0, 0x80,
357    0x48, 0x00, 0x00, 0x44, 0x80, 0x02, 0x00, 0x09, 0x06, 0x00, 0x12, 0x02,
358    0x01, 0x00, 0x10, 0x08, 0x83, 0x10, 0x45, 0x12, 0x00, 0x2c, 0x08, 0x04,
359    0x44, 0x00, 0x20, 0x20, 0xc0, 0x10, 0x20, 0x01, 0x00, 0x05, 0xc8, 0x20,
360    0x04, 0x98, 0x10, 0x08, 0x10, 0x00, 0x24, 0x02, 0x16, 0x40, 0x88, 0x00,
361    0x61, 0x88, 0x12, 0x24, 0x80, 0xa6, 0x00, 0x42, 0x00, 0x08, 0x10, 0x06,
362    0x48, 0x40, 0xa0, 0x00, 0x50, 0x20, 0x04, 0x81, 0xa4, 0x40, 0x18, 0x00,
363    0x08, 0x10, 0x80, 0x01, 0x01};
364
365    #if RSA_KEY_SIEVE && SIMULATION && RSA_INSTRUMENT
366    UINT32 PrimeIndex = 0;
367    UINT32 failedAtIteration[10] = {0};
368    UINT32 PrimeCounts[3] = {0};
369    UINT32 MillerRabinTrials[3] = {0};
370    UINT32 totalFieldsSieved[3] = {0};
371    UINT32 bitsInFieldAfterSieve[3] = {0};
372    UINT32 emptyFieldsSieved[3] = {0};
373    UINT32 noPrimeFields[3] = {0};
374    UINT32 primesChecked[3] = {0};
375    UINT16 lastSievePrime = 0;
376    #endif

```

10.2.21 RsaKeyCache.c

10.2.21.1 Introduction

This file contains the functions to implement the RSA key cache that can be used to speed up simulation.

Only one key is created for each supported key size and it is returned whenever a key of that size is requested.

If desired, the key cache can be populated from a file. This allows multiple TPM to run with the same RSA keys. Also, when doing simulation, the DRBG will use preset sequences so it is not too hard to repeat sequences for debug or profile or stress.

When the key cache is enabled, a call to `CryptRsaGenerateKey()` will call the `GetCachedRsaKey()`. If the cache is enabled and populated, then the cached key of the requested size is returned. If a key of the requested size is not available, the no key is loaded and the requested key will need to be generated. If the cache is not populated, the TPM will open a file that has the appropriate name for the type of keys required (CRT or no-CRT). If the file is the right size, it is used. If the file doesn't exist or the file does not have the correct size, the TPM will populate the cache with new keys of the required size and write the cache data to the file so that they will be available the next time.

Currently, if two simulations are being run with TPM's that have different RSA key sizes (e.g., one with 1024 and 2048 and another with 2048 and 3072, then the files will not match for the both of them and they will both try to overwrite the other's cache file. I may try to do something about this if necessary.

10.2.21.2 Includes, Types, Locals, and Defines

```

1  #include "Tpm.h"
2
3  #if USE_RSA_KEY_CACHE
4
5  #include <stdio.h>
6  #include "RsaKeyCache_fp.h"
7
8  #if CRT_FORMAT_RSA == YES
9  #define CACHE_FILE_NAME "RsaKeyCacheCrt.data"
10 #else
11 #define CACHE_FILE_NAME "RsaKeyCacheNoCrt.data"
12 #endif
13
14 typedef struct _RSA_KEY_CACHE_
15 {
16     TPM2B_PUBLIC_KEY_RSA    publicModulus;
17     TPM2B_PRIVATE_KEY_RSA   privateExponent;
18 } RSA_KEY_CACHE;
```

Determine the number of RSA key sizes for the cache

```

19 TPMI_RSA_KEY_BITS    SupportedRsaKeySizes[] = {
20 #if RSA_1024
21     1024,
22 #endif
23 #if RSA_2048
24     2048,
25 #endif
26 #if RSA_3072
27     3072,
28 #endif
29 #if RSA_4096
30     4096,
31 #endif
```

```

32     0
33 };
34
35 #define RSA_KEY_CACHE_ENTRIES (RSA_1024 + RSA_2048 + RSA_3072 + RSA_4096)

```

The key cache holds one entry for each of the supported key sizes

```

36 RSA_KEY_CACHE      s_rsaKeyCache[RSA_KEY_CACHE_ENTRIES];

```

Indicates if the key cache is loaded. It can be loaded and enabled or disabled.

```

37 BOOL              s_keyCacheLoaded = 0;

```

Indicates if the key cache is enabled

```

38 int               s_rsaKeyCacheEnabled = FALSE;

```

10.2.21.2.1 RsaKeyCacheControl()

Used to enable and disable the RSA key cache.

```

39 LIB_EXPORT void
40 RsaKeyCacheControl(
41     int             state
42 )
43 {
44     s_rsaKeyCacheEnabled = state;
45 }

```

10.2.21.2.2 InitializeKeyCache()

This will initialize the key cache and attempt to write it to a file for later use.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

46 static BOOL
47 InitializeKeyCache(
48     TPMT_PUBLIC      *publicArea,
49     TPMT_SENSITIVE   *sensitive,
50     RAND_STATE        *rand
51                     // IN: if not NULL, the deterministic
52                     //      RNG state
53 )
54 {
55     int             index;
56     TPM_KEY_BITS    keySave = publicArea->parameters.rsaDetail.keyBits;
57     BOOL            OK = TRUE;
58     s_rsaKeyCacheEnabled = FALSE;
59     for(index = 0; OK && index < RSA_KEY_CACHE_ENTRIES; index++)
60     {
61         publicArea->parameters.rsaDetail.keyBits
62             = SupportedRsaKeySizes[index];
63         OK = (CryptRsaGenerateKey(publicArea, sensitive, rand) == TPM_RC_SUCCESS);
64         if(OK)
65         {
66             s_rsaKeyCache[index].publicModulus = publicArea->unique.rsa;
67             s_rsaKeyCache[index].privateExponent = sensitive->sensitive.rsa;

```



```

68     }
69 }
70 publicArea->parameters.rsaDetail.keyBits = keySave;
71 s_keyCacheLoaded = OK;
72 #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
73     if(OK)
74     {
75         FILE                *cacheFile;
76         const char          *fn = CACHE_FILE_NAME;
77
78 #if defined _MSC_VER
79         if(fopen_s(&cacheFile, fn, "w+b") != 0)
80 #else
81         cacheFile = fopen(fn, "w+b");
82         if(NULL == cacheFile)
83 #endif
84         {
85             printf("Can't open %s for write.\n", fn);
86         }
87     else
88     {
89         fseek(cacheFile, 0, SEEK_SET);
90         if(fwrite(s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
91            != sizeof(s_rsaKeyCache))
92         {
93             printf("Error writing cache to %s.", fn);
94         }
95     }
96     if(cacheFile)
97         fclose(cacheFile);
98 }
99 #endif
100 return s_keyCacheLoaded;
101 }

```

10.2.21.2.3 KeyCacheLoaded()

Checks that key cache is loaded.

Return Value	Meaning
TRUE(1)	cache loaded
FALSE(0)	cache not loaded

```

102 static BOOL
103 KeyCacheLoaded(
104     TPMT_PUBLIC      *publicArea,
105     TPMT_SENSITIVE   *sensitive,
106     RAND_STATE       *rand          // IN: if not NULL, the deterministic
107                                     // RNG state
108 )
109 {
110 #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
111     if(!s_keyCacheLoaded)
112     {
113         FILE                *cacheFile;
114         const char *        fn = CACHE_FILE_NAME;
115 #if defined _MSC_VER && 1
116         if(fopen_s(&cacheFile, fn, "r+b") == 0)
117 #else
118         cacheFile = fopen(fn, "r+b");
119         if(NULL != cacheFile)
120 #endif
121 #endif

```

```

121     {
122         fseek(cacheFile, 0L, SEEK_END);
123         if(ftell(cacheFile) == sizeof(s_rsaKeyCache))
124         {
125             fseek(cacheFile, 0L, SEEK_SET);
126             s_keyCacheLoaded = (
127                 fread(&s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
128                 == sizeof(s_rsaKeyCache));
129         }
130         fclose(cacheFile);
131     }
132 }
133 #endif
134 if(!s_keyCacheLoaded)
135     s_rsaKeyCacheEnabled = InitializeKeyCache(publicArea, sensitive, rand);
136 return s_keyCacheLoaded;
137 }

```

10.2.21.2.4 GetCachedRsaKey()

Return Value	Meaning
TRUE(1)	key loaded
FALSE(0)	key not loaded

```

138 BOOL
139 GetCachedRsaKey(
140     TPMT_PUBLIC      *publicArea,
141     TPMT_SENSITIVE   *sensitive,
142     RAND_STATE        *rand          // IN: if not NULL, the deterministic
143                                     // RNG state
144 )
145 {
146     int keyBits = publicArea->parameters.rsaDetail.keyBits;
147     int index;
148     //
149     if(KeyCacheLoaded(publicArea, sensitive, rand))
150     {
151         for(index = 0; index < RSA_KEY_CACHE_ENTRIES; index++)
152         {
153             if((s_rsaKeyCache[index].publicModulus.t.size * 8) == keyBits)
154             {
155                 publicArea->unique.rsa = s_rsaKeyCache[index].publicModulus;
156                 sensitive->sensitive.rsa = s_rsaKeyCache[index].privateExponent;
157                 return TRUE;
158             }
159         }
160         return FALSE;
161     }
162     return s_keyCacheLoaded;
163 }
164 #endif // defined SIMULATION && defined USE_RSA_KEY_CACHE

```

10.2.22 Ticket.c

10.2.22.1 Introduction

10.2.22.2 Includes

```
1 #include "Tpm.h"
```

10.2.22.3 Functions

10.2.22.3.1 TicketIsSafe()

This function indicates if producing a ticket is safe. It checks if the leading bytes of an input buffer is TPM_GENERATED_VALUE or its substring of canonical form. If so, it is not safe to produce ticket for an input buffer claiming to be TPM generated buffer

Return Value	Meaning
TRUE(1)	safe to produce ticket
FALSE(0)	not safe to produce ticket

```
2  BOOL
3  TicketIsSafe(
4      TPM2B          *buffer
5  )
6  {
7      TPM_CONSTANTS32 valueToCompare = TPM_GENERATED_VALUE;
8      BYTE             bufferToCompare[sizeof(valueToCompare)];
9      BYTE             *marshalBuffer;
10     //
11     // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume
12     // it is not safe to generate a ticket
13     if(buffer->size < sizeof(valueToCompare))
14         return FALSE;
15     marshalBuffer = bufferToCompare;
16     TPM_CONSTANTS32_Marshal(&valueToCompare, &marshalBuffer, NULL);
17     if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))
18         return FALSE;
19     else
20         return TRUE;
21 }
```

10.2.22.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```
22 void
23 TicketComputeVerified(
24     TPMI_RH_HIERARCHY hierarchy,    // IN: hierarchy constant for ticket
25     TPM2B_DIGEST       *digest,     // IN: digest
26     TPM2B_NAME         *keyName,    // IN: name of key that signed the values
27     TPMT_TK_VERIFIED   *ticket,     // OUT: verified ticket
28 )
29 {
30     TPM2B_PROOF         *proof;
31     HMAC_STATE          hmacState;
32     //
33     // Fill in ticket fields
```

```

34     ticket->tag = TPM_ST_VERIFIED;
35     ticket->hierarchy = hierarchy;
36     proof = HierarchyGetProof(hierarchy);
37
38     // Start HMAC using the proof value of the hierarchy as the HMAC key
39     ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
40                                             &proof->b);
41     // TPM_ST_VERIFIED
42     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
43     // digest
44     CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
45     // key name
46     CryptDigestUpdate2B(&hmacState.hashState, &keyName->b);
47     // done
48     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
49
50     return;
51 }

```

10.2.22.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

```

52 void
53 TicketComputeAuth(
54     TPM_ST          type,           // IN: the type of ticket.
55     TPMI_RH_HIERARCHY hierarchy,    // IN: hierarchy constant for ticket
56     UINT64          timeout,        // IN: timeout
57     BOOL            expiresOnReset, // IN: flag to indicate if ticket expires on
58                                     // TPM Reset
59     TPM2B_DIGEST    *cpHashA,       // IN: input cpHashA
60     TPM2B_NONCE     *policyRef,      // IN: input policyRef
61     TPM2B_NAME       *entityName,    // IN: name of entity
62     TPMT_TK_AUTH     *ticket,        // OUT: Created ticket
63 )
64 {
65     TPM2B_PROOF      *proof;
66     HMAC_STATE        hmacState;
67
68     //
69     // Get proper proof
70     proof = HierarchyGetProof(hierarchy);
71
72     // Fill in ticket fields
73     ticket->tag = type;
74     ticket->hierarchy = hierarchy;
75
76     // Start HMAC with hierarchy proof as the HMAC key
77     ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
78                                             &proof->b);
79     // TPM_ST_AUTH_SECRET or TPM_ST_AUTH_SIGNED,
80     CryptDigestUpdateInt(&hmacState, sizeof(UINT16), ticket->tag);
81     // cpHash
82     CryptDigestUpdate2B(&hmacState.hashState, &cpHashA->b);
83     // policyRef
84     CryptDigestUpdate2B(&hmacState.hashState, &policyRef->b);
85     // keyName
86     CryptDigestUpdate2B(&hmacState.hashState, &entityName->b);
87     // timeout
88     CryptDigestUpdateInt(&hmacState, sizeof(timeout), timeout);
89     if(timeout != 0)
90     {
91         // epoch
92         CryptDigestUpdateInt(&hmacState.hashState, sizeof(CLOCK_NONCE),
93                             g_timeEpoch);
94     }
95 }

```

```

93         // reset count
94         if(expiresOnReset)
95             CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
96                                 gp.totalResetCount);
97     }
98     // done
99     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
100
101     return;
102 }

```

10.2.22.3.4 TicketComputeHashCheck()

This function creates a TPMT_TK_HASHCHECK ticket.

```

103 void
104 TicketComputeHashCheck(
105     TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy constant for ticket
106     TPM_ALG_ID hashAlg, // IN: the hash algorithm for 'digest'
107     TPM2B_DIGEST *digest, // IN: input digest
108     TPMT_TK_HASHCHECK *ticket // OUT: Created ticket
109 )
110 {
111     TPM2B_PROOF *proof;
112     HMAC_STATE hmacState;
113     //
114     // Get proper proof
115     proof = HierarchyGetProof(hierarchy);
116
117     // Fill in ticket fields
118     ticket->tag = TPM_ST_HASHCHECK;
119     ticket->hierarchy = hierarchy;
120
121     // Start HMAC using hierarchy proof as HMAC key
122     ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
123                                             &proof->b);
124     // TPM ST_HASHCHECK
125     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
126     // hash algorithm
127     CryptDigestUpdateInt(&hmacState, sizeof(hashAlg), hashAlg);
128     // digest
129     CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
130     // done
131     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
132
133     return;
134 }

```

10.2.22.3.5 TicketComputeCreation()

This function creates a TPMT_TK_CREATION ticket.

```

135 void
136 TicketComputeCreation(
137     TPMI_RH_HIERARCHY hierarchy, // IN: hierarchy for ticket
138     TPM2B_NAME *name, // IN: object name
139     TPM2B_DIGEST *creation, // IN: creation hash
140     TPMT_TK_CREATION *ticket // OUT: created ticket
141 )
142 {
143     TPM2B_PROOF *proof;
144     HMAC_STATE hmacState;
145

```

```
146     // Get proper proof
147     proof = HierarchyGetProof(hierarchy);
148
149     // Fill in ticket fields
150     ticket->tag = TPM_ST_CREATION;
151     ticket->hierarchy = hierarchy;
152
153     // Start HMAC using hierarchy proof as HMAC key
154     ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
155                                             &proof->b);
156     // TPM_ST_CREATION
157     CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
158     // name if provided
159     if(name != NULL)
160         CryptDigestUpdate2B(&hmacState.hashState, &name->b);
161     // creation hash
162     CryptDigestUpdate2B(&hmacState.hashState, &creation->b);
163     // Done
164     CryptHmacEnd2B(&hmacState, &ticket->digest.b);
165
166     return;
167 }
```

10.2.23 TpmAsn1.c

10.2.23.1 Includes

```

1  #include "Tpm.h"
2  #define _OIDS_
3  #include "OIDS.h"
4  #include "TpmASN1.h"
5  #include "TpmASN1_fp.h"

```

10.2.23.2 Unmarshaling Functions

10.2.23.2.1 ASN1UnmarshalContextInitialize()

Function does standard initialization of a context.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

6  BOOL
7  ASN1UnmarshalContextInitialize(
8      ASN1UnmarshalContext *ctx,
9      INT16 size,
10     BYTE *buffer
11 )
12 {
13     VERIFY(buffer != NULL);
14     VERIFY(size > 0);
15     ctx->buffer = buffer;
16     ctx->size = size;
17     ctx->offset = 0;
18     ctx->tag = 0xFF;
19     return TRUE;
20 Error:
21     return FALSE;
22 }

```

10.2.23.2.2 ASN1DecodeLength()

This function extracts the length of an element from *buffer* starting at *offset*.

Return Value	Meaning
≥ 0	the extracted length
< 0	an error

```

23  INT16
24  ASN1DecodeLength(
25      ASN1UnmarshalContext *ctx
26  )
27  {
28      BYTE first; // Next octet in buffer
29      INT16 value;
30      //
31      VERIFY(ctx->offset < ctx->size);
32      first = NEXT_OCTET(ctx);

```



```

33 // If the number of octets of the entity is larger than 127, then the first octet
34 // is the number of octets in the length specifier.
35 if(first >= 0x80)
36 {
37     // Make sure that this length field is contained with the structure being
38     // parsed
39     CHECK_SIZE(ctx, (first & 0x7F));
40     if(first == 0x82)
41     {
42         // Two octets of size
43         // get the next value
44         value = (INT16)NEXT_OCTET(ctx);
45         // Make sure that the result will fit in an INT16
46         VERIFY(value < 0x0080);
47         // Shift up and add next octet
48         value = (value << 8) + NEXT_OCTET(ctx);
49     }
50     else if(first == 0x81)
51         value = NEXT_OCTET(ctx);
52     // Sizes larger than will fit in a INT16 are an error
53     else
54         goto Error;
55 }
56 else
57     value = first;
58 // Make sure that the size defined something within the current context
59 CHECK_SIZE(ctx, value);
60 return value;
61 Error:
62 ctx->size = -1; // Makes everything fail from now on.
63 return -1;
64 }

```

10.2.23.2.3 ASN1NextTag()

This function extracts the next type from *buffer* starting at *offset*. It advances *offset* as it parses the type and the length of the type. It returns the length of the type. On return, the *length* octets starting at *offset* are the octets of the type.

Return Value	Meaning
>=0	the number of octets in <i>type</i>
<0	an error

```

65 INT16
66 ASN1NextTag(
67     ASN1UnmarshalContext *ctx
68 )
69 {
70     // A tag to get?
71     VERIFY(ctx->offset < ctx->size);
72     // Get it
73     ctx->tag = NEXT_OCTET(ctx);
74     // Make sure that it is not an extended tag
75     VERIFY((ctx->tag & 0x1F) != 0x1F);
76     // Get the length field and return that
77     return ASN1DecodeLength(ctx);
78 }
79 Error:
80 // Attempt to read beyond the end of the context or an illegal tag
81 ctx->size = -1; // Persistent failure
82 ctx->tag = 0xFF;
83 return -1;

```

84 }

10.2.23.2.4 ASN1GetBitStringValue()

Try to parse a bit string of up to 32 bits from a value that is expected to be a bit string. The bit string is left justified so that the MSb of the input is the MSb of the returned value. If there is a general parsing error, the *context*→*size* is set to -1.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

85  BOOL
86  ASN1GetBitStringValue(
87      ASN1UnmarshalContext  *ctx,
88      UINT32                 *val
89  )
90  {
91      int                shift;
92      INT16             length;
93      UINT32            value = 0;
94      int                inputBits;
95      //
96      length = ASN1NextTag(ctx);
97      VERIFY(length >= 1);
98      VERIFY(ctx->tag == ASN1_BITSTRING);
99      // Get the shift value for the bit field (how many bits to lop off of the end)
100     shift = NEXT_OCTET(ctx);
101     length--;
102     // Get the number of bits in the input
103     inputBits = (8 * length) - shift;
104     // the shift count has to make sense
105     VERIFY((shift < 8) && ((length > 0) || (shift == 0)));
106     // if there are any bytes left
107     for(; length > 1; length--)
108     {
109         // for all but the last octet, just shift and add the new octet
110         VERIFY((value & 0xFF000000) == 0); // can't loose significant bits
111         value = (value << 8) + NEXT_OCTET(ctx);
112     }
113     if(length == 1)
114     {
115         // for the last octet, just shift the accumulated value enough to
116         // accept the significant bits in the last octet and shift the last
117         // octet down
118         VERIFY(((value & (0xFF000000 << (8 - shift)))) == 0);
119         value = (value << (8 - shift)) + (NEXT_OCTET(ctx) >> shift);
120     }
121     // 'Left justify' the result
122     if(inputBits > 0)
123         value <<= (32 - inputBits);
124     *val = value;
125     return TRUE;
126 Error:
127     ctx->size = -1;
128     return FALSE;
129 }

```

10.2.23.3 Marshaling Functions

10.2.23.3.1 Introduction

Marshaling of an ASN.1 structure is accomplished from the bottom up. That is, the things that will be at the end of the structure are added last. To manage the collecting of the relative sizes, start a context for the outermost container, if there is one, and then placing items in from the bottom up. If the bottom-most item is also within a structure, create a nested context by calling `ASN1StartMarshalingContext()`.

The context control structure contains a *buffer* pointer, an *offset*, an *end* and a stack. *offset* is the offset from the start of the buffer of the last added byte. When *offset* reaches 0, the buffer is full. *offset* is a signed value so that, when it becomes negative, there is an overflow. Only two functions are allowed to move bytes into the buffer: `ASN1PushByte()` and `ASN1PushBytes()`. These functions make sure that no data is written beyond the end of the buffer.

When a new context is started, the current value of *end* is pushed on the stack and *end* is set to *offset*. As bytes are added, *offset* gets smaller. At any time, the count of bytes in the current context is simply *end* - *offset*.

Since starting a new context involves setting *end* = *offset*, the number of bytes in the context starts at 0. The nominal way of ending a context is to use *end* - *offset* to set the length value, and then a tag is added to the buffer. Then the previous *end* value is popped meaning that the context just ended becomes a member of the now current context.

The nominal strategy for building a completed ASN.1 structure is to push everything into the buffer and then move everything to the start of the buffer. The move is simple as the size of the move is the initial *end* value minus the final *offset* value. The destination is *buffer* and the source is *buffer* + *offset*. As Skippy would say **Easy peasy, Joe.**

It is not necessary to provide a buffer into which the data is placed. If no buffer is provided, then the marshaling process will return values needed for marshaling. One strategy for filling the buffer would be to execute the process for building the structure without using a buffer. This would return the overall size of the structure. Then that amount of data could be allocated for the buffer and the fill process executed again with the data going into the buffer. At the end, the data would be in its final resting place.

10.2.23.3.2 ASN1InitializeMarshalContext()

This creates a structure for handling marshaling of an ASN.1 formatted data structure.

```

133 void
134 ASN1InitializeMarshalContext(
135     ASN1MarshalContext *ctx,
136     INT16 length,
137     BYTE *buffer
138 )
139 {
140     ctx->buffer = buffer;
141     if(buffer)
142         ctx->offset = length;
143     else
144         ctx->offset = INT16_MAX;
145     ctx->end = ctx->offset;
146     ctx->depth = -1;
147 }
```

10.2.23.3.3 ASN1StartMarshalContext()

This starts a new constructed element. It is constructed on *top* of the value that was previously placed in the structure.

```

148 void
149 ASN1StartMarshalContext(
150     ASN1MarshalContext *ctx
151 )
152 {
153     pAssert((ctx->depth + 1) < MAX_DEPTH);
154     ctx->depth++;
155     ctx->ends[ctx->depth] = ctx->end;
156     ctx->end = ctx->offset;
157 }

```

10.2.23.3.4 ASN1EndMarshalContext()

This function restores the end pointer for an encapsulating structure.

Return Value	Meaning
> 0	the size of the encapsulated structure that was just ended
<= 0	an error

```

158 INT16
159 ASN1EndMarshalContext(
160     ASN1MarshalContext *ctx
161 )
162 {
163     INT16 length;
164     pAssert(ctx->depth >= 0);
165     length = ctx->end - ctx->offset;
166     ctx->end = ctx->ends[ctx->depth--];
167     if((ctx->depth == -1) && (ctx->buffer))
168     {
169         MemoryCopy(ctx->buffer, ctx->buffer + ctx->offset, ctx->end - ctx->offset);
170     }
171     return length;
172 }

```

10.2.23.3.5 ASN1EndEncapsulation()

This function puts a tag and length in the buffer. In this function, an embedded BIT_STRING is assumed to be a collection of octets. To indicate that all bits are used, a byte of zero is prepended. If a raw bit-string is needed, a new function like ASN1PushInteger() would be needed.

Return Value	Meaning
> 0	number of octets in the encapsulation
== 0	failure

```

173 UINT16
174 ASN1EndEncapsulation(
175     ASN1MarshalContext *ctx,
176     BYTE tag
177 )
178 {
179     // only add a leading zero for an encapsulated BIT STRING
180     if (tag == ASN1_BITSTRING)
181         ASN1PushByte(ctx, 0);
182     ASN1PushTagAndLength(ctx, tag, ctx->end - ctx->offset);
183     return ASN1EndMarshalContext(ctx);
184 }

```

10.2.23.3.6 ASN1PushByte()

```

185  BOOL
186  ASN1PushByte(
187      ASN1MarshalContext    *ctx,
188      BYTE                   b
189  )
190  {
191      if(ctx->offset > 0)
192      {
193          ctx->offset -= 1;
194          if(ctx->buffer)
195              ctx->buffer[ctx->offset] = b;
196          return TRUE;
197      }
198      ctx->offset = -1;
199      return FALSE;
200  }

```

10.2.23.3.7 ASN1PushBytes()

Push some raw bytes onto the buffer. *count* cannot be zero.

Return Value	Meaning
> 0	count bytes
== 0	failure unless count was zero

```

201  INT16
202  ASN1PushBytes(
203      ASN1MarshalContext    *ctx,
204      INT16                  count,
205      const BYTE             *buffer
206  )
207  {
208      // make sure that count is not negative which would mess up the math; and that
209      // if there is a count, there is a buffer
210      VERIFY((count >= 0) && ((buffer != NULL) || (count == 0)));
211      // back up the offset to determine where the new octets will get pushed
212      ctx->offset -= count;
213      // can't go negative
214      VERIFY(ctx->offset >= 0);
215      // if there are buffers, move the data, otherwise, assume that this is just a
216      // test.
217      if(count && buffer && ctx->buffer)
218          MemoryCopy(&ctx->buffer[ctx->offset], buffer, count);
219      return count;
220  Error:
221      ctx->offset = -1;
222      return 0;
223  }

```

10.2.23.3.8 ASN1PushNull()

Return Value	Meaning
> 0	count bytes
== 0	failure unless count was zero

```

224  INT16

```

```

225 ASN1PushNull(
226     ASN1MarshalContext    *ctx
227 )
228 {
229     ASN1PushByte(ctx, 0);
230     ASN1PushByte(ctx, ASN1_NULL);
231     return (ctx->offset >= 0) ? 2 : 0;
232 }

```

10.2.23.3.9 ASN1PushLength()

Push a length value. This will only handle length values that fit in an INT16.

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

233 INT16
234 ASN1PushLength(
235     ASN1MarshalContext    *ctx,
236     INT16                  len
237 )
238 {
239     UINT16                  start = ctx->offset;
240     VERIFY(len >= 0);
241     if(len <= 127)
242         ASN1PushByte(ctx, (BYTE)len);
243     else
244     {
245         ASN1PushByte(ctx, (BYTE)(len & 0xFF));
246         len >>= 8;
247         if(len == 0)
248             ASN1PushByte(ctx, 0x81);
249         else
250         {
251             ASN1PushByte(ctx, (BYTE)(len));
252             ASN1PushByte(ctx, 0x82);
253         }
254     }
255     goto Exit;
256 Error:
257     ctx->offset = -1;
258 Exit:
259     return (ctx->offset > 0) ? start - ctx->offset : 0;
260 }

```

10.2.23.3.10 ASN1PushTagAndLength()

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

261 INT16
262 ASN1PushTagAndLength(
263     ASN1MarshalContext    *ctx,
264     BYTE                  tag,
265     INT16                  length
266 )
267 {

```

```

268     INT16      bytes;
269     bytes = ASN1PushLength(ctx, length);
270     bytes += (INT16)ASN1PushByte(ctx, tag);
271     return (ctx->offset < 0) ? 0 : bytes;
272 }

```

10.2.23.3.11 ASN1PushTaggedOctetString()

This function will push a random octet string.

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

273  INT16
274  ASN1PushTaggedOctetString(
275      ASN1MarshalContext    *ctx,
276      INT16                  size,
277      const BYTE             *string,
278      BYTE                   tag
279  )
280  {
281      ASN1PushBytes(ctx, size, string);
282      // PushTagAndLength just tells how many octets it added so the total size of this
283      // element is the sum of those octets and input size.
284      size += ASN1PushTagAndLength(ctx, tag, size);
285      return size;
286  }

```

10.2.23.3.12 ASN1PushUINT()

This function pushes an native-endian integer value. This just changes a native-endian integer into a big-endian byte string and calls ASN1PushInteger(). That function will remove leading zeros and make sure that the number is positive.

