

Trusted Platform Module Library

Part 4: Supporting Routines

Family “2.0”

Level 00 Revision 01.55

April 15, 2019

Committee Draft

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.

Contact: admin@trustedcomputinggroup.org

TCG PUBLIC REVIEW

Copyright © TCG 2006-2019

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

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	4
4.2.3.1	Simple Types and Structures	4
4.2.3.2	Union Types	4
4.2.3.3	Arrays	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	BaseTypes.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	92
5.8	CompilerDependencies.h	99
5.9	Global.h	101
5.9.1	Description	101
5.9.2	Includes	101
5.9.3	Loaded Object Structures	102
5.9.3.1	Description	102
5.9.3.2	OBJECT_ATTRIBUTES	102
5.9.3.3	OBJECT Structure	103
5.9.3.4	HASH_OBJECT Structure	104
5.9.3.5	ANY_OBJECT	104
5.9.4	AUTH_DUP Types	104
5.9.5	Active Session Context	104
5.9.5.1	Description	104
5.9.5.2	SESSION_ATTRIBUTES	105
5.9.5.3	SESSION Structure	105
5.9.6	PCR	106
5.9.6.1	PCR_SAVE Structure	106
5.9.6.2	PCR_POLICY	107
5.9.6.3	PCR_AUTHVALUE	107
5.9.7	STARTUP_TYPE	107
5.9.8	NV	108

5.9.8.1	NV_INDEX.....	108
5.9.8.2	NV_REF	108
5.9.8.3	NV_PIN	108
5.9.9	COMMIT_INDEX_MASK.....	108
5.9.10	RAM Global Values	109
5.9.10.1	Description	109
5.9.10.2	Crypto Self-Test Values	109
5.9.10.3	g_rcIndex	109
5.9.10.4	g_exclusiveAuditSession	109
5.9.10.5	g_time	109
5.9.10.6	g_timeEpoch	109
5.9.10.7	g_phEnable	110
5.9.10.8	g_pcrReConfig.....	110
5.9.10.9	g_DRTMHandle	110
5.9.10.10	g_DrtmPreStartup.....	110
5.9.10.11	g_StartupLocality3.....	110
5.9.10.12	TPM_SU_NONE	110
5.9.10.13	TPM_SU_DA_USED	110
5.9.10.14	Startup Flags.....	111
5.9.10.15	g_daUsed	111
5.9.10.16	g_updateNV	111
5.9.10.17	g_powerWasLost.....	111
5.9.10.18	g_clearOrderly.....	111
5.9.10.19	g_prevOrderlyState	112
5.9.10.20	g_nvOk.....	112
5.9.10.21	g_platformUnique	112
5.9.11	Persistent Global Values	112
5.9.11.1	Description	112
5.9.11.2	PERSISTENT_DATA	112
5.9.11.3	ORDERLY_DATA	115
5.9.11.4	STATE_CLEAR_DATA	115
5.9.11.5	State Reset Data	116
5.9.12	NV Layout	118
5.9.13	Global Macro Definitions	118
5.9.14	From CryptTest.c.....	120
5.9.15	From Manufacture.c	120
5.9.16	Private data.....	120
5.9.16.1	From SessionProcess.c	120
5.9.16.2	From DA.c	121
5.9.16.3	From NV.c	121
5.9.16.4	From Object.c.....	122
5.9.16.5	From PCR.c.....	122
5.9.16.6	From Session.c.....	123
5.9.16.7	From IoBuffers.c.....	123
5.9.16.8	From TPMFail.c	123
5.9.16.9	From CommandCodeAttributes.c	124
5.10	GpMacros.h.....	125
5.10.1	Introduction	125
5.10.2	For Self-test	125
5.10.3	For Failures.....	125
5.10.4	Derived from Vendor-specific values	126
5.10.5	Compile-time Checks	126
5.11	InternalRoutines.h	130
5.12	LibSupport.h.....	132

5.13	NV.h.....	133
5.13.1	Index Type Definitions	133
5.13.2	Attribute Macros	133
5.13.3	Orderly RAM Values	134
5.14	PRNG_TestVectors.h	136
5.15	SelfTest.h.....	137
5.15.1	Introduction	137
5.15.2	Defines.....	137
5.16	SupportLibraryFunctionPrototypes_fp.h	139
5.16.1	Introduction	139
5.16.2	SupportLibInit()	139
5.16.3	MathLibraryCompatibilityCheck()	139
5.16.4	BnModMult().....	139
5.16.5	BnMult()	139
5.16.6	BnDiv().....	139
5.16.7	BnMod()	140
5.16.8	BnGcd().....	140
5.16.9	BnModExp()	140
5.16.10	BnModInverse().....	140
5.16.11	BnEccModMult()	140
5.16.12	BnEccModMult2()	140
5.16.13	BnEccAdd()	140
5.16.14	BnCurveInitialize().....	141
5.16.14.1	BnCurveFree().....	141
5.17	TPMB.h	142
5.18	Tpm.h.....	143
5.19	TpmBuildSwitches.h	144
5.20	TpmError.h.....	150
5.21	TpmTypes.h	151
5.22	VendorString.h	186
5.23	swap.h	187
6	Main	189
6.1	Introduction	189
6.2	ExecCommand.c	189
6.2.1	Introduction	189
6.2.2	Includes	189
6.2.3	ExecuteCommand()	189
6.3	CommandDispatcher.c	195
6.3.1	Introduction	195
6.3.1.1	Includes and Typedefs.....	195
6.3.1.2	Marshal/Unmarshal Functions.....	197
6.3.1.2.1	ParseHandleBuffer().....	197
6.3.1.2.2	CommandDispatcher().....	198
6.4	SessionProcess.c.....	202
6.4.1	Introduction	202
6.4.2	Includes and Data Definitions	202
6.4.3	Authorization Support Functions	202
6.4.3.1	IsDAExempted()	202
6.4.3.2	IncrementLockout().....	203
6.4.3.3	IsSessionBindEntity()	204

6.4.3.4	IsPolicySessionRequired()	205
6.4.3.5	IsAuthValueAvailable()	206
6.4.3.6	IsAuthPolicyAvailable()	208
6.4.4	Session Parsing Functions	209
6.4.4.1	ClearCpRpHashes()	209
6.4.4.2	GetCpHashPointer()	210
6.4.4.3	GetRpHashPointer()	210
6.4.4.4	ComputeCpHash()	211
6.4.4.5	GetCpHash()	212
6.4.4.6	CompareTemplateHash()	212
6.4.4.7	CompareNameHash()	213
6.4.4.8	CheckPWAuthSession()	213
6.4.4.9	ComputeCommandHMAC()	214
6.4.4.10	CheckSessionHMAC()	216
6.4.4.11	CheckPolicyAuthSession()	216
6.4.4.12	RetrieveSessionData()	219
6.4.4.13	CheckLockedOut()	221
6.4.4.14	CheckAuthSession()	222
6.4.4.15	CheckCommandAudit()	225
6.4.4.16	ParseSessionBuffer()	225
6.4.4.17	CheckAuthNoSession()	228
6.4.5	Response Session Processing	229
6.4.5.1	Introduction	229
6.4.5.2	ComputeRpHash()	229
6.4.5.3	InitAuditSession()	229
6.4.5.4	UpdateAuditDigest	230
6.4.5.5	Audit()	230
6.4.5.6	CommandAudit()	230
6.4.5.7	UpdateAuditSessionStatus()	231
6.4.5.8	ComputeResponseHMAC()	232
6.4.5.9	UpdateInternalSession()	233
6.4.5.10	BuildSingleResponseAuth()	234
6.4.5.11	UpdateAllNonceTPM()	234
6.4.5.12	BuildResponseSession()	234
6.4.5.13	SessionRemoveAssociationToHandle()	236
7	Command Support Functions	237
7.1	Introduction	237
7.2	Attestation Command Support (Attest_spt.c)	237
7.2.1	Includes	237
7.2.2	Functions	237
7.2.2.1	FillInAttestInfo()	237
7.2.2.2	SignAttestInfo()	238
7.2.2.3	IsSigningObject()	239
7.3	Context Management Command Support (Context_spt.c)	240
7.3.1	Includes	240
7.3.2	Functions	240
7.3.2.1	ComputeContextProtectionKey()	240
7.3.2.2	ComputeContextIntegrity()	241
7.3.2.3	SequenceDataExport()	242
7.3.2.4	SequenceDataImport()	242
7.4	Policy Command Support (Policy_spt.c)	243
7.4.1	Includes	243

7.4.2	Functions	243
7.4.2.1	PolicyParameterChecks()	243
7.4.2.2	PolicyContextUpdate()	244
7.4.2.3	ComputeAuthTimeout()	245
7.4.2.4	PolicyDigestClear()	245
7.5	NV Command Support (NV_spt.c)	248
7.5.1	Includes	248
7.5.2	Functions	248
7.5.2.1	NvReadAccessChecks()	248
7.5.2.2	NvWriteAccessChecks()	249
7.5.2.3	NvClearOrderly()	249
7.5.2.4	NvIsPinPassIndex()	250
7.6	Object Command Support (Object_spt.c)	251
7.6.1	Includes	251
7.6.2	Local Functions	251
7.6.2.1	GetIV2BSize()	251
7.6.2.2	ComputeProtectionKeyParms()	251
7.6.2.3	ComputeOuterIntegrity()	252
7.6.2.4	ComputeInnerIntegrity()	253
7.6.2.5	ProduceInnerIntegrity()	253
7.6.2.6	CheckInnerIntegrity()	254
7.6.3	Public Functions	255
7.6.3.1	AdjustAuthSize()	255
7.6.3.2	AreAttributesForParent()	255
7.6.3.3	CreateChecks()	255
7.6.3.4	SchemeChecks	256
7.6.3.5	PublicAttributesValidation()	260
7.6.3.6	FillInCreationData()	261
7.6.3.7	GetSeedForKDF()	262
7.6.3.8	ProduceOuterWrap()	263
7.6.3.9	UnwrapOuter()	264
7.6.3.10	MarshalSensitive()	265
7.6.3.11	SensitiveToPrivate()	266
7.6.3.12	PrivateToSensitive()	267
7.6.3.13	SensitiveToDuplicate()	268
7.6.3.14	DuplicateToSensitive()	270
7.6.3.15	SecretToCredential()	272
7.6.3.16	CredentialToSecret()	273
7.6.3.17	MemoryRemoveTrailingZeros()	274
7.6.3.18	SetLabelAndContext()	274
7.6.3.19	UnmarshalToPublic()	275
7.6.3.20	ObjectSetExternal()	275
7.7	Encrypt Decrypt Support (EncryptDecrypt_spt.c)	277
8	Subsystem	279
8.1	CommandAudit.c	279
8.1.1	Introduction	279
8.1.2	Includes	279
8.1.3	Functions	279
8.1.3.1	CommandAuditPreInstall_Init()	279
8.1.3.2	CommandAuditStartup()	279
8.1.3.3	CommandAuditSet()	280

8.1.3.4	CommandAuditClear()	280
8.1.3.5	CommandAuditIsRequired()	281
8.1.3.6	CommandAuditCapGetCCList()	281
8.1.3.7	CommandAuditGetDigest	282
8.2	DA.c	284
8.2.1	Introduction	284
8.2.2	Includes and Data Definitions	284
8.2.3	Functions	284
8.2.3.1	DAPreInstall_Init()	284
8.2.3.2	DAStartup()	284
8.2.3.3	DARegisterFailure()	285
8.2.3.4	DASelfHeal()	286
8.3	Hierarchy.c	288
8.3.1	Introduction	288
8.3.2	Includes	288
8.3.3	Functions	288
8.3.3.1	HierarchyPreInstall()	288
8.3.3.2	HierarchyStartup()	289
8.3.3.3	HierarchyGetProof()	289
8.3.3.4	HierarchyGetPrimarySeed()	290
8.3.3.5	HierarchyIsEnabled()	290
8.4	NvDynamic.c	292
8.4.1	Introduction	292
8.4.2	Includes, Defines and Data Definitions	292
8.4.3	Local Functions	292
8.4.3.1	NvNext()	292
8.4.3.2	NvNextByType()	293
8.4.3.3	NvNextIndex()	293
8.4.3.4	NvNextEvict()	294
8.4.3.5	NvGetEnd()	294
8.4.3.6	NvGetFreeBytes	294
8.4.3.7	NvTestSpace()	294
8.4.3.8	NvWriteNvListEnd()	295
8.4.3.9	NvAdd()	296
8.4.3.10	NvDelete()	297
8.4.4	RAM-based NV Index Data Access Functions	298
8.4.4.1	Introduction	298
8.4.4.2	NvRamNext()	298
8.4.4.3	NvRamGetEnd()	298
8.4.4.4	NvRamTestSpaceIndex()	299
8.4.4.5	NvRamGetIndex()	299
8.4.4.6	NvUpdateIndexOrderlyData()	299
8.4.4.7	NvAddRAM()	300
8.4.4.8	NvDeleteRAM()	300
8.4.4.9	NvReadIndex()	301
8.4.4.10	NvReadObject()	301
8.4.4.11	NvFindEvict()	301
8.4.4.12	NvIndexIsDefined()	302
8.4.4.13	NvConditionallyWrite()	302
8.4.4.14	NvReadNvIndexAttributes()	303
8.4.4.15	NvReadRamIndexAttributes()	303
8.4.4.16	NvWriteNvIndexAttributes()	303
8.4.4.17	NvWriteRamIndexAttributes()	303

8.4.5	Externally Accessible Functions	304
8.4.5.1	NvIsPlatformPersistentHandle()	304
8.4.5.2	NvIsOwnerPersistentHandle()	304
8.4.5.3	NvIndexIsAccessible()	304
8.4.5.4	NvGetEvictObject()	305
8.4.5.5	NvIndexCacheInit()	306
8.4.5.6	NvGetIndexData()	306
8.4.5.7	NvHashIndexData()	307
8.4.5.8	NvGetUINT64Data()	307
8.4.5.9	NvWriteIndexAttributes()	308
8.4.5.10	NvWriteIndexAuth()	308
8.4.5.11	NvGetIndexInfo()	309
8.4.5.12	NvWriteIndexData()	309
8.4.5.13	NvWriteUINT64Data()	311
8.4.5.14	NvGetIndexName()	311
8.4.5.15	NvGetNameByIndexHandle()	312
8.4.5.16	NvDefineIndex()	312
8.4.5.17	NvAddEvictObject()	313
8.4.5.18	NvDeleteIndex()	314
8.4.5.19	NvDeleteEvict()	314
8.4.5.20	NvFlushHierarchy()	315
8.4.5.21	NvSetGlobalLock()	316
8.4.5.22	InsertSort()	317
8.4.5.23	NvCapGetPersistent()	317
8.4.5.24	NvCapGetIndex()	318
8.4.5.25	NvCapGetIndexNumber()	319
8.4.5.26	NvCapGetPersistentNumber()	319
8.4.5.27	NvCapGetPersistentAvail()	320
8.4.5.28	NvCapGetCounterNumber()	320
8.4.5.29	NvSetStartupAttributes()	320
8.4.5.30	NvEntityStartup()	321
8.4.5.31	NvCapGetCounterAvail()	322
8.4.5.32	NvFindHandle()	323
8.4.6	NV Max Counter	323
8.4.6.1	Introduction	323
8.4.6.2	NvReadMaxCount()	323
8.4.6.3	NvUpdateMaxCount()	323
8.4.6.4	NvSetMaxCount()	324
8.4.6.5	NvGetMaxCount()	324
8.5	NvReserved.c	325
8.5.1	Introduction	325
8.5.2	Includes, Defines	325
8.5.3	Functions	325
8.5.3.1	NvInitStatic()	325
8.5.3.2	NvCheckState()	326
8.5.3.3	NvCommit	326
8.5.3.4	NvPowerOn()	326
8.5.3.5	NvManufacture()	327
8.5.3.6	NvRead()	327
8.5.3.7	NvWrite()	327
8.5.3.8	NvUpdatePersistent()	328
8.5.3.9	NvClearPersistent()	328
8.5.3.10	NvReadPersistent()	328
8.6	Object.c	329

8.6.1	Introduction	329
8.6.2	Includes and Data Definitions	329
8.6.3	Functions	329
8.6.3.1	ObjectFlush()	329
8.6.3.2	ObjectSetInUse()	329
8.6.3.3	ObjectStartup()	329
8.6.3.4	ObjectCleanupEvict()	330
8.6.3.5	IsObjectPresent()	330
8.6.3.6	ObjectIsSequence()	330
8.6.3.7	HandleToObject()	331
8.6.3.8	GetQualifiedName()	331
8.6.3.9	ObjectGetHierarchy()	332
8.6.3.10	GetHierarchy()	332
8.6.3.11	FindEmptyObjectSlot()	333
8.6.3.12	ObjectAllocateSlot()	333
8.6.3.13	ObjectSetLoadedAttributes()	333
8.6.3.14	ObjectLoad()	335
8.6.3.15	AllocateSequenceSlot()	336
8.6.3.16	ObjectCreateHMACSequence()	337
8.6.3.17	ObjectCreateHashSequence()	337
8.6.3.18	ObjectCreateEventSequence()	338
8.6.3.19	ObjectTerminateEvent()	338
8.6.3.20	ObjectContextLoad()	339
8.6.3.21	FlushObject()	340
8.6.3.22	ObjectFlushHierarchy()	340
8.6.3.23	ObjectLoadEvict()	341
8.6.3.24	ObjectComputeName()	341
8.6.3.25	PublicMarshalAndComputeName()	342
8.6.3.26	ComputeQualifiedName()	342
8.6.3.27	ObjectIsStorage()	343
8.6.3.28	ObjectCapGetLoaded()	343
8.6.3.29	ObjectCapGetTransientAvail()	344
8.6.3.30	ObjectGetPublicAttributes()	345
8.7	PCR.c	346
8.7.1	Introduction	346
8.7.2	Includes, Defines, and Data Definitions	346
8.7.3	Functions	346
8.7.3.1	PCRBelongsAuthGroup()	346
8.7.3.2	PCRBelongsPolicyGroup()	347
8.7.3.3	PCRBelongsTCBGroup()	347
8.7.3.4	PCRPolicyIsAvailable()	348
8.7.3.5	PCRGetAuthValue()	348
8.7.3.6	PCRGetAuthPolicy()	349
8.7.3.7	PCRSimStart()	349
8.7.3.8	GetSavedPcrPointer()	350
8.7.3.9	PcrIsAllocated()	351
8.7.3.10	GetPcrPointer()	351
8.7.3.11	IsPcrSelected()	352
8.7.3.12	FilterPcr()	352
8.7.3.13	PcrDrtm()	353
8.7.3.14	PCR_ClearAuth()	353
8.7.3.15	PCRStartup()	354
8.7.3.16	PCRStateSave()	355
8.7.3.17	PCRIsStateSaved()	356
8.7.3.18	PCRIsResetAllowed()	356
8.7.3.19	PCRChanged()	357

8.7.3.20	PCRIsExtendAllowed()	357
8.7.3.21	PCRExtend()	358
8.7.3.22	PCRComputeCurrentDigest()	358
8.7.3.23	PCRRead()	359
8.7.3.24	PcrWrite()	360
8.7.3.25	PCRAllocate()	361
8.7.3.26	PCRSetValue()	362
8.7.3.27	PCRResetDynamics	363
8.7.3.28	PCRCapGetAllocation()	364
8.7.3.29	PCRSetSelectBit()	364
8.7.3.30	PCRGetProperty()	364
8.7.3.31	PCRCapGetProperties()	366
8.7.3.32	PCRCapGetHandles()	367
8.8	PP.c	369
8.8.1	Introduction	369
8.8.2	Includes	369
8.8.3	Functions	369
8.8.3.1	PhysicalPresencePreInstall_Init()	369
8.8.3.2	PhysicalPresenceCommandSet()	369
8.8.3.3	PhysicalPresenceCommandClear()	370
8.8.3.4	PhysicalPresencelsRequired()	370
8.8.3.5	PhysicalPresenceCapGetCCList()	370
8.9	Session.c	372
8.9.1	Introduction	372
8.9.2	Includes, Defines, and Local Variables	373
8.9.3	File Scope Function -- ContextIdSetOldest()	373
8.9.4	Startup Function -- SessionStartup()	374
8.9.5	Access Functions	374
8.9.5.1	SessionIsLoaded()	374
8.9.5.2	SessionIsSaved()	375
8.9.5.3	SequenceNumberForSavedContextIsValid()	376
8.9.5.4	SessionPCRValuesCurrent()	376
8.9.5.5	SessionGet()	376
8.9.6	Utility Functions	377
8.9.6.1	ContextIdSessionCreate()	377
8.9.6.2	SessionCreate()	378
8.9.6.3	SessionContextSave()	380
8.9.6.4	SessionContextLoad()	381
8.9.6.5	SessionFlush()	383
8.9.6.6	SessionComputeBoundEntity()	383
8.9.6.7	SessionSetStartTime()	384
8.9.6.8	SessionResetPolicyData()	384
8.9.6.9	SessionCapGetLoaded()	385
8.9.6.10	SessionCapGetSaved()	386
8.9.6.11	SessionCapGetLoadedNumber()	387
8.9.6.12	SessionCapGetLoadedAvail()	387
8.9.6.13	SessionCapGetActiveNumber()	387
8.9.6.14	SessionCapGetActiveAvail()	388
8.10	Time.c	389
8.10.1	Introduction	389
8.10.2	Includes	389
8.10.3	Functions	389
8.10.3.1	TimePowerOn()	389

8.10.3.2	TimeNewEpoch()	389
8.10.3.3	TimeStartup()	389
8.10.3.4	TimeClockUpdate()	390
8.10.3.5	TimeUpdate()	390
8.10.3.6	TimeUpdateToCurrent()	391
8.10.3.7	TimeSetAdjustRate()	391
8.10.3.8	TimeGetMarshaled()	392
8.10.3.9	TimeFillInfo	392
9	Support	394
9.1	AlgorithmCap.c	394
9.1.1	Description	394
9.1.2	Includes and Defines	394
9.1.3	AlgorithmCapGetImplemented()	396
9.1.4	AlgorithmGetImplementedVector()	396
9.2	Bits.c	398
9.2.1	Introduction	398
9.2.2	Includes	398
9.2.3	Functions	398
9.2.3.1	TestBit()	398
9.2.3.2	SetBit()	398
9.2.3.3	ClearBit()	398
9.3	CommandCodeAttributes.c	400
9.3.1	Introduction	400
9.3.2	Includes and Defines	400
9.3.3	Command Attribute Functions	400
9.3.3.1	NextImplementedIndex()	400
9.3.3.2	GetClosestCommandIndex()	401
9.3.3.3	CommandCodeToComandIndex()	403
9.3.3.4	GetNextCommandIndex()	404
9.3.3.5	GetCommandCode()	404
9.3.3.6	CommandAuthRole()	405
9.3.3.7	EncryptSize()	405
9.3.3.8	DecryptSize()	406
9.3.3.9	IsSessionAllowed()	406
9.3.3.10	IsHandleInResponse()	406
9.3.3.11	IsWriteOperation()	406
9.3.3.12	IsReadOperation()	407
9.3.3.13	CommandCapGetCCList()	408
9.3.3.14	IsVendorCommand()	408
9.4	Entity.c	410
9.4.1	Description	410
9.4.2	Includes	410
9.4.3	Functions	410
9.4.3.1	EntityGetLoadStatus()	410
9.4.3.2	EntityGetAuthValue()	412
9.4.3.3	EntityGetAuthPolicy()	414
9.4.3.4	EntityGetName()	415
9.4.3.5	EntityGetHierarchy()	416
9.5	Global.c	418
9.5.1	Description	418
9.5.2	Defines and Includes	418