Return Value	Meaning
> 0	count bytes
== 0	failure unless count was zero

```

287  INT16
288  ASN1PushUINT(
289      ASN1MarshalContext    *ctx,
290      UINT32                 integer
291  )
292  {
293      BYTE                  marshaled[4];
294      UINT32_TO_BYTE_ARRAY(integer, marshaled);
295      return ASN1PushInteger(ctx, 4, marshaled);
296  }

```

10.2.23.3.13 ASN1PushInteger()

Push a big-endian integer on the end of the buffer

Return Value	Meaning
> 0	the number of bytes marshaled for the integer
== 0	failure

```

297  INT16
298  ASN1PushInteger(
299      ASN1MarshalContext  *ctx,          // IN/OUT: buffer context
300      INT16                iLen,          // IN: octets of the integer
301      BYTE                 *integer       // IN: big-endian integer
302  )
303  {
304      // no leading 0's
305      while((*integer == 0) && (--iLen > 0))
306          integer++;
307      // Move the bytes to the buffer
308      ASN1PushBytes(ctx, iLen, integer);
309      // if needed, add a leading byte of 0 to make the number positive
310      if(*integer & 0x80)
311          iLen += (INT16)ASN1PushByte(ctx, 0);
312      // PushTagAndLength just tells how many octets it added so the total size of this
313      // element is the sum of those octets and the adjusted input size.
314      iLen += ASN1PushTagAndLength(ctx, ASN1_INTEGER, iLen);
315      return iLen;
316  }

```

10.2.23.3.14 ASN1PushOID()

This function is used to add an OID. An OID is 0x06 followed by a byte of size followed by size bytes. This is used to avoid having to do anything special in the definition of an OID.

Return Value	Meaning
> 0	the number of bytes marshaled for the integer
== 0	failure

```

317  INT16
318  ASN1PushOID(
319      ASN1MarshalContext  *ctx,          *ctx,
320      const BYTE           *OID          *OID
321  )
322  {
323      if((*OID == ASN1_OBJECT_IDENTIFIER) && ((OID[1] & 0x80) == 0))
324      {
325          return ASN1PushBytes(ctx, OID[1] + 2, OID);
326      }
327      ctx->offset = -1;
328      return 0;
329  }

```

10.2.24 X509_ECC.c

10.2.24.1 Includes

```

1  #include "Tpm.h"
2  #include "X509.h"
3  #include "OIDs.h"
4  #include "TpmASN1_fp.h"
5  #include "X509_ECC_fp.h"
6  #include "X509_spt_fp.h"
7  #include "CryptHash_fp.h"

```

10.2.24.2 Functions

10.2.24.2.1 X509PushPoint()

This seems like it might be used more than once so...

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

8  INT16
9  X509PushPoint(
10     ASN1MarshalContext    *ctx,
11     TPMS_ECC_POINT        *p
12 )
13 {
14     // Push a bit string containing the public key. For now, push the x, and y
15     // coordinates of the public point, bottom up
16     ASN1StartMarshalContext(ctx); // BIT STRING
17     {
18         ASN1PushBytes(ctx, p->y.t.size, p->y.t.buffer);
19         ASN1PushBytes(ctx, p->x.t.size, p->x.t.buffer);
20         ASN1PushByte(ctx, 0x04);
21     }
22     return ASN1EndEncapsulation(ctx, ASN1_BITSTRING); // Ends BIT STRING
23 }

```

10.2.24.2.2 X509AddSigningAlgorithmECC()

This creates the signing algorithm data.

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

24  INT16
25  X509AddSigningAlgorithmECC(
26     OBJECT                *signKey,
27     TPMT_SIG_SCHEME        *scheme,
28     ASN1MarshalContext    *ctx
29 )
30 {
31     PHASH_DEF              hashDef = CryptGetHashDef(scheme->details.any.hashAlg);
32     //

```

```

33     NOT_REFERENCED(signKey);
34     // If the desired hashAlg definition wasn't found...
35     if(hashDef->hashAlg != scheme->details.any.hashAlg)
36         return 0;
37
38     switch(scheme->scheme)
39     {
40 #if ALG_ECDSA
41         case TPM_ALG_ECDSA:
42             // Make sure that we have an OID for this hash and ECC
43             if((hashDef->ECDSA)[0] != ASN1_OBJECT_IDENTIFIER)
44                 break;
45             // if this is just an implementation check, indicate that this
46             // combination is supported
47             if(!ctx)
48                 return 1;
49             ASN1StartMarshalContext(ctx);
50             ASN1PushOID(ctx, hashDef->ECDSA);
51             return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
52 #endif // ALG_ECDSA
53         default:
54             break;
55     }
56     return 0;
57 }

```

10.2.24.2.3 X509AddPublicECC()

This function will add the *publicKey* description to the DER data. If ctx is NULL, then no data is transferred and this function will indicate if the TPM has the values for DER-encoding of the public key.

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

58 INT16
59 X509AddPublicECC(
60     OBJECT *object,
61     ASN1MarshalContext *ctx
62 )
63 {
64     const BYTE *curveOid =
65         CryptEccGetOID(object->publicArea.parameters.eccDetail.curveID);
66     if((curveOid == NULL) || (*curveOid != ASN1_OBJECT_IDENTIFIER))
67         return 0;
68     //
69     //
70     // SEQUENCE (2 elem) 1st
71     // SEQUENCE (2 elem) 2nd
72     // OBJECT IDENTIFIER 1.2.840.10045.2.1 ecPublicKey (ANSI X9.62 public key type)
73     // OBJECT IDENTIFIER 1.2.840.10045.3.1.7 prime256v1 (ANSI X9.62 named curve)
74     // BIT STRING (520 bit) 00000100101000011101010101011100100110101000100000010...
75     //
76     // If this is a check to see if the key can be encoded, it can.
77     // Need to mark the end sequence
78     if(ctx == NULL)
79         return 1;
80     ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
81     {
82         X509PushPoint(ctx, &object->publicArea.unique.ecc); // BIT STRING
83         ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 2nd
84     }

```

```
85         ASN1PushOID(ctx, curveOid); // curve dependent
86         ASN1PushOID(ctx, OID_ECC_PUBLIC); // (1.2.840.10045.2.1)
87     }
88     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 2nd
89 }
90 return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 1st
91 }
```

DRAFT

10.2.25 X509_RSA.c

10.2.25.1 Includes

```

1  #include "Tpm.h"
2  #include "X509.h"
3  #include "TpmASN1_fp.h"
4  #include "X509_RSA_fp.h"
5  #include "X509_spt_fp.h"
6  #include "CryptHash_fp.h"
7  #include "CryptRsa_fp.h"

```

10.2.25.2 Functions

```

8  #if ALG_RSA

```

10.2.25.2.1 X509AddSigningAlgorithmRSA()

This creates the signing algorithm data.

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

9  INT16
10 X509AddSigningAlgorithmRSA(
11     OBJECT *signKey,
12     TPMT_SIG_SCHEME *scheme,
13     ASN1MarshalContext *ctx
14 )
15 {
16     TPM_ALG_ID hashAlg = scheme->details.any.hashAlg;
17     PHASH_DEF hashDef = CryptGetHashDef(hashAlg);
18     //
19     NOT_REFERENCED(signKey);
20     // return failure if hash isn't implemented
21     if(hashDef->hashAlg != hashAlg)
22         return 0;
23     switch(scheme->scheme)
24     {
25     case TPM_ALG_RSASSA:
26     {
27         // if the hash is implemented but there is no PKCS1 OID defined
28         // then this is not a valid signing combination.
29         if(hashDef->PKCS1[0] != ASN1_OBJECT_IDENTIFIER)
30             break;
31         if(ctx == NULL)
32             return 1;
33         return X509PushAlgorithmIdentifierSequence(ctx, hashDef->PKCS1);
34     }
35     case TPM_ALG_RSAPSS:
36         // leave if this is just an implementation check
37         if(ctx == NULL)
38             return 1;
39         // In the case of SHA1, everything is default and RFC4055 says that
40         // implementations that do signature generation MUST omit the parameter
41         // when defaults are used. )-:
42         if(hashDef->hashAlg == TPM_ALG_SHA1)
43         {
44             return X509PushAlgorithmIdentifierSequence(ctx, OID_RSAPSS);

```

```

45     }
46     else
47     {
48         // Going to build something that looks like:
49         // SEQUENCE (2 elem)
50         //   OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
51         //   SEQUENCE (3 elem)
52         //     [0] (1 elem)
53         //       SEQUENCE (2 elem)
54         //         OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
55         //         NULL
56         //     [1] (1 elem)
57         //       SEQUENCE (2 elem)
58         //         OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
59         //         SEQUENCE (2 elem)
60         //           OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
61         //           NULL
62         //     [2] (1 elem) salt length
63         //     INTEGER 32
64
65         // The indentation is just to keep track of where we are in the
66         // structure
67         ASN1StartMarshalContext(ctx); // SEQUENCE (2 elements)
68         {
69             ASN1StartMarshalContext(ctx); // SEQUENCE (3 elements)
70             {
71                 // [2] (1 elem) salt length
72                 // INTEGER 32
73                 ASN1StartMarshalContext(ctx);
74                 {
75                     INT16 saltSize =
76                         CryptRsaPssSaltSize((INT16)hashDef->digestSize,
77                         (INT16)signKey->publicArea.unique.rsa.t.size);
78                     ASN1PushUINT(ctx, saltSize);
79                 }
80                 ASN1EndEncapsulation(ctx, ASN1_APPLICATION_SPECIFIC + 2);
81
82                 // Add the mask generation algorithm
83                 // [1] (1 elem)
84                 // SEQUENCE (2 elem) 1st
85                 //   OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
86                 //   SEQUENCE (2 elem) 2nd
87                 //     OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
88                 //     NULL
89                 ASN1StartMarshalContext(ctx); // mask context [1] (1 elem)
90                 {
91                     ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
92                     // Handle the 2nd Sequence (sequence (object, null))
93                     {
94                         // This adds a NULL, then an OID and a SEQUENCE
95                         // wrapper.
96                         X509PushAlgorithmIdentifierSequence(ctx,
97                         hashDef->OID);
98                         // add the pkcs1-MGF OID
99                         ASN1PushOID(ctx, OID_MGF1);
100                     }
101                     // End outer sequence
102                     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
103                 }
104                 // End the [1]
105                 ASN1EndEncapsulation(ctx, ASN1_APPLICATION_SPECIFIC + 1);
106
107                 // Add the hash algorithm
108                 // [0] (1 elem)
109                 // SEQUENCE (2 elem) (done by
110                 // X509PushAlgorithmIdentifierSequence)

```

```

111         // OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256 (NIST)
112         // NULL
113         ASN1StartMarshalContext(ctx); // [0] (1 elem)
114         {
115             X509PushAlgorithmIdentifierSequence(ctx, hashDef->OID);
116         }
117         ASN1EndEncapsulation(ctx, (ASN1_APPLICATION_SPECIFIC + 0));
118     }
119     // SEQUENCE (3 elements) end
120     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
121
122     // RSA PSS OID
123     // OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
124     ASN1PushOID(ctx, OID_RSAPSS);
125 }
126 // End Sequence (2 elements)
127 return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
128 }
129 default:
130     break;
131 }
132 return 0;
133 }

```

10.2.25.2.2 X509AddPublicRSA()

This function will add the *publicKey* description to the DER data. If *fillPtr* is NULL, then no data is transferred and this function will indicate if the TPM has the values for DER-encoding of the public key.

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

134 INT16
135 X509AddPublicRSA(
136     OBJECT *object,
137     ASN1MarshalContext *ctx
138 )
139 {
140     UINT32 exp = object->publicArea.parameters.rsaDetail.exponent;
141     //
142
143     // If this is a check to see if the key can be encoded, it can.
144     // Need to mark the end sequence
145     if(ctx == NULL)
146         return 1;
147     ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
148     ASN1StartMarshalContext(ctx); // BIT STRING
149     ASN1StartMarshalContext(ctx); // SEQUENCE *(2 elem) 3rd
150
151     // Get public exponent in big-endian byte order.
152     if(exp == 0)
153         exp = RSA_DEFAULT_PUBLIC_EXPONENT;
154
155     // Push a 4 byte integer. This might get reduced if there are leading zeros or
156     // extended if the high order byte is negative.
157     ASN1PushUINT(ctx, exp);
158     // Push the public key as an integer
159     ASN1PushInteger(ctx, object->publicArea.unique.rsa.t.size,
160                    object->publicArea.unique.rsa.t.buffer);
161     // Embed this in a SEQUENCE tag and length in for the key, exponent sequence
162     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // SEQUENCE (3rd)

```



```
163
164 // Embed this in a BIT STRING
165 ASN1EndEncapsulation(ctx, ASN1_BITSTRING);
166
167 // Now add the formatted SEQUENCE for the RSA public key OID. This is a
168 // fully constructed value so it doesn't need to have a context started
169 X509PushAlgorithmIdentifierSequence(ctx, OID_PKCS1_PUB);
170
171 return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
172 }
173 #endif // ALG_RSA
```

DRAFT

10.2.26 X509_spt.c

10.2.26.1 Includes

```

1  #include "Tpm.h"
2  #include "TpmASN1.h"
3  #include "TpmASN1_fp.h"
4  #define _X509_SPT_
5  #include "X509.h"
6  #include "X509_spt_fp.h"
7  #if ALG_RSA
8  #   include "X509_RSA_fp.h"
9  #endif // ALG_RSA
10 #if ALG_ECC
11 #   include "X509_ECC_fp.h"
12 #endif // ALG_ECC
13 #if ALG_SM2
14 //#   include "X509_SM2_fp.h"
15 #endif // ALG_RSA

```

10.2.26.2 Unmarshaling Functions

10.2.26.2.1 X509FindExtensionByOID()

This will search a list of X509 extensions to find an extension with the requested OID. If the extension is found, the output context (*ctx*) is set up to point to the OID in the extension.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure (could be catastrophic)

```

16  BOOL
17  X509FindExtensionByOID(
18      ASN1UnmarshalContext *ctxIn,           // IN: the context to search
19      ASN1UnmarshalContext *ctx,             // OUT: the extension context
20      const BYTE *OID,                       // IN: oid to search for
21  )
22  {
23      INT16 length;
24      //
25      pAssert(ctxIn != NULL);
26      // Make the search non-destructive of the input if ctx provided. Otherwise, use
27      // the provided context.
28      if (ctx == NULL)
29          ctx = ctxIn;
30      // if the provided search context is different from the context of the extension,
31      // then copy the search context to the search context.
32      else if (ctx != ctxIn)
33          *ctx = *ctxIn;
34      // Now, search in the extension context
35      for(; ctx->size > ctx->offset; ctx->offset += length)
36      {
37          VERIFY((length = ASN1NextTag(ctx)) >= 0);
38          // If this is not a constructed sequence, then it doesn't belong
39          // in the extensions.
40          VERIFY(ctx->tag == ASN1_CONSTRUCTED_SEQUENCE);
41          // Make sure that this entry could hold the OID
42          if (length >= OID_SIZE(OID))
43          {
44              // See if this is a match for the provided object identifier.

```

```

45         if (MemoryEqual(OID, &(ctx->buffer[ctx->offset]), OID_SIZE(OID)))
46         {
47             // Return with ' ctx' set to point to the start of the OID with the
size
48             // set to be the size of the SEQUENCE
49             ctx->buffer += ctx->offset;
50             ctx->offset = 0;
51             ctx->size = length;
52             return TRUE;
53         }
54     }
55 }
56 VERIFY(ctx->offset == ctx->size);
57 return FALSE;
58 Error:
59     ctxIn->size = -1;
60     ctx->size = -1;
61     return FALSE;
62 }

```

10.2.26.2.2 X509GetExtensionBits()

This function will extract a bit field from an extension. If the extension doesn't contain a bit string, it will fail.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

63 UINT32
64 X509GetExtensionBits(
65     ASN1UnmarshalContext      *ctx,
66     UINT32                    *value
67 )
68 {
69     INT16    length;
70 //
71 while (((length = ASN1NextTag(ctx)) > 0) && (ctx->size > ctx->offset))
72 {
73     // Since this is an extension, the extension value will be in an OCTET STRING
74     if (ctx->tag == ASN1_OCTET_STRING)
75     {
76         return ASN1GetBitStringValue(ctx, value);
77     }
78     ctx->offset += length;
79 }
80 ctx->size = -1;
81 return FALSE;
82 }

```

10.2.26.2.3 X509ProcessExtensions()

This function is used to process the TPMA_OBJECT and KeyUsage() extensions. It is not in the CertifyX509.c code because it makes the code harder to follow.

Error Return	Meaning
TPM_RCS_ATTRIBUTES	the attributes of object are not consistent with the extension setting
TPM_RC_VALUE	problem parsing the extensions

```

83  TPM_RC
84  X509ProcessExtensions(
85      OBJECT          *object,          // IN: The object with the attributes to
86                                     //      check
87      stringRef        *extension       // IN: The start and length of the extensions
88  )
89  {
90      ASN1UnmarshalContext    ctx;
91      ASN1UnmarshalContext    extensionCtx;
92      INT16                   length;
93      UINT32                  value;
94      TPMA_OBJECT              attributes = object->publicArea.objectAttributes;
95  //
96      if(!ASN1UnmarshalContextInitialize(&ctx, extension->len, extension->buf)
97          || ((length = ASN1NextTag(&ctx)) < 0)
98          || (ctx.tag != X509_EXTENSIONS))
99          return TPM_RCS_VALUE;
100      if( ((length = ASN1NextTag(&ctx)) < 0)
101          || (ctx.tag != (ASN1_CONSTRUCTED_SEQUENCE)))
102          return TPM_RCS_VALUE;
103
104      // Get the extension for the TPMA_OBJECT if there is one
105      if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_TCG_TPMA_OBJECT) &&
106          X509GetExtensionBits(&extensionCtx, &value))
107      {
108          // If an keyAttributes extension was found, it must be exactly the same as the
109          // attributes of the object.
110          // NOTE: MemoryEqual() is used rather than a simple UINT32 compare to avoid
111          // type-punned pointer warning/error.
112          if(!MemoryEqual(&value, &attributes, sizeof(value)))
113              return TPM_RCS_ATTRIBUTES;
114      }
115      // Make sure the failure to find the value wasn't because of a fatal error
116      else if(extensionCtx.size < 0)
117          return TPM_RCS_VALUE;
118
119      // Get the keyUsage extension. This one is required
120      if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_KEY_USAGE_EXTENSION) &&
121          X509GetExtensionBits(&extensionCtx, &value))
122      {
123          x509KeyUsageUnion    keyUsage;
124          BOOL                  badSign;
125          BOOL                  badDecrypt;
126          BOOL                  badFixedTPM;
127          BOOL                  badRestricted;
128
129      //
130          keyUsage.integer = value;
131          // For KeyUsage:
132          // 1) 'sign' is SET if Key Usage includes signing
133          badSign = ((KEY_USAGE_SIGN.integer & keyUsage.integer) != 0)
134                  && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign);
135          // 2) 'decrypt' is SET if Key Usage includes decryption uses
136          badDecrypt = ((KEY_USAGE_DECRYPT.integer & keyUsage.integer) != 0)
137                     && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt);
138          // 3) 'fixedTPM' is SET if Key Usage is non-repudiation
139          badFixedTPM = IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE,
140                                  nonrepudiation)

```

```

141         && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM);
142     // 4)'restricted' is SET if Key Usage is for key agreement.
143     badRestricted = IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE, keyAgreement)
144         && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted);
145     if(badSign || badDecrypt || badFixedTPM || badRestricted)
146         return TPM_RCS_VALUE;
147     }
148     else
149         // The KeyUsage extension is required
150         return TPM_RCS_VALUE;
151
152     return TPM_RC_SUCCESS;
153 }

```

10.2.26.3 Marshaling Functions

10.2.26.3.1 X509AddSigningAlgorithm()

This creates the signing algorithm data.

Return Value	Meaning
> 0	number of octets added
<= 0	failure

```

154 INT16
155 X509AddSigningAlgorithm(
156     ASN1MarshalContext *ctx,
157     OBJECT *signKey,
158     TPMT_SIG_SCHEME *scheme
159 )
160 {
161     switch(signKey->publicArea.type)
162     {
163     #if ALG_RSA
164         case TPM_ALG_RSA:
165             return X509AddSigningAlgorithmRSA(signKey, scheme, ctx);
166     #endif // ALG_RSA
167     #if ALG_ECC
168         case TPM_ALG_ECC:
169             return X509AddSigningAlgorithmECC(signKey, scheme, ctx);
170     #endif // ALG_ECC
171     #if ALG_SM2
172         case TPM_ALG_SM2:
173             break; // no signing algorithm for SM2 yet
174             // return X509AddSigningAlgorithmSM2(signKey, scheme, ctx);
175     #endif // ALG_SM2
176         default:
177             break;
178     }
179     return 0;
180 }

```

10.2.26.3.2 X509AddPublicKey()

This function will add the *publicKey* description to the DER data. If *fillPtr* is NULL, then no data is transferred and this function will indicate if the TPM has the values for DER-encoding of the public key.

Return Value	Meaning
> 0	number of octets added
== 0	failure

```

181 INT16
182 X509AddPublicKey(
183     ASN1MarshalContext *ctx,
184     OBJECT *object
185 )
186 {
187     switch(object->publicArea.type)
188     {
189     #if ALG_RSA
190         case TPM_ALG_RSA:
191             return X509AddPublicRSA(object, ctx);
192     #endif
193     #if ALG_ECC
194         case TPM_ALG_ECC:
195             return X509AddPublicECC(object, ctx);
196     #endif
197     #if ALG_SM2
198         case TPM_ALG_SM2:
199             break;
200     #endif
201     default:
202         break;
203     }
204     return FALSE;
205 }

```

10.2.26.3.3 X509PushAlgorithmIdentifierSequence()

The function adds the algorithm identifier sequence.

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

206 INT16
207 X509PushAlgorithmIdentifierSequence(
208     ASN1MarshalContext *ctx,
209     const BYTE *OID
210 )
211 {
212     // An algorithm ID sequence is:
213     // SEQUENCE
214     //     OID
215     //     NULL
216     ASN1StartMarshalContext(ctx); // hash algorithm
217     ASN1PushNull(ctx);
218     ASN1PushOID(ctx, OID);
219     return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
220 }

```

10.2.27 AC_spt.c

10.2.27.1 Includes and Structures

```

1  #include "Tpm.h"
2  #include "AC_spt_fp.h"
3
4  #if 1          // This is the simulated AC data. This should be present in an actual
5                // implementation.
6
7  typedef struct {
8      TPMI_RH_AC      ac;
9      TPML_AC_CAPABILITIES *acData;
10
11  } acCapabilities;
12
13  TPML_AC_CAPABILITIES acData0001 = {1,
14      {{TPM_AT_PV1, 0x01234567}}};
15
16  acCapabilities ac[1] = { {0x0001, &acData0001} };
17
18  #define NUM_AC  (sizeof(ac) / sizeof(acCapabilities))
19
20  #endif // 1 The simulated AC data

```

10.2.27.2 Functions

10.2.27.2.1 AcToCapabilities()

This function returns a pointer to a list of AC capabilities.

```

21  TPML_AC_CAPABILITIES *
22  AcToCapabilities(
23      TPMI_RH_AC      component      // IN: component
24  )
25  {
26      UINT32      index;
27      //
28      for(index = 0; index < NUM_AC; index++)
29      {
30          if(ac[index].ac == component)
31              return ac[index].acData;
32      }
33      return NULL;
34  }

```

10.2.27.2.2 AcIsAccessible()

Function to determine if an AC handle references an actual AC

```

35  BOOL
36  AcIsAccessible(
37      TPM_HANDLE      acHandle
38  )
39  {
40      // In this implementation, the AC exists if there are some capabilities to go
41      // with the handle
42      return AcToCapabilities(acHandle) != NULL;
43  }

```


10.2.27.2.3 AcCapabilitiesGet()

This function returns a list of capabilities associated with an AC

Return Value	Meaning
YES	if there are more handles available
NO	all the available handles has been returned

```

44  TPMI_YES_NO
45  AcCapabilitiesGet(
46      TPMI_RH_AC          component,      // IN: the component
47      TPM_AT              type,          // IN: start capability type
48      TPML_AC_CAPABILITIES *capabilityList // OUT: list of handle
49  )
50  {
51      TPMI_YES_NO          more = NO;
52      UINT32               i;
53      TPML_AC_CAPABILITIES *capabilities = AcToCapabilities(component);
54
55      pAssert(HandleGetType(component) == TPM_HT_AC);
56
57      // Initialize output handle list
58      capabilityList->count = 0;
59
60      if(capabilities != NULL)
61      {
62          // Find the first capability less than or equal to type
63          for(i = 0; i < capabilities->count; i++)
64          {
65              if(capabilities->acCapabilities[i].tag >= type)
66              {
67                  // copy the capabilities until we run out or fill the list
68                  for(; (capabilityList->count < MAX_AC_CAPABILITIES)
69                      && (i < capabilities->count); i++)
70                  {
71                      capabilityList->acCapabilities[capabilityList->count]
72                          = capabilities->acCapabilities[i];
73                      capabilityList->count++;
74                  }
75                  more = i < capabilities->count;
76              }
77          }
78      }
79      return more;
80  }

```

10.2.27.2.4 AcSendObject()

Stub to handle sending of an AC object

```

81  TPM_RC
82  AcSendObject(
83      TPM_HANDLE          acHandle,      // IN: Handle of AC receiving object
84      OBJECT              *object,      // IN: object structure to send
85      TPMS_AC_OUTPUT      *acDataOut    // OUT: results of operation
86  )
87  {
88      NOT_REFERENCED(object);
89      NOT_REFERENCED(acHandle);
90      acDataOut->tag = TPM_AT_ERROR; // indicate that the response contains an
91                                   // error code
92      acDataOut->data = TPM_AE_NONE; // but there is no error.

```

```
93  
94     return TPM_RC_SUCCESS;  
95 }
```

DRAFT

10.2.28 CryptEccCrypt.c

10.2.28.1 Includes and Defines

```

1  #include "Tpm.h"
2
3  #if CC_ECC_Encrypt || CC_ECC_Encrypt

```

10.2.28.2 Functions

10.2.28.2.1 CryptEccSelectScheme()

This function is used by TPM2_ECC_Decrypt and TPM2_ECC_Encrypt. It sets scheme either the input scheme or the key scheme. If they key scheme is not TPM_ALG_NULL then the input scheme must be TPM_ALG_NULL or the same as the key scheme. If not, then the function returns FALSE.

Return Value	Meaning
TRUE	<i>scheme</i> is set
FALSE	<i>scheme</i> is not valid (it may have been changed).

```

4  BOOL
5  CryptEccSelectScheme(
6      OBJECT          *key,           //IN: key containing default scheme
7      TPMT_KDF_SCHEME *scheme        // IN: a decrypt scheme
8  )
9  {
10     TPMT_KDF_SCHEME *keyScheme = &key->publicArea.parameters.eccDetail.kdf;
11
12     // Get sign object pointer
13     if(scheme->scheme == TPM_ALG_NULL)
14         *scheme = *keyScheme;
15     if(keyScheme->scheme == TPM_ALG_NULL)
16         keyScheme = scheme;
17     return (scheme->scheme != TPM_ALG_NULL &&
18             (keyScheme->scheme == scheme->scheme
19              && keyScheme->details.anyKdf.hashAlg == scheme->details.anyKdf.hashAlg));
20 }

```

10.2.28.2.2 CryptEccEncrypt()

This function performs ECC-based data obfuscation. The only scheme that is currently supported is MGF1 based. See Part 1, Annex D for details.

Error Return	Meaning
TPM_RC_CURVE	unsupported curve
TPM_RC_HASH	hash not allowed
TPM_RC_SCHEME	<i>scheme</i> is not supported
TPM_RC_NO_RESULT	internal error in big number processing

```

21  LIB_EXPORT TPM_RC
22  CryptEccEncrypt(
23      OBJECT          *key,           // IN: public key of recipient
24      TPMT_KDF_SCHEME *scheme,        // IN: scheme to use.
25      TPM2B_MAX_BUFFER *plainText,    // IN: the text to obfuscate

```

```

26     TPMS_ECC_POINT          *c1,           // OUT: public ephemeral key
27     TPM2B_MAX_BUFFER        *c2,           // OUT: obfuscated text
28     TPM2B_DIGEST            *c3           // OUT: digest of ephemeral key
29                                     // and plainText
30 )
31 {
32     CURVE_INITIALIZED(E, key->publicArea.parameters.eccDetail.curveID);
33     POINT_INITIALIZED(PB, &key->publicArea.unique.ecc);
34     POINT_VAR(Px, MAX_ECC_KEY_BITS);
35     TPMS_ECC_POINT          p2;
36     ECC_NUM(D);
37     TPM2B_TYPE(2ECC, MAX_ECC_KEY_BYTES * 2);
38     TPM2B_2ECC              z;
39     int                     i;
40     HASH_STATE              hashState;
41     TPM_RC                  retVal = TPM_RC_SUCCESS;
42     //
43     #if defined DEBUG_ECC_ENCRYPT && DEBUG_ECC_ENCRYPT == YES
44     RND_DEBUG                dbg;
45     // This value is one less than the value from the reference so that it
46     // will become the correct value after having one added
47     TPM2B_ECC_PARAMETER k = {24, {
48         0x38, 0x4F, 0x30, 0x35, 0x30, 0x73, 0xAE, 0xEC,
49         0xE7, 0xA1, 0x65, 0x43, 0x30, 0xA9, 0x62, 0x04,
50         0xD3, 0x79, 0x82, 0xA3, 0xE1, 0x5B, 0x2C, 0xB4}};
51     RND_DEBUG_Instantiate(&dbg, &k.b);
52     # define RANDOM          (RAND_STATE *) &dbg
53
54     #else
55     # define RANDOM          NULL
56     #endif
57     if (E == NULL)
58         ERROR_RETURN(TPM_RC_CURVE);
59     if (TPM_ALG_KDF2 != scheme->scheme)
60         ERROR_RETURN(TPM_RC_SCHEME);
61     // generate an ephemeral key from a random k
62     if (!BnEccGenerateKeyPair(D, Px, E, RANDOM)
63         // C1 is the public part of the ephemeral key
64         || !BnPointTo2B(c1, Px, E)
65         // Compute P2
66         || (BnPointMult(Px, PB, D, NULL, NULL, E) != TPM_RC_SUCCESS)
67         || !BnPointTo2B(&p2, Px, E))
68         ERROR_RETURN(TPM_RC_NO_RESULT);
69
70     //Compute the C3 value hash(x2 || M || y2)
71     if (0 == CryptHashStart(&hashState, scheme->details.mgf1.hashAlg))
72         ERROR_RETURN(TPM_RC_HASH);
73     CryptDigestUpdate2B(&hashState, &p2.x.b);
74     CryptDigestUpdate2B(&hashState, &plainText->b);
75     CryptDigestUpdate2B(&hashState, &p2.y.b);
76     c3->t.size = CryptHashEnd(&hashState, sizeof(c3->t.buffer), c3->t.buffer);
77
78     MemoryCopy2B(&z.b, &p2.x.b, sizeof(z.t.buffer));
79     MemoryConcat2B(&z.b, &p2.y.b, sizeof(z.t.buffer));
80     // Generate the mask value from MGF1 and put it in the return buffer
81     c2->t.size = CryptMGF_KDF(plainText->t.size, c2->t.buffer,
82         scheme->details.mgf1.hashAlg, z.t.size, z.t.buffer, 1);
83     // XOR the plainText into the generated mask to create the obfuscated data
84     for (i = 0; i < plainText->t.size; i++)
85         c2->t.buffer[i] ^= plainText->t.buffer[i];
86 Exit:
87     CURVE_FREE(E);
88     return retVal;
89 }

```

10.2.28.2.3 CryptEccDecrypt()

This function performs ECC decryption and integrity check of the input data.

Error Return	Meaning
TPM_RC_CURVE	unsupported curve
TPM_RC_HASH	hash not allowed
TPM_RC_SCHEME	<i>scheme</i> is not supported
TPM_RC_NO_RESULT	internal error in big number processing
TPM_RC_VALUE	C3 did not match hash of recovered data

```

90  LIB_EXPORT TPM_RC
91  CryptEccDecrypt(
92      OBJECT                *key,                // IN: key used for data recovery
93      TPMT_KDF_SCHEME        *scheme,            // IN: scheme to use.
94      TPM2B_MAX_BUFFER        *plainText,        // OUT: the recovered text
95      TPMS_ECC_POINT          *c1,                // IN: public ephemeral key
96      TPM2B_MAX_BUFFER        *c2,                // IN: obfuscated text
97      TPM2B_DIGEST            *c3                // IN: digest of ephemeral key
98                                     // and plainText
99  )
100 {
101     CURVE_INITIALIZED(E, key->publicArea.parameters.eccDetail.curveID);
102     ECC_INITIALIZED(D, &key->sensitive.sensitive.ecc.b);
103     POINT_INITIALIZED(C1, c1);
104     TPMS_ECC_POINT          p2;
105     TPM2B_TYPE(2ECC, MAX_ECC_KEY_BYTES * 2);
106     TPM2B_DIGEST            check;
107     TPM2B_2ECC              z;
108     int                      i;
109     HASH_STATE               hashState;
110     TPM_RC                   retVal = TPM_RC_SUCCESS;
111     //
112     if (E == NULL)
113         ERROR_RETURN(TPM_RC_CURVE);
114     if (TPM_ALG_KDF2 != scheme->scheme)
115         ERROR_RETURN(TPM_RC_SCHEME);
116     // Generate the Z value
117     BnPointMult(C1, C1, D, NULL, NULL, E);
118     BnPointTo2B(&p2, C1, E);
119
120     // Start the hash to check the algorithm
121     if (0 == CryptHashStart(&hashState, scheme->details.mgf1.hashAlg))
122         ERROR_RETURN(TPM_RC_HASH);
123     CryptDigestUpdate2B(&hashState, &p2.x.b);
124
125     MemoryCopy2B(&z.b, &p2.x.b, sizeof(z.t.buffer));
126     MemoryConcat2B(&z.b, &p2.y.b, sizeof(z.t.buffer));
127
128     // Generate the mask
129     plainText->t.size = CryptMGF_KDF(c2->t.size, plainText->t.buffer,
130                                     scheme->details.mgf1.hashAlg, z.t.size,
131                                     z.t.buffer, 1);
132     // XOR the obfuscated data into the generated mask to create the plainText data
133     for (i = 0; i < plainText->t.size; i++)
134         plainText->t.buffer[i] ^= c2->t.buffer[i];
135
136     // Complete the hash and verify the data
137     CryptDigestUpdate2B(&hashState, &plainText->b);
138     CryptDigestUpdate2B(&hashState, &p2.y.b);
139     check.t.size = CryptHashEnd(&hashState, sizeof(check.t.buffer), check.t.buffer);

```

```
140     if (!MemoryEqual2B(&check.b, &c3->b))
141         ERROR_RETURN(TPM_RC_VALUE);
142 Exit:
143     CURVE_FREE(E);
144     return retVal;
145 }
146 #endif // CC_ECC_Encrypt || CC_ECC_Encrypt
```

DRAFT

Annex A (informative) Implementation Dependent

A.1 Introduction

This header file 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.

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.2 TpmProfile.h

```
1  #ifndef _TPM_PROFILE_H_
2  #define _TPM_PROFILE_H_
```

TPM 2.0 Part 2: Table 4 - Defines for Logic Values

```
3  #undef TRUE
4  #define TRUE 1
5  #undef FALSE
6  #define FALSE 0
7  #undef YES
8  #define YES 1
9  #undef NO
10 #define NO 0
11 #undef SET
12 #define SET 1
13 #undef CLEAR
14 #define CLEAR 0
```

Vendor -Specific: Table 1 - Defines for Processor Values

```
15 #ifndef BIG_ENDIAN_TPM
16 #define BIG_ENDIAN_TPM NO
17 #endif
18 #ifndef LITTLE_ENDIAN_TPM
19 #define LITTLE_ENDIAN_TPM !BIG_ENDIAN_TPM
20 #endif
21 #ifndef MOST_SIGNIFICANT_BIT_0
22 #define MOST_SIGNIFICANT_BIT_0 NO
23 #endif
24 #ifndef LEAST_SIGNIFICANT_BIT_0
25 #define LEAST_SIGNIFICANT_BIT_0 !MOST_SIGNIFICANT_BIT_0
26 #endif
27 #ifndef AUTO_ALIGN
28 #define AUTO_ALIGN NO
29 #endif
```

Vendor -Specific: Table 4 - Defines for Implemented Curves


```

30  #ifndef ECC_NIST_P192
31  #define ECC_NIST_P192          NO
32  #endif
33  #ifndef ECC_NIST_P224
34  #define ECC_NIST_P224          NO
35  #endif
36  #ifndef ECC_NIST_P256
37  #define ECC_NIST_P256          YES
38  #endif
39  #ifndef ECC_NIST_P384
40  #define ECC_NIST_P384          YES
41  #endif
42  #ifndef ECC_NIST_P521
43  #define ECC_NIST_P521          NO
44  #endif
45  #ifndef ECC_BN_P256
46  #define ECC_BN_P256           YES
47  #endif
48  #ifndef ECC_BN_P638
49  #define ECC_BN_P638           NO
50  #endif
51  #ifndef ECC_SM2_P256
52  #define ECC_SM2_P256           YES
53  #endif

```

Vendor -Specific: Table 6 - Defines for Implemented ACT

```

54  #ifndef RH_ACT_0
55  #define RH_ACT_0              YES
56  #endif
57  #ifndef RH_ACT_1
58  #define RH_ACT_1              NO
59  #endif
60  #ifndef RH_ACT_A
61  #define RH_ACT_A              YES
62  #endif

```

Vendor -Specific: Table 7 - Defines for Implementation Values

```

63  #ifndef FIELD_UPGRADE_IMPLEMENTED
64  #define FIELD_UPGRADE_IMPLEMENTED  NO
65  #endif
66  #ifndef HASH_LIB
67  #define HASH_LIB              Ossl
68  #endif
69  #ifndef SYM_LIB
70  #define SYM_LIB               Ossl
71  #endif
72  #ifndef MATH_LIB
73  #define MATH_LIB              Ossl
74  #endif
75  #ifndef IMPLEMENTATION_PCR
76  #define IMPLEMENTATION_PCR     24
77  #endif
78  #ifndef PLATFORM_PCR
79  #define PLATFORM_PCR          24
80  #endif
81  #ifndef DRTM_PCR
82  #define DRTM_PCR              17
83  #endif
84  #ifndef HCRTM_PCR
85  #define HCRTM_PCR             0
86  #endif
87  #ifndef NUM_LOCALITIES
88  #define NUM_LOCALITIES         5

```

```

89  #endif
90  #ifndef MAX_HANDLE_NUM
91  #define MAX_HANDLE_NUM          3
92  #endif
93  #ifndef MAX_ACTIVE_SESSIONS
94  #define MAX_ACTIVE_SESSIONS     64
95  #endif
96  #ifndef CONTEXT_SLOT
97  #define CONTEXT_SLOT            UINT16
98  #endif
99  #ifndef MAX_LOADED_SESSIONS
100 #define MAX_LOADED_SESSIONS     3
101 #endif
102 #ifndef MAX_SESSION_NUM
103 #define MAX_SESSION_NUM         3
104 #endif
105 #ifndef MAX_LOADED_OBJECTS
106 #define MAX_LOADED_OBJECTS      3
107 #endif
108 #ifndef MIN_EVICT_OBJECTS
109 #define MIN_EVICT_OBJECTS       2
110 #endif
111 #ifndef NUM_POLICY_PCR_GROUP
112 #define NUM_POLICY_PCR_GROUP    1
113 #endif
114 #ifndef NUM_AUTHVALUE_PCR_GROUP
115 #define NUM_AUTHVALUE_PCR_GROUP 1
116 #endif
117 #ifndef MAX_CONTEXT_SIZE
118 #define MAX_CONTEXT_SIZE        1344
119 #endif
120 #ifndef MAX_DIGEST_BUFFER
121 #define MAX_DIGEST_BUFFER       1024
122 #endif
123 #ifndef MAX_NV_INDEX_SIZE
124 #define MAX_NV_INDEX_SIZE       2048
125 #endif
126 #ifndef MAX_NV_BUFFER_SIZE
127 #define MAX_NV_BUFFER_SIZE      1024
128 #endif
129 #ifndef MAX_CAP_BUFFER
130 #define MAX_CAP_BUFFER          1024
131 #endif
132 #ifndef NV_MEMORY_SIZE
133 #define NV_MEMORY_SIZE          16384
134 #endif
135 #ifndef MIN_COUNTER_INDICES
136 #define MIN_COUNTER_INDICES     8
137 #endif
138 #ifndef NUM_STATIC_PCR
139 #define NUM_STATIC_PCR          16
140 #endif
141 #ifndef MAX_ALG_LIST_SIZE
142 #define MAX_ALG_LIST_SIZE       64
143 #endif
144 #ifndef PRIMARY_SEED_SIZE
145 #define PRIMARY_SEED_SIZE       32
146 #endif
147 #ifndef CONTEXT_ENCRYPT_ALGORITHM
148 #define CONTEXT_ENCRYPT_ALGORITHM AES
149 #endif
150 #ifndef NV_CLOCK_UPDATE_INTERVAL
151 #define NV_CLOCK_UPDATE_INTERVAL 12
152 #endif
153 #ifndef NUM_POLICY_PCR
154 #define NUM_POLICY_PCR          1

```

```

155 #endif
156 #ifndef MAX_COMMAND_SIZE
157 #define MAX_COMMAND_SIZE 4096
158 #endif
159 #ifndef MAX_RESPONSE_SIZE
160 #define MAX_RESPONSE_SIZE 4096
161 #endif
162 #ifndef ORDERLY_BITS
163 #define ORDERLY_BITS 8
164 #endif
165 #ifndef MAX_SYM_DATA
166 #define MAX_SYM_DATA 128
167 #endif
168 #ifndef MAX_RNG_ENTROPY_SIZE
169 #define MAX_RNG_ENTROPY_SIZE 64
170 #endif
171 #ifndef RAM_INDEX_SPACE
172 #define RAM_INDEX_SPACE 512
173 #endif
174 #ifndef RSA_DEFAULT_PUBLIC_EXPONENT
175 #define RSA_DEFAULT_PUBLIC_EXPONENT 0x00010001
176 #endif
177 #ifndef ENABLE_PCR_NO_INCREMENT
178 #define ENABLE_PCR_NO_INCREMENT YES
179 #endif
180 #ifndef CRT_FORMAT_RSA
181 #define CRT_FORMAT_RSA YES
182 #endif
183 #ifndef VENDOR_COMMAND_COUNT
184 #define VENDOR_COMMAND_COUNT 0
185 #endif
186 #ifndef MAX_VENDOR_BUFFER_SIZE
187 #define MAX_VENDOR_BUFFER_SIZE 1024
188 #endif
189 #ifndef SIZE_OF_X509_SERIAL_NUMBER
190 #define SIZE_OF_X509_SERIAL_NUMBER 20
191 #endif
192 #ifndef PRIVATE_VENDOR_SPECIFIC_BYTES
193 #define PRIVATE_VENDOR_SPECIFIC_BYTES RSA_PRIVATE_SIZE
194 #endif

```

Vendor -Specific: Table 2 - Defines for Implemented Algorithms

```

195 #ifndef ALG_AES
196 #define ALG_AES ALG_YES
197 #endif
198 #ifndef ALG_CAMELLIA
199 #define ALG_CAMELLIA ALG_YES
200 #endif
201 #ifndef ALG_CBC
202 #define ALG_CBC ALG_YES
203 #endif
204 #ifndef ALG_CFB
205 #define ALG_CFB ALG_YES
206 #endif
207 #ifndef ALG_CMAC
208 #define ALG_CMAC ALG_YES
209 #endif
210 #ifndef ALG_CTR
211 #define ALG_CTR ALG_YES
212 #endif
213 #ifndef ALG_ECB
214 #define ALG_ECB ALG_YES
215 #endif
216 #ifndef ALG_ECC

```

```

217 #define ALG_ECC                ALG_YES
218 #endif
219 #ifndef ALG_ECDSA
220 #define ALG_ECDSA                (ALG_YES && ALG_ECC)
221 #endif
222 #ifndef ALG_ECDH
223 #define ALG_ECDH                (ALG_YES && ALG_ECC)
224 #endif
225 #ifndef ALG_ECDSA
226 #define ALG_ECDSA                (ALG_YES && ALG_ECC)
227 #endif
228 #ifndef ALG_ECMQV
229 #define ALG_ECMQV                (ALG_NO && ALG_ECC)
230 #endif
231 #ifndef ALG_ECSCHNORR
232 #define ALG_ECSCHNORR            (ALG_YES && ALG_ECC)
233 #endif
234 #ifndef ALG_HMAC
235 #define ALG_HMAC                ALG_YES
236 #endif
237 #ifndef ALG_KDF1_SP800_108
238 #define ALG_KDF1_SP800_108        ALG_YES
239 #endif
240 #ifndef ALG_KDF1_SP800_56A
241 #define ALG_KDF1_SP800_56A        (ALG_YES && ALG_ECC)
242 #endif
243 #ifndef ALG_KDF2
244 #define ALG_KDF2                ALG_YES
245 #endif
246 #ifndef ALG_KEYEDHASH
247 #define ALG_KEYEDHASH            ALG_YES
248 #endif
249 #ifndef ALG_MGF1
250 #define ALG_MGF1                ALG_YES
251 #endif
252 #ifndef ALG_OAEP
253 #define ALG_OAEP                (ALG_YES && ALG_RSA)
254 #endif
255 #ifndef ALG_OFB
256 #define ALG_OFB                ALG_YES
257 #endif
258 #ifndef ALG_RSA
259 #define ALG_RSA                ALG_YES
260 #endif
261 #ifndef ALG_RSAES
262 #define ALG_RSAES                (ALG_YES && ALG_RSA)
263 #endif
264 #ifndef ALG_RSAPSS
265 #define ALG_RSAPSS                (ALG_YES && ALG_RSA)
266 #endif
267 #ifndef ALG_RSASSA
268 #define ALG_RSASSA                (ALG_YES && ALG_RSA)
269 #endif
270 #ifndef ALG_SHA
271 #define ALG_SHA                ALG_NO
272 #endif
273 #ifndef ALG_SHA1
274 #define ALG_SHA1                ALG_YES
275 #endif
276 #ifndef ALG_SHA256
277 #define ALG_SHA256                ALG_YES
278 #endif
279 #ifndef ALG_SHA384
280 #define ALG_SHA384                ALG_YES
281 #endif
282 #ifndef ALG_SHA3_256

```

```

283 #define ALG_SHA3_256          ALG_NO
284 #endif
285 #ifndef ALG_SHA3_384
286 #define ALG_SHA3_384          ALG_NO
287 #endif
288 #ifndef ALG_SHA3_512
289 #define ALG_SHA3_512          ALG_NO
290 #endif
291 #ifndef ALG_SHA512
292 #define ALG_SHA512            ALG_NO
293 #endif
294 #ifndef ALG_SM2
295 #define ALG_SM2                (ALG_YES && ALG_ECC)
296 #endif
297 #ifndef ALG_SM3_256
298 #define ALG_SM3_256            ALG_YES
299 #endif
300 #ifndef ALG_SM4
301 #define ALG_SM4                ALG_YES
302 #endif
303 #ifndef ALG_SYMCIPHER
304 #define ALG_SYMCIPHER          ALG_YES
305 #endif
306 #ifndef ALG_TDES
307 #define ALG_TDES                ALG_NO
308 #endif
309 #ifndef ALG_XOR
310 #define ALG_XOR                ALG_YES
311 #endif

```

TCG Algorithm Registry: Table 3 - Defines for RSA Asymmetric Cipher Algorithm Constants

```

312 #ifndef RSA_1024
313 #define RSA_1024                (ALG_RSA && YES)
314 #endif
315 #ifndef RSA_2048
316 #define RSA_2048                (ALG_RSA && YES)
317 #endif
318 #ifndef RSA_3072
319 #define RSA_3072                (ALG_RSA && NO)
320 #endif
321 #ifndef RSA_4096
322 #define RSA_4096                (ALG_RSA && NO)
323 #endif
324 #ifndef RSA_16384
325 #define RSA_16384              (ALG_RSA && NO)
326 #endif

```

TCG Algorithm Registry: Table 21 - Defines for AES Symmetric Cipher Algorithm Constants

```

327 #ifndef AES_128
328 #define AES_128                (ALG_AES && YES)
329 #endif
330 #ifndef AES_192
331 #define AES_192                (ALG_AES && NO)
332 #endif
333 #ifndef AES_256
334 #define AES_256                (ALG_AES && YES)
335 #endif

```

TCG Algorithm Registry: Table 22 - Defines for SM4 Symmetric Cipher Algorithm Constants

```

336 #ifndef SM4_128
337 #define SM4_128                (ALG_SM4 && YES)

```

338 **#endif**

TCG Algorithm Registry: Table 23 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```

339 #ifndef CAMELLIA_128
340 #define CAMELLIA_128 (ALG_CAMELLIA && YES)
341 #endif
342 #ifndef CAMELLIA_192
343 #define CAMELLIA_192 (ALG_CAMELLIA && NO)
344 #endif
345 #ifndef CAMELLIA_256
346 #define CAMELLIA_256 (ALG_CAMELLIA && YES)
347 #endif

```

TCG Algorithm Registry: Table 24 - Defines for TDES Symmetric Cipher Algorithm Constants

```

348 #ifndef TDES_128
349 #define TDES_128 (ALG_TDES && YES)
350 #endif
351 #ifndef TDES_192
352 #define TDES_192 (ALG_TDES && YES)
353 #endif

```

Vendor -Specific: Table 5 - Defines for Implemented Commands

```

354 #ifndef CC_ACT_SetTimeout
355 #define CC_ACT_SetTimeout CC_YES
356 #endif
357 #ifndef CC_AC_GetCapability
358 #define CC_AC_GetCapability CC_YES
359 #endif
360 #ifndef CC_AC_Send
361 #define CC_AC_Send CC_YES
362 #endif
363 #ifndef CC_ActivateCredential
364 #define CC_ActivateCredential CC_YES
365 #endif
366 #ifndef CC_Certify
367 #define CC_Certify CC_YES
368 #endif
369 #ifndef CC_CertifyCreation
370 #define CC_CertifyCreation CC_YES
371 #endif
372 #ifndef CC_CertifyX509
373 #define CC_CertifyX509 CC_YES
374 #endif
375 #ifndef CC_ChangeEPS
376 #define CC_ChangeEPS CC_YES
377 #endif
378 #ifndef CC_ChangePPS
379 #define CC_ChangePPS CC_YES
380 #endif
381 #ifndef CC_Clear
382 #define CC_Clear CC_YES
383 #endif
384 #ifndef CC_ClearControl
385 #define CC_ClearControl CC_YES
386 #endif
387 #ifndef CC_ClockRateAdjust
388 #define CC_ClockRateAdjust CC_YES
389 #endif
390 #ifndef CC_ClockSet
391 #define CC_ClockSet CC_YES
392 #endif

```

```
393 #ifndef CC_Commit
394 #define CC_Commit (CC_YES && ALG_ECC)
395 #endif
396 #ifndef CC_ContextLoad
397 #define CC_ContextLoad CC_YES
398 #endif
399 #ifndef CC_ContextSave
400 #define CC_ContextSave CC_YES
401 #endif
402 #ifndef CC_Create
403 #define CC_Create CC_YES
404 #endif
405 #ifndef CC_CreateLoaded
406 #define CC_CreateLoaded CC_YES
407 #endif
408 #ifndef CC_CreatePrimary
409 #define CC_CreatePrimary CC_YES
410 #endif
411 #ifndef CC_DictionaryAttackLockReset
412 #define CC_DictionaryAttackLockReset CC_YES
413 #endif
414 #ifndef CC_DictionaryAttackParameters
415 #define CC_DictionaryAttackParameters CC_YES
416 #endif
417 #ifndef CC_Duplicate
418 #define CC_Duplicate CC_YES
419 #endif
420 #ifndef CC_ECC_Decrypt
421 #define CC_ECC_Decrypt (CC_YES && ALG_ECC)
422 #endif
423 #ifndef CC_ECC_Encrypt
424 #define CC_ECC_Encrypt (CC_YES && ALG_ECC)
425 #endif
426 #ifndef CC_ECC_Parameters
427 #define CC_ECC_Parameters (CC_YES && ALG_ECC)
428 #endif
429 #ifndef CC_ECDH_KeyGen
430 #define CC_ECDH_KeyGen (CC_YES && ALG_ECC)
431 #endif
432 #ifndef CC_ECDH_ZGen
433 #define CC_ECDH_ZGen (CC_YES && ALG_ECC)
434 #endif
435 #ifndef CC_EC_Ephemeral
436 #define CC_EC_Ephemeral (CC_YES && ALG_ECC)
437 #endif
438 #ifndef CC_EncryptDecrypt
439 #define CC_EncryptDecrypt CC_YES
440 #endif
441 #ifndef CC_EncryptDecrypt2
442 #define CC_EncryptDecrypt2 CC_YES
443 #endif
444 #ifndef CC_EventSequenceComplete
445 #define CC_EventSequenceComplete CC_YES
446 #endif
447 #ifndef CC_EvictControl
448 #define CC_EvictControl CC_YES
449 #endif
450 #ifndef CC_FieldUpgradeData
451 #define CC_FieldUpgradeData CC_NO
452 #endif
453 #ifndef CC_FieldUpgradeStart
454 #define CC_FieldUpgradeStart CC_NO
455 #endif
456 #ifndef CC_FirmwareRead
457 #define CC_FirmwareRead CC_NO
458 #endif
```



```

459 #ifndef CC_FlushContext
460 #define CC_FlushContext CC_YES
461 #endif
462 #ifndef CC_GetCapability
463 #define CC_GetCapability CC_YES
464 #endif
465 #ifndef CC_GetCommandAuditDigest
466 #define CC_GetCommandAuditDigest CC_YES
467 #endif
468 #ifndef CC_GetRandom
469 #define CC_GetRandom CC_YES
470 #endif
471 #ifndef CC_GetSessionAuditDigest
472 #define CC_GetSessionAuditDigest CC_YES
473 #endif
474 #ifndef CC_GetTestResult
475 #define CC_GetTestResult CC_YES
476 #endif
477 #ifndef CC_GetTime
478 #define CC_GetTime CC_YES
479 #endif
480 #ifndef CC_HMAC
481 #define CC_HMAC (CC_YES && !ALG_CMAC)
482 #endif
483 #ifndef CC_HMAC_Start
484 #define CC_HMAC_Start (CC_YES && !ALG_CMAC)
485 #endif
486 #ifndef CC_Hash
487 #define CC_Hash CC_YES
488 #endif
489 #ifndef CC_HashSequenceStart
490 #define CC_HashSequenceStart CC_YES
491 #endif
492 #ifndef CC_HierarchyChangeAuth
493 #define CC_HierarchyChangeAuth CC_YES
494 #endif
495 #ifndef CC_HierarchyControl
496 #define CC_HierarchyControl CC_YES
497 #endif
498 #ifndef CC_Import
499 #define CC_Import CC_YES
500 #endif
501 #ifndef CC_IncrementalSelfTest
502 #define CC_IncrementalSelfTest CC_YES
503 #endif
504 #ifndef CC_Load
505 #define CC_Load CC_YES
506 #endif
507 #ifndef CC_LoadExternal
508 #define CC_LoadExternal CC_YES
509 #endif
510 #ifndef CC_MAC
511 #define CC_MAC (CC_YES && ALG_CMAC)
512 #endif
513 #ifndef CC_MAC_Start
514 #define CC_MAC_Start (CC_YES && ALG_CMAC)
515 #endif
516 #ifndef CC_MakeCredential
517 #define CC_MakeCredential CC_YES
518 #endif
519 #ifndef CC_NV_Certify
520 #define CC_NV_Certify CC_YES
521 #endif
522 #ifndef CC_NV_ChangeAuth
523 #define CC_NV_ChangeAuth CC_YES
524 #endif