9.6	Handle.c.....	419
9.6.1	Description	419
9.6.2	Includes	419
9.6.3	Functions	419
9.6.3.1	HandleGetType()	419
9.6.3.2	NextPermanentHandle()	419
9.6.3.3	PermanentCapGetHandles()	420
9.6.3.4	PermanentHandleGetPolicy()	420
9.7	IoBuffers.c.....	422
9.7.1	Includes and Data Definitions	422
9.7.2	Buffers and Functions	422
9.7.2.1	MemoryIoBufferAllocationReset()	422
9.7.2.2	MemoryIoBufferZero()	422
9.7.2.3	MemoryGetInBuffer()	422
9.7.2.4	MemoryGetOutBuffer()	423
9.7.2.5	IsLabelProperlyFormatted()	423
9.8	Locality.c.....	424
9.8.1	Includes	424
9.8.2	LocalityGetAttributes()	424
9.9	Manufacture.c	425
9.9.1	Description	425
9.9.2	Includes and Data Definitions	425
9.9.3	Functions	425
9.9.3.1	TPM_Manufacture()	425
9.9.3.2	TPM_TearDown()	426
9.9.3.3	TpmEndSimulation()	427
9.10	Marshal.c	428
9.10.1	Introduction	428
9.10.2	Unmarshal and Marshal a Value	428
9.10.3	Unmarshal and Marshal a Union	429
9.10.4	Unmarshal and Marshal a Structure	431
9.10.5	Unmarshal and Marshal an Array	432
9.10.6	TPM2B Handling	434
9.11	MathOnByteBuffers.c	435
9.11.1	Introduction	435
9.11.2	Functions	435
9.11.2.1	UnsignedCmpB.....	435
9.11.2.2	SignedCompareB()	435
9.11.2.3	ModExpB.....	436
9.11.2.4	DivideB()	437
9.11.2.5	AdjustNumberB()	438
9.11.2.6	ShiftLeft()	438
9.11.2.7	IsNumeric().....	439
9.12	Memory.c	440
9.12.1	Description	440
9.12.2	Includes and Data Definitions	440
9.12.3	Functions	440
9.12.3.1	MemoryCopy()	440
9.12.3.2	MemoryEqual()	440

9.12.3.3	MemoryCopy2B()	441
9.12.3.4	MemoryConcat2B()	441
9.12.3.5	MemoryEqual2B()	441
9.12.3.6	MemorySet()	442
9.12.3.7	MemoryPad2B()	442
9.12.3.8	Uint16ToByteArray()	442
9.12.3.9	Uint32ToByteArray()	442
9.12.3.10	Uint64ToByteArray()	443
9.12.3.11	ByteArrayToUint8()	443
9.12.3.12	ByteArrayToUint16()	443
9.12.3.13	ByteArrayToUint32()	443
9.12.3.14	ByteArrayToUint64()	444
9.13	Power.c	445
9.13.1	Description	445
9.13.2	Includes and Data Definitions	445
9.13.3	Functions	445
9.13.3.1	TPMInit()	445
9.13.3.2	TPMRegisterStartup()	445
9.13.3.3	TPMIsStarted()	445
9.14	PropertyCap.c	447
9.14.1	Description	447
9.14.2	Includes	447
9.14.3	Functions	447
9.14.3.1	TPMPropertyIsDefined()	447
9.14.3.2	TPMCapGetProperties()	454
9.15	Response.c	456
9.15.1	Description	456
9.15.2	Includes and Defines	456
9.15.3	BuildResponseHeader()	456
9.16	ResponseCodeProcessing.c	457
9.16.1	Description	457
9.16.2	Includes and Defines	457
9.16.3	RcSafeAddToResult()	457
9.17	TpmFail.c	458
9.17.1	Includes, Defines, and Types	458
9.17.2	Typedefs	458
9.17.3	Local Functions	459
9.17.3.1	MarshalUint16()	459
9.17.3.2	MarshalUint32()	459
9.17.3.3	Unmarshal32()	459
9.17.3.4	Unmarshal16()	460
9.17.4	Public Functions	460
9.17.4.1	SetForceFailureMode()	460
9.17.4.2	TpmLogFailure()	460
9.17.4.3	TpmFail()	461
9.17.4.4	TpmFailureMode	461
9.17.4.5	UnmarshalFail()	464
10	Cryptographic Functions	465
10.1	Headers	465

10.1.1	BnValues.h	465
10.1.1.1	Introduction	465
10.1.1.2	Defines	465
10.1.2	CryptEcc.h	470
10.1.2.1	Introduction	470
10.1.2.2	Structures	470
10.1.3	CryptHash.h	471
10.1.3.1	Introduction	471
10.1.3.2	Hash-related Structures	471
10.1.3.3	HMAC State Structures	474
10.1.4	CryptRand.h	476
10.1.4.1	Introduction	476
10.1.4.2	DRBG Structures and Defines	476
10.1.5	CryptRsa.h	479
10.1.6	CryptTest.h	480
10.1.7	HashTestData.h	481
10.1.8	KdfTestData.h	482
10.1.9	RsaTestData.h	483
10.1.10	SymmetricTestData.h	489
10.1.11	SymmetricTest.h	491
10.1.11.1	Introduction	491
10.1.11.2	Symmetric Test Structures	491
10.1.12	EccTestData.h	492
10.1.13	CryptSym.h	494
10.1.13.1	Introduction	494
10.1.13.2	Includes, Defines, and Typedefs	494
10.1.14	OIDS.h	496
10.1.15	TpmAsn1.h	500
10.1.15.1	Introduction	500
10.1.15.2	Includes	500
10.1.15.3	Defined Constants	500
10.1.15.3.1	ASN.1 Universal Types (Class 00b)	500
10.1.15.4	Macros	500
10.1.15.4.1	Unmarshaling Macros	500
10.1.15.4.2	Marshaling Macros	501
10.1.15.5	Structures	501
10.1.16	X509.h	502
10.1.16.1	Introduction	502
10.1.16.2	Includes	502
10.1.16.3	Defined Constants	502
10.1.16.3.1	X509 Application-specific types	502
10.1.16.4	Structures	502
10.1.16.5	Global X509 Constants	503
10.1.17	MinMax.h	504
10.1.18	TpmAlgorithmDefines.h	505
10.2	Source	511

10.2.1	AlgorithmTests.c	511
10.2.1.1	Introduction	511
10.2.1.2	Includes and Defines	511
10.2.1.3	Hash Tests	511
10.2.1.3.1	Description	511
10.2.1.3.2	TestHash()	511
10.2.1.4	Symmetric Test Functions	513
10.2.1.4.1	Makelv()	513
10.2.1.4.2	TestSymmetricAlgorithm()	513
10.2.1.4.3	AllSymsAreDone()	514
10.2.1.4.4	AllModesAreDone()	514
10.2.1.4.5	TestSymmetric()	514
10.2.1.5	RSA Tests	515
10.2.1.5.1	Introduction	516
10.2.1.5.2	RsaKeyInitialize()	516
10.2.1.5.3	TestRsaEncryptDecrypt()	516
10.2.1.5.4	TestRsaSignAndVerify()	518
10.2.1.5.5	TestRSA()	519
10.2.1.6	ECC Tests	520
10.2.1.6.1	LoadEccParameter()	520
10.2.1.6.2	LoadEccPoint()	520
10.2.1.6.3	TestECDH()	520
10.2.1.6.4	TestEccSignAndVerify()	521
10.2.1.6.5	TestKDFa()	522
10.2.1.6.6	TestEcc()	522
10.2.1.6.7	TestAlgorithm()	523
10.2.2	BnConvert.c	527
10.2.2.1	Introduction	527
10.2.2.2	Includes	527
10.2.2.3	Functions	527
10.2.2.3.1	BnFromBytes()	527
10.2.2.3.2	BnFrom2B()	528
10.2.2.3.3	BnFromHex()	528
10.2.2.3.4	BnToBytes()	529
10.2.2.3.5	BnTo2B()	530
10.2.2.3.6	BnPointFrom2B()	530
10.2.2.3.7	BnPointTo2B()	530
10.2.3	BnMath.c	532
10.2.3.1	Introduction	532
10.2.3.2	Includes	532
10.2.3.3	Functions	532
10.2.3.3.1	AddSame()	532
10.2.3.3.2	CarryProp()	533
10.2.3.3.3	BnAdd()	533
10.2.3.3.4	BnAddWord()	534
10.2.3.3.5	SubSame()	534
10.2.3.3.6	BorrowProp()	535
10.2.3.3.7	BnSub()	535
10.2.3.3.8	BnSubWord()	535
10.2.3.3.9	BnUnsignedCmp()	536
10.2.3.3.10	BnUnsignedCmpWord()	536

10.2.3.3.11	BnModWord()	537
10.2.3.3.12	Msb()	537
10.2.3.3.13	BnMsb()	537
10.2.3.3.14	BnSizeInBits()	538
10.2.3.3.15	BnSetWord()	538
10.2.3.3.16	BnSetBit()	538
10.2.3.3.17	BnTestBit()	539
10.2.3.3.18	BnMaskBits()	539
10.2.3.3.19	BnShiftRight()	540
10.2.3.3.20	BnGetRandomBits()	540
10.2.3.3.21	BnGenerateRandomInRange()	541
10.2.4	BnMemory.c	543
10.2.4.1	Introduction	543
10.2.4.2	Includes	543
10.2.4.3	Functions	543
10.2.4.3.1	BnSetTop()	543
10.2.4.3.2	BnClearTop()	543
10.2.4.3.3	BnInitializeWord()	544
10.2.4.3.4	BnInit()	544
10.2.4.3.5	BnCopy()	544
10.2.4.3.6	BnPointCopy()	545
10.2.4.3.7	BnInitializePoint()	545
10.2.5	CryptCmac.c	546
10.2.5.1	Introduction	546
10.2.5.2	Includes, Defines, and Typedefs	546
10.2.5.3	Functions	546
10.2.5.3.1	CryptCmacStart()	546
10.2.5.3.2	CryptCmacData()	546
10.2.5.3.3	CryptCmacEnd()	547
10.2.6	CryptUtil.c	549
10.2.6.1	Introduction	549
10.2.6.2	Includes	549
10.2.6.3	Hash/HMAC Functions	549
10.2.6.3.1	CryptHmacSign()	549
10.2.6.3.2	CryptHMACVerifySignature()	549
10.2.6.3.3	CryptGenerateKeyedHash()	550
10.2.6.3.4	CryptIsSchemeAnonymous()	551
10.2.6.4	Symmetric Functions	551
10.2.6.4.1	ParmDecryptSym()	551
10.2.6.4.2	ParmEncryptSym()	552
10.2.6.4.3	CryptGenerateKeySymmetric()	553
10.2.6.4.4	CryptXORObfuscation()	554
10.2.6.5	Initialization and shut down	554
10.2.6.5.1	CryptInit()	554
10.2.6.5.2	CryptStartup()	555
10.2.6.6	Algorithm-Independent Functions	556
10.2.6.6.1	Introduction	556
10.2.6.6.2	CryptIsAsymAlgorithm()	556
10.2.6.6.3	CryptSecretEncrypt()	556
10.2.6.6.4	CryptSecretDecrypt()	558
10.2.6.6.5	CryptParameterEncryption()	562

10.2.6.6.6	CryptParameterDecryption()	563
10.2.6.6.7	CryptComputeSymmetricUnique()	564
10.2.6.6.8	CryptCreateObject()	564
10.2.6.6.9	CryptGetSignHashAlg()	566
10.2.6.6.10	CryptIsSplitSign()	567
10.2.6.6.11	CryptIsAsymSignScheme()	568
10.2.6.6.12	CryptIsAsymDecryptScheme()	569
10.2.6.6.13	CryptSelectSignScheme()	570
10.2.6.6.14	CryptSign()	571
10.2.6.6.15	CryptValidateSignature()	572
10.2.6.6.16	CryptGetTestResult	573
10.2.6.6.17	CryptIsUniqueSizeValid()	573
10.2.6.6.18	CryptIsSensitiveSizeValid()	574
10.2.6.6.19	CryptValidateKeys()	575
10.2.6.6.20	CryptAlgSetImplemented()	578
10.2.6.6.21	CryptSelectMac()	579
10.2.6.6.22	CryptMacIsValidForKey()	580
10.2.6.6.23	CryptSmacIsValidAlg()	580
10.2.6.6.24	CryptSymModelsValid()	580
10.2.7	CryptSelfTest.c	582
10.2.7.1	Introduction	582
10.2.7.2	Functions	582
10.2.7.2.1	RunSelfTest()	582
10.2.7.2.2	CryptSelfTest()	582
10.2.7.2.3	CryptIncrementalSelfTest()	583
10.2.7.2.4	CryptInitializeToTest()	584
10.2.7.2.5	CryptTestAlgorithm()	584
10.2.8	CryptEccData.c	586
10.2.9	CryptDes.c	595
10.2.9.1	Introduction	595
10.2.9.2	Includes, Defines, and Typedefs	595
10.2.9.2.1	CryptSetOddByteParity()	595
10.2.9.2.2	CryptDesIsWeakKey()	596
10.2.9.2.3	CryptDesValidateKey()	596
10.2.9.2.4	CryptGenerateKeyDes()	597
10.2.10	CryptEccKeyExchange.c	598
10.2.10.1	Introduction	598
10.2.10.2	Functions	598
10.2.10.2.1	avf1()	598
10.2.10.2.2	C_2_2_MQV()	598
10.2.10.2.3	C_2_2_ECDH()	600
10.2.10.2.4	CryptEcc2PhaseKeyExchange()	600
10.2.10.2.5	ComputeWForSM2()	601
10.2.10.2.6	avfSm2()	602
10.2.10.2.7	SM2KeyExchange()	602
10.2.11	CryptEccMain.c	604
10.2.11.1	Includes and Defines	604
10.2.11.2	Functions	604
10.2.11.2.1	CryptEcclnit()	604
10.2.11.2.2	CryptEccStartup()	604
10.2.11.2.3	ClearPoint2B(generic)	604
10.2.11.2.4	CryptEccGetParametersByCurveId()	605

10.2.11.2.5	CryptEccGetKeySizeForCurve()	605
10.2.11.2.6	GetCurveData()	605
10.2.11.2.7	CryptEccGetOID()	606
10.2.11.2.8	CryptEccGetCurveByIndex()	606
10.2.11.2.9	CryptEccGetParameter()	606
10.2.11.2.10	CryptCapGetECCCurve()	607
10.2.11.2.11	CryptGetCurveSignScheme()	608
10.2.11.2.12	CryptGenerateR()	608
10.2.11.2.13	CryptCommit()	610
10.2.11.2.14	CryptEndCommit()	610
10.2.11.2.15	CryptEccGetParameters()	610
10.2.11.2.16	BnGetCurvePrime()	611
10.2.11.2.17	BnGetCurveOrder()	611
10.2.11.2.18	BnIsOnCurve()	611
10.2.11.2.19	BnIsValidPrivateEcc()	612
10.2.11.2.20	BnPointMul()	612
10.2.11.2.21	BnEccGetPrivate()	613
10.2.11.2.22	BnEccGenerateKeyPair()	614
10.2.11.2.23	CryptEccNewKeyPair	614
10.2.11.2.24	CryptEccPointMultiply()	615
10.2.11.2.25	CryptEcclIsPointOnCurve()	616
10.2.11.2.26	CryptEccGenerateKey()	616
10.2.12	CryptEccSignature.c	618
10.2.12.1	Includes and Defines	618
10.2.12.2	Utility Functions	618
10.2.12.2.1	EcdsaDigest()	618
10.2.12.2.2	BnSchnorrSign()	618
10.2.12.3	Signing Functions	619
10.2.12.3.1	BnSignEcdsa()	619
10.2.12.3.2	BnSignEcdaa()	620
10.2.12.3.3	SchnorrReduce()	622
10.2.12.3.4	SchnorrEcc()	622
10.2.12.3.5	BnHexEqual()	623
10.2.12.3.6	BnSignEcSm2()	624
10.2.12.3.7	CryptEccSign()	625
10.2.12.3.8	BnValidateSignatureEcdsa()	627
10.2.12.3.9	BnValidateSignatureEcSm2()	628
10.2.12.3.10	BnValidateSignatureEcSchnorr()	629
10.2.12.3.11	CryptEccValidateSignature()	630
10.2.12.3.12	CryptEccCommitCompute()	631
10.2.13	CryptHash.c	633
10.2.13.1	Description	633
10.2.13.2	Includes, Defines, and Types	633
10.2.13.3	Obligatory Initialization Functions	633
10.2.13.3.1	CryptHashInit()	633
10.2.13.3.2	CryptHashStartup()	634
10.2.13.4	Hash Information Access Functions	634
10.2.13.4.1	Introduction	634
10.2.13.4.2	CryptGetHashDef()	634
10.2.13.4.3	CryptHashIsValidAlg()	634
10.2.13.4.4	CryptHashGetAlgByIndex()	635
10.2.13.4.5	CryptHashGetDigestSize()	635
10.2.13.4.6	CryptHashGetBlockSize()	635

10.2.13.4.7	CryptHashGetOid()	636
10.2.13.4.8	CryptHashGetContextAlg()	636
10.2.13.5	State Import and Export	636
10.2.13.5.1	CryptHashCopyState	636
10.2.13.5.2	CryptHashExportState()	637
10.2.13.5.3	CryptHashImportState()	637
10.2.13.6	State Modification Functions	638
10.2.13.6.1	HashEnd()	638
10.2.13.6.2	CryptHashStart()	639
10.2.13.6.3	CryptDigestUpdate()	639
10.2.13.6.4	CryptHashEnd()	640
10.2.13.6.5	CryptHashBlock()	640
10.2.13.6.6	CryptDigestUpdate2B()	640
10.2.13.6.7	CryptHashEnd2B()	641
10.2.13.6.8	CryptDigestUpdateInt()	641
10.2.13.7	HMAC Functions	642
10.2.13.7.1	CryptHmacStart()	642
10.2.13.7.2	CryptHmacEnd()	643
10.2.13.7.3	CryptHmacStart2B()	643
10.2.13.7.4	CryptHmacEnd2B()	644
10.2.13.8	Mask and Key Generation Functions	644
10.2.13.8.1	CryptMGF1()	644
10.2.13.8.2	CryptKDFa()	645
10.2.13.8.3	CryptKDFe()	647
10.2.14	CryptPrime.c	649
10.2.14.1	Introduction	649
10.2.14.2	Functions	649
10.2.14.2.1	Root2()	649
10.2.14.2.2	IsPrimeInt()	649
10.2.14.2.3	BnIsProbablyPrime()	650
10.2.14.2.4	MillerRabinRounds()	650
10.2.14.2.5	MillerRabin()	651
10.2.14.2.6	RsaCheckPrime()	652
10.2.14.2.7	AdjustPrimeCandidate()	653
10.2.14.2.8	BnGeneratePrimeForRSA()	653
10.2.15	CryptPrimeSieve.c	655
10.2.15.1	Includes and defines	655
10.2.15.2	Functions	655
10.2.15.2.1	RsaAdjustPrimeLimit()	655
10.2.15.2.2	RsaNextPrime()	655
10.2.15.2.3	BitsInArray()	657
10.2.15.2.4	FindNthSetBit()	657
10.2.15.2.5	PrimeSieve()	658
10.2.15.2.6	SetFieldSize()	660
10.2.15.2.7	PrimeSelectWithSieve()	660
10.2.16	CryptRand.c	664
10.2.16.1	Introduction	664
10.2.16.2	Derivation Functions	664
10.2.16.2.1	Description	664

10.2.16.2.2	Derivation Function Defines and Structures	665
10.2.16.2.3	DfCompute()	665
10.2.16.2.4	DfStart()	665
10.2.16.2.5	DfUpdate()	666
10.2.16.2.6	DfEnd()	666
10.2.16.2.7	DfBuffer()	667
10.2.16.2.8	DRBG_GetEntropy()	667
10.2.16.2.9	IncrementIv()	668
10.2.16.2.10	EncryptDRBG()	668
10.2.16.2.11	DRBG_Update()	669
10.2.16.2.12	DRBG_Reseed()	670
10.2.16.2.13	DRBG_SelfTest()	671
10.2.16.3	Public Interface	672
10.2.16.3.1	Description	672
10.2.16.3.2	CryptRandomStir()	672
10.2.16.3.3	CryptRandomGenerate()	673
10.2.16.3.4	DRBG_Generate()	678
10.2.16.3.5	DRBG_Instantiate()	680
10.2.16.3.6	DRBG_Uninstantiate()	681
10.2.17	CryptRsa.c	682
10.2.17.1	Introduction	682
10.2.17.2	Includes	682
10.2.17.3	Obligatory Initialization Functions	682
10.2.17.3.1	CryptRsaInit()	682
10.2.17.3.2	CryptRsaStartup()	682
10.2.17.4	Internal Functions	682
10.2.17.4.1	RsaInitializeExponent()	682
10.2.17.4.2	MakePgreaterThanQ()	683
10.2.17.4.3	PackExponent()	683
10.2.17.4.4	UnpackExponent()	684
10.2.17.4.5	ComputePrivateExponent()	684
10.2.17.4.6	RsaPrivateKeyOp()	685
10.2.17.4.7	RSAEP()	685
10.2.17.4.8	RSADP()	686
10.2.17.4.9	OaepEncode()	687
10.2.17.4.10	OaepDecode()	688
10.2.17.4.11	PKCS1v1_5Encode()	689
10.2.17.4.12	RS_AES_Decode()	690
10.2.17.4.13	CryptRsaPssSaltSize()	691
10.2.17.4.14	PssEncode()	691
10.2.17.4.15	PssDecode()	692
10.2.17.4.16	MakeDerTag()	694
10.2.17.4.17	RSASSA_Encode()	695
10.2.17.4.18	RSASSA_Decode()	696
10.2.17.5	Externally Accessible Functions	697
10.2.17.5.1	CryptRsaSelectScheme()	697
10.2.17.5.2	CryptRsaLoadPrivateExponent()	697
10.2.17.5.3	CryptRsaEncrypt()	698
10.2.17.5.4	CryptRsaDecrypt()	700
10.2.17.5.5	CryptRsaSign()	701
10.2.17.5.6	CryptRsaValidateSignature()	702
10.2.17.5.7	CryptRsaGenerateKey()	703
10.2.18	CryptSmac.c	706

10.2.18.1 Introduction	706
10.2.18.2 Includes, Defines, and Typedefs	706
10.2.18.2.1 CryptSmacStart()	706
10.2.18.2.2 CryptMacStart()	706
10.2.18.2.3 CryptMacEnd()	707
10.2.18.2.4 CryptMacEnd2B()	707
10.2.19 CryptSym.c	708
10.2.19.1 Introduction	708
10.2.19.2 Includes, Defines, and Typedefs	708
10.2.19.3 Initialization and Data Access Functions	708
10.2.19.3.1 CryptSymInit()	708
10.2.19.3.2 CryptSymStartup()	708
10.2.19.3.3 CryptGetSymmetricBlockSize()	708
10.2.19.4 Symmetric Encryption	709
10.2.19.4.1 CryptSymmetricDecrypt()	712
10.2.19.4.2 CryptSymKeyValidate()	715
10.2.20 PrimeData.c	716
10.2.21 RsaKeyCache.c	722
10.2.21.1 Introduction	722
10.2.21.2 Includes, Types, Locals, and Defines	722
10.2.21.2.1 RsaKeyCacheControl()	723
10.2.21.2.2 InitializeKeyCache()	723
10.2.21.2.3 KeyCacheLoaded()	724
10.2.21.2.4 GetCachedRsaKey()	725
10.2.22 Ticket.c	726
10.2.22.1 Introduction	726
10.2.22.2 Includes	726
10.2.22.3 Functions	726
10.2.22.3.1 TicketIsSafe()	726
10.2.22.3.2 TicketComputeVerified()	726
10.2.22.3.3 TicketComputeAuth()	727
10.2.22.3.4 TicketComputeHashCheck()	728
10.2.22.3.5 TicketComputeCreation()	728
10.2.23 TpmAsn1.c	730
10.2.23.1 Includes	730
10.2.23.2 Unmarshaling Functions	730
10.2.23.2.1 ASN1UnmarshalContextInitialize()	730
10.2.23.2.2 ASN1DecodeLength()	730
10.2.23.2.3 ASN1NextTag()	731
10.2.23.2.4 ASN1GetBitStringValue()	732
10.2.23.3 Marshaling Functions	732
10.2.23.3.1 Introduction	732
10.2.23.3.2 ASN1InitialializeMarshalContext()	733
10.2.23.3.3 ASN1StartMarshalContext()	733
10.2.23.3.4 ASN1EndMarshalContext()	734
10.2.23.3.5 ASN1EndEncapsulation()	734
10.2.23.3.6 ASN1PushByte()	734
10.2.23.3.7 ASN1PushBytes()	735
10.2.23.3.8 ASN1PushNull()	735