```

```
525 #ifndef CC_NV_DefineSpace
526 #define CC_NV_DefineSpace CC_YES
527 #endif
528 #ifndef CC_NV_Extend
529 #define CC_NV_Extend CC_YES
530 #endif
531 #ifndef CC_NV_GlobalWriteLock
532 #define CC_NV_GlobalWriteLock CC_YES
533 #endif
534 #ifndef CC_NV_Increment
535 #define CC_NV_Increment CC_YES
536 #endif
537 #ifndef CC_NV_Read
538 #define CC_NV_Read CC_YES
539 #endif
540 #ifndef CC_NV_ReadLock
541 #define CC_NV_ReadLock CC_YES
542 #endif
543 #ifndef CC_NV_ReadPublic
544 #define CC_NV_ReadPublic CC_YES
545 #endif
546 #ifndef CC_NV_SetBits
547 #define CC_NV_SetBits CC_YES
548 #endif
549 #ifndef CC_NV_UndefineSpace
550 #define CC_NV_UndefineSpace CC_YES
551 #endif
552 #ifndef CC_NV_UndefineSpaceSpecial
553 #define CC_NV_UndefineSpaceSpecial CC_YES
554 #endif
555 #ifndef CC_NV_Write
556 #define CC_NV_Write CC_YES
557 #endif
558 #ifndef CC_NV_WriteLock
559 #define CC_NV_WriteLock CC_YES
560 #endif
561 #ifndef CC_ObjectChangeAuth
562 #define CC_ObjectChangeAuth CC_YES
563 #endif
564 #ifndef CC_PCR_Allocate
565 #define CC_PCR_Allocate CC_YES
566 #endif
567 #ifndef CC_PCR_Event
568 #define CC_PCR_Event CC_YES
569 #endif
570 #ifndef CC_PCR_Extend
571 #define CC_PCR_Extend CC_YES
572 #endif
573 #ifndef CC_PCR_Read
574 #define CC_PCR_Read CC_YES
575 #endif
576 #ifndef CC_PCR_Reset
577 #define CC_PCR_Reset CC_YES
578 #endif
579 #ifndef CC_PCR_SetAuthPolicy
580 #define CC_PCR_SetAuthPolicy CC_YES
581 #endif
582 #ifndef CC_PCR_SetAuthValue
583 #define CC_PCR_SetAuthValue CC_YES
584 #endif
585 #ifndef CC_PP_Commands
586 #define CC_PP_Commands CC_YES
587 #endif
588 #ifndef CC_PolicyAuthValue
589 #define CC_PolicyAuthValue CC_YES
590 #endif
```

```
591 #ifndef CC_PolicyAuthorize
592 #define CC_PolicyAuthorize CC_YES
593 #endif
594 #ifndef CC_PolicyAuthorizeNV
595 #define CC_PolicyAuthorizeNV CC_YES
596 #endif
597 #ifndef CC_PolicyCommandCode
598 #define CC_PolicyCommandCode CC_YES
599 #endif
600 #ifndef CC_PolicyCounterTimer
601 #define CC_PolicyCounterTimer CC_YES
602 #endif
603 #ifndef CC_PolicyCpHash
604 #define CC_PolicyCpHash CC_YES
605 #endif
606 #ifndef CC_PolicyDuplicationSelect
607 #define CC_PolicyDuplicationSelect CC_YES
608 #endif
609 #ifndef CC_PolicyGetDigest
610 #define CC_PolicyGetDigest CC_YES
611 #endif
612 #ifndef CC_PolicyLocality
613 #define CC_PolicyLocality CC_YES
614 #endif
615 #ifndef CC_PolicyNV
616 #define CC_PolicyNV CC_YES
617 #endif
618 #ifndef CC_PolicyNameHash
619 #define CC_PolicyNameHash CC_YES
620 #endif
621 #ifndef CC_PolicyNvWritten
622 #define CC_PolicyNvWritten CC_YES
623 #endif
624 #ifndef CC_PolicyOR
625 #define CC_PolicyOR CC_YES
626 #endif
627 #ifndef CC_PolicyPCR
628 #define CC_PolicyPCR CC_YES
629 #endif
630 #ifndef CC_PolicyPassword
631 #define CC_PolicyPassword CC_YES
632 #endif
633 #ifndef CC_PolicyPhysicalPresence
634 #define CC_PolicyPhysicalPresence CC_YES
635 #endif
636 #ifndef CC_PolicyRestart
637 #define CC_PolicyRestart CC_YES
638 #endif
639 #ifndef CC_PolicySecret
640 #define CC_PolicySecret CC_YES
641 #endif
642 #ifndef CC_PolicySigned
643 #define CC_PolicySigned CC_YES
644 #endif
645 #ifndef CC_PolicyTemplate
646 #define CC_PolicyTemplate CC_YES
647 #endif
648 #ifndef CC_PolicyTicket
649 #define CC_PolicyTicket CC_YES
650 #endif
651 #ifndef CC_Policy_AC_SendSelect
652 #define CC_Policy_AC_SendSelect CC_YES
653 #endif
654 #ifndef CC_Quote
655 #define CC_Quote CC_YES
656 #endif
```

```

657 #ifndef CC_RSA_Decrypt
658 #define CC_RSA_Decrypt (CC_YES && ALG_RSA)
659 #endif
660 #ifndef CC_RSA_Encrypt
661 #define CC_RSA_Encrypt (CC_YES && ALG_RSA)
662 #endif
663 #ifndef CC_ReadClock
664 #define CC_ReadClock CC_YES
665 #endif
666 #ifndef CC_ReadPublic
667 #define CC_ReadPublic CC_YES
668 #endif
669 #ifndef CC_Rewrap
670 #define CC_Rewrap CC_YES
671 #endif
672 #ifndef CC_SelfTest
673 #define CC_SelfTest CC_YES
674 #endif
675 #ifndef CC_SequenceComplete
676 #define CC_SequenceComplete CC_YES
677 #endif
678 #ifndef CC_SequenceUpdate
679 #define CC_SequenceUpdate CC_YES
680 #endif
681 #ifndef CC_SetAlgorithmSet
682 #define CC_SetAlgorithmSet CC_YES
683 #endif
684 #ifndef CC_SetCommandCodeAuditStatus
685 #define CC_SetCommandCodeAuditStatus CC_YES
686 #endif
687 #ifndef CC_SetPrimaryPolicy
688 #define CC_SetPrimaryPolicy CC_YES
689 #endif
690 #ifndef CC_Shutdown
691 #define CC_Shutdown CC_YES
692 #endif
693 #ifndef CC_Sign
694 #define CC_Sign CC_YES
695 #endif
696 #ifndef CC_StartAuthSession
697 #define CC_StartAuthSession CC_YES
698 #endif
699 #ifndef CC_Startup
700 #define CC_Startup CC_YES
701 #endif
702 #ifndef CC_StirRandom
703 #define CC_StirRandom CC_YES
704 #endif
705 #ifndef CC_TestParms
706 #define CC_TestParms CC_YES
707 #endif
708 #ifndef CC_Unseal
709 #define CC_Unseal CC_YES
710 #endif
711 #ifndef CC_Vendor_TCG_Test
712 #define CC_Vendor_TCG_Test CC_YES
713 #endif
714 #ifndef CC_VerifySignature
715 #define CC_VerifySignature CC_YES
716 #endif
717 #ifndef CC_ZGen_2Phase
718 #define CC_ZGen_2Phase (CC_YES && ALG_ECC)
719 #endif
720
721 #endif // _TPM_PROFILE_H_

```

A.3 TpmSizeChecks.c

A.3.1. Includes, Defines, and Types

```

1  #include    "Tpm.h"
2  #include    <stdio.h>
3  #include    <assert.h>
4
5  #if RUNTIME_SIZE_CHECKS
6
7  #if TABLE_DRIVEN_MARSHAL
8  extern uint32_t    MarshalDataSize;
9  #endif
10
11  static      int    once = 0;

```

A.3.2. TpmSizeChecks()

This function is used during the development process to make sure that the vendor-specific values result in a consistent implementation. When possible, the code contains `#if` to do compile-time checks. However, in some cases, the values require the use of `sizeof()` and that can't be used in an `#if`.

```

12  BOOL
13  TpmSizeChecks(
14      void
15  )
16  {
17      BOOL        PASS = TRUE;
18  #if DEBUG
19  //
20  if(once++ != 0)
21      return 1;
22  {
23      UINT32      maxAsymSecurityStrength = MAX_ASYM_SECURITY_STRENGTH;
24      UINT32      maxHashSecurityStrength = MAX_HASH_SECURITY_STRENGTH;
25      UINT32      maxSymSecurityStrength = MAX_SYM_SECURITY_STRENGTH;
26      UINT32      maxSecurityStrengthBits = MAX_SECURITY_STRENGTH_BITS;
27      UINT32      proofSize = PROOF_SIZE;
28      UINT32      compliantProofSize = COMPLIANT_PROOF_SIZE;
29      UINT32      compliantPrimarySeedSize = COMPLIANT_PRIMARY_SEED_SIZE;
30      UINT32      primarySeedSize = PRIMARY_SEED_SIZE;
31
32      UINT32      cmacState = sizeof(tpmCmacState_t);
33      UINT32      hashState = sizeof(HASH_STATE);
34      UINT32      keyScheduleSize = sizeof(tpmCryptKeySchedule_t);
35  //
36      NOT_REFERENCED(cmacState);
37      NOT_REFERENCED(hashState);
38      NOT_REFERENCED(keyScheduleSize);
39      NOT_REFERENCED(maxAsymSecurityStrength);
40      NOT_REFERENCED(maxHashSecurityStrength);
41      NOT_REFERENCED(maxSymSecurityStrength);
42      NOT_REFERENCED(maxSecurityStrengthBits);
43      NOT_REFERENCED(proofSize);
44      NOT_REFERENCED(compliantProofSize);
45      NOT_REFERENCED(compliantPrimarySeedSize);
46      NOT_REFERENCED(primarySeedSize);
47
48      {
49          TPMT_SENSITIVE    *p;
50          // This assignment keeps compiler from complaining about a conditional
51          // comparison being between two constants
52          UINT16            max_rsa_key_bytes = MAX_RSA_KEY_BYTES;

```

```

53         if((max_rsa_key_bytes / 2) != (sizeof(p->sensitive.rsa.t.buffer) / 5))
54         {
55             printf("Sensitive part of TPMT_SENSITIVE is undersized. May be caused"
56                 " by use of wrong version of Part 2.\n");
57             PASS = FALSE;
58         }
59     }
60 #if TABLE_DRIVEN_MARSHAL
61     printf("sizeof(MarshalData) = %zu\n", sizeof(MarshalData_st));
62 #endif
63
64     printf("Size of OBJECT = %zu\n", sizeof(OBJECT));
65     printf("Size of components in TPMT_SENSITIVE = %zu\n",
66         sizeof(TPMT_SENSITIVE));
67     printf("    TPMI_ALG_PUBLIC                %zu\n", sizeof(TPMI_ALG_PUBLIC));
68     printf("    TPM2B_AUTH                        %zu\n", sizeof(TPM2B_AUTH));
69     printf("    TPM2B_DIGEST                        %zu\n", sizeof(TPM2B_DIGEST));
70     printf("    TPMU_SENSITIVE_COMPOSITE          %zu\n",
71         sizeof(TPMU_SENSITIVE_COMPOSITE));
72
73     // Make sure that the size of the context blob is large enough for the largest
74     // context
75     // TPMS_CONTEXT_DATA contains two TPM2B values. That is not how this is
76     // implemented. Rather, the size field of the TPM2B_CONTEXT_DATA is used to
77     // determine the amount of data in the encrypted data. That part is not
78     // independently sized. This makes the actual size 2 bytes smaller than
79     // calculated using Part 2. Since this is opaque to the caller, it is not
80     // necessary to fix. The actual size is returned by TPM2_GetCapabilities().
81
82     // Initialize output handle. At the end of command action, the output
83     // handle of an object will be replaced, while the output handle
84     // for a session will be the same as input
85
86     // Get the size of fingerprint in context blob. The sequence value in
87     // TPMS_CONTEXT structure is used as the fingerprint
88     {
89         UINT32 fingerprintSize = sizeof(UINT64);
90         UINT32 integritySize = sizeof(UINT16)
91             + CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
92         UINT32 biggestObject = MAX(MAX(sizeof(HASH_OBJECT), sizeof(OBJECT)),
93             sizeof(SESSION));
94         UINT32 biggestContext = fingerprintSize + integritySize + biggestObject;
95
96         // round required size up to nearest 8 byte boundary.
97         biggestContext = 8 * ((biggestContext + 7) / 8);
98
99         if(MAX_CONTEXT_SIZE < biggestContext)
100         {
101             printf("MAX_CONTEXT_SIZE needs to be increased to at least %d (%d)\n",
102                 biggestContext, MAX_CONTEXT_SIZE);
103             PASS = FALSE;
104         }
105         else if (MAX_CONTEXT_SIZE > biggestContext)
106         {
107             printf("MAX_CONTEXT_SIZE can be reduced to %d (%d)\n",
108                 biggestContext, MAX_CONTEXT_SIZE);
109         }
110     }
111     {
112         union u
113         {
114             TPMA_OBJECT          attributes;
115             UINT32                uint32Value;
116         } u;
117         // these are defined so that compiler doesn't complain about conditional
118         // expressions comparing two constants.

```

```

118         int                aSize = sizeof(u.attributes);
119         int                uSize = sizeof(u.uint32Value);
120         u.uint32Value = 0;
121         SET_ATTRIBUTE(u.attributes, TPMA_OBJECT, fixedTPM);
122         if(u.uint32Value != 2)
123         {
124             printf("The bit allocation in a TPMA_OBJECT is not as expected");
125             PASS = FALSE;
126         }
127         if(aSize != uSize) // comparison of two sizeof() values annoys compiler
128         {
129             printf("A TPMA_OBJECT is not the expected size.");
130             PASS = FALSE;
131         }
132     }
133     // Check that the platform implements each of the ACT that the TPM thinks are
134     // present
135     {
136         uint32_t            act;
137         for(act = 0; act < 16; act++)
138         {
139             switch(act)
140             {
141                 FOR_EACH_ACT(CASE_ACT_NUMBER)
142                     if(!_plat__ACT_GetImplemented(act))
143                     {
144                         printf("TPM_RH_ACT_%1X is not implemented by platform\n",
145                             act);
146                         PASS = FALSE;
147                     }
148                 default:
149                     break;
150             }
151         }
152     }
153     #endif // DEBUG
154     return (PASS);
155 }
156 #endif // RUNTIME_SIZE_CHECKS

```


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 OpenSSL-Specific Files

B.2.1. Introduction

The following files are specific to a port that uses the OpenSSL library for cryptographic functions.

B.2.2. Header Files

B.2.2.1. TpmToOsslHash.h

B.2.2.1.1. Introduction

This header file is used to *splice* the OpenSSL hash code into the TPM code.

```

1  #ifndef HASH_LIB_DEFINED
2  #define HASH_LIB_DEFINED
3
4  #define HASH_LIB_OSSL
5
6  #include <openssl/evp.h>
7  #include <openssl/sha.h>
8
9  #if ALG_SM3_256
10 #   if defined(OPENSSL_NO_SM3) || OPENSSL_VERSION_NUMBER < 0x10101010L
11 #       undef ALG_SM3_256
12 #       define ALG_SM3_256 ALG_NO
13 #   elif OPENSSL_VERSION_NUMBER >= 0x10200000L
14 #       include <openssl/sm3.h>
15 #   else
16       // OpenSSL 1.1.1 keeps smX.h headers in the include/crypto directory,
17       // and they do not get installed as part of the libssl package
18 #       define SM3_LBLOCK      (64/4)
19
20       typedef struct SM3state_st {
21           unsigned int A, B, C, D, E, F, G, H;
22           unsigned int Nl, Nh;
23           unsigned int data[SM3_LBLOCK];
24           unsigned int num;
25       } SM3_CTX;
26
27       int sm3_init(SM3_CTX *c);
28       int sm3_update(SM3_CTX *c, const void *data, size_t len);
29       int sm3_final(unsigned char *md, SM3_CTX *c);
30 #   endif // OpenSSL < 1.2
31 #endif // ALG_SM3_256
32
33 #include <openssl/oss1_typ.h>
34
35 #define HASH_ALIGNMENT  RADIX_BYTES

```

B.2.2.1.2. Links to the OpenSSL HASH code

Redefine the internal name used for each of the hash state structures to the name used by the library. These defines need to be known in all parts of the TPM so that the structure sizes can be properly computed when needed.

```

36 #define tpmHashStateSHA1_t      SHA_CTX
37 #define tpmHashStateSHA256_t   SHA256_CTX

```

```

38 #define tpmHashStateSHA384_t    SHA512_CTX
39 #define tpmHashStateSHA512_t    SHA512_CTX
40 #define tpmHashStateSM3_256_t    SM3_CTX

```

The defines below are only needed when compiling CryptHash.c or CryptSmac.c. This isolation is primarily to avoid name space collision. However, if there is a real collision, it will likely show up when the linker tries to put things together.

```

41 #ifndef _CRYPT_HASH_C_
42
43 typedef BYTE          *PBYTE;
44 typedef const BYTE    *PCBYTE;

```

Define the interface between CryptHash.c to the functions provided by the library. For each method, define the calling parameters of the method and then define how the method is invoked in CryptHash.c.

All hashes are required to have the same calling sequence. If they don't, create a simple adaptation function that converts from the **standard** form of the call to the form used by the specific hash (and then send a nasty letter to the person who wrote the hash function for the library).

The macro that calls the method also defines how the parameters get swizzled between the default form (in CryptHash.c) and the library form.

Initialize the hash context

```

45 #define HASH_START_METHOD_DEF    void (HASH_START_METHOD) (PANY_HASH_STATE state)
46 #define HASH_START(hashState)
47     ((hashState)->def->method.start) (&(hashState)->state);

```

Add data to the hash

```

48 #define HASH_DATA_METHOD_DEF
49     void (HASH_DATA_METHOD) (PANY_HASH_STATE state,
50                             PCBYTE buffer,
51                             size_t size)
52 #define HASH_DATA(hashState, dInSize, dIn)
53     ((hashState)->def->method.data) (&(hashState)->state, dIn, dInSize)

```

Finalize the hash and get the digest

```

54 #define HASH_END_METHOD_DEF
55     void (HASH_END_METHOD) (BYTE *buffer, PANY_HASH_STATE state)
56 #define HASH_END(hashState, buffer)
57     ((hashState)->def->method.end) (buffer, &(hashState)->state)

```

Copy the hash context

NOTE For import, export, and copy, memcpy() is used since there is no reformatting necessary between the internal and external forms.

```

58 #define HASH_STATE_COPY_METHOD_DEF
59     void (HASH_STATE_COPY_METHOD) (PANY_HASH_STATE to,
60                                     PCANY_HASH_STATE from,
61                                     size_t size)
62 #define HASH_STATE_COPY(hashStateOut, hashStateIn)
63     ((hashStateIn)->def->method.copy) (&(hashStateOut)->state,
64                                         &(hashStateIn)->state,
65                                         (hashStateIn)->def->contextSize)

```

Copy (with reformatting when necessary) an internal hash structure to an external blob

```

66 #define HASH_STATE_EXPORT_METHOD_DEF

```

```

67         void (HASH_STATE_EXPORT_METHOD) (BYTE *to,           \
68                                           PCANY_HASH_STATE from, \
69                                           size_t size)         \
70 #define HASH_STATE_EXPORT(to, hashStateFrom)                 \
71     ((hashStateFrom)->def->method.copyOut)                   \
72     (&((BYTE *) (to))[offsetof(HASH_STATE, state)]),         \
73     &(hashStateFrom)->state,                                  \
74     (hashStateFrom)->def->contextSize)

```

Copy from an external blob to an internal formate (with reformatting when necessary

```

75 #define HASH_STATE_IMPORT_METHOD_DEF                          \
76     void (HASH_STATE_IMPORT_METHOD) (PANY_HASH_STATE to,     \
77                                       const BYTE *from,        \
78                                       size_t size)             \
79 #define HASH_STATE_IMPORT(hashStateTo, from)                  \
80     ((hashStateTo)->def->method.copyIn)                        \
81     (&(hashStateTo)->state,                                    \
82     &((const BYTE *) (from))[offsetof(HASH_STATE, state)]), \
83     (hashStateTo)->def->contextSize)

```

Function aliases. The code in CryptHash.c uses the internal designation for the functions. These need to be translated to the function names of the library.

```

84 #define tpmHashStart_SHA1          SHA1_Init    // external name of the
85                                     // initialization method
86 #define tpmHashData_SHA1           SHA1_Update
87 #define tpmHashEnd_SHA1            SHA1_Final
88 #define tpmHashStateCopy_SHA1      memcpy
89 #define tpmHashStateExport_SHA1    memcpy
90 #define tpmHashStateImport_SHA1    memcpy
91 #define tpmHashStart_SHA256        SHA256_Init
92 #define tpmHashData_SHA256         SHA256_Update
93 #define tpmHashEnd_SHA256          SHA256_Final
94 #define tpmHashStateCopy_SHA256    memcpy
95 #define tpmHashStateExport_SHA256  memcpy
96 #define tpmHashStateImport_SHA256  memcpy
97 #define tpmHashStart_SHA384        SHA384_Init
98 #define tpmHashData_SHA384         SHA384_Update
99 #define tpmHashEnd_SHA384          SHA384_Final
100 #define tpmHashStateCopy_SHA384    memcpy
101 #define tpmHashStateExport_SHA384  memcpy
102 #define tpmHashStateImport_SHA384  memcpy
103 #define tpmHashStart_SHA512        SHA512_Init
104 #define tpmHashData_SHA512         SHA512_Update
105 #define tpmHashEnd_SHA512          SHA512_Final
106 #define tpmHashStateCopy_SHA512    memcpy
107 #define tpmHashStateExport_SHA512  memcpy
108 #define tpmHashStateImport_SHA_512 memcpy
109 #define tpmHashStart_SM3_256        sm3_init
110 #define tpmHashData_SM3_256         sm3_update
111 #define tpmHashEnd_SM3_256          sm3_final
112 #define tpmHashStateCopy_SM3_256    memcpy
113 #define tpmHashStateExport_SM3_256  memcpy
114 #define tpmHashStateImport_SM3_256  memcpy
115
116 #endif // _CRYPT_HASH_C_
117
118 #define LibHashInit()

```

This definition would change if there were something to report

```

119 #define HashLibSimulationEnd()
120

```

121 `#endif // HASH_LIB_DEFINED`

DRAFT

B.2.2.2. TpmToOsslMath.h

B.2.2.2.1. Introduction

This file contains the structure definitions used for ECC in the OpenSSL version of the code. These definitions would change, based on the library. The ECC-related structures that cross the TPM interface are defined in TpmTypes.h

```

1  #ifndef MATH_LIB_DEFINED
2  #define MATH_LIB_DEFINED
3
4  #define MATH_LIB_OSSL
5
6  #include <openssl/evp.h>
7  #include <openssl/ec.h>
8
9  #define SYMMETRIC_ALIGNMENT RADIX_BYTES
10
11 #if OPENSSL_VERSION_NUMBER >= 0x10200000L
12     // Check the bignum_st definition in crypto/bn/bn_lcl.h and either update the
13     // version check or provide the new definition for this version.
14 #error Untested OpenSSL version
15 #elif OPENSSL_VERSION_NUMBER >= 0x10100000L
16     // from crypto/bn/bn_lcl.h
17     struct bignum_st {
18         BN_ULONG *d;
19         int top;
20
21         int dmax;
22         int neg;
23         int flags;
24     };
25 #else
26 #define EC_POINT_get_affine_coordinates EC_POINT_get_affine_coordinates_GFp
27 #define EC_POINT_set_affine_coordinates EC_POINT_set_affine_coordinates_GFp
28 #endif // OPENSSL_VERSION_NUMBER
29
30 #include <openssl/bn.h>

```

B.2.2.2.2. Macros and Defines

Make sure that the library is using the correct size for a crypt word

```

31 #if defined THIRTY_TWO_BIT && (RADIX_BITS != 32) \
32    || ((defined SIXTY_FOUR_BIT_LONG || defined SIXTY_FOUR_BIT) \
33        && (RADIX_BITS != 64))
34 #error Ossl library is using different radix
35 #endif

```

Allocate a local BIGNUM value. For the allocation, a *bigNum* structure is created as is a local BIGNUM. The *bigNum* is initialized and then the BIGNUM is set to reference the local value.

```

36 #define BIG_VAR(name, bits) \
37     BN_VAR(name##Bn, (bits)); \
38     BIGNUM _##name; \
39     BIGNUM *name = BigInitialized(&_##name, \
40                                   BnInit(name##Bn, \
41                                           BYTES_TO_CRYPT_WORDS(sizeof(_##name##Bn.d))))
41

```

Allocate a BIGNUM and initialize with the values in a *bigNum* initializer

```

42 #define BIG_INITIALIZED(name, initializer)          \
43     BIGNUM _##name;                                \
44     BIGNUM *name = BigInitialized(&_##name, initializer)
45
46 typedef struct
47 {
48     const ECC_CURVE_DATA *C;    // the TPM curve values
49     EC_GROUP *G;               // group parameters
50     BN_CTX *CTX;               // the context for the math (this might not be
51                               // the context in which the curve was created>;
52 } OSSL_CURVE_DATA;
53
54 typedef OSSL_CURVE_DATA *bigCurve;
55
56 #define AccessCurveData(E) ((E)->C)
57
58 #include "TpmToOsslSupport_fp.h"

```

Start and end a context within which the OpenSSL memory management works

```

59 #define OSSL_ENTER() BN_CTX *CTX = OsslContextEnter()
60 #define OSSL_LEAVE() OsslContextLeave(CTX)

```

Start and end a context that spans multiple ECC functions. This is used so that the group for the curve can persist across multiple frames.

```

61 #define CURVE_INITIALIZED(name, initializer)          \
62     OSSL_CURVE_DATA _##name;                        \
63     bigCurve name = BnCurveInitialize(&_##name, initializer)
64 #define CURVE_FREE(name) BnCurveFree(name)

```

Start and end a local stack frame within the context of the curve frame

```

65 #define ECC_ENTER() BN_CTX *CTX = OsslPushContext(E->CTX)
66 #define ECC_LEAVE() OsslPopContext(CTX)
67
68 #define BN_NEW() BnNewVariable(CTX)

```

This definition would change if there were something to report

```

69 #define MathLibSimulationEnd()
70
71 #endif // MATH_LIB_DEFINED

```


B.2.2.3. TpmToOsslSym.h

B.2.2.3.1. Introduction

This header file is used to *splice* the OpenSSL library into the TPM code.

The support required of a library are a hash module, a block cipher module and portions of a big number library.

All of the library-dependent headers should have the same guard to that only the first one gets defined.

```

1  #ifndef SYM_LIB_DEFINED
2  #define SYM_LIB_DEFINED
3
4  #define SYM_LIB_OSSL
5
6  #include <openssl/aes.h>
7
8  #if ALG_TDES
9  #include <openssl/des.h>
10 #endif
11
12 #if ALG_SM4
13 #   if defined(OPENSSL_NO_SM4) || OPENSSL_VERSION_NUMBER < 0x10101010L
14 #       undef ALG_SM4
15 #       define ALG_SM4    ALG_NO
16 #   elif OPENSSL_VERSION_NUMBER >= 0x10200000L
17 #       include <openssl/sm4.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
22 #       define SM4_KEY_SCHEDULE    32
23
24       typedef struct SM4_KEY_st {
25           uint32_t rk[SM4_KEY_SCHEDULE];
26       } SM4_KEY;
27
28       int SM4_set_key(const uint8_t *key, SM4_KEY *ks);
29       void SM4_encrypt(const uint8_t *in, uint8_t *out, const SM4_KEY *ks);
30       void SM4_decrypt(const uint8_t *in, uint8_t *out, const SM4_KEY *ks);
31 #   endif // OpenSSL < 1.2
32 #endif // ALG_SM4
33
34 #if ALG_CAMELLIA
35 #include <openssl/camellia.h>
36 #endif
37
38 #include <openssl/bn.h>
39 #include <openssl/ssl_typ.h>

```

B.2.2.3.2. Links to the OpenSSL symmetric algorithms.

The Crypt functions that call the block encryption function use the parameters in the order:

- 1) *keySchedule*
- 2) *in buffer*
- 3) *out buffer* Since open SSL uses the order in *encryptoCall_t* above, need to swizzle the values to the order required by the library.

```
40 #define SWIZZLE(keySchedule, in, out)
```

\

```
41      (const BYTE *) (in), (BYTE *) (out), (void *) (keySchedule)
```

Define the order of parameters to the library functions that do block encryption and decryption.

```
42  typedef void (*TpmCryptSetSymKeyCall_t) (
43      const BYTE *in,
44      BYTE *out,
45      void *keySchedule
46  );
47
48  #define SYM_ALIGNMENT    RADIX_BYTES
```

B.2.2.3.3. Links to the OpenSSL AES code

Macros to set up the encryption/decryption key schedules

AES:

```
49  #define TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule) \
50      AES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *) (schedule)) \
51  #define TpmCryptSetDecryptKeyAES(key, keySizeInBits, schedule) \
52      AES_set_decrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *) (schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly. Currently, only used by CryptSym.c and CryptRand.c

When using these calls, to call the AES block encryption code, the caller should use: TpmCryptEncryptAES(SWIZZLE(keySchedule, in, out));

```
53  #define TpmCryptEncryptAES          AES_encrypt
54  #define TpmCryptDecryptAES          AES_decrypt
55  #define tpmKeyScheduleAES          AES_KEY
```

B.2.2.3.4. Links to the OpenSSL DES code

```
56  #if ALG_TDES
57  #include "TpmToOsslDesSupport_fp.h"
58  #endif
59
60  #define TpmCryptSetEncryptKeyTDES(key, keySizeInBits, schedule) \
61      TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *) (schedule)) \
62  #define TpmCryptSetDecryptKeyTDES(key, keySizeInBits, schedule) \
63      TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *) (schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly. Currently, only used by CryptRand.c

```
64  #define TpmCryptEncryptTDES          TDES_encrypt
65  #define TpmCryptDecryptTDES          TDES_decrypt
66  #define tpmKeyScheduleTDES          DES_key_schedule
```

B.2.2.3.5. Links to the OpenSSL SM4 code

Macros to set up the encryption/decryption key schedules

```
67  #define TpmCryptSetEncryptKeySM4(key, keySizeInBits, schedule) \
68      SM4_set_key((key), (tpmKeyScheduleSM4 *) (schedule)) \
69  #define TpmCryptSetDecryptKeySM4(key, keySizeInBits, schedule) \
70      SM4_set_key((key), (tpmKeyScheduleSM4 *) (schedule))
```

Macros to alias encryption calls to specific algorithms. This should be used sparingly.

```

71 #define TpmCryptEncryptSM4          SM4_encrypt
72 #define TpmCryptDecryptSM4        SM4_decrypt
73 #define tpmKeyScheduleSM4         SM4_KEY

```

B.2.2.3.6. Links to the OpenSSL CAMELLIA code

Macros to set up the encryption/decryption key schedules

```

74 #define TpmCryptSetEncryptKeyCAMELLIA(key, keySizeInBits, schedule) \
75     Camellia_set_key((key), (keySizeInBits), (tpmKeyScheduleCAMELLIA *) (schedule))
76 #define TpmCryptSetDecryptKeyCAMELLIA(key, keySizeInBits, schedule) \
77     Camellia_set_key((key), (keySizeInBits), (tpmKeyScheduleCAMELLIA *) (schedule))

```

Macros to alias encryption calls to specific algorithms. This should be used sparingly.

```

78 #define TpmCryptEncryptCAMELLIA      Camellia_encrypt
79 #define TpmCryptDecryptCAMELLIA      Camellia_decrypt
80 #define tpmKeyScheduleCAMELLIA      CAMELLIA_KEY

```

Forward reference

```

81 typedef union tpmCryptKeySchedule_t tpmCryptKeySchedule_t;

```

This definition would change if there were something to report

```

82 #define SymLibSimulationEnd()
83
84 #endif // SYM_LIB_DEFINED

```

B.2.3. Source Files

B.2.3.1. TpmToOsslDesSupport.c

B.2.3.1.1. Introduction

The functions in this file are used for initialization of the interface to the OpenSSL library.