10.2.23.3.9	ASN1PushLength()	736
10.2.23.3.10	ASN1PushTagAndLength()	736
10.2.23.3.11	ASN1PushTaggedOctetString()	737
10.2.23.3.12	ASN1PushUINT()	737
10.2.23.3.13	ASN1PushInteger	737
10.2.23.3.14	ASN1PushOID()	738
10.2.24	X509_ECC.c	739
10.2.24.1	Includes	739
10.2.24.2	Functions	739
10.2.24.2.1	X509PushPoint()	739
10.2.24.2.2	X509AddSigningAlgorithmECC()	739
10.2.24.2.3	X509AddPublicECC()	740
10.2.25	X509_RSA.c	742
10.2.25.1	Includes	742
10.2.25.2	Functions	742
10.2.25.2.1	X509AddSigningAlgorithmRSA()	742
10.2.25.2.2	X509AddPublicRSA()	744
10.2.26	X509_spt.c	746
10.2.26.1	Includes	746
10.2.26.2	Unmarshaling Functions	746
10.2.26.2.1	X509FindExtensionOID()	746
10.2.26.2.2	X509GetExtensionBits()	747
10.2.26.2.3	X509ProcessExtensions()	747
10.2.26.3	Marshaling Functions	749
10.2.26.3.1	X509AddSigningAlgorithm()	749
10.2.26.3.2	X509AddPublicKey()	749
10.2.26.3.3	X509PushAlgorithmIdentifierSequence()	750
10.2.27	AC_spt.c	751
10.2.27.1	Includes	751
10.2.27.1.1	AcToCapabilities()	751
10.2.27.1.2	AcIsAccessible()	751
10.2.27.1.3	AcCapabilitiesGet()	751
10.2.27.1.4	AcSendObject()	752
Annex A (informative)	Implementation Dependent	754
A.1	Introduction	754
A.2	TpmProfile.h	754
Annex B (informative)	Library-Specific	767
B.1	Introduction	767
B.2	OpenSSL-Specific Files	768
B.2.1	Introduction	768
B.2.2	Header Files	768
B.2.2.1	TpmToOsslHash.h	768
B.2.2.1.1	Introduction	768
B.2.2.1.2	Links to the OpenSSL HASH code	768
B.2.2.2	TpmToOsslMath.h	771
B.2.2.2.1	Introduction	771
B.2.2.2.2	Macros and Defines	771

B.2.2.3. TpmToOsslSym.h	773
B.2.2.3.1. Introduction	773
B.2.2.3.2. Links to the OpenSSL AES code.....	773
B.2.3. Source Files	775
B.2.3.1. TpmToOsslDesSupport.c.....	775
B.2.3.1.1. Introduction	775
B.2.3.1.2. Defines and Includes	775
B.2.3.1.3. Functions.....	775
B.2.3.2. TpmToOsslMath.c	777
B.2.3.2.1. Introduction	777
B.2.3.2.2. Introduction	777
B.2.3.2.3. Includes and Defines	777
B.2.3.2.4. Functions.....	777
B.2.3.2.5. BnEccAdd()	787
B.2.3.3. TpmToOsslSupport.c	788
B.2.3.3.1. Introduction	788
B.2.3.3.2. Defines and Includes	788
Annex C (informative) Simulation Environment	790
C.1 Introduction	790
C.2 Cancel.c.....	790
C.2.1. Description	790
C.2.2. Includes, Typedefs, Structures, and Defines	790
C.2.3. Functions	790
C.2.3.1. _plat_IsCanceled()	790
C.2.3.2. _plat_SetCancel().....	790
C.2.3.3. _plat_ClearCancel().....	791
C.3 Clock.c.....	792
C.3.1. Description	792
C.3.2. Includes and Data Definitions	792
C.3.3. Simulator Functions.....	792
C.3.3.1. Introduction	792
C.3.3.2. _plat_TimerReset().....	792
C.3.3.3. _plat_TimerRestart().....	792
C.3.4. Functions Used by TPM	793
C.3.4.1. Introduction	793
C.3.4.2. _plat_RealTime()	793
C.3.4.3. _plat_TimerRead().....	793
C.3.4.4. _plat_TimerWasReset()	795
C.3.4.5. _plat_TimerWasStopped()	795
C.3.4.6. _plat_ClockAdjustRate()	795
C.4 Entropy.c.....	797
C.4.1. Includes and Local Values	797
C.4.2. Functions	797
C.4.2.1. rand32()	797
C.4.2.2. _plat_GetEntropy()	797
C.5 LocalityPlat.c.....	800
C.5.1. Includes	800

C.5.2. Functions	800
C.5.2.1. _plat_LocalityGet()	800
C.5.2.2. _plat_LocalitySet()	800
C.6 NVMem.c	801
C.6.1. Description	801
C.6.2. Includes and Local	801
C.6.3. Functions	801
C.6.3.1. NvFileOpen()	801
C.6.3.2. NvFileCommit()	801
C.6.3.3. NvFileSize()	802
C.6.3.4. _plat_NvErrors()	802
C.6.3.5. _plat_NVEnable()	803
C.6.3.6. _plat_NVDisable()	804
C.6.3.7. _plat_IsNvAvailable()	804
C.6.3.8. _plat_NvMemoryRead()	804
C.6.3.9. _plat_NvIsDifferent()	805
C.6.3.10. _plat_NvMemoryWrite()	805
C.6.3.11. _plat_NvMemoryClear()	805
C.6.3.12. _plat_NvMemoryMove()	806
C.6.3.13. _plat_NvCommit()	806
C.6.3.14. _plat_SetNvAvail()	806
C.6.3.15. _plat_ClearNvAvail()	806
C.7 PowerPlat.c	808
C.7.1. Includes and Function Prototypes	808
C.7.2. Functions	808
C.7.2.1. _plat_Signal_PowerOn()	808
C.7.2.2. _plat_WasPowerLost()	808
C.7.2.3. _plat_Signal_Reset()	808
C.7.2.4. _plat_Signal_PowerOff()	809
C.8 PlatformData.h	810
C.9 PlatformData.c	812
C.9.1. Description	812
C.9.2. Includes	812
C.10 PPPlat.c	813
C.10.1. Description	813
C.10.2. Includes	813
C.10.3. Functions	813
C.10.3.1. _plat_PhysicalPresenceAsserted()	813
C.10.3.2. _plat_Signal_PhysicalPresenceOn()	813
C.10.3.3. _plat_Signal_PhysicalPresenceOff()	813
C.11 RunCommand.c	814
C.11.1. Introduction	814
C.11.2. Includes and locals	814
C.11.3. Functions	814
C.11.3.1. _plat_RunCommand()	814
C.11.3.2. _plat_Fail()	814
C.12 Unique.c	815
C.12.1. Introduction	815
C.12.2. Includes	815

C.12.3. _plat__GetUnique()	815
C.13 DebugHelpers.c.....	816
C.13.1. Description	816
C.13.2. Includes and Local	816
C.13.2.1. DebugFileOpen()	816
C.14 Platform.h	818
Annex D (informative) Remote Procedure Interface	819
D.1 Introduction	819
D.2 TpmTcpProtocol.h	820
D.2.1. Introduction	820
D.2.2. Typedefs and Defines	820
D.2.3. TPM Commands	820
D.2.4. Enumerations and Structures	820
D.3 TcpServer.c.....	822
D.3.1. Description	822
D.3.2. Includes, Locals, Defines and Function Prototypes	822
D.3.3. Functions	822
D.3.3.1. CreateSocket()	822
D.3.3.2. PlatformServer()	823
D.3.3.3. PlatformSvcRoutine().....	824
D.3.3.4. PlatformSignalService()	825
D.3.3.5. RegularCommandService().....	826
D.3.3.6. StartTcpServer()	826
D.3.3.7. ReadBytes()	827
D.3.3.8. WriteBytes()	827
D.3.3.9. WriteUINT32()	828
D.3.3.10. ReadVarBytes()	828
D.3.3.11. WriteVarBytes()	829
D.3.3.12. TpmServer()	829
D.4 TPMCmdp.c	832
D.4.1. Description	832
D.4.2. Includes and Data Definitions	832
D.4.3. Functions	832
D.4.3.1. Signal_PowerOn()	832
D.4.3.2. Signal_Restart()	833
D.4.3.3. Signal_PowerOff()	833
D.4.3.4. _rpc__ForceFailureMode().....	833
D.4.3.5. _rpc__Signal_PhysicalPresenceOn().....	833
D.4.3.6. _rpc__Signal_PhysicalPresenceOff().....	834
D.4.3.7. _rpc__Signal_Hash_Start().....	834
D.4.3.8. _rpc__Signal_Hash_Data().....	834
D.4.3.9. _rpc__Signal_HashEnd()	834
D.4.3.10. _rpc__Send_Command()	835
D.4.3.11. _rpc__Signal_CancelOn().....	835
D.4.3.12. _rpc__Signal_CancelOff().....	835
D.4.3.13. _rpc__Signal_NvOn()	836
D.4.3.14. _rpc__Signal_NvOff()	836
D.4.3.15. _rpc__RsaKeyCacheControl().....	836
D.4.3.16. _rpc__Shutdown()	837
D.5 TPMCmnds.c.....	838
D.5.1. Description	838

D.5.2. Includes, Defines, Data Definitions, and Function Prototypes	838
D.5.3. Functions	838
D.5.3.1. Usage()	838
D.5.3.2. main().....	838

DRAFT

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 to 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 tables in the TPM 2.0 Part 2 Annexes are constructed so that they can be processed by a program. The program that processes these tables in the TPM 2.0 Part 2 Annexes is called "The TPM 2.0 Part 2 Configuration Parser."

The tables in the TPM 2.0 Part 2 Annexes determine the configuration of a TPM implementation. These tables may be modified by an implementer to describe the algorithms and commands to be executed in by a specific implementation as well as to set implementation limits such as the number of PCR, sizes of buffers, etc.

The TPM 2.0 Part 2 Configuration Parser produces a set of structures and definitions that are used by the TPM 2.0 Part 2 Structure Parser.

4.2 Structure Parser

4.2.1 Introduction

The program that processes the tables in TPM 2.0 Part 2 (other than the table in the annexes) 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 sections 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**.

4.3 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  #ifndef _Create_FP_H_
3  #define _Create_FP_H_
```

Input structure definition

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

Output structure definition

```
11 typedef struct {
12     TPM2B_PRIVATE      outPrivate;
13     TPM2B_PUBLIC        outPublic;
14     TPM2B_CREATION_DATA creationData;
```

```

15     TPM2B_DIGEST          creationHash;
16     TPMT_TK_CREATION      creationTicket;
17 } Create_Out;

```

Response code modifiers

```

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

```

Function prototype

```

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

- allocate all structures as packed to a byte boundary (this may not be possible if the processor does not handle unaligned accesses); or
- 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 section 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  typedef uint8_t      UINT8;
7  typedef uint8_t      BYTE;
8  typedef int8_t       INT8;
9  typedef int          BOOL;
10 typedef uint16_t     UINT16;
11 typedef int16_t      INT16;
12 typedef uint32_t     UINT32;
13 typedef int32_t      INT32;
14 typedef uint64_t     UINT64;
15 typedef int64_t      INT64;
16 #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  #define      MAX_CAP_DATA      (MAX_CAP_BUFFER - sizeof(TPM_CAP) - sizeof(UINT32))
4  #define      MAX_CAP_ALGS      (MAX_CAP_DATA / sizeof(TPMS_ALG_PROPERTY))
5  #define      MAX_CAP_HANDLES   (MAX_CAP_DATA / sizeof(TPM_HANDLE))
6  #define      MAX_CAP_CC        (MAX_CAP_DATA / sizeof(TPM_CC))
7  #define      MAX_TPM_PROPERTIES (MAX_CAP_DATA / sizeof(TPMS_TAGGED_PROPERTY))
8  #define      MAX_PCR_PROPERTIES (MAX_CAP_DATA / sizeof(TPMS_TAGGED_PCR_SELECT))
9  #define      MAX_ECC_CURVES    (MAX_CAP_DATA / sizeof(TPM_ECC_CURVE))
10 #define      MAX_TAGGED_POLICIES (MAX_CAP_DATA / sizeof(TPMS_TAGGED_POLICY))
11 #define      MAX_AC_CAPABILITIES (MAX_CAP_DATA / sizeof(TPMS_AC_OUTPUT))
12 #endif

```


5.4 CommandAttributeData.h

This file should only be included by CommandCodeAttributes.c

```

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

```

8  const TPMA_CC    s_ccAttr [] = {
9  #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
10     TPMA_CC_INITIALIZER(0x011F, 0, 1, 0, 0, 2, 0, 0, 0),
11 #endif
12 #if (PAD_LIST || CC_EvictControl)
13     TPMA_CC_INITIALIZER(0x0120, 0, 1, 0, 0, 2, 0, 0, 0),
14 #endif
15 #if (PAD_LIST || CC_HierarchyControl)
16     TPMA_CC_INITIALIZER(0x0121, 0, 1, 1, 0, 1, 0, 0, 0),
17 #endif
18 #if (PAD_LIST || CC_NV_UndefineSpace)
19     TPMA_CC_INITIALIZER(0x0122, 0, 1, 0, 0, 2, 0, 0, 0),
20 #endif
21 #if (PAD_LIST )
22     TPMA_CC_INITIALIZER(0x0123, 0, 0, 0, 0, 0, 0, 0, 0),
23 #endif
24 #if (PAD_LIST || CC_ChangeEPS)
25     TPMA_CC_INITIALIZER(0x0124, 0, 1, 1, 0, 1, 0, 0, 0),
26 #endif
27 #if (PAD_LIST || CC_ChangePPS)
28     TPMA_CC_INITIALIZER(0x0125, 0, 1, 1, 0, 1, 0, 0, 0),
29 #endif
30 #if (PAD_LIST || CC_Clear)
31     TPMA_CC_INITIALIZER(0x0126, 0, 1, 1, 0, 1, 0, 0, 0),
32 #endif
33 #if (PAD_LIST || CC_ClearControl)
34     TPMA_CC_INITIALIZER(0x0127, 0, 1, 0, 0, 1, 0, 0, 0),
35 #endif
36 #if (PAD_LIST || CC_ClockSet)
37     TPMA_CC_INITIALIZER(0x0128, 0, 1, 0, 0, 1, 0, 0, 0),
38 #endif
39 #if (PAD_LIST || CC_HierarchyChangeAuth)
40     TPMA_CC_INITIALIZER(0x0129, 0, 1, 0, 0, 1, 0, 0, 0),
41 #endif
42 #if (PAD_LIST || CC_NV_DefineSpace)
43     TPMA_CC_INITIALIZER(0x012A, 0, 1, 0, 0, 1, 0, 0, 0),
44 #endif
45 #if (PAD_LIST || CC_PCR_Allocate)
46     TPMA_CC_INITIALIZER(0x012B, 0, 1, 0, 0, 1, 0, 0, 0),
47 #endif
48 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
49     TPMA_CC_INITIALIZER(0x012C, 0, 1, 0, 0, 1, 0, 0, 0),
50 #endif
51 #if (PAD_LIST || CC_PP_Commands)
52     TPMA_CC_INITIALIZER(0x012D, 0, 1, 0, 0, 1, 0, 0, 0),
53 #endif
54 #if (PAD_LIST || CC_SetPrimaryPolicy)
55     TPMA_CC_INITIALIZER(0x012E, 0, 1, 0, 0, 1, 0, 0, 0),
56 #endif

```

```

57  #if (PAD_LIST || CC_FieldUpgradeStart)
58      TPMA_CC_INITIALIZER(0x012F, 0, 0, 0, 0, 2, 0, 0, 0),
59  #endif
60  #if (PAD_LIST || CC_ClockRateAdjust)
61      TPMA_CC_INITIALIZER(0x0130, 0, 0, 0, 0, 1, 0, 0, 0),
62  #endif
63  #if (PAD_LIST || CC_CreatePrimary)
64      TPMA_CC_INITIALIZER(0x0131, 0, 0, 0, 0, 1, 1, 0, 0),
65  #endif
66  #if (PAD_LIST || CC_NV_GlobalWriteLock)
67      TPMA_CC_INITIALIZER(0x0132, 0, 1, 0, 0, 1, 0, 0, 0),
68  #endif
69  #if (PAD_LIST || CC_GetCommandAuditDigest)
70      TPMA_CC_INITIALIZER(0x0133, 0, 1, 0, 0, 2, 0, 0, 0),
71  #endif
72  #if (PAD_LIST || CC_NV_Increment)
73      TPMA_CC_INITIALIZER(0x0134, 0, 1, 0, 0, 2, 0, 0, 0),
74  #endif
75  #if (PAD_LIST || CC_NV_SetBits)
76      TPMA_CC_INITIALIZER(0x0135, 0, 1, 0, 0, 2, 0, 0, 0),
77  #endif
78  #if (PAD_LIST || CC_NV_Extend)
79      TPMA_CC_INITIALIZER(0x0136, 0, 1, 0, 0, 2, 0, 0, 0),
80  #endif
81  #if (PAD_LIST || CC_NV_Write)
82      TPMA_CC_INITIALIZER(0x0137, 0, 1, 0, 0, 2, 0, 0, 0),
83  #endif
84  #if (PAD_LIST || CC_NV_WriteLock)
85      TPMA_CC_INITIALIZER(0x0138, 0, 1, 0, 0, 2, 0, 0, 0),
86  #endif
87  #if (PAD_LIST || CC_DictionaryAttackLockReset)
88      TPMA_CC_INITIALIZER(0x0139, 0, 1, 0, 0, 1, 0, 0, 0),
89  #endif
90  #if (PAD_LIST || CC_DictionaryAttackParameters)
91      TPMA_CC_INITIALIZER(0x013A, 0, 1, 0, 0, 1, 0, 0, 0),
92  #endif
93  #if (PAD_LIST || CC_NV_ChangeAuth)
94      TPMA_CC_INITIALIZER(0x013B, 0, 1, 0, 0, 1, 0, 0, 0),
95  #endif
96  #if (PAD_LIST || CC_PCR_Event)
97      TPMA_CC_INITIALIZER(0x013C, 0, 1, 0, 0, 1, 0, 0, 0),
98  #endif
99  #if (PAD_LIST || CC_PCR_Reset)
100      TPMA_CC_INITIALIZER(0x013D, 0, 1, 0, 0, 1, 0, 0, 0),
101  #endif
102  #if (PAD_LIST || CC_SequenceComplete)
103      TPMA_CC_INITIALIZER(0x013E, 0, 0, 0, 1, 1, 0, 0, 0),
104  #endif
105  #if (PAD_LIST || CC_SetAlgorithmSet)
106      TPMA_CC_INITIALIZER(0x013F, 0, 1, 0, 0, 1, 0, 0, 0),
107  #endif
108  #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
109      TPMA_CC_INITIALIZER(0x0140, 0, 1, 0, 0, 1, 0, 0, 0),
110  #endif
111  #if (PAD_LIST || CC_FieldUpgradeData)
112      TPMA_CC_INITIALIZER(0x0141, 0, 1, 0, 0, 0, 0, 0, 0),
113  #endif
114  #if (PAD_LIST || CC_IncrementalSelfTest)
115      TPMA_CC_INITIALIZER(0x0142, 0, 1, 0, 0, 0, 0, 0, 0),
116  #endif
117  #if (PAD_LIST || CC_SelfTest)
118      TPMA_CC_INITIALIZER(0x0143, 0, 1, 0, 0, 0, 0, 0, 0),
119  #endif
120  #if (PAD_LIST || CC_Startup)
121      TPMA_CC_INITIALIZER(0x0144, 0, 1, 0, 0, 0, 0, 0, 0),
122  #endif