B.2.3.1.2. Defines and Includes

```
1  #include "Tpm.h"
2
3  #if (defined SYM_LIB_OSSL) && ALG_TDES
```

B.2.3.1.3. Functions

B.2.3.1.3.1. TDES_set_encrypt_key()

This function makes creation of a TDES key look like the creation of a key for any of the other OpenSSL block ciphers. It will create three key schedules, one for each of the DES keys. If there are only two keys, then the third schedule is a copy of the first.

```
4  void
5  TDES_set_encrypt_key(
6      const BYTE                *key,
7      UINT16                    keySizeInBits,
8      tpmKeyScheduleTDES       *keySchedule
9  )
10 {
11     DES_set_key_unchecked((const_DES_cblock *)key, &keySchedule[0]);
12     DES_set_key_unchecked((const_DES_cblock *)&key[8], &keySchedule[1]);
13     // If is two-key, copy the schedule for K1 into K3, otherwise, compute the
14     // the schedule for K3
15     if(keySizeInBits == 128)
16         keySchedule[2] = keySchedule[0];
17     else
18         DES_set_key_unchecked((const_DES_cblock *)&key[16],
19                               &keySchedule[2]);
20 }
```

B.2.3.1.3.2. TDES_encrypt()

The TPM code uses one key schedule. For TDES, the schedule contains three schedules. OpenSSL wants the schedules referenced separately. This function does that.

```
21 void TDES_encrypt(
22     const BYTE                *in,
23     BYTE                      *out,
24     tpmKeyScheduleTDES       *ks
25 )
26 {
27     DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,
28                     &ks[0], &ks[1], &ks[2],
29                     DES_ENCRYPT);
30 }
```

B.2.3.1.3.3. TDES_decrypt()

As with TDES_encrypt() this function bridges between the TPM single schedule model and the OpenSSL three schedule model.

```
31 void TDES_decrypt(  
32     const BYTE      *in,  
33     BYTE            *out,  
34     tpmKeyScheduleTDES *ks  
35 )  
36 {  
37     DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,  
38                     &ks[0], &ks[1], &ks[2],  
39                     DES_DECRYPT);  
40 }  
41 #endif // SYM_LIB_OSSL
```

B.2.3.2. TpmToOsslMath.c

B.2.3.2.1. Introduction

The functions in this file provide the low-level interface between the TPM code and the big number and elliptic curve math routines in OpenSSL.

Most math on big numbers require a context. The context contains the memory in which OpenSSL creates and manages the big number values. When a OpenSSL math function will be called that modifies a BIGNUM value, that value must be created in an OpenSSL context. The first line of code in such a function must be: `OSSL_ENTER()`; and the last operation before returning must be `OSSL_LEAVE()`. OpenSSL variables can then be created with `BnNewVariable()`. Constant values to be used by OpenSSL are created from the *bigNum* values passed to the functions in this file. Space for the BIGNUM control block is allocated in the stack of the function and then it is initialized by calling `BigInitialized()`. That function sets up the values in the BIGNUM structure and sets the data pointer to point to the data in the `bignum_t`. This is only used when the value is known to be a constant in the called function.

Because the allocations of constants is on the local stack and the `OSSL_ENTER()/OSSL_LEAVE()` pair flushes everything created in OpenSSL memory, there should be no chance of a memory leak.

B.2.3.2.2. Includes and Defines

```
1  #include "Tpm.h"
2
3  #ifndef MATH_LIB_OSSL
4  #include "TpmToOsslMath_fp.h"
```

B.2.3.2.3. Functions

B.2.3.2.3.1. OsslToTpmBn()

This function converts an OpenSSL BIGNUM to a TPM bignum. In this implementation it is assumed that OpenSSL uses a different control structure but the same data layout -- an array of native-endian words in little-endian order.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure because value will not fit or OpenSSL variable doesn't exist

```
5  BOOL
6  OsslToTpmBn(
7      bigNum          bn,
8      BIGNUM          *osslBn
9  )
10 {
11     VERIFY(osslBn != NULL);
12     // If the bn is NULL, it means that an output value pointer was NULL meaning that
13     // the results is simply to be discarded.
14     if(bn != NULL)
15     {
16         int          i;
17         //
18         VERIFY((unsigned)osslBn->top <= BnGetAllocated(bn));
19         for(i = 0; i < osslBn->top; i++)
20             bn->d[i] = osslBn->d[i];
21         BnSetTop(bn, osslBn->top);
```

```

22     }
23     return TRUE;
24 Error:
25     return FALSE;
26 }

```

B.2.3.2.3.2. BigInitialized()

This function initializes an OSSL BIGNUM from a TPM *bigConst*. Do not use this for values that are passed to OpenSSL() when they are not declared as const in the function prototype. Instead, use BnNewVariable().

```

27 BIGNUM *
28 BigInitialized(
29     BIGNUM          *toInit,
30     bigConst        initializer
31 )
32 {
33     if(initializer == NULL)
34         FAIL(FATAL_ERROR_PARAMETER);
35     if(toInit == NULL || initializer == NULL)
36         return NULL;
37     toInit->d = (BN_ULONG *)&initializer->d[0];
38     toInit->dmax = (int)initializer->allocated;
39     toInit->top = (int)initializer->size;
40     toInit->neg = 0;
41     toInit->flags = 0;
42     return toInit;
43 }
44 #ifndef OSSL_DEBUG
45 # define BIGNUM_PRINT(label, bn, eol)
46 # define DEBUG_PRINT(x)
47 #else
48 # define DEBUG_PRINT(x)    printf("%s", x)
49 # define BIGNUM_PRINT(label, bn, eol) BIGNUM_print((label), (bn), (eol))

```

B.2.3.2.3.3. BIGNUM_print()

```

50 static void
51 BIGNUM_print(
52     const char      *label,
53     const BIGNUM    *a,
54     BOOL            eol
55 )
56 {
57     BN_ULONG        *d;
58     int              i;
59     int              notZero = FALSE;
60
61     if(label != NULL)
62         printf("%s", label);
63     if(a == NULL)
64     {
65         printf("NULL");
66         goto done;
67     }
68     if (a->neg)
69         printf("-");
70     for(i = a->top, d = &a->d[i - 1]; i > 0; i--)
71     {
72         int          j;
73         BN_ULONG      l = *d--;
74         for(j = BN_BITS2 - 8; j >= 0; j -= 8)

```



```

75     {
76         BYTE    b = (BYTE)((l >> j) & 0xFF);
77         notZero = notZero || (b != 0);
78         if(notZero)
79             printf("%02x", b);
80     }
81     if(!notZero)
82         printf("0");
83 }
84 done:
85     if(eol)
86         printf("\n");
87     return;
88 }
89 #endif

```

B.2.3.2.3.4. BnNewVariable()

This function allocates a new variable in the provided context. If the context does not exist or the allocation fails, it is a catastrophic failure.

```

90 static BIGNUM *
91 BnNewVariable(
92     BN_CTX          *CTX
93 )
94 {
95     BIGNUM          *new;
96     //
97     // This check is intended to protect against calling this function without
98     // having initialized the CTX.
99     if((CTX == NULL) || ((new = BN_CTX_get(CTX)) == NULL))
100         FAIL(FATAL_ERROR_ALLOCATION);
101     return new;
102 }
103 #if LIBRARY_COMPATIBILITY_CHECK

```

B.2.3.2.3.5. MathLibraryCompatibilityCheck()

```

104 BOOL
105 MathLibraryCompatibilityCheck(
106     void
107 )
108 {
109     OSSL_ENTER();
110     BIGNUM          *osslTemp = BnNewVariable(CTX);
111     crypt_uword_t    i;
112     BYTE            test[] = {0x1F, 0x1E, 0x1D, 0x1C, 0x1B, 0x1A, 0x19, 0x18,
113                               0x17, 0x16, 0x15, 0x14, 0x13, 0x12, 0x11, 0x10,
114                               0x0F, 0x0E, 0x0D, 0x0C, 0x0B, 0x0A, 0x09, 0x08,
115                               0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00};
116     BN_VAR(tpmTemp, sizeof(test) * 8); // allocate some space for a test value
117     //
118     // Convert the test data to a bigNum
119     BnFromBytes(tpmTemp, test, sizeof(test));
120     // Convert the test data to an OpenSSL BIGNUM
121     BN_bin2bn(test, sizeof(test), osslTemp);
122     // Make sure the values are consistent
123     VERIFY(osslTemp->top == (int)tpmTemp->size);
124     for(i = 0; i < tpmTemp->size; i++)
125         VERIFY(osslTemp->d[i] == tpmTemp->d[i]);
126     OSSL_LEAVE();
127     return 1;
128 Error:

```

```

129     return 0;
130 }
131 #endif

```

B.2.3.2.3.6. BnModMult()

This function does a modular multiply. It first does a multiply and then a divide and returns the remainder of the divide.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

132 LIB_EXPORT BOOL
133 BnModMult(
134     bigNum          result,
135     bigConst        op1,
136     bigConst        op2,
137     bigConst        modulus
138 )
139 {
140     OSSL_ENTER();
141     BOOL            OK = TRUE;
142     BIGNUM          *bnResult = BN_NEW();
143     BIGNUM          *bnTemp = BN_NEW();
144     BIG_INITIALIZED(bnOp1, op1);
145     BIG_INITIALIZED(bnOp2, op2);
146     BIG_INITIALIZED(bnMod, modulus);
147     //
148     VERIFY(BN_mul(bnTemp, bnOp1, bnOp2, CTX));
149     VERIFY(BN_div(NULL, bnResult, bnTemp, bnMod, CTX));
150     VERIFY(OsslToTpmBn(result, bnResult));
151     goto Exit;
152 Error:
153     OK = FALSE;
154 Exit:
155     OSSL_LEAVE();
156     return OK;
157 }

```

B.2.3.2.3.7. BnMult()

Multiplies two numbers

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

158 LIB_EXPORT BOOL
159 BnMult(
160     bigNum          result,
161     bigConst        multiplicand,
162     bigConst        multiplier
163 )
164 {
165     OSSL_ENTER();
166     BIGNUM          *bnTemp = BN_NEW();
167     BOOL            OK = TRUE;
168     BIG_INITIALIZED(bnA, multiplicand);

```

```

169     BIG_INITIALIZED(bnB, multiplier);
170     //
171     VERIFY(BN_mul(bnTemp, bnA, bnB, CTX));
172     VERIFY(OsslToTpmBn(result, bnTemp));
173     goto Exit;
174 Error:
175     OK = FALSE;
176 Exit:
177     OSSL_LEAVE();
178     return OK;
179 }

```

B.2.3.2.3.8. BnDiv()

This function divides two *bigNum* values. The function returns FALSE if there is an error in the operation.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

180 LIB_EXPORT BOOL
181 BnDiv(
182     bigNum          quotient,
183     bigNum          remainder,
184     bigConst        dividend,
185     bigConst        divisor
186 )
187 {
188     OSSL_ENTER();
189     BIGNUM          *bnQ = BN_NEW();
190     BIGNUM          *bnR = BN_NEW();
191     BOOL            OK = TRUE;
192     BIG_INITIALIZED(bnDend, dividend);
193     BIG_INITIALIZED(bnSor, divisor);
194     //
195     if(BnEqualZero(divisor))
196         FAIL(FATAL_ERROR_DIVIDE_ZERO);
197     VERIFY(BN_div(bnQ, bnR, bnDend, bnSor, CTX));
198     VERIFY(OsslToTpmBn(quotient, bnQ));
199     VERIFY(OsslToTpmBn(remainder, bnR));
200     DEBUG_PRINT("In BnDiv:\n");
201     BIGNUM_PRINT("    bnDividend: ", bnDend, TRUE);
202     BIGNUM_PRINT("    bnDivisor: ", bnSor, TRUE);
203     BIGNUM_PRINT("    bnQuotient: ", bnQ, TRUE);
204     BIGNUM_PRINT("    bnRemainder: ", bnR, TRUE);
205     goto Exit;
206 Error:
207     OK = FALSE;
208 Exit:
209     OSSL_LEAVE();
210     return OK;
211 }
212 #if ALG_RSA

```

B.2.3.2.3.9. BnGcd()

Get the greatest common divisor of two numbers

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

213 LIB_EXPORT BOOL
214 BnGcd(
215     bigNum    gcd,           // OUT: the common divisor
216     bigConst   number1,      // IN:
217     bigConst   number2       // IN:
218 )
219 {
220     OSSL_ENTER();
221     BIGNUM      *bnGcd = BN_NEW();
222     BOOL        OK = TRUE;
223     BIG_INITIALIZED(bn1, number1);
224     BIG_INITIALIZED(bn2, number2);
225     //
226     VERIFY(BN_gcd(bnGcd, bn1, bn2, CTX));
227     VERIFY(OsslToTpmBn(gcd, bnGcd));
228     goto Exit;
229 Error:
230     OK = FALSE;
231 Exit:
232     OSSL_LEAVE();
233     return OK;
234 }

```

B.2.3.2.3.10. BnModExp()

Do modular exponentiation using *bigNum* values. The conversion from a *bignum_t* to a *bigNum* is trivial as they are based on the same structure

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

235 LIB_EXPORT BOOL
236 BnModExp(
237     bigNum      result,       // OUT: the result
238     bigConst    number,       // IN: number to exponentiate
239     bigConst    exponent,     // IN:
240     bigConst    modulus       // IN:
241 )
242 {
243     OSSL_ENTER();
244     BIGNUM      *bnResult = BN_NEW();
245     BOOL        OK = TRUE;
246     BIG_INITIALIZED(bnN, number);
247     BIG_INITIALIZED(bnE, exponent);
248     BIG_INITIALIZED(bnM, modulus);
249     //
250     VERIFY(BN_mod_exp(bnResult, bnN, bnE, bnM, CTX));
251     VERIFY(OsslToTpmBn(result, bnResult));
252     goto Exit;
253 Error:
254     OK = FALSE;
255 Exit:
256     OSSL_LEAVE();
257     return OK;
258 }

```

B.2.3.2.3.11. BnModInverse()

Modular multiplicative inverse

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

259 LIB_EXPORT BOOL
260 BnModInverse(
261     bigNum          result,
262     bigConst         number,
263     bigConst         modulus
264 )
265 {
266     OSSL_ENTER();
267     BIGNUM          *bnResult = BN_NEW();
268     BOOL            OK = TRUE;
269     BIG_INITIALIZED(bnN, number);
270     BIG_INITIALIZED(bnM, modulus);
271     //
272     VERIFY(BN_mod_inverse(bnResult, bnN, bnM, CTX) != NULL);
273     VERIFY(OsslToTpmBn(result, bnResult));
274     goto Exit;
275 Error:
276     OK = FALSE;
277 Exit:
278     OSSL_LEAVE();
279     return OK;
280 }
281 #endif // ALG_RSA
282
283 #if ALG_ECC

```

B.2.3.2.3.12. PointFromOssl()Function to copy the point result from an OSSL function to a *bigNum*

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

284 static BOOL
285 PointFromOssl(
286     bigPoint         pOut,          // OUT: resulting point
287     EC_POINT         *pIn,          // IN: the point to return
288     bigCurve          E              // IN: the curve
289 )
290 {
291     BIGNUM          *x = NULL;
292     BIGNUM          *y = NULL;
293     BOOL            OK;
294     BN_CTX_start(E->CTX);
295     //
296     x = BN_CTX_get(E->CTX);
297     y = BN_CTX_get(E->CTX);
298
299     if(y == NULL)
300         FAIL(FATAL_ERROR_ALLOCATION);
301     // If this returns false, then the point is at infinity

```

```

302     OK = EC_POINT_get_affine_coordinates_GFp(E->G, pIn, x, y, E->CTX);
303     if(OK)
304     {
305         OsslToTpmBn(pOut->x, x);
306         OsslToTpmBn(pOut->y, y);
307         BnSetWord(pOut->z, 1);
308     }
309     else
310         BnSetWord(pOut->z, 0);
311     BN_CTX_end(E->CTX);
312     return OK;
313 }

```

B.2.3.2.3.13. EcPointInitialized()

Allocate and initialize a point.

```

314 static EC_POINT *
315 EcPointInitialized(
316     pointConst      initializer,
317     bigCurve         E
318 )
319 {
320     EC_POINT         *P = NULL;
321
322     if(initializer != NULL)
323     {
324         BIG_INITIALIZED(bnX, initializer->x);
325         BIG_INITIALIZED(bnY, initializer->y);
326         if(E == NULL)
327             FAIL(FATAL_ERROR_ALLOCATION);
328         P = EC_POINT_new(E->G);
329         if(!EC_POINT_set_affine_coordinates_GFp(E->G, P, bnX, bnY, E->CTX))
330             P = NULL;
331     }
332     return P;
333 }

```

B.2.3.2.3.14. BnCurveInitialize()

This function initializes the OpenSSL curve information structure. This structure points to the TPM-defined values for the curve, to the context for the number values in the frame, and to the OpenSSL-defined group values.

Return Value	Meaning*
NULL	the TPM_ECC_CURVE is not valid or there was a problem in initializing the curve data
non-NULL	points to <i>E</i>

```

334 LIB_EXPORT bigCurve
335 BnCurveInitialize(
336     bigCurve         E,           // IN: curve structure to initialize
337     TPM_ECC_CURVE    curveId      // IN: curve identifier
338 )
339 {
340     const ECC_CURVE_DATA *C = GetCurveData(curveId);
341     if(C == NULL)
342         E = NULL;
343     if(E != NULL)
344     {
345         // This creates the OpenSSL memory context that stays in effect as long as the

```

```

346     // curve (E) is defined.
347     OSSL_ENTER();                                     // if the allocation fails, the TPM fails
348     EC_POINT *P = NULL;
349     BIG_INITIALIZED(bnP, C->prime);
350     BIG_INITIALIZED(bnA, C->a);
351     BIG_INITIALIZED(bnB, C->b);
352     BIG_INITIALIZED(bnX, C->base.x);
353     BIG_INITIALIZED(bnY, C->base.y);
354     BIG_INITIALIZED(bnN, C->order);
355     BIG_INITIALIZED(bnH, C->h);
356     //
357     E->C = C;
358     E->CTX = CTX;
359
360     // initialize EC group, associate a generator point and initialize the point
361     // from the parameter data
362     // Create a group structure
363     E->G = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, CTX);
364     VERIFY(E->G != NULL);
365
366     // Allocate a point in the group that will be used in setting the
367     // generator. This is not needed after the generator is set.
368     P = EC_POINT_new(E->G);
369     VERIFY(P != NULL);
370
371     // Need to use this in case Montgomery method is being used
372     VERIFY(EC_POINT_set_affine_coordinates_GFp(E->G, P, bnX, bnY, CTX));
373     // Now set the generator
374     VERIFY(EC_GROUP_set_generator(E->G, P, bnN, bnH));
375
376     EC_POINT_free(P);
377     goto Exit;
378 Error:
379     EC_POINT_free(P);
380     BnCurveFree(E);
381     E = NULL;
382 }
383 Exit:
384     return E;
385 }

```

B.2.3.2.3.15. BnCurveFree()

This function will free the allocated components of the curve and end the frame in which the curve data exists

```

386 LIB_EXPORT void
387 BnCurveFree(
388     bigCurve      E
389 )
390 {
391     if(E)
392     {
393         EC_GROUP_free(E->G);
394         OsslContextLeave(E->CTX);
395     }
396 }

```

B.2.3.2.3.16. BnEccModMult()

This function does a point multiply of the form $R = [d]S$

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation; treat as result being point at infinity

```

397 LIB_EXPORT BOOL
398 BnEccModMult(
399     bigPoint          R,          // OUT: computed point
400     pointConst        S,          // IN: point to multiply by 'd' (optional)
401     bigConst          d,          // IN: scalar for [d]S
402     bigCurve          E
403 )
404 {
405     EC_POINT          *pR = EC_POINT_new(E->G);
406     EC_POINT          *pS = EcPointInitialized(S, E);
407     BIG_INITIALIZED(bnD, d);
408
409     if(S == NULL)
410         EC_POINT_mul(E->G, pR, bnD, NULL, NULL, E->CTX);
411     else
412         EC_POINT_mul(E->G, pR, NULL, pS, bnD, E->CTX);
413     PointFromOssl(R, pR, E);
414     EC_POINT_free(pR);
415     EC_POINT_free(pS);
416     return !BnEqualZero(R->z);
417 }

```

B.2.3.2.3.17. BnEccModMult2()

This function does a point multiply of the form $R = [d]G + [u]Q$

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation; treat as result being point at infinity

```

418 LIB_EXPORT BOOL
419 BnEccModMult2(
420     bigPoint          R,          // OUT: computed point
421     pointConst        S,          // IN: optional point
422     bigConst          d,          // IN: scalar for [d]S or [d]G
423     pointConst        Q,          // IN: second point
424     bigConst          u,          // IN: second scalar
425     bigCurve          E          // IN: curve
426 )
427 {
428     EC_POINT          *pR = EC_POINT_new(E->G);
429     EC_POINT          *pS = EcPointInitialized(S, E);
430     BIG_INITIALIZED(bnD, d);
431     EC_POINT          *pQ = EcPointInitialized(Q, E);
432     BIG_INITIALIZED(bnU, u);
433
434     if(S == NULL || S == (pointConst)&(AccessCurveData(E)->base))
435         EC_POINT_mul(E->G, pR, bnD, pQ, bnU, E->CTX);
436     else
437     {
438         const EC_POINT *points[2];
439         const BIGNUM   *scalars[2];
440         points[0] = pS;
441         points[1] = pQ;
442         scalars[0] = bnD;
443         scalars[1] = bnU;

```

```

444     EC_POINTs_mul(E->G, pR, NULL, 2, points, scalars, E->CTX);
445 }
446 PointFromOssl(R, pR, E);
447 EC_POINT_free(pR);
448 EC_POINT_free(pS);
449 EC_POINT_free(pQ);
450 return !BnEqualZero(R->z);
451 }

```

B.2.3.2.4. BnEccAdd()

This function does addition of two points.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation; treat as result being point at infinity

```

452 LIB_EXPORT BOOL
453 BnEccAdd(
454     bigPoint          R,          // OUT: computed point
455     pointConst        S,          // IN: point to multiply by 'd'
456     pointConst        Q,          // IN: second point
457     bigCurve          E           // IN: curve
458 )
459 {
460     EC_POINT          *pR = EC_POINT_new(E->G);
461     EC_POINT          *pS = EcPointInitialized(S, E);
462     EC_POINT          *pQ = EcPointInitialized(Q, E);
463 //
464     EC_POINT_add(E->G, pR, pS, pQ, E->CTX);
465
466     PointFromOssl(R, pR, E);
467     EC_POINT_free(pR);
468     EC_POINT_free(pS);
469     EC_POINT_free(pQ);
470     return !BnEqualZero(R->z);
471 }
472 #endif // ALG_ECC
473
474 #endif // MATHLIB_OSSL

```

B.2.3.3. TpmToOsslSupport.c**B.2.3.3.1. Introduction**

The functions in this file are used for initialization of the interface to the OpenSSL library.

B.2.3.3.2. Defines and Includes

```

1  #include "Tpm.h"
2
3  #if defined(HASH_LIB_OSSL) || defined(MATH_LIB_OSSL) || defined(SYM_LIB_OSSL)

```

Used to pass the pointers to the correct sub-keys

```

4  typedef const BYTE *desKeyPointers[3];

```

B.2.3.3.2.1. SupportLibInit()

This does any initialization required by the support library.

```

5  LIB_EXPORT int
6  SupportLibInit(
7      void
8  )
9  {
10     return TRUE;
11 }

```

B.2.3.3.2.2. OsslContextEnter()

This function is used to initialize an OpenSSL context at the start of a function that will call to an OpenSSL math function.

```

12 BN_CTX *
13 OsslContextEnter(
14     void
15 )
16 {
17     BN_CTX          *CTX = BN_CTX_new();
18     //
19     return OsslPushContext(CTX);
20 }

```

B.2.3.3.2.3. OsslContextLeave()

This is the companion function to OsslContextEnter().

```

21 void
22 OsslContextLeave(
23     BN_CTX          *CTX
24 )
25 {
26     OsslPopContext(CTX);
27     BN_CTX_free(CTX);
28 }

```

B.2.3.3.2.4. OsslPushContext()

This function is used to create a frame in a context. All values allocated within this context after the frame is started will be automatically freed when the context (OsslPopContext())

```
29  BN_CTX *
30  OsslPushContext (
31      BN_CTX      *CTX
32  )
33  {
34      if (CTX == NULL)
35          FAIL(FATAL_ERROR_ALLOCATION);
36      BN_CTX_start (CTX);
37      return CTX;
38  }
```

B.2.3.3.2.5. OsslPopContext()

This is the companion function to OsslPushContext().

```
39  void
40  OsslPopContext (
41      BN_CTX      *CTX
42  )
43  {
44      // BN_CTX_end can't be called with NULL. It will blow up.
45      if (CTX != NULL)
46          BN_CTX_end (CTX);
47  }
48  #endif // HASH_LIB_OSSL || MATH_LIB_OSSL || SYM_LIB_OSSL
```

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.2 Cancel.c

C.2.1. Description

This module simulates the cancel pins on the TPM.

C.2.2. Includes, Typedefs, Structures, and Defines

```
1  #include "Platform.h"
```

C.2.3. Functions

C.2.3.1. `_plat__IsCanceled()`

Check if the cancel flag is set

Return Value	Meaning
TRUE(1)	if cancel flag is set
FALSE(0)	if cancel flag is not set

```
2  LIB_EXPORT int
3  _plat__IsCanceled(
4      void
5  )
6  {
7      // return cancel flag
8      return s_isCanceled;
9  }
```

C.2.3.2. `_plat__SetCancel()`

Set cancel flag.

```
10 LIB_EXPORT void
11 _plat__SetCancel(
12     void
13 )
14 {
15     s_isCanceled = TRUE;
16     return;
17 }
```

C.2.3.3. _plat__ClearCancel()

Clear cancel flag

```
18  LIB_EXPORT void
19  _plat__ClearCancel(
20      void
21  )
22  {
23      s_isCanceled = FALSE;
24      return;
25  }
```

DRAFT

C.3 Clock.c

C.3.1. Description

This file contains the routines that are used by the simulator to mimic a hardware clock on a TPM.

In this implementation, all the time values are measured in millisecond. However, the precision of the clock functions may be implementation dependent.

C.3.2. Includes and Data Definitions

```
1  #include <assert.h>
2  #include "Platform.h"
3  #include "TpmFail_fp.h"
```

C.3.3. Simulator Functions

C.3.3.1. Introduction

This set of functions is intended to be called by the simulator environment in order to simulate hardware events.

C.3.3.2. _plat__TimerReset()

This function sets current system clock time as t0 for counting TPM time. This function is called at a power on event to reset the clock. When the clock is reset, the indication that the clock was stopped is also set.

```
4  LIB_EXPORT void
5  _plat__TimerReset(
6      void
7  )
8  {
9      s_lastSystemTime = 0;
10     s_tpmTime = 0;
11     s_adjustRate = CLOCK_NOMINAL;
12     s_timerReset = TRUE;
13     s_timerStopped = TRUE;
14     return;
15 }
```

C.3.3.3. _plat__TimerRestart()

This function should be called in order to simulate the restart of the timer should it be stopped while power is still applied.

```
16 LIB_EXPORT void
17 _plat__TimerRestart(
18     void
19 )
20 {
21     s_timerStopped = TRUE;
22     return;
23 }
```


C.3.4. Functions Used by TPM

C.3.4.1. Introduction

These functions are called by the TPM code. They should be replaced by appropriated hardware functions.

```
24 #include <time.h>
25 clock_t      debugTime;
```

C.3.4.2. `_plat__RealTime()`

This is another, probably futile, attempt to define a portable function that will return a 64-bit clock value that has *mSec* resolution.

```
26 LIB_EXPORT uint64_t
27 _plat__RealTime(
28     void
29 )
30 {
31     clock64_t      time;
32 #ifdef _MSC_VER
33     struct _timeb   sysTime;
34 //
35     _ftime_s(&sysTime);
36     time = (clock64_t)(sysTime.time) * 1000 + sysTime.millitm;
37     // set the time back by one hour if daylight savings
38     if(sysTime.dstflag)
39         time -= 1000 * 60 * 60; // mSec/sec * sec/min * min/hour = ms/hour
40 #else
41     // hopefully, this will work with most UNIX systems
42     struct timespec  systime;
43 //
44     clock_gettime(CLOCK_MONOTONIC, &systime);
45     time = (clock64_t)systime.tv_sec * 1000 + (systime.tv_nsec / 1000000);
46 #endif
47     return time;
48 }
```

C.3.4.3. `_plat__TimerRead()`

This function provides access to the tick timer of the platform. The TPM code uses this value to drive the TPM Clock.

The tick timer is supposed to run when power is applied to the device. This timer should not be reset by time events including `_TPM_Init`. It should only be reset when TPM power is re-applied.

If the TPM is run in a protected environment, that environment may provide the tick time to the TPM as long as the time provided by the environment is not allowed to go backwards. If the time provided by the system can go backwards during a power discontinuity, then the `_plat__Signal_PowerOn()` should call `_plat__TimerReset()`.