```

```
123 #if (PAD_LIST || CC_Shutdown)
124     TPMA_CC_INITIALIZER(0x0145, 0, 1, 0, 0, 0, 0, 0, 0),
125 #endif
126 #if (PAD_LIST || CC_StirRandom)
127     TPMA_CC_INITIALIZER(0x0146, 0, 1, 0, 0, 0, 0, 0, 0),
128 #endif
129 #if (PAD_LIST || CC_ActivateCredential)
130     TPMA_CC_INITIALIZER(0x0147, 0, 0, 0, 0, 2, 0, 0, 0),
131 #endif
132 #if (PAD_LIST || CC_Certify)
133     TPMA_CC_INITIALIZER(0x0148, 0, 0, 0, 0, 2, 0, 0, 0),
134 #endif
135 #if (PAD_LIST || CC_PolicyNV)
136     TPMA_CC_INITIALIZER(0x0149, 0, 0, 0, 0, 3, 0, 0, 0),
137 #endif
138 #if (PAD_LIST || CC_CertifyCreation)
139     TPMA_CC_INITIALIZER(0x014A, 0, 0, 0, 0, 2, 0, 0, 0),
140 #endif
141 #if (PAD_LIST || CC_Duplicate)
142     TPMA_CC_INITIALIZER(0x014B, 0, 0, 0, 0, 2, 0, 0, 0),
143 #endif
144 #if (PAD_LIST || CC_GetTime)
145     TPMA_CC_INITIALIZER(0x014C, 0, 0, 0, 0, 2, 0, 0, 0),
146 #endif
147 #if (PAD_LIST || CC_GetSessionAuditDigest)
148     TPMA_CC_INITIALIZER(0x014D, 0, 0, 0, 0, 3, 0, 0, 0),
149 #endif
150 #if (PAD_LIST || CC_NV_Read)
151     TPMA_CC_INITIALIZER(0x014E, 0, 0, 0, 0, 2, 0, 0, 0),
152 #endif
153 #if (PAD_LIST || CC_NV_ReadLock)
154     TPMA_CC_INITIALIZER(0x014F, 0, 1, 0, 0, 2, 0, 0, 0),
155 #endif
156 #if (PAD_LIST || CC_ObjectChangeAuth)
157     TPMA_CC_INITIALIZER(0x0150, 0, 0, 0, 0, 2, 0, 0, 0),
158 #endif
159 #if (PAD_LIST || CC_PolicySecret)
160     TPMA_CC_INITIALIZER(0x0151, 0, 0, 0, 0, 2, 0, 0, 0),
161 #endif
162 #if (PAD_LIST || CC_Rewrap)
163     TPMA_CC_INITIALIZER(0x0152, 0, 0, 0, 0, 2, 0, 0, 0),
164 #endif
165 #if (PAD_LIST || CC_Create)
166     TPMA_CC_INITIALIZER(0x0153, 0, 0, 0, 0, 1, 0, 0, 0),
167 #endif
168 #if (PAD_LIST || CC_ECDH_ZGen)
169     TPMA_CC_INITIALIZER(0x0154, 0, 0, 0, 0, 1, 0, 0, 0),
170 #endif
171 #if (PAD_LIST || (CC_HMAC || CC_MAC))
172     TPMA_CC_INITIALIZER(0x0155, 0, 0, 0, 0, 1, 0, 0, 0),
173 #endif
174 #if (PAD_LIST || CC_Import)
175     TPMA_CC_INITIALIZER(0x0156, 0, 0, 0, 0, 1, 0, 0, 0),
176 #endif
177 #if (PAD_LIST || CC_Load)
178     TPMA_CC_INITIALIZER(0x0157, 0, 0, 0, 0, 1, 1, 0, 0),
179 #endif
180 #if (PAD_LIST || CC_Quote)
181     TPMA_CC_INITIALIZER(0x0158, 0, 0, 0, 0, 1, 0, 0, 0),
182 #endif
183 #if (PAD_LIST || CC_RSA_Decrypt)
184     TPMA_CC_INITIALIZER(0x0159, 0, 0, 0, 0, 1, 0, 0, 0),
185 #endif
186 #if (PAD_LIST )
187     TPMA_CC_INITIALIZER(0x015A, 0, 0, 0, 0, 0, 0, 0, 0),
188 #endif
```

```

189 #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
190     TPMA_CC_INITIALIZER(0x015B, 0, 0, 0, 0, 1, 1, 0, 0),
191 #endif
192 #if (PAD_LIST || CC_SequenceUpdate)
193     TPMA_CC_INITIALIZER(0x015C, 0, 0, 0, 0, 1, 0, 0, 0),
194 #endif
195 #if (PAD_LIST || CC_Sign)
196     TPMA_CC_INITIALIZER(0x015D, 0, 0, 0, 0, 1, 0, 0, 0),
197 #endif
198 #if (PAD_LIST || CC_Unseal)
199     TPMA_CC_INITIALIZER(0x015E, 0, 0, 0, 0, 1, 0, 0, 0),
200 #endif
201 #if (PAD_LIST )
202     TPMA_CC_INITIALIZER(0x015F, 0, 0, 0, 0, 0, 0, 0, 0),
203 #endif
204 #if (PAD_LIST || CC_PolicySigned)
205     TPMA_CC_INITIALIZER(0x0160, 0, 0, 0, 0, 2, 0, 0, 0),
206 #endif
207 #if (PAD_LIST || CC_ContextLoad)
208     TPMA_CC_INITIALIZER(0x0161, 0, 0, 0, 0, 0, 1, 0, 0),
209 #endif
210 #if (PAD_LIST || CC_ContextSave)
211     TPMA_CC_INITIALIZER(0x0162, 0, 0, 0, 0, 1, 0, 0, 0),
212 #endif
213 #if (PAD_LIST || CC_ECDH_KeyGen)
214     TPMA_CC_INITIALIZER(0x0163, 0, 0, 0, 0, 1, 0, 0, 0),
215 #endif
216 #if (PAD_LIST || CC_EncryptDecrypt)
217     TPMA_CC_INITIALIZER(0x0164, 0, 0, 0, 0, 1, 0, 0, 0),
218 #endif
219 #if (PAD_LIST || CC_FlushContext)
220     TPMA_CC_INITIALIZER(0x0165, 0, 0, 0, 0, 0, 0, 0, 0),
221 #endif
222 #if (PAD_LIST )
223     TPMA_CC_INITIALIZER(0x0166, 0, 0, 0, 0, 0, 0, 0, 0),
224 #endif
225 #if (PAD_LIST || CC_LoadExternal)
226     TPMA_CC_INITIALIZER(0x0167, 0, 0, 0, 0, 0, 1, 0, 0),
227 #endif
228 #if (PAD_LIST || CC_MakeCredential)
229     TPMA_CC_INITIALIZER(0x0168, 0, 0, 0, 0, 1, 0, 0, 0),
230 #endif
231 #if (PAD_LIST || CC_NV_ReadPublic)
232     TPMA_CC_INITIALIZER(0x0169, 0, 0, 0, 0, 1, 0, 0, 0),
233 #endif
234 #if (PAD_LIST || CC_PolicyAuthorize)
235     TPMA_CC_INITIALIZER(0x016A, 0, 0, 0, 0, 1, 0, 0, 0),
236 #endif
237 #if (PAD_LIST || CC_PolicyAuthValue)
238     TPMA_CC_INITIALIZER(0x016B, 0, 0, 0, 0, 1, 0, 0, 0),
239 #endif
240 #if (PAD_LIST || CC_PolicyCommandCode)
241     TPMA_CC_INITIALIZER(0x016C, 0, 0, 0, 0, 1, 0, 0, 0),
242 #endif
243 #if (PAD_LIST || CC_PolicyCounterTimer)
244     TPMA_CC_INITIALIZER(0x016D, 0, 0, 0, 0, 1, 0, 0, 0),
245 #endif
246 #if (PAD_LIST || CC_PolicyCpHash)
247     TPMA_CC_INITIALIZER(0x016E, 0, 0, 0, 0, 1, 0, 0, 0),
248 #endif
249 #if (PAD_LIST || CC_PolicyLocality)
250     TPMA_CC_INITIALIZER(0x016F, 0, 0, 0, 0, 1, 0, 0, 0),
251 #endif
252 #if (PAD_LIST || CC_PolicyNameHash)
253     TPMA_CC_INITIALIZER(0x0170, 0, 0, 0, 0, 1, 0, 0, 0),
254 #endif

```

```
255 #if (PAD_LIST || CC_PolicyOR)
256     TPMA_CC_INITIALIZER(0x0171, 0, 0, 0, 0, 1, 0, 0, 0),
257 #endif
258 #if (PAD_LIST || CC_PolicyTicket)
259     TPMA_CC_INITIALIZER(0x0172, 0, 0, 0, 0, 1, 0, 0, 0),
260 #endif
261 #if (PAD_LIST || CC_ReadPublic)
262     TPMA_CC_INITIALIZER(0x0173, 0, 0, 0, 0, 1, 0, 0, 0),
263 #endif
264 #if (PAD_LIST || CC_RSA_Encrypt)
265     TPMA_CC_INITIALIZER(0x0174, 0, 0, 0, 0, 1, 0, 0, 0),
266 #endif
267 #if (PAD_LIST )
268     TPMA_CC_INITIALIZER(0x0175, 0, 0, 0, 0, 0, 0, 0, 0),
269 #endif
270 #if (PAD_LIST || CC_StartAuthSession)
271     TPMA_CC_INITIALIZER(0x0176, 0, 0, 0, 0, 2, 1, 0, 0),
272 #endif
273 #if (PAD_LIST || CC_VerifySignature)
274     TPMA_CC_INITIALIZER(0x0177, 0, 0, 0, 0, 1, 0, 0, 0),
275 #endif
276 #if (PAD_LIST || CC_ECC_Parameters)
277     TPMA_CC_INITIALIZER(0x0178, 0, 0, 0, 0, 0, 0, 0, 0),
278 #endif
279 #if (PAD_LIST || CC_FirmwareRead)
280     TPMA_CC_INITIALIZER(0x0179, 0, 0, 0, 0, 0, 0, 0, 0),
281 #endif
282 #if (PAD_LIST || CC_GetCapability)
283     TPMA_CC_INITIALIZER(0x017A, 0, 0, 0, 0, 0, 0, 0, 0),
284 #endif
285 #if (PAD_LIST || CC_GetRandom)
286     TPMA_CC_INITIALIZER(0x017B, 0, 0, 0, 0, 0, 0, 0, 0),
287 #endif
288 #if (PAD_LIST || CC_GetTestResult)
289     TPMA_CC_INITIALIZER(0x017C, 0, 0, 0, 0, 0, 0, 0, 0),
290 #endif
291 #if (PAD_LIST || CC_Hash)
292     TPMA_CC_INITIALIZER(0x017D, 0, 0, 0, 0, 0, 0, 0, 0),
293 #endif
294 #if (PAD_LIST || CC_PCR_Read)
295     TPMA_CC_INITIALIZER(0x017E, 0, 0, 0, 0, 0, 0, 0, 0),
296 #endif
297 #if (PAD_LIST || CC_PolicyPCR)
298     TPMA_CC_INITIALIZER(0x017F, 0, 0, 0, 0, 1, 0, 0, 0),
299 #endif
300 #if (PAD_LIST || CC_PolicyRestart)
301     TPMA_CC_INITIALIZER(0x0180, 0, 0, 0, 0, 1, 0, 0, 0),
302 #endif
303 #if (PAD_LIST || CC_ReadClock)
304     TPMA_CC_INITIALIZER(0x0181, 0, 0, 0, 0, 0, 0, 0, 0),
305 #endif
306 #if (PAD_LIST || CC_PCR_Extend)
307     TPMA_CC_INITIALIZER(0x0182, 0, 1, 0, 0, 1, 0, 0, 0),
308 #endif
309 #if (PAD_LIST || CC_PCR_SetAuthValue)
310     TPMA_CC_INITIALIZER(0x0183, 0, 0, 0, 0, 1, 0, 0, 0),
311 #endif
312 #if (PAD_LIST || CC_NV_Certify)
313     TPMA_CC_INITIALIZER(0x0184, 0, 0, 0, 0, 3, 0, 0, 0),
314 #endif
315 #if (PAD_LIST || CC_EventSequenceComplete)
316     TPMA_CC_INITIALIZER(0x0185, 0, 1, 0, 1, 2, 0, 0, 0),
317 #endif
318 #if (PAD_LIST || CC_HashSequenceStart)
319     TPMA_CC_INITIALIZER(0x0186, 0, 0, 0, 0, 0, 1, 0, 0),
320 #endif
```

```

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

```

This is the command code attribute structure.

```

377  const COMMAND_ATTRIBUTES s_commandAttributes [] = {
378  #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
379      (COMMAND_ATTRIBUTES)(CC_NV_UndefineSpaceSpecial * // 0x011F
380      (IS_IMPLEMENTED+HANDLE_1_ADMIN+HANDLE_2_USER+PP_COMMAND)),
381  #endif
382  #if (PAD_LIST || CC_EvictControl)

```



```

383         (COMMAND_ATTRIBUTES) (CC_EvictControl * // 0x0120
384         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
385 #endif
386 #if (PAD_LIST || CC_HierarchyControl)
387         (COMMAND_ATTRIBUTES) (CC_HierarchyControl * // 0x0121
388         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
389 #endif
390 #if (PAD_LIST || CC_NV_UndefineSpace)
391         (COMMAND_ATTRIBUTES) (CC_NV_UndefineSpace * // 0x0122
392         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
393 #endif
394 #if (PAD_LIST )
395         (COMMAND_ATTRIBUTES) (0), // 0x0123
396 #endif
397 #if (PAD_LIST || CC_ChangeEPS)
398         (COMMAND_ATTRIBUTES) (CC_ChangeEPS * // 0x0124
399         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
400 #endif
401 #if (PAD_LIST || CC_ChangePPS)
402         (COMMAND_ATTRIBUTES) (CC_ChangePPS * // 0x0125
403         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
404 #endif
405 #if (PAD_LIST || CC_Clear)
406         (COMMAND_ATTRIBUTES) (CC_Clear * // 0x0126
407         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
408 #endif
409 #if (PAD_LIST || CC_ClearControl)
410         (COMMAND_ATTRIBUTES) (CC_ClearControl * // 0x0127
411         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
412 #endif
413 #if (PAD_LIST || CC_ClockSet)
414         (COMMAND_ATTRIBUTES) (CC_ClockSet * // 0x0128
415         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
416 #endif
417 #if (PAD_LIST || CC_HierarchyChangeAuth)
418         (COMMAND_ATTRIBUTES) (CC_HierarchyChangeAuth * // 0x0129
419         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
420 #endif
421 #if (PAD_LIST || CC_NV_DefineSpace)
422         (COMMAND_ATTRIBUTES) (CC_NV_DefineSpace * // 0x012A
423         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
424 #endif
425 #if (PAD_LIST || CC_PCR_Allocate)
426         (COMMAND_ATTRIBUTES) (CC_PCR_Allocate * // 0x012B
427         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
428 #endif
429 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
430         (COMMAND_ATTRIBUTES) (CC_PCR_SetAuthPolicy * // 0x012C
431         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
432 #endif
433 #if (PAD_LIST || CC_PP_Commands)
434         (COMMAND_ATTRIBUTES) (CC_PP_Commands * // 0x012D
435         (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),
436 #endif
437 #if (PAD_LIST || CC_SetPrimaryPolicy)
438         (COMMAND_ATTRIBUTES) (CC_SetPrimaryPolicy * // 0x012E
439         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
440 #endif
441 #if (PAD_LIST || CC_FieldUpgradeStart)
442         (COMMAND_ATTRIBUTES) (CC_FieldUpgradeStart * // 0x012F
443         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),
444 #endif
445 #if (PAD_LIST || CC_ClockRateAdjust)
446         (COMMAND_ATTRIBUTES) (CC_ClockRateAdjust * // 0x0130
447         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
448 #endif

```

```

449 #if (PAD_LIST || CC_CreatePrimary)
450     (COMMAND_ATTRIBUTES) (CC_CreatePrimary * // 0x0131
451         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
452 #endif
453 #if (PAD_LIST || CC_NV_GlobalWriteLock)
454     (COMMAND_ATTRIBUTES) (CC_NV_GlobalWriteLock * // 0x0132
455         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
456 #endif
457 #if (PAD_LIST || CC_GetCommandAuditDigest)
458     (COMMAND_ATTRIBUTES) (CC_GetCommandAuditDigest * // 0x0133
459         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
460 #endif
461 #if (PAD_LIST || CC_NV_Increment)
462     (COMMAND_ATTRIBUTES) (CC_NV_Increment * // 0x0134
463         (IS_IMPLEMENTED+HANDLE_1_USER)),
464 #endif
465 #if (PAD_LIST || CC_NV_SetBits)
466     (COMMAND_ATTRIBUTES) (CC_NV_SetBits * // 0x0135
467         (IS_IMPLEMENTED+HANDLE_1_USER)),
468 #endif
469 #if (PAD_LIST || CC_NV_Extend)
470     (COMMAND_ATTRIBUTES) (CC_NV_Extend * // 0x0136
471         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
472 #endif
473 #if (PAD_LIST || CC_NV_Write)
474     (COMMAND_ATTRIBUTES) (CC_NV_Write * // 0x0137
475         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
476 #endif
477 #if (PAD_LIST || CC_NV_WriteLock)
478     (COMMAND_ATTRIBUTES) (CC_NV_WriteLock * // 0x0138
479         (IS_IMPLEMENTED+HANDLE_1_USER)),
480 #endif
481 #if (PAD_LIST || CC_DictionaryAttackLockReset)
482     (COMMAND_ATTRIBUTES) (CC_DictionaryAttackLockReset * // 0x0139
483         (IS_IMPLEMENTED+HANDLE_1_USER)),
484 #endif
485 #if (PAD_LIST || CC_DictionaryAttackParameters)
486     (COMMAND_ATTRIBUTES) (CC_DictionaryAttackParameters * // 0x013A
487         (IS_IMPLEMENTED+HANDLE_1_USER)),
488 #endif
489 #if (PAD_LIST || CC_NV_ChangeAuth)
490     (COMMAND_ATTRIBUTES) (CC_NV_ChangeAuth * // 0x013B
491         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),
492 #endif
493 #if (PAD_LIST || CC_PCR_Event)
494     (COMMAND_ATTRIBUTES) (CC_PCR_Event * // 0x013C
495         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
496 #endif
497 #if (PAD_LIST || CC_PCR_Reset)
498     (COMMAND_ATTRIBUTES) (CC_PCR_Reset * // 0x013D
499         (IS_IMPLEMENTED+HANDLE_1_USER)),
500 #endif
501 #if (PAD_LIST || CC_SequenceComplete)
502     (COMMAND_ATTRIBUTES) (CC_SequenceComplete * // 0x013E
503         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
504 #endif
505 #if (PAD_LIST || CC_SetAlgorithmSet)
506     (COMMAND_ATTRIBUTES) (CC_SetAlgorithmSet * // 0x013F
507         (IS_IMPLEMENTED+HANDLE_1_USER)),
508 #endif
509 #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
510     (COMMAND_ATTRIBUTES) (CC_SetCommandCodeAuditStatus * // 0x0140
511         (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
512 #endif
513 #if (PAD_LIST || CC_FieldUpgradeData)
514     (COMMAND_ATTRIBUTES) (CC_FieldUpgradeData * // 0x0141

```



```

515         (IS_IMPLEMENTED+DECRYPT_2)),
516 #endif
517 #if (PAD_LIST || CC_IncrementalSelfTest)
518     (COMMAND_ATTRIBUTES)(CC_IncrementalSelfTest * // 0x0142
519         (IS_IMPLEMENTED)),
520 #endif
521 #if (PAD_LIST || CC_SelfTest)
522     (COMMAND_ATTRIBUTES)(CC_SelfTest * // 0x0143
523         (IS_IMPLEMENTED)),
524 #endif
525 #if (PAD_LIST || CC_Startup)
526     (COMMAND_ATTRIBUTES)(CC_Startup * // 0x0144
527         (IS_IMPLEMENTED+NO_SESSIONS)),
528 #endif
529 #if (PAD_LIST || CC_Shutdown)
530     (COMMAND_ATTRIBUTES)(CC_Shutdown * // 0x0145
531         (IS_IMPLEMENTED)),
532 #endif
533 #if (PAD_LIST || CC_StirRandom)
534     (COMMAND_ATTRIBUTES)(CC_StirRandom * // 0x0146
535         (IS_IMPLEMENTED+DECRYPT_2)),
536 #endif
537 #if (PAD_LIST || CC_ActivateCredential)
538     (COMMAND_ATTRIBUTES)(CC_ActivateCredential * // 0x0147
539         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
540 #endif
541 #if (PAD_LIST || CC_Certify)
542     (COMMAND_ATTRIBUTES)(CC_Certify * // 0x0148
543         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
544 #endif
545 #if (PAD_LIST || CC_PolicyNV)
546     (COMMAND_ATTRIBUTES)(CC_PolicyNV * // 0x0149
547         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL)),
548 #endif
549 #if (PAD_LIST || CC_CertifyCreation)
550     (COMMAND_ATTRIBUTES)(CC_CertifyCreation * // 0x014A
551         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
552 #endif
553 #if (PAD_LIST || CC_Duplicate)
554     (COMMAND_ATTRIBUTES)(CC_Duplicate * // 0x014B
555         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),
556 #endif
557 #if (PAD_LIST || CC_GetTime)
558     (COMMAND_ATTRIBUTES)(CC_GetTime * // 0x014C
559         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
560 #endif
561 #if (PAD_LIST || CC_GetSessionAuditDigest)
562     (COMMAND_ATTRIBUTES)(CC_GetSessionAuditDigest * // 0x014D
563         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
564 #endif
565 #if (PAD_LIST || CC_NV_Read)
566     (COMMAND_ATTRIBUTES)(CC_NV_Read * // 0x014E
567         (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
568 #endif
569 #if (PAD_LIST || CC_NV_ReadLock)
570     (COMMAND_ATTRIBUTES)(CC_NV_ReadLock * // 0x014F
571         (IS_IMPLEMENTED+HANDLE_1_USER)),
572 #endif
573 #if (PAD_LIST || CC_ObjectChangeAuth)
574     (COMMAND_ATTRIBUTES)(CC_ObjectChangeAuth * // 0x0150
575         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),
576 #endif
577 #if (PAD_LIST || CC_PolicySecret)
578     (COMMAND_ATTRIBUTES)(CC_PolicySecret * // 0x0151
579         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL+ENCRYPT_2)),
580 #endif

```

```

581 #if (PAD_LIST || CC_Rewrap)
582     (COMMAND_ATTRIBUTES) (CC_Rewrap * // 0x0152
583         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
584 #endif
585 #if (PAD_LIST || CC_Create)
586     (COMMAND_ATTRIBUTES) (CC_Create * // 0x0153
587         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
588 #endif
589 #if (PAD_LIST || CC_ECDH_ZGen)
590     (COMMAND_ATTRIBUTES) (CC_ECDH_ZGen * // 0x0154
591         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
592 #endif
593 #if (PAD_LIST || (CC_HMAC || CC_MAC))
594     (COMMAND_ATTRIBUTES) ((CC_HMAC || CC_MAC) * // 0x0155
595         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
596 #endif
597 #if (PAD_LIST || CC_Import)
598     (COMMAND_ATTRIBUTES) (CC_Import * // 0x0156
599         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
600 #endif
601 #if (PAD_LIST || CC_Load)
602     (COMMAND_ATTRIBUTES) (CC_Load * // 0x0157
603         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),
604 #endif
605 #if (PAD_LIST || CC_Quote)
606     (COMMAND_ATTRIBUTES) (CC_Quote * // 0x0158
607         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
608 #endif
609 #if (PAD_LIST || CC_RSA_Decrypt)
610     (COMMAND_ATTRIBUTES) (CC_RSA_Decrypt * // 0x0159
611         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
612 #endif
613 #if (PAD_LIST )
614     (COMMAND_ATTRIBUTES) (0), // 0x015A
615 #endif
616 #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
617     (COMMAND_ATTRIBUTES) ((CC_HMAC_Start || CC_MAC_Start) * // 0x015B
618         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),
619 #endif
620 #if (PAD_LIST || CC_SequenceUpdate)
621     (COMMAND_ATTRIBUTES) (CC_SequenceUpdate * // 0x015C
622         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
623 #endif
624 #if (PAD_LIST || CC_Sign)
625     (COMMAND_ATTRIBUTES) (CC_Sign * // 0x015D
626         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
627 #endif
628 #if (PAD_LIST || CC_Unseal)
629     (COMMAND_ATTRIBUTES) (CC_Unseal * // 0x015E
630         (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
631 #endif
632 #if (PAD_LIST )
633     (COMMAND_ATTRIBUTES) (0), // 0x015F
634 #endif
635 #if (PAD_LIST || CC_PolicySigned)
636     (COMMAND_ATTRIBUTES) (CC_PolicySigned * // 0x0160
637         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL+ENCRYPT_2)),
638 #endif
639 #if (PAD_LIST || CC_ContextLoad)
640     (COMMAND_ATTRIBUTES) (CC_ContextLoad * // 0x0161
641         (IS_IMPLEMENTED+NO_SESSIONS+R_HANDLE)),
642 #endif
643 #if (PAD_LIST || CC_ContextSave)
644     (COMMAND_ATTRIBUTES) (CC_ContextSave * // 0x0162
645         (IS_IMPLEMENTED+NO_SESSIONS)),
646 #endif

```

```

647 #if (PAD_LIST || CC_ECDH_KeyGen)
648     (COMMAND_ATTRIBUTES) (CC_ECDH_KeyGen
649         (IS_IMPLEMENTED+ENCRYPT_2)),
650 #endif
651 #if (PAD_LIST || CC_EncryptDecrypt)
652     (COMMAND_ATTRIBUTES) (CC_EncryptDecrypt
653         (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
654 #endif
655 #if (PAD_LIST || CC_FlushContext)
656     (COMMAND_ATTRIBUTES) (CC_FlushContext
657         (IS_IMPLEMENTED+NO_SESSIONS)),
658 #endif
659 #if (PAD_LIST )
660     (COMMAND_ATTRIBUTES) (0),
661 #endif
662 #if (PAD_LIST || CC_LoadExternal)
663     (COMMAND_ATTRIBUTES) (CC_LoadExternal
664         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
665 #endif
666 #if (PAD_LIST || CC_MakeCredential)
667     (COMMAND_ATTRIBUTES) (CC_MakeCredential
668         (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
669 #endif
670 #if (PAD_LIST || CC_NV_ReadPublic)
671     (COMMAND_ATTRIBUTES) (CC_NV_ReadPublic
672         (IS_IMPLEMENTED+ENCRYPT_2)),
673 #endif
674 #if (PAD_LIST || CC_PolicyAuthorize)
675     (COMMAND_ATTRIBUTES) (CC_PolicyAuthorize
676         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
677 #endif
678 #if (PAD_LIST || CC_PolicyAuthValue)
679     (COMMAND_ATTRIBUTES) (CC_PolicyAuthValue
680         (IS_IMPLEMENTED+ALLOW_TRIAL)),
681 #endif
682 #if (PAD_LIST || CC_PolicyCommandCode)
683     (COMMAND_ATTRIBUTES) (CC_PolicyCommandCode
684         (IS_IMPLEMENTED+ALLOW_TRIAL)),
685 #endif
686 #if (PAD_LIST || CC_PolicyCounterTimer)
687     (COMMAND_ATTRIBUTES) (CC_PolicyCounterTimer
688         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
689 #endif
690 #if (PAD_LIST || CC_PolicyCpHash)
691     (COMMAND_ATTRIBUTES) (CC_PolicyCpHash
692         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
693 #endif
694 #if (PAD_LIST || CC_PolicyLocality)
695     (COMMAND_ATTRIBUTES) (CC_PolicyLocality
696         (IS_IMPLEMENTED+ALLOW_TRIAL)),
697 #endif
698 #if (PAD_LIST || CC_PolicyNameHash)
699     (COMMAND_ATTRIBUTES) (CC_PolicyNameHash
700         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
701 #endif
702 #if (PAD_LIST || CC_PolicyOR)
703     (COMMAND_ATTRIBUTES) (CC_PolicyOR
704         (IS_IMPLEMENTED+ALLOW_TRIAL)),
705 #endif
706 #if (PAD_LIST || CC_PolicyTicket)
707     (COMMAND_ATTRIBUTES) (CC_PolicyTicket
708         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
709 #endif
710 #if (PAD_LIST || CC_ReadPublic)
711     (COMMAND_ATTRIBUTES) (CC_ReadPublic
712         (IS_IMPLEMENTED+ENCRYPT_2)),

```

```

713 #endif
714 #if (PAD_LIST || CC_RSA_Encrypt)
715     (COMMAND_ATTRIBUTES) (CC_RSA_Encrypt * // 0x0174
716     (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
717 #endif
718 #if (PAD_LIST )
719     (COMMAND_ATTRIBUTES) (0), // 0x0175
720 #endif
721 #if (PAD_LIST || CC_StartAuthSession)
722     (COMMAND_ATTRIBUTES) (CC_StartAuthSession * // 0x0176
723     (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
724 #endif
725 #if (PAD_LIST || CC_VerifySignature)
726     (COMMAND_ATTRIBUTES) (CC_VerifySignature * // 0x0177
727     (IS_IMPLEMENTED+DECRYPT_2)),
728 #endif
729 #if (PAD_LIST || CC_ECC_Parameters)
730     (COMMAND_ATTRIBUTES) (CC_ECC_Parameters * // 0x0178
731     (IS_IMPLEMENTED)),
732 #endif
733 #if (PAD_LIST || CC_FirmwareRead)
734     (COMMAND_ATTRIBUTES) (CC_FirmwareRead * // 0x0179
735     (IS_IMPLEMENTED+ENCRYPT_2)),
736 #endif
737 #if (PAD_LIST || CC_GetCapability)
738     (COMMAND_ATTRIBUTES) (CC_GetCapability * // 0x017A
739     (IS_IMPLEMENTED)),
740 #endif
741 #if (PAD_LIST || CC_GetRandom)
742     (COMMAND_ATTRIBUTES) (CC_GetRandom * // 0x017B
743     (IS_IMPLEMENTED+ENCRYPT_2)),
744 #endif
745 #if (PAD_LIST || CC_GetTestResult)
746     (COMMAND_ATTRIBUTES) (CC_GetTestResult * // 0x017C
747     (IS_IMPLEMENTED+ENCRYPT_2)),
748 #endif
749 #if (PAD_LIST || CC_Hash)
750     (COMMAND_ATTRIBUTES) (CC_Hash * // 0x017D
751     (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
752 #endif
753 #if (PAD_LIST || CC_PCR_Read)
754     (COMMAND_ATTRIBUTES) (CC_PCR_Read * // 0x017E
755     (IS_IMPLEMENTED)),
756 #endif
757 #if (PAD_LIST || CC_PolicyPCR)
758     (COMMAND_ATTRIBUTES) (CC_PolicyPCR * // 0x017F
759     (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
760 #endif
761 #if (PAD_LIST || CC_PolicyRestart)
762     (COMMAND_ATTRIBUTES) (CC_PolicyRestart * // 0x0180
763     (IS_IMPLEMENTED+ALLOW_TRIAL)),
764 #endif
765 #if (PAD_LIST || CC_ReadClock)
766     (COMMAND_ATTRIBUTES) (CC_ReadClock * // 0x0181
767     (IS_IMPLEMENTED)),
768 #endif
769 #if (PAD_LIST || CC_PCR_Extend)
770     (COMMAND_ATTRIBUTES) (CC_PCR_Extend * // 0x0182
771     (IS_IMPLEMENTED+HANDLE_1_USER)),
772 #endif
773 #if (PAD_LIST || CC_PCR_SetAuthValue)
774     (COMMAND_ATTRIBUTES) (CC_PCR_SetAuthValue * // 0x0183
775     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
776 #endif
777 #if (PAD_LIST || CC_NV_Certify)
778     (COMMAND_ATTRIBUTES) (CC_NV_Certify * // 0x0184

```

```

779         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
780 #endif
781 #if (PAD_LIST || CC_EventSequenceComplete)
782     (COMMAND_ATTRIBUTES)(CC_EventSequenceComplete * // 0x0185
783         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER)),
784 #endif
785 #if (PAD_LIST || CC_HashSequenceStart)
786     (COMMAND_ATTRIBUTES)(CC_HashSequenceStart * // 0x0186
787         (IS_IMPLEMENTED+DECRYPT_2+R_HANDLE)),
788 #endif
789 #if (PAD_LIST || CC_PolicyPhysicalPresence)
790     (COMMAND_ATTRIBUTES)(CC_PolicyPhysicalPresence * // 0x0187
791         (IS_IMPLEMENTED+ALLOW_TRIAL)),
792 #endif
793 #if (PAD_LIST || CC_PolicyDuplicationSelect)
794     (COMMAND_ATTRIBUTES)(CC_PolicyDuplicationSelect * // 0x0188
795         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
796 #endif
797 #if (PAD_LIST || CC_PolicyGetDigest)
798     (COMMAND_ATTRIBUTES)(CC_PolicyGetDigest * // 0x0189
799         (IS_IMPLEMENTED+ALLOW_TRIAL+ENCRYPT_2)),
800 #endif
801 #if (PAD_LIST || CC_TestParms)
802     (COMMAND_ATTRIBUTES)(CC_TestParms * // 0x018A
803         (IS_IMPLEMENTED)),
804 #endif
805 #if (PAD_LIST || CC_Commit)
806     (COMMAND_ATTRIBUTES)(CC_Commit * // 0x018B
807         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
808 #endif
809 #if (PAD_LIST || CC_PolicyPassword)
810     (COMMAND_ATTRIBUTES)(CC_PolicyPassword * // 0x018C
811         (IS_IMPLEMENTED+ALLOW_TRIAL)),
812 #endif
813 #if (PAD_LIST || CC_ZGen_2Phase)
814     (COMMAND_ATTRIBUTES)(CC_ZGen_2Phase * // 0x018D
815         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
816 #endif
817 #if (PAD_LIST || CC_EC_Ephemeral)
818     (COMMAND_ATTRIBUTES)(CC_EC_Ephemeral * // 0x018E
819         (IS_IMPLEMENTED+ENCRYPT_2)),
820 #endif
821 #if (PAD_LIST || CC_PolicyNvWritten)
822     (COMMAND_ATTRIBUTES)(CC_PolicyNvWritten * // 0x018F
823         (IS_IMPLEMENTED+ALLOW_TRIAL)),
824 #endif
825 #if (PAD_LIST || CC_PolicyTemplate)
826     (COMMAND_ATTRIBUTES)(CC_PolicyTemplate * // 0x0190
827         (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
828 #endif
829 #if (PAD_LIST || CC_CreateLoaded)
830     (COMMAND_ATTRIBUTES)(CC_CreateLoaded * // 0x0191
831         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
832 #endif
833 #if (PAD_LIST || CC_PolicyAuthorizeNV)
834     (COMMAND_ATTRIBUTES)(CC_PolicyAuthorizeNV * // 0x0192
835         (IS_IMPLEMENTED+HANDLE_1_USER+ALLOW_TRIAL)),
836 #endif
837 #if (PAD_LIST || CC_EncryptDecrypt2)
838     (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt2 * // 0x0193
839         (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
840 #endif
841 #if (PAD_LIST || CC_AC_GetCapability)
842     (COMMAND_ATTRIBUTES)(CC_AC_GetCapability * // 0x0194
843         (IS_IMPLEMENTED)),
844 #endif

```

```
845  #if (PAD_LIST || CC_AC_Send)
846      (COMMAND_ATTRIBUTES) (CC_AC_Send * // 0x0195
847      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+HANDLE_2_USER)),
848  #endif
849  #if (PAD_LIST || CC_Policy_AC_SendSelect)
850      (COMMAND_ATTRIBUTES) (CC_Policy_AC_SendSelect * // 0x0196
851      (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
852  #endif
853  #if (PAD_LIST || CC_CertifyX509)
854      (COMMAND_ATTRIBUTES) (CC_CertifyX509 * // 0x0197
855      (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
856  #endif
857  #if (PAD_LIST || CC_Vendor_TCG_Test)
858      (COMMAND_ATTRIBUTES) (CC_Vendor_TCG_Test * // 0x0000
859      (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
860  #endif
861      0
862  };
863  #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  typedef uint16_t COMMAND_ATTRIBUTES;
4  #define NOT_IMPLEMENTED ((COMMAND_ATTRIBUTES) 0)
5  #define ENCRYPT_2 ((COMMAND_ATTRIBUTES) 1 << 0)
6  #define ENCRYPT_4 ((COMMAND_ATTRIBUTES) 1 << 1)
7  #define DECRYPT_2 ((COMMAND_ATTRIBUTES) 1 << 2)
8  #define DECRYPT_4 ((COMMAND_ATTRIBUTES) 1 << 3)
9  #define HANDLE_1_USER ((COMMAND_ATTRIBUTES) 1 << 4)
10 #define HANDLE_1_ADMIN ((COMMAND_ATTRIBUTES) 1 << 5)
11 #define HANDLE_1_DUP ((COMMAND_ATTRIBUTES) 1 << 6)
12 #define HANDLE_2_USER ((COMMAND_ATTRIBUTES) 1 << 7)
13 #define PP_COMMAND ((COMMAND_ATTRIBUTES) 1 << 8)
14 #define IS_IMPLEMENTED ((COMMAND_ATTRIBUTES) 1 << 9)
15 #define NO_SESSIONS ((COMMAND_ATTRIBUTES) 1 << 10)
16 #define NV_COMMAND ((COMMAND_ATTRIBUTES) 1 << 11)
17 #define PP_REQUIRED ((COMMAND_ATTRIBUTES) 1 << 12)
18 #define R_HANDLE ((COMMAND_ATTRIBUTES) 1 << 13)
19 #define ALLOW_TRIAL ((COMMAND_ATTRIBUTES) 1 << 14)
20 #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 sorter. ;-)

```
4  # define UNMARSHAL_DISPATCH(name)      (UNMARSHAL_t)name##_Unmarshal
5  # define MARSHAL_DISPATCH(name)        (MARSHAL_t)##name##_Marshal
6  # define _UNMARSHAL_T_                  UNMARSHAL_t
7  # define _MARSHAL_T_                    MARSHAL_t
```

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

```
8  const _UNMARSHAL_T_ UnmarshalArray[] = {
9  #define TPMI_DH_CONTEXT_H_UNMARSHAL      0
10     UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
11  #define TPMI_RH_AC_H_UNMARSHAL           (TPMI_DH_CONTEXT_H_UNMARSHAL + 1)
12     UNMARSHAL_DISPATCH(TPMI_RH_AC),
13  #define TPMI_RH_CLEAR_H_UNMARSHAL        (TPMI_RH_AC_H_UNMARSHAL + 1)
14     UNMARSHAL_DISPATCH(TPMI_RH_CLEAR),
15  #define TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL (TPMI_RH_CLEAR_H_UNMARSHAL + 1)
16     UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY_AUTH),
17  #define TPMI_RH_LOCKOUT_H_UNMARSHAL       (TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL + 1)
18     UNMARSHAL_DISPATCH(TPMI_RH_LOCKOUT),
19  #define TPMI_RH_NV_AUTH_H_UNMARSHAL       (TPMI_RH_LOCKOUT_H_UNMARSHAL + 1)
20     UNMARSHAL_DISPATCH(TPMI_RH_NV_AUTH),
21  #define TPMI_RH_NV_INDEX_H_UNMARSHAL      (TPMI_RH_NV_AUTH_H_UNMARSHAL + 1)
22     UNMARSHAL_DISPATCH(TPMI_RH_NV_INDEX),
23  #define TPMI_RH_PLATFORM_H_UNMARSHAL      (TPMI_RH_NV_INDEX_H_UNMARSHAL + 1)
24     UNMARSHAL_DISPATCH(TPMI_RH_PLATFORM),
25  #define TPMI_RH_PROVISION_H_UNMARSHAL     (TPMI_RH_PLATFORM_H_UNMARSHAL + 1)
26     UNMARSHAL_DISPATCH(TPMI_RH_PROVISION),
27  #define TPMI_SH_HMAC_H_UNMARSHAL          (TPMI_RH_PROVISION_H_UNMARSHAL + 1)
28     UNMARSHAL_DISPATCH(TPMI_SH_HMAC),
29  #define TPMI_SH_POLICY_H_UNMARSHAL        (TPMI_SH_HMAC_H_UNMARSHAL + 1)
30     UNMARSHAL_DISPATCH(TPMI_SH_POLICY),
31  // HANDLE_FIRST_FLAG_TYPE is the first handle that needs a flag when called.
32  #define HANDLE_FIRST_FLAG_TYPE           (TPMI_SH_POLICY_H_UNMARSHAL + 1)
33  #define TPMI_DH_ENTITY_H_UNMARSHAL        (TPMI_SH_POLICY_H_UNMARSHAL + 1)
34     UNMARSHAL_DISPATCH(TPMI_DH_ENTITY),
35  #define TPMI_DH_OBJECT_H_UNMARSHAL        (TPMI_DH_ENTITY_H_UNMARSHAL + 1)
36     UNMARSHAL_DISPATCH(TPMI_DH_OBJECT),
37  #define TPMI_DH_PARENT_H_UNMARSHAL        (TPMI_DH_OBJECT_H_UNMARSHAL + 1)
38     UNMARSHAL_DISPATCH(TPMI_DH_PARENT),
39  #define TPMI_DH_PCR_H_UNMARSHAL           (TPMI_DH_PARENT_H_UNMARSHAL + 1)
40     UNMARSHAL_DISPATCH(TPMI_DH_PCR),
41  #define TPMI_RH_ENDORSEMENT_H_UNMARSHAL   (TPMI_DH_PCR_H_UNMARSHAL + 1)
42     UNMARSHAL_DISPATCH(TPMI_RH_ENDORSEMENT),
43  #define TPMI_RH_HIERARCHY_H_UNMARSHAL     (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)
44     UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
45  // PARAMETER_FIRST_TYPE marks the end of the handle list.
46  #define PARAMETER_FIRST_TYPE              (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
47  #define TPM2B_DATA_P_UNMARSHAL           (TPMI_RH_HIERARCHY_H_UNMARSHAL + 1)
```



```
48     UNMARSHAL_DISPATCH(TPM2B_DATA),
49 #define TPM2B_DIGEST_P_UNMARSHAL (TPM2B_DATA_P_UNMARSHAL + 1)
50     UNMARSHAL_DISPATCH(TPM2B_DIGEST),
51 #define TPM2B_ECC_PARAMETER_P_UNMARSHAL (TPM2B_DIGEST_P_UNMARSHAL + 1)
52     UNMARSHAL_DISPATCH(TPM2B_ECC_PARAMETER),
53 #define TPM2B_ECC_POINT_P_UNMARSHAL (TPM2B_ECC_PARAMETER_P_UNMARSHAL + 1)
54     UNMARSHAL_DISPATCH(TPM2B_ECC_POINT),
55 #define TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL (TPM2B_ECC_POINT_P_UNMARSHAL + 1)
56     UNMARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
57 #define TPM2B_EVENT_P_UNMARSHAL (TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL + 1)
58     UNMARSHAL_DISPATCH(TPM2B_EVENT),
59 #define TPM2B_ID_OBJECT_P_UNMARSHAL (TPM2B_EVENT_P_UNMARSHAL + 1)
60     UNMARSHAL_DISPATCH(TPM2B_ID_OBJECT),
61 #define TPM2B_IV_P_UNMARSHAL (TPM2B_ID_OBJECT_P_UNMARSHAL + 1)
62     UNMARSHAL_DISPATCH(TPM2B_IV),
63 #define TPM2B_MAX_BUFFER_P_UNMARSHAL (TPM2B_IV_P_UNMARSHAL + 1)
64     UNMARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
65 #define TPM2B_MAX_NV_BUFFER_P_UNMARSHAL (TPM2B_MAX_BUFFER_P_UNMARSHAL + 1)
66     UNMARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
67 #define TPM2B_NAME_P_UNMARSHAL (TPM2B_MAX_NV_BUFFER_P_UNMARSHAL + 1)
68     UNMARSHAL_DISPATCH(TPM2B_NAME),
69 #define TPM2B_NV_PUBLIC_P_UNMARSHAL (TPM2B_NAME_P_UNMARSHAL + 1)
70     UNMARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
71 #define TPM2B_PRIVATE_P_UNMARSHAL (TPM2B_NV_PUBLIC_P_UNMARSHAL + 1)
72     UNMARSHAL_DISPATCH(TPM2B_PRIVATE),
73 #define TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL (TPM2B_PRIVATE_P_UNMARSHAL + 1)
74     UNMARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
75 #define TPM2B_SENSITIVE_P_UNMARSHAL (TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL + 1)
76     UNMARSHAL_DISPATCH(TPM2B_SENSITIVE),
77 #define TPM2B_SENSITIVE_CREATE_P_UNMARSHAL (TPM2B_SENSITIVE_P_UNMARSHAL + 1)
78     UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_CREATE),
79 #define TPM2B_SENSITIVE_DATA_P_UNMARSHAL (TPM2B_SENSITIVE_CREATE_P_UNMARSHAL + 1)
80     UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
81 #define TPM2B_TEMPLATE_P_UNMARSHAL (TPM2B_SENSITIVE_DATA_P_UNMARSHAL + 1)
82     UNMARSHAL_DISPATCH(TPM2B_TEMPLATE),
83 #define TPM2B_TIMEOUT_P_UNMARSHAL (TPM2B_TEMPLATE_P_UNMARSHAL + 1)
84     UNMARSHAL_DISPATCH(TPM2B_TIMEOUT),
85 #define TPMT_DH_CONTEXT_P_UNMARSHAL (TPM2B_TIMEOUT_P_UNMARSHAL + 1)
86     UNMARSHAL_DISPATCH(TPMT_DH_CONTEXT),
87 #define TPMT_DH_PERSISTENT_P_UNMARSHAL (TPMT_DH_CONTEXT_P_UNMARSHAL + 1)
88     UNMARSHAL_DISPATCH(TPMT_DH_PERSISTENT),
89 #define TPMT_ECC_CURVE_P_UNMARSHAL (TPMT_DH_PERSISTENT_P_UNMARSHAL + 1)
90     UNMARSHAL_DISPATCH(TPMT_ECC_CURVE),
91 #define TPMT_YES_NO_P_UNMARSHAL (TPMT_ECC_CURVE_P_UNMARSHAL + 1)
92     UNMARSHAL_DISPATCH(TPMT_YES_NO),
93 #define TPML_ALG_P_UNMARSHAL (TPMT_YES_NO_P_UNMARSHAL + 1)
94     UNMARSHAL_DISPATCH(TPML_ALG),
95 #define TPML_CC_P_UNMARSHAL (TPML_ALG_P_UNMARSHAL + 1)
96     UNMARSHAL_DISPATCH(TPML_CC),
97 #define TPML_DIGEST_P_UNMARSHAL (TPML_CC_P_UNMARSHAL + 1)
98     UNMARSHAL_DISPATCH(TPML_DIGEST),
99 #define TPML_DIGEST_VALUES_P_UNMARSHAL (TPML_DIGEST_P_UNMARSHAL + 1)
100     UNMARSHAL_DISPATCH(TPML_DIGEST_VALUES),
101 #define TPML_PCR_SELECTION_P_UNMARSHAL (TPML_DIGEST_VALUES_P_UNMARSHAL + 1)
102     UNMARSHAL_DISPATCH(TPML_PCR_SELECTION),
103 #define TPMS_CONTEXT_P_UNMARSHAL (TPML_PCR_SELECTION_P_UNMARSHAL + 1)
104     UNMARSHAL_DISPATCH(TPMS_CONTEXT),
105 #define TPMT_PUBLIC_PARMS_P_UNMARSHAL (TPMS_CONTEXT_P_UNMARSHAL + 1)
106     UNMARSHAL_DISPATCH(TPMT_PUBLIC_PARMS),
107 #define TPMT_TK_AUTH_P_UNMARSHAL (TPMT_PUBLIC_PARMS_P_UNMARSHAL + 1)
108     UNMARSHAL_DISPATCH(TPMT_TK_AUTH),
109 #define TPMT_TK_CREATION_P_UNMARSHAL (TPMT_TK_AUTH_P_UNMARSHAL + 1)
110     UNMARSHAL_DISPATCH(TPMT_TK_CREATION),
111 #define TPMT_TK_HASHCHECK_P_UNMARSHAL (TPMT_TK_CREATION_P_UNMARSHAL + 1)
112     UNMARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
113 #define TPMT_TK_VERIFIED_P_UNMARSHAL (TPMT_TK_HASHCHECK_P_UNMARSHAL + 1)
```