```
49 LIB_EXPORT uint64_t
50 _plat__TimerRead(
51     void
52 )
53 {
54 #ifdef HARDWARE_CLOCK
55 #error "need a definition for reading the hardware clock"
56     return HARDWARE_CLOCK
```

```

57  #else
58      clock64_t          timeDiff;
59      clock64_t          adjustedTimeDiff;
60      clock64_t          timeNow;
61      clock64_t          readjustedTimeDiff;
62
63      // This produces a timeNow that is basically locked to the system clock.
64      timeNow = _plat_RealTime();
65
66      // if this hasn't been initialized, initialize it
67      if(s_lastSystemTime == 0)
68      {
69          s_lastSystemTime = timeNow;
70          debugTime = clock();
71          s_lastReportedTime = 0;
72          s_realTimePrevious = 0;
73      }
74      // The system time can bounce around and that's OK as long as we don't allow
75      // time to go backwards. When the time does appear to go backwards, set
76      // lastSystemTime to be the new value and then update the reported time.
77      if(timeNow < s_lastReportedTime)
78          s_lastSystemTime = timeNow;
79      s_lastReportedTime = s_lastReportedTime + timeNow - s_lastSystemTime;
80      s_lastSystemTime = timeNow;
81      timeNow = s_lastReportedTime;
82
83      // The code above produces a timeNow that is similar to the value returned
84      // by Clock(). The difference is that timeNow does not max out, and it is
85      // at a ms. rate rather than at a CLOCKS_PER_SEC rate. The code below
86      // uses that value and does the rate adjustment on the time value.
87      // If there is no difference in time, then skip all the computations
88      if(s_realTimePrevious >= timeNow)
89          return s_tpmTime;
90      // Compute the amount of time since the last update of the system clock
91      timeDiff = timeNow - s_realTimePrevious;
92
93      // Do the time rate adjustment and conversion from CLOCKS_PER_SEC to mSec
94      adjustedTimeDiff = (timeDiff * CLOCK_NOMINAL) / ((uint64_t)s_adjustRate);
95
96      // update the TPM time with the adjusted timeDiff
97      s_tpmTime += (clock64_t)adjustedTimeDiff;
98
99      // Might have some rounding error that would loose CLOCKS. See what is not
100     // being used. As mentioned above, this could result in putting back more than
101     // is taken out. Here, we are trying to recreate timeDiff.
102     readjustedTimeDiff = (adjustedTimeDiff * (uint64_t)s_adjustRate )
103                          / CLOCK_NOMINAL;
104
105     // adjusted is now converted back to being the amount we should advance the
106     // previous sampled time. It should always be less than or equal to timeDiff.
107     // That is, we could not have use more time than we started with.
108     s_realTimePrevious = s_realTimePrevious + readjustedTimeDiff;
109
110     #ifdef DEBUGGING_TIME
111         // Put this in so that TPM time will pass much faster than real time when
112         // doing debug.
113         // A value of 1000 for DEBUG_TIME_MULTIPLIER will make each ms into a second
114         // A good value might be 100
115         return (s_tpmTime * DEBUG_TIME_MULTIPLIER);
116     #endif
117     return s_tpmTime;
118 #endif
119 }

```

C.3.4.4. `_plat__TimerWasReset()`

This function is used to interrogate the flag indicating if the tick timer has been reset.

If the *resetFlag* parameter is SET, then the flag will be CLEAR before the function returns.

```

120  LIB_EXPORT int
121  _plat__TimerWasReset(
122      void
123  )
124  {
125      int      retVal = s_timerReset;
126      s_timerReset = FALSE;
127      return retVal;
128  }

```

C.3.4.5. `_plat__TimerWasStopped()`

This function is used to interrogate the flag indicating if the tick timer has been stopped. If so, this is typically a reason to roll the nonce.

This function will CLEAR the *s_timerStopped* flag before returning. This provides functionality that is similar to status register that is cleared when read. This is the model used here because it is the one that has the most impact on the TPM code as the flag can only be accessed by one entity in the TPM. Any other implementation of the hardware can be made to look like a read-once register.

```

129  LIB_EXPORT int
130  _plat__TimerWasStopped(
131      void
132  )
133  {
134      int      retVal = s_timerStopped;
135      s_timerStopped = FALSE;
136      return retVal;
137  }

```

C.3.4.6. `_plat__ClockAdjustRate()`

Adjust the clock rate

```

138  LIB_EXPORT void
139  _plat__ClockAdjustRate(
140      int      adjust          // IN: the adjust number. It could be positive
141                               // or negative
142  )
143  {
144      // We expect the caller should only use a fixed set of constant values to
145      // adjust the rate
146      switch(adjust)
147      {
148          case CLOCK_ADJUST_COARSE:
149              s_adjustRate += CLOCK_ADJUST_COARSE;
150              break;
151          case -CLOCK_ADJUST_COARSE:
152              s_adjustRate -= CLOCK_ADJUST_COARSE;
153              break;
154          case CLOCK_ADJUST_MEDIUM:
155              s_adjustRate += CLOCK_ADJUST_MEDIUM;
156              break;
157          case -CLOCK_ADJUST_MEDIUM:
158              s_adjustRate -= CLOCK_ADJUST_MEDIUM;
159              break;

```

```
160     case CLOCK_ADJUST_FINE:
161         s_adjustRate += CLOCK_ADJUST_FINE;
162         break;
163     case -CLOCK_ADJUST_FINE:
164         s_adjustRate -= CLOCK_ADJUST_FINE;
165         break;
166     default:
167         // ignore any other values;
168         break;
169 }
170 if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
171     s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
172 if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
173     s_adjustRate = CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT;
174
175 return;
176 }
```

C.4 Entropy.c

C.4.1. Includes and Local Values

```

1  #define _CRT_RAND_S
2  #include <stdlib.h>
3  #include <memory.h>
4  #include <time.h>
5  #include "Platform.h"
6
7  #ifndef _MSC_VER
8  #include <process.h>
9  #else
10 #include <unistd.h>
11 #endif

```

This is the last 32-bits of hardware entropy produced. We have to check to see that two consecutive 32-bit values are not the same because according to FIPS 140-2, annex C:

"If each call to an RNG produces blocks of n bits (where n > 15), the first n-bit block generated after power-up, initialization, or reset shall not be used, but shall be saved for comparison with the next n-bit block to be generated. Each subsequent generation of an n-bit block shall be compared with the previously generated block. The test shall fail if any two compared n-bit blocks are equal."

```

12 extern uint32_t      lastEntropy;

```

C.4.2. Functions

C.4.2.1. rand32()

Local function to get a 32-bit random number

```

13 static uint32_t
14 rand32(
15     void
16 )
17 {
18     uint32_t    rndNum = rand();
19     #if RAND_MAX < UINT16_MAX
20         // If the maximum value of the random number is a 15-bit number, then shift it up
21         // 15 bits, get 15 more bits, shift that up 2 and then XOR in another value to get
22         // a full 32 bits.
23         rndNum = (rndNum << 15) ^ rand();
24         rndNum = (rndNum << 2) ^ rand();
25     #elif RAND_MAX == UINT16_MAX
26         // If the maximum size is 16-bits, shift it and add another 16 bits
27         rndNum = (rndNum << 16) ^ rand();
28     #elif RAND_MAX < UINT32_MAX
29         // If 31 bits, then shift 1 and include another random value to get the extra bit
30         rndNum = (rndNum << 1) ^ rand();
31     #endif
32     return rndNum;
33 }

```

C.4.2.2. _plat__GetEntropy()

This function is used to get available hardware entropy. In a hardware implementation of this function, there would be no call to the system to get entropy.

Return Value	Meaning
< 0	hardware failure of the entropy generator, this is sticky
>= 0	the returned amount of entropy (bytes)

```

34 LIB_EXPORT int32_t
35 _plat_GetEntropy(
36     unsigned char    *entropy,          // output buffer
37     uint32_t         amount             // amount requested
38 )
39 {
40     uint32_t         rndNum;
41     int32_t          ret;
42     //
43     if(amount == 0)
44     {
45         // Seed the platform entropy source if the entropy source is software. There
46         // is no reason to put a guard macro (#if or #ifdef) around this code because
47         // this code would not be here if someone was changing it for a system with
48         // actual hardware.
49         //
50         // NOTE 1: The following command does not provide proper cryptographic
51         // entropy. Its primary purpose to make sure that different instances of the
52         // simulator, possibly started by a script on the same machine, are seeded
53         // differently. Vendors of the actual TPMs need to ensure availability of
54         // proper entropy using their platform-specific means.
55         //
56         // NOTE 2: In debug builds by default the reference implementation will seed
57         // its RNG deterministically (without using any platform provided randomness).
58         // See the USE_DEBUG_RNG macro and DRBG_GetEntropy() function.
59 #ifdef MSC_VER
60         srand((unsigned)_plat_RealTime() ^ _getpid());
61 #else
62         srand((unsigned)_plat_RealTime() ^ getpid());
63 #endif
64         lastEntropy = rand32();
65         ret = 0;
66     }
67     else
68     {
69         rndNum = rand32();
70         if(rndNum == lastEntropy)
71         {
72             ret = -1;
73         }
74         else
75         {
76             lastEntropy = rndNum;
77             // Each process will have its random number generator initialized
78             // according to the process id and the initialization time. This is not a
79             // lot of entropy so, to add a bit more, XOR the current time value into
80             // the returned entropy value.
81             // NOTE: the reason for including the time here rather than have it in
82             // in the value assigned to lastEntropy is that rand() could be broken and
83             // using the time would in the lastEntropy value would hide this.
84             rndNum ^= (uint32_t)_plat_RealTime();
85
86             // Only provide entropy 32 bits at a time to test the ability
87             // of the caller to deal with partial results.
88             ret = MIN(amount, sizeof(rndNum));
89             memcpy(entropy, &rndNum, ret);
90         }
91     }
92     return ret;

```

93 }

DRAFT

C.5 LocalityPlat.c

C.5.1. Includes

```
1  #include "Platform.h"
```

C.5.2. Functions

C.5.2.1. _plat__LocalityGet()

Get the most recent command locality in locality value form. This is an integer value for locality and not a locality structure. The locality can be 0-4 or 32-255. 5-31 is not allowed.

```
2  LIB_EXPORT unsigned char
3  _plat__LocalityGet(
4      void
5  )
6  {
7      return s_locality;
8  }
```

C.5.2.2. _plat__LocalitySet()

Set the most recent command locality in locality value form

```
9  LIB_EXPORT void
10 _plat__LocalitySet(
11     unsigned char    locality
12 )
13 {
14     if(locality > 4 && locality < 32)
15         locality = 0;
16     s_locality = locality;
17     return;
18 }
```

C.6 NVMem.c

C.6.1. Description

This file contains the NV read and write access methods. This implementation uses RAM/file and does not manage the RAM/file as NV blocks. The implementation may become more sophisticated over time.

C.6.2. Includes and Local

```

1  #include <memory.h>
2  #include <string.h>
3  #include <assert.h>
4  #include "Platform.h"
5  #if FILE_BACKED_NV
6  #   include <stdio.h>
7  static FILE      *s_NvFile = NULL;
8  static int       s_NeedsManufacture = FALSE;
9  #endif

```

C.6.3. Functions

```

10 #if FILE_BACKED_NV

```

C.6.3.1. NvFileOpen()

This function opens the file used to hold the NV image.

Return Value	Meaning
≥ 0	success
-1	error

```

11 static int
12 NvFileOpen(
13     const char    *mode
14 )
15 {
16 #if defined(NV_FILE_PATH)
17 #   define TO_STRING(s) TO_STRING_IMPL(s)
18 #   define TO_STRING_IMPL(s) #s
19     const char* s_NvFilePath = TO_STRING(NV_FILE_PATH);
20 #   undef TO_STRING
21 #   undef TO_STRING_IMPL
22 #else
23     const char* s_NvFilePath = "NVChip";
24 #endif
25
26     // Try to open an exist NVChip file for read/write
27 #   if defined _MSC_VER && 1
28     if(fopen_s(&s_NvFile, s_NvFilePath, mode) != 0)
29         s_NvFile = NULL;
30 #   else
31     s_NvFile = fopen(s_NvFilePath, mode);
32 #   endif
33     return (s_NvFile == NULL) ? -1 : 0;
34 }

```

C.6.3.2. NvFileCommit()

Write all of the contents of the NV image to a file.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

35  static int
36  NvFileCommit(
37      void
38  )
39  {
40      int      OK;
41      // If NV file is not available, return failure
42      if(s_NvFile == NULL)
43          return 1;
44      // Write RAM data to NV
45      fseek(s_NvFile, 0, SEEK_SET);
46      OK = (NV_MEMORY_SIZE == fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NvFile));
47      OK = OK && (0 == fflush(s_NvFile));
48      assert(OK);
49      return OK;
50  }

```

C.6.3.3. NvFileSize()

This function gets the size of the NV file and puts the file pointer where desired using the seek method values. SEEK_SET => beginning; SEEK_CUR => current position and SEEK_END => to the end of the file.

```

51  static long
52  NvFileSize(
53      int      leaveAt
54  )
55  {
56      long      fileSize;
57      long      filePos = ftell(s_NvFile);
58      //
59      assert(NULL != s_NvFile);
60
61      int fseek_result = fseek(s_NvFile, 0, SEEK_END);
62      NOT_REFERENCED(fseek_result); // Fix compiler warning for NDEBUD
63      assert(fseek_result == 0);
64      fileSize = ftell(s_NvFile);
65      assert(fileSize >= 0);
66      switch(leaveAt)
67      {
68          case SEEK_SET:
69              filePos = 0;
70          case SEEK_CUR:
71              fseek(s_NvFile, filePos, SEEK_SET);
72              break;
73          case SEEK_END:
74              break;
75          default:
76              assert(FALSE);
77              break;
78      }
79      return fileSize;
80  }

```

81 **#endif**

C.6.3.4. `_plat__NvErrors()`

This function is used by the simulator to set the error flags in the NV subsystem to simulate an error in the NV loading process

```

82  LIB_EXPORT void
83  _plat__NvErrors(
84      int             recoverable,
85      int             unrecoverable
86  )
87  {
88      s_NV_unrecoverable = unrecoverable;
89      s_NV_recoverable   = recoverable;
90  }
```

C.6.3.5. `_plat__NVEnable()`

Enable NV memory.

This version just pulls in data from a file. In a real TPM, with NV on chip, this function would verify the integrity of the saved context. If the NV memory was not on chip but was in something like RPMB, the NV state would be read in, decrypted and integrity checked.

The recovery from an integrity failure depends on where the error occurred. If it was in the state that is discarded by TPM Reset, then the error is recoverable if the TPM is reset. Otherwise, the TPM must go into failure mode.

Return Value	Meaning
0	if success
> 0	if receive recoverable error
<0	if unrecoverable error

```

91  LIB_EXPORT int
92  _plat__NVEnable(
93      void             *platParameter // IN: platform specific parameters
94  )
95  {
96      NOT_REFERENCED(platParameter); // to keep compiler quiet
97      //
98      // Start assuming everything is OK
99      s_NV_unrecoverable = FALSE;
100     s_NV_recoverable   = FALSE;
101     #if FILE_BACKED_NV
102     if(s_NvFile != NULL)
103         return 0;
104     // Initialize all the bytes in the ram copy of the NV
105     _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
106     // If the file exists
107     if(NvFileOpen("r+b") >= 0)
108     {
109         long     fileSize = NvFileSize(SEEK_SET); // get the file size and leave the
110                                                     // file pointer at the start
111     }
112     //
113     // If the size is right, read the data
114     if (NV_MEMORY_SIZE == fileSize)
115     {
116         s_NeedsManufacture =
```

```

117         fread(s_NV, 1, NV_MEMORY_SIZE, s_NvFile) != NV_MEMORY_SIZE;
118     }
119     else
120     {
121         NvFileCommit();    // for any other size, initialize it
122         s_NeedsManufacture = TRUE;
123     }
124 }
125 // If NVChip file does not exist, try to create it for read/write.
126 else if(NvFileOpen("w+b") >= 0)
127 {
128     NvFileCommit();        // Initialize the file
129     s_NeedsManufacture = TRUE;
130 }
131 assert(NULL != s_NvFile);    // Just in case we are broken for some reason.
132 #endif
133 // NV contents have been initialized and the error checks have been performed. For
134 // simulation purposes, use the signaling interface to indicate if an error is
135 // to be simulated and the type of the error.
136 if(s_NV_unrecoverable)
137     return -1;
138 return s_NV_recoverable;
139 }

```

C.6.3.6. _plat__NVDisable()

Disable NV memory

```

140 LIB_EXPORT void
141 _plat__NVDisable(
142     int delete    // IN: If TRUE, delete the NV contents.
143 )
144 {
145     #if FILE_BACKED_NV
146     if(NULL != s_NvFile)
147     {
148         fclose(s_NvFile);    // Close NV file
149         // Alternative to deleting the file is to set its size to 0. This will not
150         // match the NV size so the TPM will need to be remanufactured.
151         if(delete)
152         {
153             // Open for writing at the start. Sets the size to zero.
154             if(NvFileOpen("w") >= 0)
155             {
156                 fflush(s_NvFile);
157                 fclose(s_NvFile);
158             }
159         }
160     }
161     s_NvFile = NULL;    // Set file handle to NULL
162 #endif
163     return;
164 }

```

C.6.3.7. _plat__IsNvAvailable()

Check if NV is available

Return Value	Meaning
0	NV is available
1	NV is not available due to write failure
2	NV is not available due to rate limit

```

165 LIB_EXPORT int
166 _plat__IsNvAvailable(
167     void
168 )
169 {
170     int         retVal = 0;
171     // NV is not available if the TPM is in failure mode
172     if(!s_NvIsAvailable)
173         retVal = 1;
174 #if FILE_BACKED_NV
175     else
176         retVal = (s_NvFile == NULL);
177 #endif
178     return retVal;
179 }

```

C.6.3.8. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```

180 LIB_EXPORT void
181 _plat__NvMemoryRead(
182     unsigned int    startOffset,    // IN: read start
183     unsigned int    size,           // IN: size of bytes to read
184     void            *data           // OUT: data buffer
185 )
186 {
187     assert(startOffset + size <= NV_MEMORY_SIZE);
188     memcpy(data, &s_NV[startOffset], size);    // Copy data from RAM
189     return;
190 }

```

C.6.3.9. _plat__NvIsDifferent()

This function checks to see if the NV is different from the test value. This is so that NV will not be written if it has not changed.

Return Value	Meaning
TRUE(1)	the NV location is different from the test value
FALSE(0)	the NV location is the same as the test value

```

191 LIB_EXPORT int
192 _plat__NvIsDifferent(
193     unsigned int    startOffset,    // IN: read start
194     unsigned int    size,           // IN: size of bytes to read
195     void            *data           // IN: data buffer
196 )
197 {
198     return (memcmp(&s_NV[startOffset], data, size) != 0);
199 }

```

C.6.3.10. _plat__NvMemoryWrite()

This function is used to update NV memory. The **write** is to a memory copy of NV. At the end of the current command, any changes are written to the actual NV memory.

NOTE A useful optimization would be for this code to compare the current contents of NV with the local copy and note the blocks that have changed. Then only write those blocks when _plat__NvCommit() is called.

```

200  LIB_EXPORT int
201  _plat__NvMemoryWrite(
202      unsigned int    startOffset,    // IN: write start
203      unsigned int    size,           // IN: size of bytes to write
204      void            *data           // OUT: data buffer
205  )
206  {
207      if(startOffset + size <= NV_MEMORY_SIZE)
208      {
209          memcpy(&s_NV[startOffset], data, size);    // Copy the data to the NV image
210          return TRUE;
211      }
212      return FALSE;
213  }

```

C.6.3.11. _plat__NvMemoryClear()

Function is used to set a range of NV memory bytes to an implementation-dependent value. The value represents the erase state of the memory.

```

214  LIB_EXPORT void
215  _plat__NvMemoryClear(
216      unsigned int    start,           // IN: clear start
217      unsigned int    size             // IN: number of bytes to clear
218  )
219  {
220      assert(start + size <= NV_MEMORY_SIZE);
221      // In this implementation, assume that the erase value for NV is all 1s
222      memset(&s_NV[start], 0xff, size);
223  }

```

C.6.3.12. _plat__NvMemoryMove()

Function: Move a chunk of NV memory from source to destination This function should ensure that if there overlap, the original data is copied before it is written

```

224  LIB_EXPORT void
225  _plat__NvMemoryMove(
226      unsigned int    sourceOffset,    // IN: source offset
227      unsigned int    destOffset,      // IN: destination offset
228      unsigned int    size             // IN: size of data being moved
229  )
230  {
231      assert(sourceOffset + size <= NV_MEMORY_SIZE);
232      assert(destOffset + size <= NV_MEMORY_SIZE);
233      memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);    // Move data in RAM
234      return;
235  }

```


C.6.3.13. _plat__NvCommit()

This function writes the local copy of NV to NV for permanent store. It will write NV_MEMORY_SIZE bytes to NV. If a file is use, the entire file is written.

Return Value	Meaning
0	NV write success
non-0	NV write fail

```

236 LIB_EXPORT int
237 _plat__NvCommit(
238     void
239 )
240 {
241     #if FILE_BACKED_NV
242         return (NvFileCommit() ? 0 : 1);
243     #else
244         return 0;
245     #endif
246 }
```

C.6.3.14. _plat__SetNvAvail()

Set the current NV state to available. This function is for testing purpose only. It is not part of the platform NV logic

```

247 LIB_EXPORT void
248 _plat__SetNvAvail(
249     void
250 )
251 {
252     s_NvIsAvailable = TRUE;
253     return;
254 }
```

C.6.3.15. _plat__ClearNvAvail()

Set the current NV state to unavailable. This function is for testing purpose only. It is not part of the platform NV logic

```

255 LIB_EXPORT void
256 _plat__ClearNvAvail(
257     void
258 )
259 {
260     s_NvIsAvailable = FALSE;
261     return;
262 }
```

C.6.3.16. _plat__NVNeedsManufacture()

This function is used by the simulator to determine when the TPM's NV state needs to be manufactured.

```

263 LIB_EXPORT int
264 _plat__NVNeedsManufacture(
265     void
266 )
267 {
```

```
268  #if FILE_BACKED_NV
269      return s_NeedsManufacture;
270  #else
271      return FALSE;
272  #endif
273  }
```

DRAFT

C.7 PowerPlat.c

C.7.1. Includes and Function Prototypes

```
1  #include    "Platform.h"
2  #include    "_TPM_Init_fp.h"
```

C.7.2. Functions

C.7.2.1. _plat__Signal_PowerOn()

Signal platform power on

```
3  LIB_EXPORT int
4  _plat__Signal_PowerOn(
5      void
6  )
7  {
8      // Reset the timer
9      _plat__TimerReset();
10
11     // Need to indicate that we lost power
12     s_powerLost = TRUE;
13
14     return 0;
15 }
```

C.7.2.2. _plat__WasPowerLost()

Test whether power was lost before a _TPM_Init.

This function will clear the **hardware** indication of power loss before return. This means that there can only be one spot in the TPM code where this value gets read. This method is used here as it is the most difficult to manage in the TPM code and, if the hardware actually works this way, it is hard to make it look like anything else. So, the burden is placed on the TPM code rather than the platform code

Return Value	Meaning
TRUE(1)	power was lost
FALSE(0)	power was not lost

```
16 LIB_EXPORT int
17 _plat__WasPowerLost(
18     void
19 )
20 {
21     int      retVal = s_powerLost;
22     s_powerLost = FALSE;
23     return retVal;
24 }
```

C.7.2.3. _plat__Signal_Reset()

This a TPM reset without a power loss.

```
25 LIB_EXPORT int
26 _plat__Signal_Reset(
```

```
27     void
28     )
29     {
30         // Initialize locality
31         s_locality = 0;
32
33         // Command cancel
34         s_isCanceled = FALSE;
35
36         _TPM_Init();
37
38         // if we are doing reset but did not have a power failure, then we should
39         // not need to reload NV ...
40
41         return 0;
42     }
```

C.7.2.4. _plat__Signal_PowerOff()

Signal platform power off

```
43 LIB_EXPORT void
44 _plat__Signal_PowerOff(
45     void
46     )
47     {
48         // Prepare NV memory for power off
49         _plat__NVDisable(0);
50
51         // Disable tick ACT tick processing
52         _plat__ACT_EnableTicks(FALSE);
53
54         return;
55     }
```

C.8 PlatformData.h

This file contains the instance data for the Platform module. It is collected in this file so that the state of the module is easier to manage.

```

1  #ifndef _PLATFORM_DATA_H_
2  #define _PLATFORM_DATA_H_
3
4  #ifdef _PLATFORM_DATA_C_
5  #define EXTERN
6  #else
7  #define EXTERN extern
8  #endif

```

From Cancel.c Cancel flag. It is initialized as FALSE, which indicate the command is not being canceled

```

9  EXTERN int      s_isCanceled;
10
11 #ifndef HARDWARE_CLOCK
12 typedef uint64_t  clock64_t;

```

This is the value returned the last time that the system clock was read. This is only relevant for a simulator or virtual TPM.

```

13 EXTERN clock64_t      s_realTimePrevious;

```

These values are used to try to synthesize a long lived version of clock().

```

14 EXTERN clock64_t      s_lastSystemTime;
15 EXTERN clock64_t      s_lastReportedTime;

```

This is the rate adjusted value that is the equivalent of what would be read from a hardware register that produced rate adjusted time.

```

16 EXTERN clock64_t      s_tpmTime;
17 #endif // HARDWARE_CLOCK

```

This value indicates that the timer was reset

```

18 EXTERN int      s_timerReset;

```

This value indicates that the timer was stopped. It causes a clock discontinuity.

```

19 EXTERN int      s_timerStopped;

```

This variable records the time when _plat__TimerReset() is called. This mechanism allow us to subtract the time when TPM is power off from the total time reported by clock() function

```

20 EXTERN uint64_t      s_initClock;

```

This variable records the timer adjustment factor.

```

21 EXTERN unsigned int      s_adjustRate;

```

For LocalityPlat.c Locality of current command

```

22 EXTERN unsigned char s_locality;

```

For NVMem.c Choose if the NV memory should be backed by RAM or by file. If this macro is defined, then a file is used as NV. If it is not defined, then RAM is used to back NV memory. Comment out to use RAM.

```

23  #if (!defined VTPM) || ((VTPM != NO) && (VTPM != YES))
24  #   undef VTPM
25  #   define VTPM YES // Default: Either YES or NO
26  #endif

```

For a simulation, use a file to back up the NV

```

27  #if (!defined FILE_BACKED_NV) || ((FILE_BACKED_NV != NO) && (FILE_BACKED_NV != YES))
28  #   undef FILE_BACKED_NV
29  #   define FILE_BACKED_NV (VTPM && YES) // Default: Either YES or NO
30  #endif
31
32  #if SIMULATION
33  #   undef FILE_BACKED_NV
34  #   define FILE_BACKED_NV YES
35  #endif // SIMULATION
36
37  EXTERN unsigned char s_NV[NV_MEMORY_SIZE];
38  EXTERN int s_NvIsAvailable;
39  EXTERN int s_NV_unrecoverable;
40  EXTERN int s_NV_recoverable;

```

For PPPlat.c Physical presence. It is initialized to FALSE

```

41  EXTERN int s_physicalPresence;

```

From Power

```

42  EXTERN int s_powerLost;

```

For Entropy.c

```

43  EXTERN uint32_t lastEntropy;
44
45  #define DEFINE_ACT(N) EXTERN ACT_DATA ACT_##N;
46  FOR_EACH_ACT(DEFINE_ACT)
47
48  EXTERN int actTicksAllowed;
49
50  #endif // _PLATFORM_DATA_H_

```

C.9 PlatformData.c

C.9.1. Description

This file will instance the TPM variables that are not stack allocated. The descriptions for these variables are in Global.h for this project.

C.9.2. Includes

```
1 #define _PLATFORM_DATA_C_  
2 #include "Platform.h"
```

DRAFT

C.10 PPPlat.c

C.10.1. Description

This module simulates the physical presence interface pins on the TPM.

C.10.2. Includes

```
1  #include "Platform.h"
```

C.10.3. Functions

C.10.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

Return Value	Meaning
TRUE(1)	if physical presence is signaled
FALSE(0)	if physical presence is not signaled

```
2  LIB_EXPORT int
3  _plat__PhysicalPresenceAsserted(
4      void
5  )
6  {
7      // Do not know how to check physical presence without real hardware.
8      // so always return TRUE;
9      return s_physicalPresence;
10 }
```

C.10.3.2. _plat__Signal_PhysicalPresenceOn()

Signal physical presence on

```
11 LIB_EXPORT void
12 _plat__Signal_PhysicalPresenceOn(
13     void
14 )
15 {
16     s_physicalPresence = TRUE;
17     return;
18 }
```

C.10.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
19 LIB_EXPORT void
20 _plat__Signal_PhysicalPresenceOff(
21     void
22 )
23 {
24     s_physicalPresence = FALSE;
25     return;
26 }
```

C.11 RunCommand.c

C.11.1. Introduction

This module provides the platform specific entry and fail processing. The `_plat__RunCommand()` function is used to call to `ExecuteCommand()` in the TPM code. This function does whatever processing is necessary to set up the platform in anticipation of the call to the TPM including setup for error processing.

The `_plat__Fail()` function is called when there is a failure in the TPM. The TPM code will have set the flag to indicate that the TPM is in failure mode. This call will then recursively call `ExecuteCommand()` in order to build the failure mode response. When `ExecuteCommand()` returns to `_plat__Fail()`, the platform will do some platform specific operation to return to the environment in which the TPM is executing. For a simulator, `setjmp/longjmp` is used. For an OS, a system exit to the OS would be appropriate.

C.11.2. Includes and locals

```
1  #include "Platform.h"
2  #include <setjmp.h>
3  #include "ExecCommand_fp.h"
4
5  jmp_buf      s_jumpBuffer;
```

C.11.3. Functions

C.11.3.1. `_plat__RunCommand()`

This version of `RunCommand()` will set up a `jmp_buf` and call `ExecuteCommand()`. If the command executes without failing, it will return and `RunCommand()` will return. If there is a failure in the command, then `_plat__Fail()` is called and it will `longjmp` back to `RunCommand()` which will call `ExecuteCommand()` again. However, this time, the TPM will be in failure mode so `ExecuteCommand()` will simply build a failure response and return.

```
6  LIB_EXPORT void
7  _plat__RunCommand(
8      uint32_t      requestSize,    // IN: command buffer size
9      unsigned char *request,       // IN: command buffer
10     uint32_t      *responseSize,   // IN/OUT: response buffer size
11     unsigned char **response      // IN/OUT: response buffer
12 )
13 {
14     setjmp(s_jumpBuffer);
15     ExecuteCommand(requestSize, request, responseSize, response);
16 }
```

C.11.3.2. `_plat__Fail()`

This is the platform depended failure exit for the TPM.

```
17 LIB_EXPORT NORETURN void
18 _plat__Fail(
19     void
20 )
21 {
22     longjmp(&s_jumpBuffer[0], 1);
23 }
```

C.12 Unique.c

C.12.1. Introduction

In some implementations of the TPM, the hardware can provide a secret value to the TPM. This secret value is statistically unique to the instance of the TPM. Typical uses of this value are to provide personalization to the random number generation and as a shared secret between the TPM and the manufacturer.

C.12.2. Includes

```

1  #include "Platform.h"
2
3  const char notReallyUnique[] =
4  "This is not really a unique value. A real unique value should"
5  " be generated by the platform.";

```

C.12.3. _plat__GetUnique()

This function is used to access the platform-specific unique value. This function places the unique value in the provided buffer (*b*) and returns the number of bytes transferred. The function will not copy more data than *bSize*.

NOTE If a platform unique value has unequal distribution of uniqueness and *bSize* is smaller than the size of the unique value, the *bSize* portion with the most uniqueness should be returned.

```

6  LIB_EXPORT uint32_t
7  _plat__GetUnique(
8      uint32_t which,           // authorities (0) or details
9      uint32_t bSize,          // size of the buffer
10     unsigned char *b          // output buffer
11 )
12 {
13     const char *from = notReallyUnique;
14     uint32_t retVal = 0;
15
16     if(which == 0) // the authorities value
17     {
18         for(retVal = 0;
19             *from != 0 && retVal < bSize;
20             retVal++)
21         {
22             *b++ = *from++;
23         }
24     }
25     else
26     {
27         #define uSize sizeof(notReallyUnique)
28         b = &b[(bSize < uSize) ? bSize : uSize] - 1;
29         for(retVal = 0;
30             *from != 0 && retVal < bSize;
31             retVal++)
32         {
33             *b-- = *from++;
34         }
35     }
36     return retVal;
37 }

```

C.13 DebugHelpers.c

C.13.1. Description

This file contains the NV read and write access methods. This implementation uses RAM/file and does not manage the RAM/file as NV blocks. The implementation may become more sophisticated over time.

C.13.2. Includes and Local

```

1  #include <stdio.h>
2  #include <time.h>
3  #include "Platform.h"
4
5  #if CERTIFYX509_DEBUG
6
7  const char      *debugFileName = "DebugFile.txt";

```

C.13.2.1. fileOpen()

This exists to allow use of the *safe* version of fopen() with a MS runtime.

```

8  static FILE *
9  fileOpen(
10     const char      *fn,
11     const char      *mode
12 )
13 {
14     FILE      *f;
15     # if defined _MSC_VER
16     if(fopen_s(&f, fn, mode) != 0)
17         f = NULL;
18     # else
19     f = fopen(fn, mode);
20     # endif
21     return f;
22 }

```

C.13.2.2. DebugFileInit()

This function initializes the file containing the debug data with the time of the file creation.

Return Value	Meaning
0	success
!= 0	error

```

23  int
24  DebugFileInit(
25     void
26 )
27 {
28     FILE      *f = NULL;
29     time_t      t = time(NULL);
30     //
31     // Get current date and time.
32     # if defined _MSC_VER
33     char      timeString[100];
34     ctime_s(timeString, (size_t)sizeof(timeString), &t);
35     # else

```

```

36     char                *timeString;
37     timeString = ctime(&t);
38 #   endif
39     // Try to open the debug file
40     f = fopen(debugFileName, "w");
41     if(f)
42     {
43         // Initialize the contents with the time.
44         fprintf(f, "%s\n", timeString);
45         fclose(f);
46         return 0;
47     }
48     return -1;
49 }

```

C.13.2.3. DebugDumpBuffer()

```

50 void
51 DebugDumpBuffer(
52     int                size,
53     unsigned char      *buf,
54     const char         *identifier
55 )
56 {
57     int                i;
58     //
59     FILE *f = fopen(debugFileName, "a");
60     if(!f)
61         return;
62     if(identifier)
63         fprintf(f, "%s\n", identifier);
64     if(buf)
65     {
66         for(i = 0; i < size; i++)
67         {
68             if(((i % 16) == 0) && (i))
69                 fprintf(f, "\n");
70             fprintf(f, " %02X", buf[i]);
71         }
72         if((size % 16) != 0)
73             fprintf(f, "\n");
74     }
75     fclose(f);
76 }
77 #endif // CERTIFYX509_DEBUG

```

C.14 Platform.h

```
1  #ifndef    _PLATFORM_H_
2  #define    _PLATFORM_H_
3
4  #include "TpmBuildSwitches.h"
5  #include "BaseTypes.h"
6  #include "TPMB.h"
7  #include "MinMax.h"
8
9  #include "TpmProfile.h"
10
11 #include "PlatformACT.h"
12 #include "PlatformClock.h"
13 #include "PlatformData.h"
14 #include "Platform_fp.h"
15
16 #endif    // _PLATFORM_H_
```

C.15 PlatformACT.h

This file contains the definitions for the ACT macros and data types used in the ACT implementation.

```

1  #ifndef _PLATFORM_ACT_H_
2  #define _PLATFORM_ACT_H_
3
4  typedef struct ACT_DATA
5  {
6      uint32_t      remaining;
7      uint32_t      newValue;
8      uint8_t       signaled;
9      uint8_t       pending;
10     uint8_t        number;
11 } ACT_DATA, *P_ACT_DATA;
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 #define FOR_EACH_ACT(op)
127     IF_ACT_0_IMPLEMENTED(op)

```

```
128     IF_ACT_1_IMPLEMENTED (op)           \  
129     IF_ACT_2_IMPLEMENTED (op)           \  
130     IF_ACT_3_IMPLEMENTED (op)           \  
131     IF_ACT_4_IMPLEMENTED (op)           \  
132     IF_ACT_5_IMPLEMENTED (op)           \  
133     IF_ACT_6_IMPLEMENTED (op)           \  
134     IF_ACT_7_IMPLEMENTED (op)           \  
135     IF_ACT_8_IMPLEMENTED (op)           \  
136     IF_ACT_9_IMPLEMENTED (op)           \  
137     IF_ACT_A_IMPLEMENTED (op)           \  
138     IF_ACT_B_IMPLEMENTED (op)           \  
139     IF_ACT_C_IMPLEMENTED (op)           \  
140     IF_ACT_D_IMPLEMENTED (op)           \  
141     IF_ACT_E_IMPLEMENTED (op)           \  
142     IF_ACT_F_IMPLEMENTED (op)           \  
143  
144     #endif // _PLATFORM_ACT_H_
```

C.16 PlatformACT.c

C.16.1. Includes

```
1 #include "Platform.h"
```

C.16.2. Functions

C.16.2.1. ActSignal()

Function called when there is an ACT event to signal or unsignal

```
2 static void
3 ActSignal(
4     P_ACT_DATA      actData,
5     int              on
6 )
7 {
8     if(actData == NULL)
9         return;
10    // If this is to turn a signal on, don't do anything if it is already on. If this
11    // is to turn the signal off, do it anyway because this might be for
12    // initialization.
13    if(on && (actData->signaled == TRUE))
14        return;
15    actData->signaled = (uint8_t)on;
16
17    // If there is an action, then replace the "Do something" with the correct action.
18    // It should test 'on' to see if it is turning the signal on or off.
19    switch(actData->number)
20    {
21    #if RH_ACT_0
22        case 0: // Do something
23            return;
24    #endif
25    #if RH_ACT_1
26        case 1: // Do something
27            return;
28    #endif
29    #if RH_ACT_2
30        case 2: // Do something
31            return;
32    #endif
33    #if RH_ACT_3
34        case 3: // Do something
35            return;
36    #endif
37    #if RH_ACT_4
38        case 4: // Do something
39            return;
40    #endif
41    #if RH_ACT_5
42        case 5: // Do something
43            return;
44    #endif
45    #if RH_ACT_6
46        case 6: // Do something
47            return;
48    #endif
49    #if RH_ACT_7
50        case 7: // Do something
51            return;
```

```

52 #endif
53 #if RH_ACT_8
54     case 8: // Do something
55         return;
56 #endif
57 #if RH_ACT_9
58     case 9: // Do something
59         return;
60 #endif
61 #if RH_ACT_A
62     case 0xA: // Do something
63         return;
64 #endif
65 #if RH_ACT_B
66     case 0xB:
67         // Do something
68         return;
69 #endif
70 #if RH_ACT_C
71     case 0xC: // Do something
72         return;
73 #endif
74 #if RH_ACT_D
75     case 0xD: // Do something
76         return;
77 #endif
78 #if RH_ACT_E
79     case 0xE: // Do something
80         return;
81 #endif
82 #if RH_ACT_F
83     case 0xF: // Do something
84         return;
85 #endif
86     default:
87         return;
88 }
89 }

```

C.16.2.2. ActGetDataPointer()

```

90 static P_ACT_DATA
91 ActGetDataPointer(
92     uint32_t      act
93 )
94 {
95
96 #define RETURN_ACT_POINTER(N) if(0x##N == act) return &ACT_##N;
97
98     FOR_EACH_ACT(RETURN_ACT_POINTER)
99
100     return (P_ACT_DATA) NULL;
101 }

```

C.16.2.3. _plat__ACT_GetImplemented()

This function tests to see if an ACT is implemented. It is a belt and suspenders function because the TPM should not be calling to manipulate an ACT that is not implemented. However, this could help the simulator code which doesn't necessarily know if an ACT is implemented or not.

```

102 LIB_EXPORT int
103 _plat__ACT_GetImplemented(
104     uint32_t      act

```

```

105 )
106 {
107     return (ActGetDataPointer(act) != NULL);
108 }

```

C.16.2.4. _plat__ACT_GetRemaining()

This function returns the remaining time. If an update is pending, *newValue* is returned. Otherwise, the current counter value is returned. Note that since the timers keep running, the returned value can get stale immediately. The actual count value will be no greater than the returned value.

```

109 LIB_EXPORT uint32_t
110 _plat__ACT_GetRemaining(
111     uint32_t      act           //IN: the ACT selector
112 )
113 {
114     P_ACT_DATA    actData = ActGetDataPointer(act);
115     uint32_t      remain;
116     //
117     if(actData == NULL)
118         return 0;
119     remain = actData->remaining;
120     if(actData->pending)
121         remain = actData->newValue;
122     return remain;
123 }

```

C.16.2.5. _plat__ACT_GetSignaled()

```

124 LIB_EXPORT int
125 _plat__ACT_GetSignaled(
126     uint32_t      act           //IN: number of ACT to check
127 )
128 {
129     P_ACT_DATA    actData = ActGetDataPointer(act);
130     //
131     if(actData == NULL)
132         return 0;
133     return (int) actData->signaled;
134 }

```

C.16.2.6. _plat__ACT_SetSignaled()

```

135 LIB_EXPORT void
136 _plat__ACT_SetSignaled(
137     uint32_t      act,
138     int           on
139 )
140 {
141     ActSignal(ActGetDataPointer(act), on);
142 }

```

C.16.2.7. _plat__ACT_GetPending()

```

143 LIB_EXPORT int
144 _plat__ACT_GetPending(
145     uint32_t      act           //IN: number of ACT to check
146 )
147 {
148     P_ACT_DATA    actData = ActGetDataPointer(act);

```

```

149 //
150 if(actData == NULL)
151     return 0;
152 return (int )actData->pending;
153 }

```

C.16.2.8. _plat__ACT_UpdateCounter()

This function is used to write the *newValue* for the counter. If an update is pending, then no update occurs and the function returns FALSE. If *setSignaled* is TRUE, then the ACT signaled state is SET and if *newValue* is 0, nothing is posted.

```

154 LIB_EXPORT int
155 _plat__ACT_UpdateCounter(
156     uint32_t      act,          // IN: ACT to update
157     uint32_t      newValue     // IN: the value to post
158 )
159 {
160     P_ACT_DATA    actData = ActGetDataPointer(act);
161     //
162     if(actData == NULL)
163         // actData doesn't exist but pretend update is pending rather than indicate
164         // that a retry is necessary.
165         return TRUE;
166     // if an update is pending then return FALSE so that there will be a retry
167     if(actData->pending != 0)
168         return FALSE;
169     actData->newValue = newValue;
170     actData->pending = TRUE;
171
172     return TRUE;
173 }

```

C.16.2.9. _plat__ACT_EnableTicks()

This enables and disables the processing of the once-per-second ticks. This should be turned off (*enable* = FALSE) by *_TPM_Init* and turned on (*enable* = TRUE) by *TPM2_Startup()* after all the initializations have completed.

```

174 LIB_EXPORT void
175 _plat__ACT_EnableTicks(
176     int          enable
177 )
178 {
179     actTicksAllowed = enable;
180 }

```

C.16.2.10. ActDecrement()

If *newValue* is non-zero it is copied to *remaining* and then *newValue* is set to zero. Then *remaining* is decremented by one if it is not already zero. If the value is decremented to zero, then the associated event is signaled. If setting *remaining* causes it to be greater than 1, then the signal associated with the ACT is turned off.

```

181 static void
182 ActDecrement(
183     P_ACT_DATA    actData
184 )
185 {
186     // Check to see if there is an update pending

```

```

187     if(actData->pending)
188     {
189         // If this update will cause the count to go from non-zero to zero, set
190         // the newValue to 1 so that it will timeout when decremented below.
191         if((actData->newValue == 0) && (actData->remaining != 0))
192             actData->newValue = 1;
193         actData->remaining = actData->newValue;
194
195         // Update processed
196         actData->pending = 0;
197     }
198     // no update so countdown if the count is non-zero but not max
199     if((actData->remaining != 0) && (actData->remaining != UINT32_MAX))
200     {
201         // If this countdown causes the count to go to zero, then turn the signal for
202         // the ACT on.
203         if((actData->remaining -= 1) == 0)
204             ActSignal(actData, TRUE);
205     }
206     // If the current value of the counter is non-zero, then the signal should be
207     // off.
208     if(actData->signaled && (actData->remaining > 0))
209         ActSignal(actData, FALSE);
210 }

```

C.16.2.11. _plat__ACT_Tick()

This processes the once-per-second clock tick from the hardware. This is set up for the simulator to use the control interface to send ticks to the TPM. These ticks do not have to be on a per second basis. They can be as slow or as fast as desired so that the simulation can be tested.

```

211 LIB_EXPORT void
212 _plat__ACT_Tick(
213     void
214 )
215 {
216     // Ticks processing is turned off at certain times just to make sure that nothing
217     // strange is happening before pointers and things are
218     if(actTicksAllowed)
219     {
220         // Handle the update for each counter.
221         #define DECREMENT_COUNT(N)    ActDecrement(&ACT_##N);
222
223         FOR_EACH_ACT(DECREMENT_COUNT)
224     }
225 }

```

C.16.2.12. ActZero()

This function initializes a single ACT

```

226 static void
227 ActZero(
228     uint32_t      act,
229     P_ACT_DATA    actData
230 )
231 {
232     actData->remaining = 0;
233     actData->newValue = 0;
234     actData->pending = 0;
235     actData->number = (uint8_t)act;
236     ActSignal(actData, FALSE);

```


237 }

C.16.2.13. _plat__ACT_Initialize()

This function initializes the ACT hardware and data structures

```
238 LIB_EXPORT int
239 _plat__ACT_Initialize(
240     void
241 )
242 {
243     actTicksAllowed = 0;
244 #define ZERO_ACT(N)  ActZero(0x##N, &ACT_##N);
245     FOR_EACH_ACT(ZERO_ACT)
246
247     return TRUE;
248 }
```

C.17 PlatformClock.h

This file contains the instance data for the Platform module. It is collected in this file so that the state of the module is easier to manage.

```

1  #ifndef _PLATFORM_CLOCK_H_
2  #define _PLATFORM_CLOCK_H_
3
4  #ifndef _MSC_VER
5  #include <sys/types.h>
6  #include <sys/timeb.h>
7  #else
8  #include <sys/time.h>
9  #include <time.h>
10 #endif

```

CLOCK_NOMINAL is the number of hardware ticks per *mS*. A value of 300000 means that the nominal clock rate used to drive the hardware clock is 30 MHz. The adjustment rates are used to determine the conversion of the hardware ticks to internal hardware clock value. In practice, we would expect that there would be a hardware register will accumulated *mS*. It would be incremented by the output of a pre-scaler. The pre-scaler would divide the ticks from the clock by some value that would compensate for the difference between clock time and real time. The code in Clock does the emulation of this function.

```

11 #define      CLOCK_NOMINAL      30000

```

A 1% change in rate is 300 counts

```

12 #define      CLOCK_ADJUST_COARSE      300

```

A 0.1% change in rate is 30 counts

```

13 #define      CLOCK_ADJUST_MEDIUM      30

```

A minimum change in rate is 1 count

```

14 #define      CLOCK_ADJUST_FINE      1

```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16.7%)

```

15 #define      CLOCK_ADJUST_LIMIT      5000

```

```

16
17 #endif // _PLATFORM_CLOCK_H_

```

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.

DRAFT

D.2 TpmTcpProtocol.h

D.2.1. Introduction

TPM commands are communicated as uint8_t streams on a TCP connection. The TPM command protocol is enveloped with the interface protocol described in this file. The command is indicated by a uint32 with one of the values below. Most commands take no parameters return no TPM errors. In these cases the TPM interface protocol acknowledges that command processing is completed by returning a uint32=0. The command TPM_SIGNAL_HASH_DATA takes a uint32-prepended variable length byte array and the interface protocol acknowledges command completion with a uint32=0. Most TPM commands are enveloped using the TPM_SEND_COMMAND interface command. The parameters are as indicated below. The interface layer also appends a UIN32=0 to the TPM response for regularity.

D.2.2. Typedefs and Defines

```
1  #ifndef      TCP_TPM_PROTOCOL_H
2  #define      TCP_TPM_PROTOCOL_H
```

D.2.3. TPM Commands.

All commands acknowledge processing by returning a uint32 == 0 except where noted

```
3  #define TPM_SIGNAL_POWER_ON          1
4  #define TPM_SIGNAL_POWER_OFF        2
5  #define TPM_SIGNAL_PHYS_PRESENCE_ON 3
6  #define TPM_SIGNAL_PHYS_PRESENCE_OFF 4
7  #define TPM_SIGNAL_HASH_START        5
8  #define TPM_SIGNAL_HASH_DATA         6
9  // {uint32_t BufferSize, uint8_t[BufferSize] Buffer}
10 #define TPM_SIGNAL_HASH_END          7
11 #define TPM_SEND_COMMAND              8
12 // {uint8_t Locality, uint32_t InBufferSize, uint8_t[InBufferSize] InBuffer} -
13 >
14 // {uint32_t OutBufferSize, uint8_t[OutBufferSize] OutBuffer}
15 #define TPM_SIGNAL_CANCEL_ON          9
16 #define TPM_SIGNAL_CANCEL_OFF        10
17 #define TPM_SIGNAL_NV_ON             11
18 #define TPM_SIGNAL_NV_OFF            12
19 #define TPM_SIGNAL_KEY_CACHE_ON      13
20 #define TPM_SIGNAL_KEY_CACHE_OFF     14
21
22 #define TPM_REMOTE_HANDSHAKE          15
23 #define TPM_SET_ALTERNATIVE_RESULT    16
24
25 #define TPM_SIGNAL_RESET              17
26 #define TPM_SIGNAL_RESTART            18
27
28 #define TPM_SESSION_END               20
29 #define TPM_STOP                      21
30
31 #define TPM_GET_COMMAND_RESPONSE_SIZES 25
32
33 #define TPM_ACT_GET_SIGNED            26
34
35 #define TPM_TEST_FAILURE_MODE         30
```

D.2.4. Enumerations and Structures

```
35  enum TpmEndPointInfo
```

```
36 {
37     tpmPlatformAvailable = 0x01,
38     tpmUsesTbs = 0x02,
39     tpmInRawMode = 0x04,
40     tpmSupportsPP = 0x08
41 };
42
43 #ifdef _MSC_VER
44 #   pragma warning(push, 3)
45 #endif
```

Existing RPC interface type definitions retained so that the implementation can be re-used

```
46 typedef struct in_buffer
47 {
48     unsigned long BufferSize;
49     unsigned char *Buffer;
50 } _IN_BUFFER;
51
52 typedef unsigned char *_OUTPUT_BUFFER;
53
54 typedef struct out_buffer
55 {
56     uint32_t BufferSize;
57     _OUTPUT_BUFFER Buffer;
58 } _OUT_BUFFER;
59
60 #ifdef _MSC_VER
61 #   pragma warning(pop)
62 #endif
63
64 #ifndef WIN32
65 typedef unsigned long DWORD;
66 typedef void *LPVOID;
67 #endif
68
69 #endif
```

D.3 TcpServer.c

D.3.1. Description

This file contains the socket interface to a TPM simulator.

D.3.2. Includes, Locals, Defines and Function Prototypes

```

1  #include "TpmBuildSwitches.h"
2  #include <stdio.h>
3  #include <stdbool.h>
4
5  #ifdef _MSC_VER
6  #   pragma warning(push, 3)
7  #   include <windows.h>
8  #   include <winsock.h>
9  #   pragma warning(pop)
10  typedef int socklen_t;
11 #elif defined(__unix__)
12 #   include <string.h>
13 #   include <unistd.h>
14 #   include <errno.h>
15 #   include <stdint.h>
16 #   include <netinet/in.h>
17 #   include <sys/socket.h>
18 #   include <pthread.h>
19 #   define ZeroMemory(ptr, sz) (memset((ptr), 0, (sz)))
20 #   define closesocket(x) close(x)
21 #   define INVALID_SOCKET (-1)
22 #   define SOCKET_ERROR (-1)
23 #   define WSAGetLastError() (errno)
24 #   define INT_PTR intptr_t
25
26     typedef int SOCKET;
27 #else
28 #   error "Unsupported platform."
29 #endif
30
31 #include "TpmTcpProtocol.h"
32 #include "Manufacture_fp.h"
33 #include "TpmProfile.h"
34
35 #include "Simulator_fp.h"
36 #include "Platform_fp.h"

```

To access key cache control in TPM

```

37 void RsaKeyCacheControl(int state);
38
39 #ifndef __IGNORE_STATE__
40
41 static uint32_t ServerVersion = 1;
42
43 #define MAX_BUFFER 1048576
44 char InputBuffer[MAX_BUFFER];           //The input data buffer for the simulator.
45 char OutputBuffer[MAX_BUFFER];          //The output data buffer for the simulator.
46
47 struct
48 {
49     uint32_t    largestCommandSize;
50     uint32_t    largestCommand;
51     uint32_t    largestResponseSize;

```

```

52     uint32_t    largestResponse;
53 } CommandResponseSizes = {0};
54
55 #endif // __IGNORE_STATE__

```

D.3.3. Functions

D.3.3.1. CreateSocket()

This function creates a socket listening on *PortNumber*.

```

56 static int
57 CreateSocket(
58     int                PortNumber,
59     SOCKET             *listenSocket
60 )
61 {
62     struct             sockaddr_in MyAddress;
63     int res;
64 //
65 // Initialize Winsock
66 #ifndef _MSC_VER
67     WSADATA            wsaData;
68     res = WSASStartup(MAKEWORD(2, 2), &wsaData);
69     if(res != 0)
70     {
71         printf("WSASStartup failed with error: %d\n", res);
72         return -1;
73     }
74 #endif
75 // create listening socket
76 *listenSocket = socket(PF_INET, SOCK_STREAM, 0);
77 if(INVALID_SOCKET == *listenSocket)
78 {
79     printf("Cannot create server listen socket. Error is 0x%x\n",
80         WSAGetLastError());
81     return -1;
82 }
83 // bind the listening socket to the specified port
84 ZeroMemory(&MyAddress, sizeof(MyAddress));
85 MyAddress.sin_port = htons((short)PortNumber);
86 MyAddress.sin_family = AF_INET;
87
88 res = bind(*listenSocket, (struct sockaddr*) &MyAddress, sizeof(MyAddress));
89 if(res == SOCKET_ERROR)
90 {
91     printf("Bind error. Error is 0x%x\n", WSAGetLastError());
92     return -1;
93 }
94 // listen/wait for server connections
95 res = listen(*listenSocket, 3);
96 if(res == SOCKET_ERROR)
97 {
98     printf("Listen error. Error is 0x%x\n", WSAGetLastError());
99     return -1;
100 }
101 return 0;
102 }

```

D.3.3.2. PlatformServer()

This function processes incoming platform requests.


```

103 bool
104 PlatformServer(
105     SOCKET          s
106 )
107 {
108     bool            OK = true;
109     uint32_t        Command;
110     //
111     for(;;)
112     {
113         OK = ReadBytes(s, (char*)&Command, 4);
114         // client disconnected (or other error). We stop processing this client
115         // and return to our caller who can stop the server or listen for another
116         // connection.
117         if(!OK)
118             return true;
119         Command = ntohl(Command);
120         switch(Command)
121         {
122             case TPM_SIGNAL_POWER_ON:
123                 _rpc_Signal_PowerOn(false);
124                 break;
125             case TPM_SIGNAL_POWER_OFF:
126                 _rpc_Signal_PowerOff();
127                 break;
128             case TPM_SIGNAL_RESET:
129                 _rpc_Signal_PowerOn(true);
130                 break;
131             case TPM_SIGNAL_RESTART:
132                 _rpc_Signal_Restart();
133                 break;
134             case TPM_SIGNAL_PHYS_PRESENCE_ON:
135                 _rpc_Signal_PhysicalPresenceOn();
136                 break;
137             case TPM_SIGNAL_PHYS_PRESENCE_OFF:
138                 _rpc_Signal_PhysicalPresenceOff();
139                 break;
140             case TPM_SIGNAL_CANCEL_ON:
141                 _rpc_Signal_CancelOn();
142                 break;
143             case TPM_SIGNAL_CANCEL_OFF:
144                 _rpc_Signal_CancelOff();
145                 break;
146             case TPM_SIGNAL_NV_ON:
147                 _rpc_Signal_NvOn();
148                 break;
149             case TPM_SIGNAL_NV_OFF:
150                 _rpc_Signal_NvOff();
151                 break;
152             case TPM_SIGNAL_KEY_CACHE_ON:
153                 _rpc_RsaKeyCacheControl(true);
154                 break;
155             case TPM_SIGNAL_KEY_CACHE_OFF:
156                 _rpc_RsaKeyCacheControl(false);
157                 break;
158             case TPM_SESSION_END:
159                 // Client signaled end-of-session
160                 TpmEndSimulation();
161                 return true;
162             case TPM_STOP:
163                 // Client requested the simulator to exit
164                 return false;
165             case TPM_TEST_FAILURE_MODE:
166                 _rpc_ForceFailureMode();
167                 break;
168             case TPM_GET_COMMAND_RESPONSE_SIZES:

```

```

169         OK = WriteVarBytes(s, (char *)&CommandResponseSizes,
170                             sizeof(CommandResponseSizes));
171         memset(&CommandResponseSizes, 0, sizeof(CommandResponseSizes));
172         if(!OK)
173             return true;
174         break;
175     case TPM_ACT_GET_SIGNED:
176     {
177         uint32_t actHandle;
178         OK = ReadUINT32(s, &actHandle);
179         WriteUINT32(s, _rpc__ACT_GetSigned(actHandle));
180         break;
181     }
182     default:
183         printf("Unrecognized platform interface command %d\n",
184               (int)Command);
185         WriteUINT32(s, 1);
186         return true;
187     }
188     WriteUINT32(s, 0);
189 }
190 }

```

D.3.3.3. PlatformSvcRoutine()

This function is called to set up the socket interfaces to listen for commands.

```

191 DWORD WINAPI
192 PlatformSvcRoutine(
193     LPVOID          port
194 )
195 {
196     int                PortNumber = (int) (INT_PTR)port;
197     SOCKET             listenSocket, serverSocket;
198     struct             sockaddr_in HerAddress;
199     int                res;
200     socklen_t          length;
201     bool               continueServing;
202     //
203     res = CreateSocket(PortNumber, &listenSocket);
204     if(res != 0)
205     {
206         printf("Create platform service socket fail\n");
207         return res;
208     }
209     // Loop accepting connections one-by-one until we are killed or asked to stop
210     // Note the platform service is single-threaded so we don't listen for a new
211     // connection until the prior connection drops.
212     do
213     {
214         printf("Platform server listening on port %d\n", PortNumber);
215
216         // blocking accept
217         length = sizeof(HerAddress);
218         serverSocket = accept(listenSocket,
219                               (struct sockaddr*) &HerAddress,
220                               &length);
221         if(serverSocket == INVALID_SOCKET)
222         {
223             printf("Accept error. Error is 0x%x\n", WSAGetLastError());
224             return (DWORD)-1;
225         }
226         printf("Client accepted\n");
227     }

```

```

228         // normal behavior on client disconnection is to wait for a new client
229         // to connect
230         continueServing = PlatformServer(serverSocket);
231         closesocket(serverSocket);
232     } while(continueServing);
233
234     return 0;
235 }

```

D.3.3.4. PlatformSignalService()

This function starts a new thread waiting for platform signals. Platform signals are processed one at a time in the order in which they are received.

```

236 int
237 PlatformSignalService(
238     int          PortNumber
239 )
240 {
241     #if defined(_MSC_VER)
242         HANDLE          hPlatformSvc;
243         int              ThreadId;
244         int              port = PortNumber;
245         //
246         // Create service thread for platform signals
247         hPlatformSvc = CreateThread(NULL, 0,
248                                     (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
249                                     (LPVOID) (INT_PTR)port, 0, (LPDWORD)&ThreadId);
250         if(hPlatformSvc == NULL)
251         {
252             printf("Thread Creation failed\n");
253             return -1;
254         }
255         return 0;
256     #else
257         pthread_t        thread_id;
258         int              ret;
259         int              port = PortNumber;
260
261         ret = pthread_create(&thread_id, NULL, (void*)PlatformSvcRoutine,
262                             (LPVOID) (INT_PTR)port);
263         if (ret == -1)
264         {
265             printf("pthread_create failed: %s", strerror(ret));
266         }
267         return ret;
268     #endif // _MSC_VER
269 }

```

D.3.3.5. RegularCommandService()

This function services regular commands.

```

270 int
271 RegularCommandService(
272     int          PortNumber
273 )
274 {
275     SOCKET          listenSocket;
276     SOCKET          serverSocket;
277     struct          sockaddr_in HerAddress;
278     int             res;
279     socklen_t       length;

```

```

280     bool                continueServing;
281 //
282     res = CreateSocket(PortNumber, &listenSocket);
283     if(res != 0)
284     {
285         printf("Create platform service socket fail\n");
286         return res;
287     }
288     // Loop accepting connections one-by-one until we are killed or asked to stop
289     // Note the TPM command service is single-threaded so we don't listen for
290     // a new connection until the prior connection drops.
291     do
292     {
293         printf("TPM command server listening on port %d\n", PortNumber);
294
295         // blocking accept
296         length = sizeof(HeaderAddress);
297         serverSocket = accept(listenSocket,
298                               (struct sockaddr*) &HeaderAddress,
299                               &length);
300         if(serverSocket == INVALID_SOCKET)
301         {
302             printf("Accept error. Error is 0x%x\n", WSAGetLastError());
303             return -1;
304         }
305         printf("Client accepted\n");
306
307         // normal behavior on client disconnection is to wait for a new client
308         // to connect
309         continueServing = TpmServer(serverSocket);
310         closesocket(serverSocket);
311     } while(continueServing);
312     return 0;
313 }
314 #if RH_ACT_0

```

D.3.3.6. SimulatorTimeServiceRoutine()

This function is called to service the time *ticks*.

```

315 static unsigned long WINAPI
316 SimulatorTimeServiceRoutine(
317     LPVOID                notUsed
318 )
319 {
320     // All time is in ms
321     const int64_t         tick = 1000;
322     uint64_t               prevTime = _plat__RealTime();
323     int64_t                timeout = tick;
324
325     (void)notUsed;
326
327     while (true)
328     {
329         uint64_t curTime;
330
331         #if defined(_MSC_VER)
332             Sleep((DWORD)timeout);
333         #else
334             struct timespec req = { timeout / 1000, (timeout % 1000) * 1000 };
335             struct timespec rem;
336             nanosleep(&req, &rem);
337         #endif // _MSC_VER
338         curTime = _plat__RealTime();

```

```

339
340 // May need to issue several ticks if the Sleep() took longer than asked,
341 // or no ticks at all, it Sleep() was interrupted prematurely.
342 while (prevTime < curTime - tick / 2)
343 {
344     //printf("%05lld | %05lld\n",
345     //      prevTime % 100000, (curTime - tick / 2) % 100000);
346     _plat__ACT_Tick();
347     prevTime += (uint64_t)tick;
348 }
349 // Adjust the next timeout to keep the average interval of one second
350 timeout = tick + (prevTime - curTime);
351 //prevTime = curTime;
352 //printf("%04lld | c:%05lld | p:%05llu\n",
353 //      timeout, curTime % 100000, prevTime);
354 }
355 return 0;
356 }

```

D.3.3.7. ActTimeService()

This function starts a new thread waiting to wait for time ticks.