```

114         UNMARSHAL_DISPATCH(TPMT_TK_VERIFIED),
115 #define TPM_AT_P_UNMARSHAL (TPMT_TK_VERIFIED_P_UNMARSHAL + 1)
116         UNMARSHAL_DISPATCH(TPM_AT),
117 #define TPM_CAP_P_UNMARSHAL (TPM_AT_P_UNMARSHAL + 1)
118         UNMARSHAL_DISPATCH(TPM_CAP),
119 #define TPM_CLOCK_ADJUST_P_UNMARSHAL (TPM_CAP_P_UNMARSHAL + 1)
120         UNMARSHAL_DISPATCH(TPM_CLOCK_ADJUST),
121 #define TPM_EO_P_UNMARSHAL (TPM_CLOCK_ADJUST_P_UNMARSHAL + 1)
122         UNMARSHAL_DISPATCH(TPM_EO),
123 #define TPM_SE_P_UNMARSHAL (TPM_EO_P_UNMARSHAL + 1)
124         UNMARSHAL_DISPATCH(TPM_SE),
125 #define TPM_SU_P_UNMARSHAL (TPM_SE_P_UNMARSHAL + 1)
126         UNMARSHAL_DISPATCH(TPM_SU),
127 #define UINT16_P_UNMARSHAL (TPM_SU_P_UNMARSHAL + 1)
128         UNMARSHAL_DISPATCH(UINT16),
129 #define UINT32_P_UNMARSHAL (UINT16_P_UNMARSHAL + 1)
130         UNMARSHAL_DISPATCH(UINT32),
131 #define UINT64_P_UNMARSHAL (UINT32_P_UNMARSHAL + 1)
132         UNMARSHAL_DISPATCH(UINT64),
133 #define UINT8_P_UNMARSHAL (UINT64_P_UNMARSHAL + 1)
134         UNMARSHAL_DISPATCH(UINT8),
135 // PARAMETER_FIRST_FLAG_TYPE is the first parameter to need a flag.
136 #define PARAMETER_FIRST_FLAG_TYPE (UINT8_P_UNMARSHAL + 1)
137 #define TPM2B_PUBLIC_P_UNMARSHAL (UINT8_P_UNMARSHAL + 1)
138         UNMARSHAL_DISPATCH(TPM2B_PUBLIC),
139 #define TPMT_ALG_CIPHER_MODE_P_UNMARSHAL (TPM2B_PUBLIC_P_UNMARSHAL + 1)
140         UNMARSHAL_DISPATCH(TPMT_ALG_CIPHER_MODE),
141 #define TPMT_ALG_HASH_P_UNMARSHAL (TPMT_ALG_CIPHER_MODE_P_UNMARSHAL + 1)
142         UNMARSHAL_DISPATCH(TPMT_ALG_HASH),
143 #define TPMT_ALG_MAC_SCHEME_P_UNMARSHAL (TPMT_ALG_HASH_P_UNMARSHAL + 1)
144         UNMARSHAL_DISPATCH(TPMT_ALG_MAC_SCHEME),
145 #define TPMT_DH_PCR_P_UNMARSHAL (TPMT_ALG_MAC_SCHEME_P_UNMARSHAL + 1)
146         UNMARSHAL_DISPATCH(TPMT_DH_PCR),
147 #define TPMT_ECC_KEY_EXCHANGE_P_UNMARSHAL (TPMT_DH_PCR_P_UNMARSHAL + 1)
148         UNMARSHAL_DISPATCH(TPMT_ECC_KEY_EXCHANGE),
149 #define TPMT_RH_ENABLES_P_UNMARSHAL (TPMT_ECC_KEY_EXCHANGE_P_UNMARSHAL + 1)
150         UNMARSHAL_DISPATCH(TPMT_RH_ENABLES),
151 #define TPMT_RH_HIERARCHY_P_UNMARSHAL (TPMT_RH_ENABLES_P_UNMARSHAL + 1)
152         UNMARSHAL_DISPATCH(TPMT_RH_HIERARCHY),
153 #define TPMT_RSA_DECRYPT_P_UNMARSHAL (TPMT_RH_HIERARCHY_P_UNMARSHAL + 1)
154         UNMARSHAL_DISPATCH(TPMT_RSA_DECRYPT),
155 #define TPMT_SIGNATURE_P_UNMARSHAL (TPMT_RSA_DECRYPT_P_UNMARSHAL + 1)
156         UNMARSHAL_DISPATCH(TPMT_SIGNATURE),
157 #define TPMT_SIG_SCHEME_P_UNMARSHAL (TPMT_SIGNATURE_P_UNMARSHAL + 1)
158         UNMARSHAL_DISPATCH(TPMT_SIG_SCHEME),
159 #define TPMT_SYM_DEF_P_UNMARSHAL (TPMT_SIG_SCHEME_P_UNMARSHAL + 1)
160         UNMARSHAL_DISPATCH(TPMT_SYM_DEF),
161 #define TPMT_SYM_DEF_OBJECT_P_UNMARSHAL (TPMT_SYM_DEF_P_UNMARSHAL + 1)
162         UNMARSHAL_DISPATCH(TPMT_SYM_DEF_OBJECT)
163 // PARAMETER_LAST_TYPE is the end of the command parameter list.
164 #define PARAMETER_LAST_TYPE (TPMT_SYM_DEF_OBJECT_P_UNMARSHAL)
165 };

```

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

```

166 const _MARSHAL_T MarshalArray[] = {
167
168 #define UINT32_H_MARSHAL 0
169         MARSHAL_DISPATCH(UINT32),
170 // RESPONSE_PARAMETER_FIRST_TYPE marks the end of the response handles.
171 #define RESPONSE_PARAMETER_FIRST_TYPE (UINT32_H_MARSHAL + 1)
172 #define TPM2B_ATTEST_P_MARSHAL (UINT32_H_MARSHAL + 1)
173         MARSHAL_DISPATCH(TPM2B_ATTEST),
174 #define TPM2B_CREATION_DATA_P_MARSHAL (TPM2B_ATTEST_P_MARSHAL + 1)

```

```

175     MARSHAL_DISPATCH(TPM2B_CREATION_DATA),
176 #define TPM2B_DATA_P_MARSHAL (TPM2B_CREATION_DATA_P_MARSHAL + 1)
177     MARSHAL_DISPATCH(TPM2B_DATA),
178 #define TPM2B_DIGEST_P_MARSHAL (TPM2B_DATA_P_MARSHAL + 1)
179     MARSHAL_DISPATCH(TPM2B_DIGEST),
180 #define TPM2B_ECC_POINT_P_MARSHAL (TPM2B_DIGEST_P_MARSHAL + 1)
181     MARSHAL_DISPATCH(TPM2B_ECC_POINT),
182 #define TPM2B_ENCRYPTED_SECRET_P_MARSHAL (TPM2B_ECC_POINT_P_MARSHAL + 1)
183     MARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
184 #define TPM2B_ID_OBJECT_P_MARSHAL (TPM2B_ENCRYPTED_SECRET_P_MARSHAL + 1)
185     MARSHAL_DISPATCH(TPM2B_ID_OBJECT),
186 #define TPM2B_IV_P_MARSHAL (TPM2B_ID_OBJECT_P_MARSHAL + 1)
187     MARSHAL_DISPATCH(TPM2B_IV),
188 #define TPM2B_MAX_BUFFER_P_MARSHAL (TPM2B_IV_P_MARSHAL + 1)
189     MARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
190 #define TPM2B_MAX_NV_BUFFER_P_MARSHAL (TPM2B_MAX_BUFFER_P_MARSHAL + 1)
191     MARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
192 #define TPM2B_NAME_P_MARSHAL (TPM2B_MAX_NV_BUFFER_P_MARSHAL + 1)
193     MARSHAL_DISPATCH(TPM2B_NAME),
194 #define TPM2B_NV_PUBLIC_P_MARSHAL (TPM2B_NAME_P_MARSHAL + 1)
195     MARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
196 #define TPM2B_PRIVATE_P_MARSHAL (TPM2B_NV_PUBLIC_P_MARSHAL + 1)
197     MARSHAL_DISPATCH(TPM2B_PRIVATE),
198 #define TPM2B_PUBLIC_P_MARSHAL (TPM2B_PRIVATE_P_MARSHAL + 1)
199     MARSHAL_DISPATCH(TPM2B_PUBLIC),
200 #define TPM2B_PUBLIC_KEY_RSA_P_MARSHAL (TPM2B_PUBLIC_P_MARSHAL + 1)
201     MARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
202 #define TPM2B_SENSITIVE_DATA_P_MARSHAL (TPM2B_PUBLIC_KEY_RSA_P_MARSHAL + 1)
203     MARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
204 #define TPM2B_TIMEOUT_P_MARSHAL (TPM2B_SENSITIVE_DATA_P_MARSHAL + 1)
205     MARSHAL_DISPATCH(TPM2B_TIMEOUT),
206 #define UINT8_P_MARSHAL (TPM2B_TIMEOUT_P_MARSHAL + 1)
207     MARSHAL_DISPATCH(UINT8),
208 #define TPML_AC_CAPABILITIES_P_MARSHAL (UINT8_P_MARSHAL + 1)
209     MARSHAL_DISPATCH(TPML_AC_CAPABILITIES),
210 #define TPML_ALG_P_MARSHAL (TPML_AC_CAPABILITIES_P_MARSHAL + 1)
211     MARSHAL_DISPATCH(TPML_ALG),
212 #define TPML_DIGEST_P_MARSHAL (TPML_ALG_P_MARSHAL + 1)
213     MARSHAL_DISPATCH(TPML_DIGEST),
214 #define TPML_DIGEST_VALUES_P_MARSHAL (TPML_DIGEST_P_MARSHAL + 1)
215     MARSHAL_DISPATCH(TPML_DIGEST_VALUES),
216 #define TPML_PCR_SELECTION_P_MARSHAL (TPML_DIGEST_VALUES_P_MARSHAL + 1)
217     MARSHAL_DISPATCH(TPML_PCR_SELECTION),
218 #define TPMS_AC_OUTPUT_P_MARSHAL (TPML_PCR_SELECTION_P_MARSHAL + 1)
219     MARSHAL_DISPATCH(TPMS_AC_OUTPUT),
220 #define TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL (TPMS_AC_OUTPUT_P_MARSHAL + 1)
221     MARSHAL_DISPATCH(TPMS_ALGORITHM_DETAIL_ECC),
222 #define TPMS_CAPABILITY_DATA_P_MARSHAL \
223     (TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL + 1)
224     MARSHAL_DISPATCH(TPMS_CAPABILITY_DATA),
225 #define TPMS_CONTEXT_P_MARSHAL (TPMS_CAPABILITY_DATA_P_MARSHAL + 1)
226     MARSHAL_DISPATCH(TPMS_CONTEXT),
227 #define TPMS_TIME_INFO_P_MARSHAL (TPMS_CONTEXT_P_MARSHAL + 1)
228     MARSHAL_DISPATCH(TPMS_TIME_INFO),
229 #define TPMT_HA_P_MARSHAL (TPMS_TIME_INFO_P_MARSHAL + 1)
230     MARSHAL_DISPATCH(TPMT_HA),
231 #define TPMT_SIGNATURE_P_MARSHAL (TPMT_HA_P_MARSHAL + 1)
232     MARSHAL_DISPATCH(TPMT_SIGNATURE),
233 #define TPMT_TK_AUTH_P_MARSHAL (TPMT_SIGNATURE_P_MARSHAL + 1)
234     MARSHAL_DISPATCH(TPMT_TK_AUTH),
235 #define TPMT_TK_CREATION_P_MARSHAL (TPMT_TK_AUTH_P_MARSHAL + 1)
236     MARSHAL_DISPATCH(TPMT_TK_CREATION),
237 #define TPMT_TK_HASHCHECK_P_MARSHAL (TPMT_TK_CREATION_P_MARSHAL + 1)
238     MARSHAL_DISPATCH(TPMT_TK_HASHCHECK),
239 #define TPMT_TK_VERIFIED_P_MARSHAL (TPMT_TK_HASHCHECK_P_MARSHAL + 1)
240     MARSHAL_DISPATCH(TPMT_TK_VERIFIED),

```

```

241 #define UINT32_P_MARSHAL (TPMT_TK_VERIFIED_P_MARSHAL + 1)
242     MARSHAL_DISPATCH(UINT32),
243 #define UINT16_P_MARSHAL (UINT32_P_MARSHAL + 1)
244     MARSHAL_DISPATCH(UINT16)
245 // RESPONSE_PARAMETER_LAST_TYPE is the end of the response parameter list.
246 #define RESPONSE_PARAMETER_LAST_TYPE (UINT16_P_MARSHAL)
247 };

```

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

```

248 #define INT32_P_UNMARSHAL      UINT32_P_UNMARSHAL
249 #define TPM2B_AUTH_P_UNMARSHAL TPM2B_DIGEST_P_UNMARSHAL
250 #define TPM2B_NONCE_P_UNMARSHAL TPM2B_DIGEST_P_UNMARSHAL
251 #define TPM2B_OPERAND_P_UNMARSHAL TPM2B_DIGEST_P_UNMARSHAL
252 #define TPMA_LOCALITY_P_UNMARSHAL UINT8_P_UNMARSHAL
253 #define TPM_CC_P_UNMARSHAL      UINT32_P_UNMARSHAL
254 #define TPMI_DH_CONTEXT_H_MARSHAL UINT32_H_MARSHAL
255 #define TPMI_DH_OBJECT_H_MARSHAL  UINT32_H_MARSHAL
256 #define TPMI_SH_AUTH_SESSION_H_MARSHAL UINT32_H_MARSHAL
257 #define TPM_HANDLE_H_MARSHAL      UINT32_H_MARSHAL
258 #define TPM2B_NONCE_P_MARSHAL     TPM2B_DIGEST_P_MARSHAL
259 #define TPMI_YES_NO_P_MARSHAL     UINT8_P_MARSHAL
260 #define TPM_RC_P_MARSHAL          UINT32_P_MARSHAL
261 #if CC_Startup
262 #include "Startup_fp.h"
263 typedef TPM_RC (Startup_Entry) (
264     Startup_In *in
265 );
266 typedef const struct {
267     Startup_Entry *entry;
268     UINT16 inSize;
269     UINT16 outSize;
270     UINT16 offsetOfTypes;
271     BYTE types[3];
272 } Startup_COMMAND_DESCRIPTOR_t;
273 Startup_COMMAND_DESCRIPTOR_t _StartupData = {
274     /* entry */ &TPM2_Startup,
275     /* inSize */ (UINT16) (sizeof(Startup_In)),
276     /* outSize */ 0,
277     /* offsetOfTypes */ offsetof(Startup_COMMAND_DESCRIPTOR_t, types),
278     /* offsets */ // No parameter offsets;
279     /* types */ {TPM_SU_P_UNMARSHAL,
280                 END_OF_LIST,
281                 END_OF_LIST}
282 };
283 #define _StartupDataAddress (&_StartupData)
284 #else
285 #define _StartupDataAddress 0
286 #endif // CC_Startup
287 #if CC_Shutdown
288 #include "Shutdown_fp.h"
289 typedef TPM_RC (Shutdown_Entry) (
290     Shutdown_In *in
291 );
292 typedef const struct {
293     Shutdown_Entry *entry;
294     UINT16 inSize;
295     UINT16 outSize;
296     UINT16 offsetOfTypes;
297     BYTE types[3];
298 } Shutdown_COMMAND_DESCRIPTOR_t;
299 Shutdown_COMMAND_DESCRIPTOR_t _ShutdownData = {
300     /* entry */ &TPM2_Shutdown,
301     /* inSize */ (UINT16) (sizeof(Shutdown_In)),

```

```

302     /* outSize */ 0,
303     /* offsetOfTypes */ offsetof(Shutdown_COMMAND_DESCRIPTOR_t, types),
304     /* offsets */ // No parameter offsets;
305     /* types */ {TPM_SU_P_UNMARSHAL,
306                 END_OF_LIST,
307                 END_OF_LIST}
308 };
309 #define _ShutdownDataAddress (&_ShutdownData)
310 #else
311 #define _ShutdownDataAddress 0
312 #endif // CC_Shutdown
313 #if CC_SelfTest
314 #include "SelfTest_fp.h"
315 typedef TPM_RC (SelfTest_Entry) (
316     SelfTest_In *in
317 );
318 typedef const struct {
319     SelfTest_Entry *entry;
320     UINT16 inSize;
321     UINT16 outSize;
322     UINT16 offsetOfTypes;
323     BYTE types[3];
324 } SelfTest_COMMAND_DESCRIPTOR_t;
325 SelfTest_COMMAND_DESCRIPTOR_t _SelfTestData = {
326     /* entry */ &TPM2_SelfTest,
327     /* inSize */ (UINT16) (sizeof(SelfTest_In)),
328     /* outSize */ 0,
329     /* offsetOfTypes */ offsetof(SelfTest_COMMAND_DESCRIPTOR_t, types),
330     /* offsets */ // No parameter offsets;
331     /* types */ {TPMI_YES_NO_P_UNMARSHAL,
332                 END_OF_LIST,
333                 END_OF_LIST}
334 };
335 #define _SelfTestDataAddress (&_SelfTestData)
336 #else
337 #define _SelfTestDataAddress 0
338 #endif // CC_SelfTest
339 #if CC_IncrementalSelfTest
340 #include "IncrementalSelfTest_fp.h"
341 typedef TPM_RC (IncrementalSelfTest_Entry) (
342     IncrementalSelfTest_In *in,
343     IncrementalSelfTest_Out *out
344 );
345 typedef const struct {
346     IncrementalSelfTest_Entry *entry;
347     UINT16 inSize;
348     UINT16 outSize;
349     UINT16 offsetOfTypes;
350     BYTE types[4];
351 } IncrementalSelfTest_COMMAND_DESCRIPTOR_t;
352 IncrementalSelfTest_COMMAND_DESCRIPTOR_t _IncrementalSelfTestData = {
353     /* entry */ &TPM2_IncrementalSelfTest,
354     /* inSize */ (UINT16) (sizeof(IncrementalSelfTest_In)),
355     /* outSize */ (UINT16) (sizeof(IncrementalSelfTest_Out)),
356     /* offsetOfTypes */ offsetof(IncrementalSelfTest_COMMAND_DESCRIPTOR_t,
357 types),
358     /* offsets */ // No parameter offsets;
359     /* types */ {TPML_ALG_P_UNMARSHAL,
360                 END_OF_LIST,
361                 TPML_ALG_P_MARSHAL,
362                 END_OF_LIST}
363 };
364 #define _IncrementalSelfTestDataAddress (&_IncrementalSelfTestData)
365 #else
366 #define _IncrementalSelfTestDataAddress 0
367 #endif // CC_IncrementalSelfTest

```



```

367 #if CC_GetTestResult
368 #include "GetTestResult_fp.h"
369 typedef TPM_RC (GetTestResult_Entry) (
370     GetTestResult_Out      *out
371 );
372 typedef const struct {
373     GetTestResult_Entry    *entry;
374     UINT16                  inSize;
375     UINT16                  outSize;
376     UINT16                  offsetOfTypes;
377     UINT16                  paramOffsets[1];
378     BYTE                    types[4];
379 } GetTestResult_COMMAND_DESCRIPTOR_t;
380 GetTestResult_COMMAND_DESCRIPTOR_t _GetTestResultData = {
381     /* entry */           &TPM2_GetTestResult,
382     /* inSize */          0,
383     /* outSize */         (UINT16) (sizeof(GetTestResult_Out)),
384     /* offsetOfTypes */   offsetof(GetTestResult_COMMAND_DESCRIPTOR_t, types),
385     /* offsets */         {(UINT16) (offsetof(GetTestResult_Out, testResult))},
386     /* types */           {END_OF_LIST,
387                           TPM2B_MAX_BUFFER_P_MARSHAL,
388                           TPM_RC_P_MARSHAL,
389                           END_OF_LIST}
390 };
391 #define _GetTestResultDataAddress (&_GetTestResultData)
392 #else
393 #define _GetTestResultDataAddress 0
394 #endif // CC_GetTestResult
395 #if CC_StartAuthSession
396 #include "StartAuthSession_fp.h"
397 typedef TPM_RC (StartAuthSession_Entry) (
398     StartAuthSession_In    *in,
399     StartAuthSession_Out   *out
400 );
401 typedef const struct {
402     StartAuthSession_Entry *entry;
403     UINT16                  inSize;
404     UINT16                  outSize;
405     UINT16                  offsetOfTypes;
406     UINT16                  paramOffsets[7];
407     BYTE                    types[11];
408 } StartAuthSession_COMMAND_DESCRIPTOR_t;
409 StartAuthSession_COMMAND_DESCRIPTOR_t _StartAuthSessionData = {
410     /* entry */           &TPM2_StartAuthSession,
411     /* inSize */          (UINT16) (sizeof(StartAuthSession_In)),
412     /* outSize */         (UINT16) (sizeof(StartAuthSession_Out)),
413     /* offsetOfTypes */   offsetof(StartAuthSession_COMMAND_DESCRIPTOR_t, types),
414     /* offsets */         {(UINT16) (offsetof(StartAuthSession_In, bind)),
415                           (UINT16) (offsetof(StartAuthSession_In, nonceCaller)),
416                           (UINT16) (offsetof(StartAuthSession_In, encryptedSalt)),
417                           (UINT16) (offsetof(StartAuthSession_In, sessionType)),
418                           (UINT16) (offsetof(StartAuthSession_In, symmetric)),
419                           (UINT16) (offsetof(StartAuthSession_In, authHash)),
420                           (UINT16) (offsetof(StartAuthSession_Out, nonceTPM))},
421     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
422                           TPMI_DH_ENTITY_H_UNMARSHAL + ADD_FLAG,
423                           TPM2B_NONCE_P_UNMARSHAL,
424                           TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
425                           TPM_SE_P_UNMARSHAL,
426                           TPMT_SYM_DEF_P_UNMARSHAL + ADD_FLAG,
427                           TPMI_ALG_HASH_P_UNMARSHAL,
428                           END_OF_LIST,
429                           TPMI_SH_AUTH_SESSION_H_MARSHAL,
430                           TPM2B_NONCE_P_MARSHAL,
431                           END_OF_LIST}
432 };

```

```

433 #define _StartAuthSessionDataAddress (&_StartAuthSessionData)
434 #else
435 #define _StartAuthSessionDataAddress 0
436 #endif // CC_StartAuthSession
437 #if CC_PolicyRestart
438 #include "PolicyRestart_fp.h"
439 typedef TPM_RC (PolicyRestart_Entry)(
440     PolicyRestart_In      *in
441 );
442 typedef const struct {
443     PolicyRestart_Entry    *entry;
444     UINT16                  inSize;
445     UINT16                  outSize;
446     UINT16                  offsetOfTypes;
447     BYTE                    types[3];
448 } PolicyRestart_COMMAND_DESCRIPTOR_t;
449 PolicyRestart_COMMAND_DESCRIPTOR_t _PolicyRestartData = {
450     /* entry */           &TPM2_PolicyRestart,
451     /* inSize */          (UINT16)(sizeof(PolicyRestart_In)),
452     /* outSize */         0,
453     /* offsetOfTypes */   offsetof(PolicyRestart_COMMAND_DESCRIPTOR_t, types),
454     /* offsets */         // No parameter offsets;
455     /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
456                           END_OF_LIST,
457                           END_OF_LIST}
458 };
459 #define _PolicyRestartDataAddress (&_PolicyRestartData)
460 #else
461 #define _PolicyRestartDataAddress 0
462 #endif // CC_PolicyRestart
463 #if CC_Create
464 #include "Create_fp.h"
465 typedef TPM_RC (Create_Entry)(
466     Create_In      *in,
467     Create_Out     *out
468 );
469 typedef const struct {
470     Create_Entry    *entry;
471     UINT16          inSize;
472     UINT16          outSize;
473     UINT16          offsetOfTypes;
474     UINT16          paramOffsets[8];
475     BYTE            types[12];
476 } Create_COMMAND_DESCRIPTOR_t;
477 Create_COMMAND_DESCRIPTOR_t _CreateData = {
478     /* entry */           &TPM2_Create,
479     /* inSize */          (UINT16)(sizeof(Create_In)),
480     /* outSize */         (UINT16)(sizeof(Create_Out)),
481     /* offsetOfTypes */   offsetof(Create_COMMAND_DESCRIPTOR_t, types),
482     /* offsets */         {(UINT16)(offsetof(Create_In, inSensitive)),
483                           (UINT16)(offsetof(Create_In, inPublic)),
484                           (UINT16)(offsetof(Create_In, outsideInfo)),
485                           (UINT16)(offsetof(Create_In, creationPCR)),
486                           (UINT16)(offsetof(Create_Out, outPublic)),
487                           (UINT16)(offsetof(Create_Out, creationData)),
488                           (UINT16)(offsetof(Create_Out, creationHash)),
489                           (UINT16)(offsetof(Create_Out, creationTicket))},
490     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
491                           TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
492                           TPM2B_PUBLIC_P_UNMARSHAL,
493                           TPM2B_DATA_P_UNMARSHAL,
494                           TPML_PCR_SELECTION_P_UNMARSHAL,
495                           END_OF_LIST,
496                           TPM2B_PRIVATE_P_MARSHAL,
497                           TPM2B_PUBLIC_P_MARSHAL,
498                           TPM2B_CREATION_DATA_P_MARSHAL,

```

```

499             TPM2B_DIGEST_P MARSHAL,
500             TPMT_TK_CREATION_P_MARSHAL,
501             END_OF_LIST}
502 };
503 #define _CreateDataAddress (&_CreateData)
504 #else
505 #define _CreateDataAddress 0
506 #endif // CC_Create
507 #if CC_Load
508 #include "Load_fp.h"
509 typedef TPM_RC (Load_Entry) (
510     Load_In             *in,
511     Load_Out            *out
512 );
513 typedef const struct {
514     Load_Entry           *entry;
515     UINT16               inSize;
516     UINT16               outSize;
517     UINT16               offsetOfTypes;
518     UINT16               paramOffsets[3];
519     BYTE                 types[7];
520 } Load_COMMAND_DESCRIPTOR_t;
521 Load_COMMAND_DESCRIPTOR_t _LoadData = {
522     /* entry */           &TPM2_Load,
523     /* inSize */          (UINT16) (sizeof(Load_In)),
524     /* outSize */         (UINT16) (sizeof(Load_Out)),
525     /* offsetOfTypes */   offsetof(Load_COMMAND_DESCRIPTOR_t, types),
526     /* offsets */         {(UINT16) (offsetof(Load_In, inPrivate)),
527                           (UINT16) (offsetof(Load_In, inPublic)),
528                           (UINT16) (offsetof(Load_Out, name))},
529     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
530                           TPM2B_PRIVATE_P_UNMARSHAL,
531                           TPM2B_PUBLIC_P_UNMARSHAL,
532                           END_OF_LIST,
533                           TPM_HANDLE_H_MARSHAL,
534                           TPM2B_NAME_P_MARSHAL,
535                           END_OF_LIST}
536 };
537 #define _LoadDataAddress (&_LoadData)
538 #else
539 #define _LoadDataAddress 0
540 #endif // CC_Load
541 #if CC_LoadExternal
542 #include "LoadExternal_fp.h"
543 typedef TPM_RC (LoadExternal_Entry) (
544     LoadExternal_In      *in,
545     LoadExternal_Out     *out
546 );
547 typedef const struct {
548     LoadExternal_Entry    *entry;
549     UINT16               inSize;
550     UINT16               outSize;
551     UINT16               offsetOfTypes;
552     UINT16               paramOffsets[3];
553     BYTE                 types[7];
554 } LoadExternal_COMMAND_DESCRIPTOR_t;
555 LoadExternal_COMMAND_DESCRIPTOR_t _LoadExternalData = {
556     /* entry */           &TPM2_LoadExternal,
557     /* inSize */          (UINT16) (sizeof(LoadExternal_In)),
558     /* outSize */         (UINT16) (sizeof(LoadExternal_Out)),
559     /* offsetOfTypes */   offsetof(LoadExternal_COMMAND_DESCRIPTOR_t, types),
560     /* offsets */         {(UINT16) (offsetof(LoadExternal_In, inPublic)),
561                           (UINT16) (offsetof(LoadExternal_In, hierarchy)),
562                           (UINT16) (offsetof(LoadExternal_Out, name))},
563     /* types */           {TPM2B_SENSITIVE_P_UNMARSHAL,
564                           TPM2B_PUBLIC_P_UNMARSHAL + ADD_FLAG,

```



```

565         TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
566         END_OF_LIST,
567         TPM_HANDLE_H_MARSHAL,
568         TPM2B_NAME_P_MARSHAL,
569         END_OF_LIST}
570 };
571 #define _LoadExternalDataAddress (&LoadExternalData)
572 #else
573 #define LoadExternalDataAddress 0
574 #endif // CC_LoadExternal
575 #if CC_ReadPublic
576 #include "ReadPublic_fp.h"
577 typedef TPM_RC (ReadPublic_Entry)(
578     ReadPublic_In          *in,
579     ReadPublic_Out         *out
580 );
581 typedef const struct {
582     ReadPublic_Entry      *entry;
583     UINT16                inSize;
584     UINT16                outSize;
585     UINT16                offsetOfTypes;
586     UINT16                paramOffsets[2];
587     BYTE                  types[6];
588 } ReadPublic_COMMAND_DESCRIPTOR_t;
589 ReadPublic_COMMAND_DESCRIPTOR_t ReadPublicData = {
590     /* entry */          &TPM2_ReadPublic,
591     /* inSize */         (UINT16)(sizeof(ReadPublic_In)),
592     /* outSize */        (UINT16)(sizeof(ReadPublic_Out)),
593     /* offsetOfTypes */  offsetof(ReadPublic_COMMAND_DESCRIPTOR_t, types),
594     /* offsets */        {(UINT16)(offsetof(ReadPublic_Out, name)),
595                          (UINT16)(offsetof(ReadPublic_Out, qualifiedName))},
596     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
597                          END_OF_LIST,
598                          TPM2B_PUBLIC_P_MARSHAL,
599                          TPM2B_NAME_P_MARSHAL,
600                          TPM2B_NAME_P_MARSHAL,
601                          END_OF_LIST}
602 };
603 #define _ReadPublicDataAddress (&ReadPublicData)
604 #else
605 #define ReadPublicDataAddress 0
606 #endif // CC_ReadPublic
607 #if CC_ActivateCredential
608 #include "ActivateCredential_fp.h"
609 typedef TPM_RC (ActivateCredential_Entry)(
610     ActivateCredential_In  *in,
611     ActivateCredential_Out *out
612 );
613 typedef const struct {
614     ActivateCredential_Entry *entry;
615     UINT16                   inSize;
616     UINT16                   outSize;
617     UINT16                   offsetOfTypes;
618     UINT16                   paramOffsets[3];
619     BYTE                     types[7];
620 } ActivateCredential_COMMAND_DESCRIPTOR_t;
621 ActivateCredential_COMMAND_DESCRIPTOR_t ActivateCredentialData = {
622     /* entry */          &TPM2_ActivateCredential,
623     /* inSize */         (UINT16)(sizeof(ActivateCredential_In)),
624     /* outSize */        (UINT16)(sizeof(ActivateCredential_Out)),
625     /* offsetOfTypes */  offsetof(ActivateCredential_COMMAND_DESCRIPTOR_t,
626 types),
627     /* offsets */        {(UINT16)(offsetof(ActivateCredential_In, keyHandle)),
628                          (UINT16)(offsetof(ActivateCredential_In,
629 credentialBlob)),
630                          (UINT16)(offsetof(ActivateCredential_In, secret))},

```

```

629     /* types */
630
631     TPMI_DH_OBJECT_H_UNMARSHAL,
632     TPM2B_ID_OBJECT_P_UNMARSHAL,
633     TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
634     END_OF_LIST,
635     TPM2B_DIGEST_P_MARSHAL,
636     END_OF_LIST}
637 };
638 #define _ActivateCredentialDataAddress (&_ActivateCredentialData)
639 #else
640 #define _ActivateCredentialDataAddress 0
641 #endif // CC_ActivateCredential
642 #if CC_MakeCredential
643 #include "MakeCredential_fp.h"
644 typedef TPM_RC (MakeCredential_Entry) (
645     MakeCredential_In *in,
646     MakeCredential_Out *out
647 );
648 typedef const struct {
649     MakeCredential_Entry *entry;
650     UINT16 inSize;
651     UINT16 outSize;
652     UINT16 offsetOfTypes;
653     paramOffsets[3];
654     BYTE types[7];
655 } MakeCredential_COMMAND_DESCRIPTOR_t;
656 MakeCredential_COMMAND_DESCRIPTOR_t _MakeCredentialData = {
657     /* entry */ &TPM2_MakeCredential,
658     /* inSize */ (UINT16) (sizeof(MakeCredential_In)),
659     /* outSize */ (UINT16) (sizeof(MakeCredential_Out)),
660     /* offsetOfTypes */ offsetof(MakeCredential_COMMAND_DESCRIPTOR_t, types),
661     /* offsets */ { (UINT16) (offsetof(MakeCredential_In, credential)),
662                   (UINT16) (offsetof(MakeCredential_In, objectName)),
663                   (UINT16) (offsetof(MakeCredential_Out, secret)) },
664     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
665                 TPM2B_DIGEST_P_UNMARSHAL,
666                 TPM2B_NAME_P_UNMARSHAL,
667                 END_OF_LIST,
668                 TPM2B_ID_OBJECT_P_MARSHAL,
669                 TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
670                 END_OF_LIST }
671 };
672 #define _MakeCredentialDataAddress (&_MakeCredentialData)
673 #else
674 #define _MakeCredentialDataAddress 0
675 #endif // CC_MakeCredential
676 #if CC_Unseal
677 #include "Unseal_fp.h"
678 typedef TPM_RC (Unseal_Entry) (
679     Unseal_In *in,
680     Unseal_Out *out
681 );
682 typedef const struct {
683     Unseal_Entry *entry;
684     UINT16 inSize;
685     UINT16 outSize;
686     UINT16 offsetOfTypes;
687     BYTE types[4];
688 } Unseal_COMMAND_DESCRIPTOR_t;
689 Unseal_COMMAND_DESCRIPTOR_t _UnsealData = {
690     /* entry */ &TPM2_Unseal,
691     /* inSize */ (UINT16) (sizeof(Unseal_In)),
692     /* outSize */ (UINT16) (sizeof(Unseal_Out)),
693     /* offsetOfTypes */ offsetof(Unseal_COMMAND_DESCRIPTOR_t, types),
694     /* offsets */ // No parameter offsets;
695     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,

```

```

695                                     END_OF_LIST,
696                                     TPM2B_SENSITIVE_DATA_P_MARSHAL,
697                                     END_OF_LIST}
698 };
699 #define _UnsealDataAddress (&_UnsealData)
700 #else
701 #define _UnsealDataAddress 0
702 #endif // CC_Unseal
703 #if CC_ObjectChangeAuth
704 #include "ObjectChangeAuth_fp.h"
705 typedef TPM_RC (ObjectChangeAuth_Entry) (
706     ObjectChangeAuth_In      *in,
707     ObjectChangeAuth_Out     *out
708 );
709 typedef const struct {
710     ObjectChangeAuth_Entry  *entry;
711     UINT16                  inSize;
712     UINT16                  outSize;
713     UINT16                  offsetOfTypes;
714     UINT16                  paramOffsets[2];
715     BYTE                    types[6];
716 } ObjectChangeAuth_COMMAND_DESCRIPTOR_t;
717 ObjectChangeAuth_COMMAND_DESCRIPTOR_t _ObjectChangeAuthData = {
718     /* entry */           &TPM2_ObjectChangeAuth,
719     /* inSize */          (UINT16) (sizeof(ObjectChangeAuth_In)),
720     /* outSize */         (UINT16) (sizeof(ObjectChangeAuth_Out)),
721     /* offsetOfTypes */   offsetof(ObjectChangeAuth_COMMAND_DESCRIPTOR_t, types),
722     /* offsets */         { (UINT16) (offsetof(ObjectChangeAuth_In, parentHandle)),
723                           (UINT16) (offsetof(ObjectChangeAuth_In, newAuth)) },
724     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
725                           TPMI_DH_OBJECT_H_UNMARSHAL,
726                           TPM2B_AUTH_P_UNMARSHAL,
727                           END_OF_LIST,
728                           TPM2B_PRIVATE_P_MARSHAL,
729                           END_OF_LIST}
730 };
731 #define _ObjectChangeAuthDataAddress (&_ObjectChangeAuthData)
732 #else
733 #define _ObjectChangeAuthDataAddress 0
734 #endif // CC_ObjectChangeAuth
735 #if CC_CreateLoaded
736 #include "CreateLoaded_fp.h"
737 typedef TPM_RC (CreateLoaded_Entry) (
738     CreateLoaded_In          *in,
739     CreateLoaded_Out         *out
740 );
741 typedef const struct {
742     CreateLoaded_Entry       *entry;
743     UINT16                   inSize;
744     UINT16                   outSize;
745     UINT16                   offsetOfTypes;
746     UINT16                   paramOffsets[5];
747     BYTE                     types[9];
748 } CreateLoaded_COMMAND_DESCRIPTOR_t;
749 CreateLoaded_COMMAND_DESCRIPTOR_t _CreateLoadedData = {
750     /* entry */           &TPM2_CreateLoaded,
751     /* inSize */          (UINT16) (sizeof(CreateLoaded_In)),
752     /* outSize */         (UINT16) (sizeof(CreateLoaded_Out)),
753     /* offsetOfTypes */   offsetof(CreateLoaded_COMMAND_DESCRIPTOR_t, types),
754     /* offsets */         { (UINT16) (offsetof(CreateLoaded_In, inSensitive)),
755                           (UINT16) (offsetof(CreateLoaded_In, inPublic)),
756                           (UINT16) (offsetof(CreateLoaded_Out, outPrivate)),
757                           (UINT16) (offsetof(CreateLoaded_Out, outPublic)),
758                           (UINT16) (offsetof(CreateLoaded_Out, name)) },
759     /* types */           {TPMI_DH_PARENT_H_UNMARSHAL + ADD_FLAG,
760                           TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,

```

```

761         TPM2B_TEMPLATE_P_UNMARSHAL,
762         END_OF_LIST,
763         TPM_HANDLE_H_MARSHAL,
764         TPM2B_PRIVATE_P_MARSHAL,
765         TPM2B_PUBLIC_P_MARSHAL,
766         TPM2B_NAME_P_MARSHAL,
767         END_OF_LIST}
768 };
769 #define _CreateLoadedDataAddress (&_CreateLoadedData)
770 #else
771 #define _CreateLoadedDataAddress 0
772 #endif // CC_CreateLoaded
773 #if CC_Duplicate
774 #include "Duplicate_fp.h"
775 typedef TPM_RC (Duplicate_Entry) (
776     Duplicate_In      *in,
777     Duplicate_Out     *out
778 );
779 typedef const struct {
780     Duplicate_Entry    *entry;
781     UINT16             inSize;
782     UINT16             outSize;
783     UINT16             offsetOfTypes;
784     UINT16             paramOffsets[5];
785     BYTE              types[9];
786 } Duplicate_COMMAND_DESCRIPTOR_t;
787 Duplicate_COMMAND_DESCRIPTOR_t _DuplicateData = {
788     /* entry */           &TPM2_Duplicate,
789     /* inSize */         (UINT16) (sizeof(Duplicate_In)),
790     /* outSize */        (UINT16) (sizeof(Duplicate_Out)),
791     /* offsetOfTypes */  offsetof(Duplicate_COMMAND_DESCRIPTOR_t, types),
792     /* offsets */        { (UINT16) (offsetof(Duplicate_In, newParentHandle)),
793                          (UINT16) (offsetof(Duplicate_In, encryptionKeyIn)),
794                          (UINT16) (offsetof(Duplicate_In, symmetricAlg)),
795                          (UINT16) (offsetof(Duplicate_Out, duplicate)),
796                          (UINT16) (offsetof(Duplicate_Out, outSymSeed)) },
797     /* types */          { TPMI_DH_OBJECT_H_UNMARSHAL,
798                          TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
799                          TPM2B_DATA_P_UNMARSHAL,
800                          TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
801                          END_OF_LIST,
802                          TPM2B_DATA_P_MARSHAL,
803                          TPM2B_PRIVATE_P_MARSHAL,
804                          TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
805                          END_OF_LIST}
806 };
807 #define _DuplicateDataAddress (&_DuplicateData)
808 #else
809 #define _DuplicateDataAddress 0
810 #endif // CC_Duplicate
811 #if CC_Rewrap
812 #include "Rewrap_fp.h"
813 typedef TPM_RC (Rewrap_Entry) (
814     Rewrap_In          *in,
815     Rewrap_Out         *out
816 );
817 typedef const struct {
818     Rewrap_Entry        *entry;
819     UINT16              inSize;
820     UINT16              outSize;
821     UINT16              offsetOfTypes;
822     UINT16              paramOffsets[5];
823     BYTE               types[9];
824 } Rewrap_COMMAND_DESCRIPTOR_t;
825 Rewrap_COMMAND_DESCRIPTOR_t _RewrapData = {
826     /* entry */           &TPM2_Rewrap,

```

```

827     /* inSize      */ (UINT16) (sizeof(Rewrap_In)),
828     /* outSize     */ (UINT16) (sizeof(Rewrap_Out)),
829     /* offsetOfTypes */ offsetof(Rewrap_COMMAND_DESCRIPTOR_t, types),
830     /* offsets      */ {(UINT16) (offsetof(Rewrap_In, newParent)),
831                        (UINT16) (offsetof(Rewrap_In, inDuplicate)),
832                        (UINT16) (offsetof(Rewrap_In, name)),
833                        (UINT16) (offsetof(Rewrap_In, inSymSeed)),
834                        (UINT16) (offsetof(Rewrap_Out, outSymSeed))},
835     /* types        */ {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
836                        TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
837                        TPM2B_PRIVATE_P_UNMARSHAL,
838                        TPM2B_NAME_P_UNMARSHAL,
839                        TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
840                        END_OF_LIST,
841                        TPM2B_PRIVATE_P_MARSHAL,
842                        TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
843                        END_OF_LIST}
844 };
845 #define _RewrapDataAddress (&_RewrapData)
846 #else
847 #define _RewrapDataAddress 0
848 #endif // CC_Rewrap
849 #if CC_Import
850 #include "Import_fp.h"
851 typedef TPM_RC (Import_Entry) (
852     Import_In          *in,
853     Import_Out         *out
854 );
855 typedef const struct {
856     Import_Entry      *entry;
857     UINT16            inSize;
858     UINT16            outSize;
859     UINT16            offsetOfTypes;
860     UINT16            paramOffsets[5];
861     BYTE              types[9];
862 } Import_COMMAND_DESCRIPTOR_t;
863 Import_COMMAND_DESCRIPTOR_t _ImportData = {
864     /* entry      */ &TPM2_Import,
865     /* inSize     */ (UINT16) (sizeof(Import_In)),
866     /* outSize    */ (UINT16) (sizeof(Import_Out)),
867     /* offsetOfTypes */ offsetof(Import_COMMAND_DESCRIPTOR_t, types),
868     /* offsets     */ {(UINT16) (offsetof(Import_In, encryptionKey)),
869                      (UINT16) (offsetof(Import_In, objectPublic)),
870                      (UINT16) (offsetof(Import_In, duplicate)),
871                      (UINT16) (offsetof(Import_In, inSymSeed)),
872                      (UINT16) (offsetof(Import_In, symmetricAlg))},
873     /* types      */ {TPMI_DH_OBJECT_H_UNMARSHAL,
874                      TPM2B_DATA_P_UNMARSHAL,
875                      TPM2B_PUBLIC_P_UNMARSHAL,
876                      TPM2B_PRIVATE_P_UNMARSHAL,
877                      TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
878                      TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
879                      END_OF_LIST,
880                      TPM2B_PRIVATE_P_MARSHAL,
881                      END_OF_LIST}
882 };
883 #define _ImportDataAddress (&_ImportData)
884 #else
885 #define _ImportDataAddress 0
886 #endif // CC_Import
887 #if CC_RSA_Encrypt
888 #include "RSA_Encrypt_fp.h"
889 typedef TPM_RC (RSA_Encrypt_Entry) (
890     RSA_Encrypt_In      *in,
891     RSA_Encrypt_Out     *out
892 );

```

```

893 typedef const struct {
894     RSA_Encrypt_Entry      *entry;
895     UINT16                 inSize;
896     UINT16                 outSize;
897     UINT16                 offsetOfTypes;
898     UINT16                 paramOffsets[3];
899     BYTE                   types[7];
900 } RSA_Encrypt_COMMAND_DESCRIPTOR_t;
901 RSA_Encrypt_COMMAND_DESCRIPTOR_t _RSA_EncryptData = {
902     /* entry */           &TPM2_RSA_Encrypt,
903     /* inSize */         (UINT16) (sizeof(RSA_Encrypt_In)),
904     /* outSize */        (UINT16) (sizeof(RSA_Encrypt_Out)),
905     /* offsetOfTypes */  offsetof(RSA_Encrypt_COMMAND_DESCRIPTOR_t, types),
906     /* offsets */        {(UINT16) (offsetof(RSA_Encrypt_In, message)),
907                          (UINT16) (offsetof(RSA_Encrypt_In, inScheme)),
908                          (UINT16) (offsetof(RSA_Encrypt_In, label))},
909     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
910                          TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
911                          TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
912                          TPM2B_DATA_P_UNMARSHAL,
913                          END_OF_LIST,
914                          TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
915                          END_OF_LIST}
916 };
917 #define _RSA_EncryptDataAddress (&_RSA_EncryptData)
918 #else
919 #define _RSA_EncryptDataAddress 0
920 #endif // CC_RSA_Encrypt
921 #if CC_RSA_Decrypt
922 #include "RSA_Decrypt_fp.h"
923 typedef TPM_RC (RSA_Decrypt_Entry) (
924     RSA_Decrypt_In      *in,
925     RSA_Decrypt_Out      *out
926 );
927 typedef const struct {
928     RSA_Decrypt_Entry      *entry;
929     UINT16                 inSize;
930     UINT16                 outSize;
931     UINT16                 offsetOfTypes;
932     UINT16                 paramOffsets[3];
933     BYTE                   types[7];
934 } RSA_Decrypt_COMMAND_DESCRIPTOR_t;
935 RSA_Decrypt_COMMAND_DESCRIPTOR_t _RSA_DecryptData = {
936     /* entry */           &TPM2_RSA_Decrypt,
937     /* inSize */         (UINT16) (sizeof(RSA_Decrypt_In)),
938     /* outSize */        (UINT16) (sizeof(RSA_Decrypt_Out)),
939     /* offsetOfTypes */  offsetof(RSA_Decrypt_COMMAND_DESCRIPTOR_t, types),
940     /* offsets */        {(UINT16) (offsetof(RSA_Decrypt_In, cipherText)),
941                          (UINT16) (offsetof(RSA_Decrypt_In, inScheme)),
942                          (UINT16) (offsetof(RSA_Decrypt_In, label))},
943     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
944                          TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
945                          TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
946                          TPM2B_DATA_P_UNMARSHAL,
947                          END_OF_LIST,
948                          TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
949                          END_OF_LIST}
950 };
951 #define _RSA_DecryptDataAddress (&_RSA_DecryptData)
952 #else
953 #define _RSA_DecryptDataAddress 0
954 #endif // CC_RSA_Decrypt
955 #if CC_ECDH_KeyGen
956 #include "ECDH_KeyGen_fp.h"
957 typedef TPM_RC (ECDH_KeyGen_Entry) (
958     ECDH_KeyGen_In      *in,

```



```

959     ECDH_KeyGen_Out          *out
960 );
961 typedef const struct {
962     ECDH_KeyGen_Entry        *entry;
963     UINT16                   inSize;
964     UINT16                   outSize;
965     UINT16                   offsetOfTypes;
966     UINT16                   paramOffsets[1];
967     BYTE                     types[5];
968 } ECDH_KeyGen_COMMAND_DESCRIPTOR_t;
969 ECDH_KeyGen_COMMAND_DESCRIPTOR_t _ECDH_KeyGenData = {
970     /* entry */           &TPM2_ECDH_KeyGen,
971     /* inSize */          (UINT16) (sizeof(ECDH_KeyGen_In)),
972     /* outSize */         (UINT16) (sizeof(ECDH_KeyGen_Out)),
973     /* offsetOfTypes */   offsetof(ECDH_KeyGen_COMMAND_DESCRIPTOR_t, types),
974     /* offsets */         {(UINT16) (offsetof(ECDH_KeyGen_Out, pubPoint))},
975     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
976                           END_OF_LIST,
977                           TPM2B_ECC_POINT_P_MARSHAL,
978                           TPM2B_ECC_POINT_P_MARSHAL,
979                           END_OF_LIST};
980 };
981 #define _ECDH_KeyGenDataAddress (&_ECDH_KeyGenData)
982 #else
983 #define _ECDH_KeyGenDataAddress 0
984 #endif // CC_ECDH_KeyGen
985 #if CC_ECDH_ZGen
986 #include "ECDH_ZGen_fp.h"
987 typedef TPM_RC (ECDH_ZGen_Entry) (
988     ECDH_ZGen_In             *in,
989     ECDH_ZGen_Out            *out
990 );
991 typedef const struct {
992     ECDH_ZGen_Entry          *entry;
993     UINT16                   inSize;
994     UINT16                   outSize;
995     UINT16                   offsetOfTypes;
996     UINT16                   paramOffsets[1];
997     BYTE                     types[5];
998 } ECDH_ZGen_COMMAND_DESCRIPTOR_t;
999 ECDH_ZGen_COMMAND_DESCRIPTOR_t _ECDH_ZGenData = {
1000     /* entry */           &TPM2_ECDH_ZGen,
1001     /* inSize */          (UINT16) (sizeof(ECDH_ZGen_In)),
1002     /* outSize */         (UINT16) (sizeof(ECDH_ZGen_Out)),
1003     /* offsetOfTypes */   offsetof(ECDH_ZGen_COMMAND_DESCRIPTOR_t, types),
1004     /* offsets */         {(UINT16) (offsetof(ECDH_ZGen_In, inPoint))},
1005     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1006                           TPM2B_ECC_POINT_P_UNMARSHAL,
1007                           END_OF_LIST,
1008                           TPM2B_ECC_POINT_P_MARSHAL,
1009                           END_OF_LIST};
1010 };
1011 #define _ECDH_ZGenDataAddress (&_ECDH_ZGenData)
1012 #else
1013 #define _ECDH_ZGenDataAddress 0
1014 #endif // CC_ECDH_ZGen
1015 #if CC_ECC_Parameters
1016 #include "ECC_Parameters_fp.h"
1017 typedef TPM_RC (ECC_Parameters_Entry) (
1018     ECC_Parameters_In        *in,
1019     ECC_Parameters_Out       *out
1020 );
1021 typedef const struct {
1022     ECC_Parameters_Entry     *entry;
1023     UINT16                   inSize;
1024     UINT16                   outSize;

```