Return Value	Meaning
==0	success
!=0	failure

```

357 static int
358 ActTimeService(
359     void
360 )
361 {
362     static bool    running = false;
363     int           ret = 0;
364     if(!running)
365     {
366 #if defined(_MSC_VER)
367         HANDLE     hThr;
368         int         ThreadId;
369         //
370         printf("Starting ACT thread...\n");
371         // Don't allow ticks to be processed before TPM is manufactured.
372         _plat__ACT_EnableTicks(false);
373
374         // Create service thread for ACT internal timer
375         hThr = CreateThread(NULL, 0,
376             (LPTHREAD_START_ROUTINE)SimulatorTimeServiceRoutine,
377             (LPVOID) (INT_PTR) NULL, 0, (LPDWORD) &ThreadId);
378         if(hThr != NULL)
379             CloseHandle(hThr);
380         else
381             ret = -1;
382 #else
383         pthread_t    thread_id;
384         //
385         ret = pthread_create(&thread_id, NULL, (void*)SimulatorTimeServiceRoutine,
386             (LPVOID) (INT_PTR) NULL);
387 #endif // _MSC_VER
388
389         if(ret != 0)
390             printf("ACT thread Creation failed\n");
391         else

```

```

392         running = true;
393     }
394     return ret;
395 }
396 #endif // RH_ACT_0

```

D.3.3.8. StartTcpServer()

This is the main entry-point to the TCP server. The server listens on port specified. Note that there is no way to specify the network interface in this implementation.

```

397 int
398 StartTcpServer(
399     int          PortNumber
400 )
401 {
402     int          res;
403     //
404     #ifdef RH_ACT_0
405         // Start the Time Service routine
406         res = ActTimeService();
407         if(res != 0)
408         {
409             printf("TimeService failed\n");
410             return res;
411         }
412     #endif
413
414     // Start Platform Signal Processing Service
415     res = PlatformSignalService(PortNumber + 1);
416     if(res != 0)
417     {
418         printf("PlatformSignalService failed\n");
419         return res;
420     }
421     // Start Regular/DRTM TPM command service
422     res = RegularCommandService(PortNumber);
423     if(res != 0)
424     {
425         printf("RegularCommandService failed\n");
426         return res;
427     }
428     return 0;
429 }

```

D.3.3.9. ReadBytes()

This function reads the indicated number of bytes (*NumBytes*) into buffer from the indicated socket.

```

430 bool
431 ReadBytes(
432     SOCKET      s,
433     char        *buffer,
434     int         NumBytes
435 )
436 {
437     int          res;
438     int          numGot = 0;
439     //
440     while(numGot < NumBytes)
441     {
442         res = recv(s, buffer + numGot, NumBytes - numGot, 0);
443         if(res == -1)

```

```

444     {
445         printf("Receive error.  Error is 0x%x\n", WSAGetLastError());
446         return false;
447     }
448     if(res == 0)
449     {
450         return false;
451     }
452     numGot += res;
453 }
454 return true;
455 }

```

D.3.3.10. WriteBytes()

This function will send the indicated number of bytes (*NumBytes*) to the indicated socket

```

456 bool
457 WriteBytes(
458     SOCKET          s,
459     char            *buffer,
460     int             NumBytes
461 )
462 {
463     int             res;
464     int             numSent = 0;
465     //
466     while(numSent < NumBytes)
467     {
468         res = send(s, buffer + numSent, NumBytes - numSent, 0);
469         if(res == -1)
470         {
471             if(WSAGetLastError() == 0x2745)
472             {
473                 printf("Client disconnected\n");
474             }
475             else
476             {
477                 printf("Send error.  Error is 0x%x\n", WSAGetLastError());
478             }
479             return false;
480         }
481         numSent += res;
482     }
483     return true;
484 }

```

D.3.3.11. WriteUINT32()

Send 4 byte integer

```

485 bool
486 WriteUINT32(
487     SOCKET          s,
488     uint32_t        val
489 )
490 {
491     uint32_t netVal = htonl(val);
492     //
493     return WriteBytes(s, (char*)&netVal, 4);
494 }

```


D.3.3.12. ReadUINT32()

Function to read 4 byte integer from socket.

```

495  bool
496  ReadUINT32 (
497      SOCKET          s,
498      uint32_t        *val
499  )
500  {
501      uint32_t netVal;
502      //
503      if (!ReadBytes(s, (char*)&netVal, 4))
504          return false;
505      *val = ntohl(netVal);
506      return true;
507  }

```

D.3.3.13. ReadVarBytes()

Get a uint32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```

508  bool
509  ReadVarBytes (
510      SOCKET          s,
511      char            *buffer,
512      uint32_t        *BytesReceived,
513      int             MaxLen
514  )
515  {
516      int             length;
517      bool            res;
518      //
519      res = ReadBytes(s, (char*)&length, 4);
520      if(!res) return res;
521      length = ntohl(length);
522      *BytesReceived = length;
523      if(length > MaxLen)
524      {
525          printf("Buffer too big. Client says %d\n", length);
526          return false;
527      }
528      if(length == 0) return true;
529      res = ReadBytes(s, buffer, length);
530      if(!res) return res;
531      return true;
532  }

```

D.3.3.14. WriteVarBytes()

Send a uint32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```

533  bool
534  WriteVarBytes (
535      SOCKET          s,
536      char            *buffer,
537      int             BytesToSend
538  )
539  {
540      uint32_t        netLength = htonl(BytesToSend);

```

```

541     bool res;
542     //
543     res = WriteBytes(s, (char*)&netLength, 4);
544     if(!res)
545         return res;
546     res = WriteBytes(s, buffer, BytesToSend);
547     if(!res)
548         return res;
549     return true;
550 }

```

D.3.3.15. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```

551 bool
552 TpmServer(
553     SOCKET          s
554 )
555 {
556     uint32_t          length;
557     uint32_t          Command;
558     uint8_t          locality;
559     bool              OK;
560     int               result;
561     int               clientVersion;
562     _IN_BUFFER        InBuffer;
563     _OUT_BUFFER        OutBuffer;
564     //
565     for(;;)
566     {
567         OK = ReadBytes(s, (char*)&Command, 4);
568         // client disconnected (or other error). We stop processing this client
569         // and return to our caller who can stop the server or listen for another
570         // connection.
571         if(!OK)
572             return true;
573         Command = ntohl(Command);
574         switch(Command)
575         {
576             case TPM_SIGNAL_HASH_START:
577                 _rpc_Signal_Hash_Start();
578                 break;
579             case TPM_SIGNAL_HASH_END:
580                 _rpc_Signal_HashEnd();
581                 break;
582             case TPM_SIGNAL_HASH_DATA:
583                 OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
584                 if(!OK) return true;
585                 InBuffer.Buffer = (uint8_t*)InputBuffer;
586                 InBuffer.BufferSize = length;
587                 _rpc_Signal_Hash_Data(InBuffer);
588                 break;
589             case TPM_SEND_COMMAND:
590                 OK = ReadBytes(s, (char*)&locality, 1);
591                 if(!OK)
592                     return true;
593                 OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
594                 if(!OK)
595                     return true;
596                 InBuffer.Buffer = (uint8_t*)InputBuffer;
597                 InBuffer.BufferSize = length;
598                 OutBuffer.BufferSize = MAX_BUFFER;
599                 OutBuffer.Buffer = (_OUTPUT_BUFFER)OutputBuffer;

```

```

600         // record the number of bytes in the command if it is the largest
601         // we have seen so far.
602         if(InBuffer.BufferSize > CommandResponseSizes.largestCommandSize)
603         {
604             CommandResponseSizes.largestCommandSize = InBuffer.BufferSize;
605             memcpy(&CommandResponseSizes.largestCommand,
606                 &InputBuffer[6], sizeof(uint32_t));
607         }
608         _rpc_Send_Command(locality, InBuffer, &OutBuffer);
609         // record the number of bytes in the response if it is the largest
610         // we have seen so far.
611         if(OutBuffer.BufferSize > CommandResponseSizes.largestResponseSize)
612         {
613             CommandResponseSizes.largestResponseSize
614                 = OutBuffer.BufferSize;
615             memcpy(&CommandResponseSizes.largestResponse,
616                 &OutputBuffer[6], sizeof(uint32_t));
617         }
618         OK = WriteVarBytes(s,
619             (char*)OutBuffer.Buffer,
620             OutBuffer.BufferSize);
621         if(!OK)
622             return true;
623         break;
624     case TPM_REMOTE_HANDSHAKE:
625         OK = ReadBytes(s, (char*)&clientVersion, 4);
626         if(!OK)
627             return true;
628         if(clientVersion == 0)
629         {
630             printf("Unsupported client version (0).\n");
631             return true;
632         }
633         OK &= WriteUINT32(s, ServerVersion);
634         OK &= WriteUINT32(s, tpmInRawMode
635             | tpmPlatformAvailable | tpmSupportsPP);
636         break;
637     case TPM_SET_ALTERNATIVE_RESULT:
638         OK = ReadBytes(s, (char*)&result, 4);
639         if(!OK)
640             return true;
641         // Alternative result is not applicable to the simulator.
642         break;
643     case TPM_SESSION_END:
644         // Client signaled end-of-session
645         return true;
646     case TPM_STOP:
647         // Client requested the simulator to exit
648         return false;
649     default:
650         printf("Unrecognized TPM interface command %d\n", (int)Command);
651         return true;
652     }
653     OK = WriteUINT32(s, 0);
654     if(!OK)
655         return true;
656 }
657 }

```

D.4 TPMCmdp.c

D.4.1. Description

This file contains the functions that process the commands received on the control port or the command port of the simulator. The control port is used to allow simulation of hardware events (such as, `_TPM_Hash_Start`) to test the simulated TPM's reaction to those events. This improves code coverage of the testing.

D.4.2. Includes and Data Definitions

```

1  #include <stdbool.h>
2  #include "TpmBuildSwitches.h"
3
4  #ifndef _MSC_VER
5  #   pragma warning(push, 3)
6  #   include <windows.h>
7  #   include <winsock.h>
8  #   pragma warning(pop)
9  #elif defined(__unix__)
10 #   include "BaseTypes.h"    // on behalf of TpmFail_fp.h
11     typedef int SOCKET;
12 #else
13 #   error "Unsupported platform."
14 #endif
15
16 #include "Platform_fp.h"
17 #include "ExecCommand_fp.h"
18 #include "Manufacture_fp.h"
19 #include "_TPM_Init_fp.h"
20 #include "_TPM_Hash_Start_fp.h"
21 #include "_TPM_Hash_Data_fp.h"
22 #include "_TPM_Hash_End_fp.h"
23 #include "TpmFail_fp.h"
24
25 #include "TpmTcpProtocol.h"
26 #include "Simulator_fp.h"
27
28 static bool    s_isPowerOn = false;

```

D.4.3. Functions

D.4.3.1. Signal_PowerOn()

This function processes a power-on indication. Among other things, it calls the `_TPM_Init()` handler.

```

29 void
30 __rpc__Signal_PowerOn(
31     bool    isReset
32 )
33 {
34     // if power is on and this is not a call to do TPM reset then return
35     if(s_isPowerOn && !isReset)
36         return;
37     // If this is a reset but power is not on, then return
38     if(isReset && !s_isPowerOn)
39         return;
40     // Unless this is just a reset, pass power on signal to platform
41     if(!isReset)
42         __plat__Signal_PowerOn();

```

```

43     // Power on and reset both lead to _TPM_Init()
44     _plat__Signal_Reset();
45
46     // Set state as power on
47     s_isPowerOn = true;
48 }

```

D.4.3.2. Signal_Restart()

This function processes the clock restart indication. All it does is call the platform function.

```

49 void
50 _rpc__Signal_Restart(
51     void
52 )
53 {
54     _plat__TimerRestart();
55 }

```

D.4.3.3. Signal_PowerOff()

This function processes the power off indication. Its primary function is to set a flag indicating that the next power on indication should cause _TPM_Init() to be called.

```

56 void
57 _rpc__Signal_PowerOff(
58     void
59 )
60 {
61     if(s_isPowerOn)
62         // Pass power off signal to platform
63         _plat__Signal_PowerOff();
64     // This could be redundant, but...
65     s_isPowerOn = false;
66
67     return;
68 }

```

D.4.3.4. _rpc__ForceFailureMode()

This function is used to debug the Failure Mode logic of the TPM. It will set a flag in the TPM code such that the next call to TPM2_SelfTest() will result in a failure, putting the TPM into Failure Mode.

```

69 void
70 _rpc__ForceFailureMode(
71     void
72 )
73 {
74     SetForceFailureMode();
75     return;
76 }

```

D.4.3.5. _rpc__Signal_PhysicalPresenceOn()

This function is called to simulate activation of the physical presence **pin**.

```

77 void
78 _rpc__Signal_PhysicalPresenceOn(
79     void

```

```

80     )
81 {
82     // If TPM power is on...
83     if(s_isPowerOn)
84         // ... pass physical presence on to platform
85         _plat__Signal_PhysicalPresenceOn();
86     return;
87 }

```

D.4.3.6. _rpc__Signal_PhysicalPresenceOff()

This function is called to simulate deactivation of the physical presence **pin**.

```

88 void
89 _rpc__Signal_PhysicalPresenceOff(
90     void
91 )
92 {
93     // If TPM is power on...
94     if(s_isPowerOn)
95         // ... pass physical presence off to platform
96         _plat__Signal_PhysicalPresenceOff();
97     return;
98 }

```

D.4.3.7. _rpc__Signal_Hash_Start()

This function is called to simulate a _TPM_Hash_Start event. It will call

```

99 void
100 _rpc__Signal_Hash_Start(
101     void
102 )
103 {
104     // If TPM power is on...
105     if(s_isPowerOn)
106         // ... pass _TPM_Hash_Start signal to TPM
107         _TPM_Hash_Start();
108     return;
109 }

```

D.4.3.8. _rpc__Signal_Hash_Data()

This function is called to simulate a _TPM_Hash_Data event.

```

110 void
111 _rpc__Signal_Hash_Data(
112     _IN_BUFFER    input
113 )
114 {
115     // If TPM power is on...
116     if(s_isPowerOn)
117         // ... pass _TPM_Hash_Data signal to TPM
118         _TPM_Hash_Data(input.BufferSize, input.Buffer);
119     return;
120 }

```

D.4.3.9. _rpc__Signal_HashEnd()

This function is called to simulate a _TPM_Hash_End event.

```

121 void
122 _rpc__Signal_HashEnd(
123     void
124 )
125 {
126     // If TPM power is on...
127     if(s_isPowerOn)
128         // ... pass _TPM_HashEnd signal to TPM
129         _TPM_Hash_End();
130     return;
131 }

```

D.4.3.10. _rpc__Send_Command()

This is the interface to the TPM code.

```

132 void
133 _rpc__Send_Command(
134     unsigned char    locality,
135     _IN_BUFFER       request,
136     _OUT_BUFFER      *response
137 )
138 {
139     // If TPM is power off, reject any commands.
140     if(!s_isPowerOn)
141     {
142         response->BufferSize = 0;
143         return;
144     }
145     // Set the locality of the command so that it doesn't change during the command
146     _plat__LocalitySet(locality);
147     // Do implementation-specific command dispatch
148     _plat__RunCommand(request.BufferSize, request.Buffer,
149         &response->BufferSize, &response->Buffer);
150     return;
151 }

```

D.4.3.11. _rpc__Signal_CancelOn()

This function is used to turn on the indication to cancel a command in process. An executing command is not interrupted. The command code may periodically check this indication to see if it should abort the current command processing and returned TPM_RC_CANCELLED.

```

152 void
153 _rpc__Signal_CancelOn(
154     void
155 )
156 {
157     // If TPM power is on...
158     if(s_isPowerOn)
159         // ... set the platform canceling flag.
160         _plat__SetCancel();
161     return;
162 }

```

D.4.3.12. _rpc__Signal_CancelOff()

This function is used to turn off the indication to cancel a command in process.

```

163 void
164 _rpc__Signal_CancelOff(

```



```

165     void
166     )
167 {
168     // If TPM power is on...
169     if(s_isPowerOn)
170         // ... set the platform canceling flag.
171         _plat__ClearCancel();
172     return;
173 }

```

D.4.3.13. _rpc__Signal_NvOn()

In a system where the NV memory used by the TPM is not within the TPM, the NV may not always be available. This function turns on the indicator that indicates that NV is available.

```

174 void
175 _rpc__Signal_NvOn(
176     void
177 )
178 {
179     // If TPM power is on...
180     if(s_isPowerOn)
181         // ... make the NV available
182         _plat__SetNvAvail();
183     return;
184 }

```

D.4.3.14. _rpc__Signal_NvOff()

This function is used to set the indication that NV memory is no longer available.

```

185 void
186 _rpc__Signal_NvOff(
187     void
188 )
189 {
190     // If TPM power is on...
191     if(s_isPowerOn)
192         // ... make NV not available
193         _plat__ClearNvAvail();
194     return;
195 }
196 void RsaKeyCacheControl(int state);

```

D.4.3.15. _rpc__RsaKeyCacheControl()

This function is used to enable/disable the use of the RSA key cache during simulation.

```

197 void
198 _rpc__RsaKeyCacheControl(
199     int state
200 )
201 {
202     #if USE_RSA_KEY_CACHE
203         RsaKeyCacheControl(state);
204     #else
205         NOT_REFERENCED(state);
206     #endif
207     return;
208 }
209 #define TPM_RH_ACT_0          0x40000110

```

D.4.3.16. _rpc__ACT_GetSignaled()

This function is used to count the ACT second tick.

```
210  bool
211  _rpc__ACT_GetSignaled(
212      uint32_t actHandle
213  )
214  {
215      // If TPM power is on...
216      if (s_isPowerOn)
217          // ... query the platform
218          return _plat__ACT_GetSignaled(actHandle - TPM_RH_ACT_0);
219      return false;
220  }
```

D.5 TPMCmds.c

D.5.1. Description

This file contains the entry point for the simulator.

D.5.2. Includes, Defines, Data Definitions, and Function Prototypes

```

1  #include "TpmBuildSwitches.h"
2
3  #include <stdlib.h>
4  #include <stdio.h>
5  #include <stdint.h>
6  #include <stdbool.h>
7  #include <ctype.h>
8  #include <string.h>
9
10 #ifndef _MSC_VER
11 #   pragma warning(push, 3)
12 #   include <windows.h>
13 #   include <winsock.h>
14 #   pragma warning(pop)
15 #elif defined(__unix__)
16 #   define _strcmpi strcasecmp
17 #   typedef int SOCKET;
18 #else
19 #   error "Unsupported platform."
20 #endif
21
22 #include "TpmTcpProtocol.h"
23 #include "Manufacture_fp.h"
24 #include "Platform_fp.h"
25 #include "Simulator_fp.h"
26
27 #define PURPOSE \
28 "TPM 2.0 Reference Simulator.\n" \
29 "Copyright (c) Microsoft Corporation. All rights reserved."
30
31 #define DEFAULT_TPM_PORT 2321

```

Information about command line arguments (does not include program name)

```

32 static uint32_t    s_ArgsMask = 0;    // Bit mask of unmatched command line args
33 static int         s_Argc = 0;
34 static const char **s_Argv = NULL;

```

D.5.3. Functions

```

35 #if DEBUG

```

D.5.3.1. Assert()

This function implements a run-time assertion. Computation of its parameters must not result in any side effects, as these computations will be stripped from the release builds.

```

36 static void Assert (bool cond, const char* msg)
37 {
38     if (cond)
39         return;
40     fputs(msg, stderr);

```

```

41     exit(2);
42 }
43 #else
44 #define Assert(cond, msg)
45 #endif

```

D.5.3.2. Usage()

This function prints the proper calling sequence for the simulator.

```

46 static void
47 Usage(
48     const char      *programName
49 )
50 {
51     fprintf(stderr, "%s\n\n", PURPOSE);
52     fprintf(stderr, "Usage:  %s [PortNum] [opts]\n\n",
53         "Starts the TPM server listening on TCP port PortNum (by default %d).\n\n"
54         "An option can be in the short form (one letter preceded with '-' or '/')\n"
55         "or in the full form (preceded with '--' or no option marker at all).\n"
56         "Possible options are:\n"
57         "  -h (--help) or ? - print this message\n"
58         "  -m (--manufacture) - forces NV state of the TPM simulator to be "
59         "(re)manufactured\n",
60     programName, DEFAULT_TPM_PORT);
61     exit(1);
62 }

```

D.5.3.3. CmdLineParser_Init()

This function initializes command line option parser.

```

63 static bool
64 CmdLineParser_Init(
65     int argc,
66     char *argv[],
67     int maxOpts
68 )
69 {
70     if (argc == 1)
71         return false;
72
73     if (maxOpts && (argc - 1) > maxOpts)
74     {
75         fprintf(stderr, "No more than %d options can be specified\n\n", maxOpts);
76         Usage(argv[0]);
77     }
78     s_Argc = argc - 1;
79     s_Argv = (const char**) (argv + 1);
80     s_ArgsMask = (1 << s_Argc) - 1;
81     return true;
82 }

```

D.5.3.4. CmdLineParser_More()

Returns true if there are unparsed options still.

```

83 static bool
84 CmdLineParser_More(
85     void
86 )

```

```

87 {
88     return s_ArgsMask != 0;
89 }

```

D.5.3.5. CmdLineParser_IsOpt()

This function determines if the given command line parameter represents a valid option.

```

90 static bool
91 CmdLineParser_IsOpt(
92     const char* opt,           // Command line parameter to check
93     const char* optFull,      // Expected full name
94     const char* optShort,     // Expected short (single letter) name
95     bool dashed               // The parameter is preceded by a single dash
96 )
97 {
98     return 0 == strcmp(opt, optFull)
99         || (optShort && opt[0] == optShort[0] && opt[1] == 0)
100         || (dashed && opt[0] == '-' && 0 == strcmp(opt + 1, optFull));
101 }

```

D.5.3.6. CmdLineParser_IsOptPresent()

This function determines if the given command line parameter represents a valid option.

```

102 static bool
103 CmdLineParser_IsOptPresent(
104     const char* optFull,
105     const char* optShort
106 )
107 {
108     int i;
109     int curArgBit;
110     Assert(s_Argv != NULL,
111         "InitCmdLineOptParser(argc, argv) has not been invoked\n");
112     Assert(optFull && optFull[0],
113         "Full form of a command line option must be present.\n"
114         "If only a short (single letter) form is supported, it must be"
115         "specified as the full one.\n");
116     Assert(!optShort || (optShort[0] && !optShort[1]),
117         "If a short form of an option is specified, it must consist "
118         "of a single letter only.\n");
119
120     if (!CmdLineParser_More())
121         return false;
122
123     for (i = 0, curArgBit = 1; i < s_Argc; ++i, curArgBit <= 1)
124     {
125         const char* opt = s_Argv[i];
126         if ( (s_ArgsMask & curArgBit) && opt
127             && ( 0 == strcmp(opt, optFull)
128                 || ( opt[0] == '/' || opt[0] == '-' )
129                     && CmdLineParser_IsOpt(opt + 1, optFull, optShort,
130                                             opt[0] == '-') ) ) )
131         {
132             s_ArgsMask ^= curArgBit;
133             return true;
134         }
135     }
136     return false;
137 }

```

D.5.3.7. CmdLineParser_Done()

This function notifies the parser that no more options are needed.

```

138 static void
139 CmdLineParser_Done(
140     const char      *programName
141 )
142 {
143     char delim = ':';
144     int      i;
145     int      curArgBit;
146
147     if (!CmdLineParser_More())
148         return;
149
150     fprintf(stderr, "Command line contains unknown option%s",
151             s_ArgsMask & (s_ArgsMask - 1) ? "s" : "");
152     for (i = 0, curArgBit = 1; i < s_Argc; ++i, curArgBit <= 1)
153     {
154         if (s_ArgsMask & curArgBit)
155         {
156             fprintf(stderr, "%c %s", delim, s_Argv[i]);
157             delim = ',';
158         }
159     }
160     fprintf(stderr, "\n\n");
161     Usage(programName);
162 }

```

D.5.3.8. main()

This is the main entry point for the simulator. It registers the interface and starts listening for clients

```

163 int
164 main(
165     int      argc,
166     char     *argv[]
167 )
168 {
169     bool      manufacture = false;
170     int      PortNum = DEFAULT_TPM_PORT;
171
172     // Parse command line options
173
174     if (CmdLineParser_Init(argc, argv, 2))
175     {
176         if ( CmdLineParser_IsOptPresent("?", "?")
177             || CmdLineParser_IsOptPresent("help", "h"))
178         {
179             Usage(argv[0]);
180         }
181         if (CmdLineParser_IsOptPresent("manufacture", "m"))
182         {
183             manufacture = true;
184         }
185         if (CmdLineParser_More())
186         {
187             int      i;
188             for (i = 0; i < s_Argc; ++i)
189             {
190                 char *nptr = NULL;
191                 int portNum = (int)strtol(s_Argv[i], &nptr, 0);
192                 if (s_Argv[i] != nptr)

```

```

193         {
194             // A numeric option is found
195             if(!*nptr && portNum > 0 && portNum < 65535)
196             {
197                 PortNum = portNum;
198                 s_ArgsMask ^= 1 << i;
199                 break;
200             }
201             fprintf(stderr, "Invalid numeric option %s\n\n", s_Argv[i]);
202             Usage(argv[0]);
203         }
204     }
205 }
206 CmdLineParser_Done(argv[0]);
207 }
208 printf("LIBRARY_COMPATIBILITY_CHECK is %s\n",
209        (LIBRARY_COMPATIBILITY_CHECK ? "ON" : "OFF"));
210 // Enable NV memory
211 _plat__NVEnable(NULL);
212
213 if (manufacture || _plat__NVNeedsManufacture())
214 {
215     printf("Manufacturing NV state...\n");
216     if(TPM_Manufacture(1) != 0)
217     {
218         // if the manufacture didn't work, then make sure that the NV file doesn't
219         // survive. This prevents manufacturing failures from being ignored the
220         // next time the code is run.
221         _plat__NVDisable(1);
222         exit(1);
223     }
224     // Coverage test - repeated manufacturing attempt
225     if(TPM_Manufacture(0) != 1)
226     {
227         exit(2);
228     }
229     // Coverage test - re-manufacturing
230     TPM_TearDown();
231     if(TPM_Manufacture(1) != 0)
232     {
233         exit(3);
234     }
235 }
236 // Disable NV memory
237 _plat__NVDisable(0);
238
239 StartTcpServer(PortNum);
240 return EXIT_SUCCESS;
241 }

```