```

1025     UINT16                offsetOfTypes;
1026     BYTE                  types[4];
1027 } ECC_Parameters_COMMAND_DESCRIPTOR_t;
1028 ECC_Parameters_COMMAND_DESCRIPTOR_t _ECC_ParametersData = {
1029     /* entry */           &TPM2_ECC_Parameters,
1030     /* inSize */          (UINT16) (sizeof(ECC_Parameters_In)),
1031     /* outSize */         (UINT16) (sizeof(ECC_Parameters_Out)),
1032     /* offsetOfTypes */   offsetof(ECC_Parameters_COMMAND_DESCRIPTOR_t, types),
1033     /* offsets */         // No parameter offsets;
1034     /* types */           {TPMI_ECC_CURVE_P_UNMARSHAL,
1035                           END_OF_LIST,
1036                           TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL,
1037                           END_OF_LIST};
1038 };
1039 #define _ECC_ParametersDataAddress (&_ECC_ParametersData)
1040 #else
1041 #define _ECC_ParametersDataAddress 0
1042 #endif // CC_ECC_Parameters
1043 #if CC_ZGen_2Phase
1044 #include "ZGen_2Phase_fp.h"
1045 typedef TPM_RC (ZGen_2Phase_Entry) (
1046     ZGen_2Phase_In          *in,
1047     ZGen_2Phase_Out         *out
1048 );
1049 typedef const struct {
1050     ZGen_2Phase_Entry        *entry;
1051     UINT16                   inSize;
1052     UINT16                   outSize;
1053     UINT16                   offsetOfTypes;
1054     UINT16                   paramOffsets[5];
1055     BYTE                     types[9];
1056 } ZGen_2Phase_COMMAND_DESCRIPTOR_t;
1057 ZGen_2Phase_COMMAND_DESCRIPTOR_t _ZGen_2PhaseData = {
1058     /* entry */           &TPM2_ZGen_2Phase,
1059     /* inSize */          (UINT16) (sizeof(ZGen_2Phase_In)),
1060     /* outSize */         (UINT16) (sizeof(ZGen_2Phase_Out)),
1061     /* offsetOfTypes */   offsetof(ZGen_2Phase_COMMAND_DESCRIPTOR_t, types),
1062     /* offsets */         {(UINT16) (offsetof(ZGen_2Phase_In, inQsB)),
1063                           (UINT16) (offsetof(ZGen_2Phase_In, inQeB)),
1064                           (UINT16) (offsetof(ZGen_2Phase_In, inScheme)),
1065                           (UINT16) (offsetof(ZGen_2Phase_In, counter)),
1066                           (UINT16) (offsetof(ZGen_2Phase_Out, outZ2))},
1067     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1068                           TPM2B_ECC_POINT_P_UNMARSHAL,
1069                           TPM2B_ECC_POINT_P_UNMARSHAL,
1070                           TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL,
1071                           UINT16_P_UNMARSHAL,
1072                           END_OF_LIST,
1073                           TPM2B_ECC_POINT_P_MARSHAL,
1074                           TPM2B_ECC_POINT_P_MARSHAL,
1075                           END_OF_LIST};
1076 };
1077 #define _ZGen_2PhaseDataAddress (&_ZGen_2PhaseData)
1078 #else
1079 #define _ZGen_2PhaseDataAddress 0
1080 #endif // CC_ZGen_2Phase
1081 #if CC_EncryptDecrypt
1082 #include "EncryptDecrypt_fp.h"
1083 typedef TPM_RC (EncryptDecrypt_Entry) (
1084     EncryptDecrypt_In        *in,
1085     EncryptDecrypt_Out       *out
1086 );
1087 typedef const struct {
1088     EncryptDecrypt_Entry      *entry;
1089     UINT16                   inSize;
1090     UINT16                   outSize;

```



```

1091     UINT16                offsetOfTypes;
1092     UINT16                paramOffsets[5];
1093     BYTE                  types[9];
1094 } EncryptDecrypt_COMMAND_DESCRIPTOR_t;
1095 EncryptDecrypt_COMMAND_DESCRIPTOR_t _EncryptDecryptData = {
1096     /* entry */           &TPM2_EncryptDecrypt,
1097     /* inSize */          (UINT16) (sizeof(EncryptDecrypt_In)),
1098     /* outSize */         (UINT16) (sizeof(EncryptDecrypt_Out)),
1099     /* offsetOfTypes */   offsetof(EncryptDecrypt_COMMAND_DESCRIPTOR_t, types),
1100     /* offsets */         { (UINT16) (offsetof(EncryptDecrypt_In, decrypt)),
1101                           (UINT16) (offsetof(EncryptDecrypt_In, mode)),
1102                           (UINT16) (offsetof(EncryptDecrypt_In, ivIn)),
1103                           (UINT16) (offsetof(EncryptDecrypt_In, inData)),
1104                           (UINT16) (offsetof(EncryptDecrypt_Out, ivOut))},
1105     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1106                           TPMI_YES_NO_P_UNMARSHAL,
1107                           TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1108                           TPM2B_IV_P_UNMARSHAL,
1109                           TPM2B_MAX_BUFFER_P_UNMARSHAL,
1110                           END_OF_LIST,
1111                           TPM2B_MAX_BUFFER_P_MARSHAL,
1112                           TPM2B_IV_P_MARSHAL,
1113                           END_OF_LIST}
1114 };
1115 #define _EncryptDecryptDataAddress (&_EncryptDecryptData)
1116 #else
1117 #define _EncryptDecryptDataAddress 0
1118 #endif // CC_EncryptDecrypt
1119 #if CC_EncryptDecrypt2
1120 #include "EncryptDecrypt2_fp.h"
1121 typedef TPM_RC (EncryptDecrypt2_Entry) (
1122     EncryptDecrypt2_In      *in,
1123     EncryptDecrypt2_Out     *out
1124 );
1125 typedef const struct {
1126     EncryptDecrypt2_Entry   *entry;
1127     UINT16                  inSize;
1128     UINT16                  outSize;
1129     UINT16                  offsetOfTypes;
1130     UINT16                  paramOffsets[5];
1131     BYTE                    types[9];
1132 } EncryptDecrypt2_COMMAND_DESCRIPTOR_t;
1133 EncryptDecrypt2_COMMAND_DESCRIPTOR_t _EncryptDecrypt2Data = {
1134     /* entry */           &TPM2_EncryptDecrypt2,
1135     /* inSize */          (UINT16) (sizeof(EncryptDecrypt2_In)),
1136     /* outSize */         (UINT16) (sizeof(EncryptDecrypt2_Out)),
1137     /* offsetOfTypes */   offsetof(EncryptDecrypt2_COMMAND_DESCRIPTOR_t, types),
1138     /* offsets */         { (UINT16) (offsetof(EncryptDecrypt2_In, inData)),
1139                           (UINT16) (offsetof(EncryptDecrypt2_In, decrypt)),
1140                           (UINT16) (offsetof(EncryptDecrypt2_In, mode)),
1141                           (UINT16) (offsetof(EncryptDecrypt2_In, ivIn)),
1142                           (UINT16) (offsetof(EncryptDecrypt2_Out, ivOut))},
1143     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1144                           TPM2B_MAX_BUFFER_P_UNMARSHAL,
1145                           TPMI_YES_NO_P_UNMARSHAL,
1146                           TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1147                           TPM2B_IV_P_UNMARSHAL,
1148                           END_OF_LIST,
1149                           TPM2B_MAX_BUFFER_P_MARSHAL,
1150                           TPM2B_IV_P_MARSHAL,
1151                           END_OF_LIST}
1152 };
1153 #define _EncryptDecrypt2DataAddress (&_EncryptDecrypt2Data)
1154 #else
1155 #define _EncryptDecrypt2DataAddress 0
1156 #endif // CC_EncryptDecrypt2

```

```

1157 #if CC_Hash
1158 #include "Hash_fp.h"
1159 typedef TPM_RC (Hash_Entry) (
1160     Hash_In          *in,
1161     Hash_Out         *out
1162 );
1163 typedef const struct {
1164     Hash_Entry        *entry;
1165     UINT16            inSize;
1166     UINT16            outSize;
1167     UINT16            offsetOfTypes;
1168     UINT16            paramOffsets[3];
1169     BYTE              types[7];
1170 } Hash_COMMAND_DESCRIPTOR_t;
1171 Hash_COMMAND_DESCRIPTOR_t _HashData = {
1172     /* entry */          &TPM2_Hash,
1173     /* inSize */         (UINT16) (sizeof(Hash_In)),
1174     /* outSize */        (UINT16) (sizeof(Hash_Out)),
1175     /* offsetOfTypes */  offsetof(Hash_COMMAND_DESCRIPTOR_t, types),
1176     /* offsets */        {(UINT16) (offsetof(Hash_In, hashAlg)),
1177                          (UINT16) (offsetof(Hash_In, hierarchy)),
1178                          (UINT16) (offsetof(Hash_Out, validation))},
1179     /* types */          {TPM2B_MAX_BUFFER_P_UNMARSHAL,
1180                          TPMI_ALG_HASH_P_UNMARSHAL,
1181                          TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1182                          END_OF_LIST,
1183                          TPM2B_DIGEST_P_MARSHAL,
1184                          TPMT_TK_HASHCHECK_P_MARSHAL,
1185                          END_OF_LIST}
1186 };
1187 #define _HashDataAddress (&_HashData)
1188 #else
1189 #define _HashDataAddress 0
1190 #endif // CC_Hash
1191 #if CC_HMAC
1192 #include "HMAC_fp.h"
1193 typedef TPM_RC (HMAC_Entry) (
1194     HMAC_In          *in,
1195     HMAC_Out         *out
1196 );
1197 typedef const struct {
1198     HMAC_Entry        *entry;
1199     UINT16            inSize;
1200     UINT16            outSize;
1201     UINT16            offsetOfTypes;
1202     UINT16            paramOffsets[2];
1203     BYTE              types[6];
1204 } HMAC_COMMAND_DESCRIPTOR_t;
1205 HMAC_COMMAND_DESCRIPTOR_t _HMACData = {
1206     /* entry */          &TPM2_HMAC,
1207     /* inSize */         (UINT16) (sizeof(HMAC_In)),
1208     /* outSize */        (UINT16) (sizeof(HMAC_Out)),
1209     /* offsetOfTypes */  offsetof(HMAC_COMMAND_DESCRIPTOR_t, types),
1210     /* offsets */        {(UINT16) (offsetof(HMAC_In, buffer)),
1211                          (UINT16) (offsetof(HMAC_In, hashAlg))},
1212     /* types */          {TPMI_DH_OBJECT_H_UNMARSHAL,
1213                          TPM2B_MAX_BUFFER_P_UNMARSHAL,
1214                          TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1215                          END_OF_LIST,
1216                          TPM2B_DIGEST_P_MARSHAL,
1217                          END_OF_LIST}
1218 };
1219 #define _HMACDataAddress (&_HMACData)
1220 #else
1221 #define _HMACDataAddress 0
1222 #endif // CC_HMAC

```

```

1223 #if CC_MAC
1224 #include "MAC_fp.h"
1225 typedef TPM_RC (MAC_Entry) (
1226     MAC_In          *in,
1227     MAC_Out         *out
1228 );
1229 typedef const struct {
1230     MAC_Entry        *entry;
1231     UINT16           inSize;
1232     UINT16           outSize;
1233     UINT16           offsetOfTypes;
1234     UINT16           paramOffsets[2];
1235     BYTE             types[6];
1236 } MAC_COMMAND_DESCRIPTOR_t;
1237 MAC_COMMAND_DESCRIPTOR_t _MACData = {
1238     /* entry */           &TPM2_MAC,
1239     /* inSize */          (UINT16) (sizeof(MAC_In)),
1240     /* outSize */         (UINT16) (sizeof(MAC_Out)),
1241     /* offsetOfTypes */   offsetof(MAC_COMMAND_DESCRIPTOR_t, types),
1242     /* offsets */         {(UINT16) (offsetof(MAC_In, buffer)),
1243                          (UINT16) (offsetof(MAC_In, inScheme))},
1244     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1245                          TPM2B_MAX_BUFFER_P_UNMARSHAL,
1246                          TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1247                          END_OF_LIST,
1248                          TPM2B_DIGEST_P_MARSHAL,
1249                          END_OF_LIST};
1250 };
1251 #define _MACDataAddress (&_MACData)
1252 #else
1253 #define _MACDataAddress 0
1254 #endif // CC_MAC
1255 #if CC_GetRandom
1256 #include "GetRandom_fp.h"
1257 typedef TPM_RC (GetRandom_Entry) (
1258     GetRandom_In     *in,
1259     GetRandom_Out    *out
1260 );
1261 typedef const struct {
1262     GetRandom_Entry   *entry;
1263     UINT16            inSize;
1264     UINT16            outSize;
1265     UINT16            offsetOfTypes;
1266     BYTE              types[4];
1267 } GetRandom_COMMAND_DESCRIPTOR_t;
1268 GetRandom_COMMAND_DESCRIPTOR_t _GetRandomData = {
1269     /* entry */           &TPM2_GetRandom,
1270     /* inSize */          (UINT16) (sizeof(GetRandom_In)),
1271     /* outSize */         (UINT16) (sizeof(GetRandom_Out)),
1272     /* offsetOfTypes */   offsetof(GetRandom_COMMAND_DESCRIPTOR_t, types),
1273     /* offsets */         // No parameter offsets;
1274     /* types */           {UINT16_P_UNMARSHAL,
1275                          END_OF_LIST,
1276                          TPM2B_DIGEST_P_MARSHAL,
1277                          END_OF_LIST};
1278 };
1279 #define _GetRandomDataAddress (&_GetRandomData)
1280 #else
1281 #define _GetRandomDataAddress 0
1282 #endif // CC_GetRandom
1283 #if CC_StirRandom
1284 #include "StirRandom_fp.h"
1285 typedef TPM_RC (StirRandom_Entry) (
1286     StirRandom_In     *in
1287 );
1288 typedef const struct {

```

```

1289     StirRandom_Entry      *entry;
1290     UINT16                 inSize;
1291     UINT16                 outSize;
1292     UINT16                 offsetOfTypes;
1293     BYTE                   types[3];
1294 } StirRandom_COMMAND_DESCRIPTOR_t;
1295 StirRandom_COMMAND_DESCRIPTOR_t _StirRandomData = {
1296     /* entry */           &TPM2_StirRandom,
1297     /* inSize */          (UINT16) (sizeof(StirRandom_In)),
1298     /* outSize */         0,
1299     /* offsetOfTypes */   offsetof(StirRandom_COMMAND_DESCRIPTOR_t, types),
1300     /* offsets */         // No parameter offsets;
1301     /* types */           {TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1302                           END_OF_LIST,
1303                           END_OF_LIST};
1304 };
1305 #define _StirRandomDataAddress (&_StirRandomData)
1306 #else
1307 #define _StirRandomDataAddress 0
1308 #endif // CC_StirRandom
1309 #if CC_HMAC_Start
1310 #include "HMAC_Start_fp.h"
1311 typedef TPM_RC (HMAC_Start_Entry) (
1312     HMAC_Start_In          *in,
1313     HMAC_Start_Out         *out
1314 );
1315 typedef const struct {
1316     HMAC_Start_Entry      *entry;
1317     UINT16                 inSize;
1318     UINT16                 outSize;
1319     UINT16                 offsetOfTypes;
1320     UINT16                 paramOffsets[2];
1321     BYTE                   types[6];
1322 } HMAC_Start_COMMAND_DESCRIPTOR_t;
1323 HMAC_Start_COMMAND_DESCRIPTOR_t _HMAC_StartData = {
1324     /* entry */           &TPM2_HMAC_Start,
1325     /* inSize */          (UINT16) (sizeof(HMAC_Start_In)),
1326     /* outSize */         (UINT16) (sizeof(HMAC_Start_Out)),
1327     /* offsetOfTypes */   offsetof(HMAC_Start_COMMAND_DESCRIPTOR_t, types),
1328     /* offsets */         {(UINT16) (offsetof(HMAC_Start_In, auth)),
1329                           (UINT16) (offsetof(HMAC_Start_In, hashAlg))},
1330     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL,
1331                           TPM2B_AUTH_P_UNMARSHAL,
1332                           TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1333                           END_OF_LIST,
1334                           TPMI_DH_OBJECT_H_MARSHAL,
1335                           END_OF_LIST};
1336 };
1337 #define _HMAC_StartDataAddress (&_HMAC_StartData)
1338 #else
1339 #define _HMAC_StartDataAddress 0
1340 #endif // CC_HMAC_Start
1341 #if CC_MAC_Start
1342 #include "MAC_Start_fp.h"
1343 typedef TPM_RC (MAC_Start_Entry) (
1344     MAC_Start_In          *in,
1345     MAC_Start_Out         *out
1346 );
1347 typedef const struct {
1348     MAC_Start_Entry      *entry;
1349     UINT16                 inSize;
1350     UINT16                 outSize;
1351     UINT16                 offsetOfTypes;
1352     UINT16                 paramOffsets[2];
1353     BYTE                   types[6];
1354 } MAC_Start_COMMAND_DESCRIPTOR_t;

```

```

1355 MAC_Start_COMMAND_DESCRIPTOR_t _MAC_StartData = {
1356     /* entry */ &TPM2_MAC_Start,
1357     /* inSize */ (UINT16) (sizeof(MAC_Start_In)),
1358     /* outSize */ (UINT16) (sizeof(MAC_Start_Out)),
1359     /* offsetOfTypes */ offsetof(MAC_Start_COMMAND_DESCRIPTOR_t, types),
1360     /* offsets */ { (UINT16) (offsetof(MAC_Start_In, auth)),
1361                   (UINT16) (offsetof(MAC_Start_In, inScheme)) },
1362     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
1363                 TPM2B_AUTH_P_UNMARSHAL,
1364                 TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
1365                 END_OF_LIST,
1366                 TPMI_DH_OBJECT_H_MARSHAL,
1367                 END_OF_LIST };
1368 };
1369 #define _MAC_StartDataAddress (&_MAC_StartData)
1370 #else
1371 #define _MAC_StartDataAddress 0
1372 #endif // CC_MAC_Start
1373 #if CC_HashSequenceStart
1374 #include "HashSequenceStart_fp.h"
1375 typedef TPM_RC (HashSequenceStart_Entry) (
1376     HashSequenceStart_In *in,
1377     HashSequenceStart_Out *out
1378 );
1379 typedef const struct {
1380     HashSequenceStart_Entry *entry;
1381     UINT16 inSize;
1382     UINT16 outSize;
1383     UINT16 offsetOfTypes;
1384     UINT16 paramOffsets[1];
1385     BYTE types[5];
1386 } HashSequenceStart_COMMAND_DESCRIPTOR_t;
1387 HashSequenceStart_COMMAND_DESCRIPTOR_t _HashSequenceStartData = {
1388     /* entry */ &TPM2_HashSequenceStart,
1389     /* inSize */ (UINT16) (sizeof(HashSequenceStart_In)),
1390     /* outSize */ (UINT16) (sizeof(HashSequenceStart_Out)),
1391     /* offsetOfTypes */ offsetof(HashSequenceStart_COMMAND_DESCRIPTOR_t,
1392     types),
1393     /* offsets */ { (UINT16) (offsetof(HashSequenceStart_In, hashAlg)) },
1394     /* types */ { TPM2B_AUTH_P_UNMARSHAL,
1395                 TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1396                 END_OF_LIST,
1397                 TPMI_DH_OBJECT_H_MARSHAL,
1398                 END_OF_LIST };
1399 };
1400 #define _HashSequenceStartDataAddress (&_HashSequenceStartData)
1401 #else
1402 #define _HashSequenceStartDataAddress 0
1403 #endif // CC_HashSequenceStart
1404 #if CC_SequenceUpdate
1405 #include "SequenceUpdate_fp.h"
1406 typedef TPM_RC (SequenceUpdate_Entry) (
1407     SequenceUpdate_In *in
1408 );
1409 typedef const struct {
1410     SequenceUpdate_Entry *entry;
1411     UINT16 inSize;
1412     UINT16 outSize;
1413     UINT16 offsetOfTypes;
1414     UINT16 paramOffsets[1];
1415     BYTE types[4];
1416 } SequenceUpdate_COMMAND_DESCRIPTOR_t;
1417 SequenceUpdate_COMMAND_DESCRIPTOR_t _SequenceUpdateData = {
1418     /* entry */ &TPM2_SequenceUpdate,
1419     /* inSize */ (UINT16) (sizeof(SequenceUpdate_In)),
1420     /* outSize */ 0,

```

```

1420     /* offsetOfTypes */      offsetof(SequenceUpdate_COMMAND_DESCRIPTOR_t, types),
1421     /* offsets */           { (UINT16) (offsetof(SequenceUpdate_In, buffer)) },
1422     /* types */             { TPMI_DH_OBJECT_H_UNMARSHAL,
1423                             TPM2B_MAX_BUFFER_P_UNMARSHAL,
1424                             END_OF_LIST,
1425                             END_OF_LIST };
1426 };
1427 #define _SequenceUpdateDataAddress (&_SequenceUpdateData)
1428 #else
1429 #define _SequenceUpdateDataAddress 0
1430 #endif // CC_SequenceUpdate
1431 #if CC_SequenceComplete
1432 #include "SequenceComplete_fp.h"
1433 typedef TPM_RC (SequenceComplete_Entry) (
1434     SequenceComplete_In      *in,
1435     SequenceComplete_Out     *out
1436 );
1437 typedef const struct {
1438     SequenceComplete_Entry  *entry;
1439     UINT16                  inSize;
1440     UINT16                  outSize;
1441     UINT16                  offsetOfTypes;
1442     UINT16                  paramOffsets[3];
1443     BYTE                    types[7];
1444 } SequenceComplete_COMMAND_DESCRIPTOR_t;
1445 SequenceComplete_COMMAND_DESCRIPTOR_t _SequenceCompleteData = {
1446     /* entry */              &TPM2_SequenceComplete,
1447     /* inSize */             (UINT16) (sizeof(SequenceComplete_In)),
1448     /* outSize */            (UINT16) (sizeof(SequenceComplete_Out)),
1449     /* offsetOfTypes */      offsetof(SequenceComplete_COMMAND_DESCRIPTOR_t, types),
1450     /* offsets */            { (UINT16) (offsetof(SequenceComplete_In, buffer)),
1451                             (UINT16) (offsetof(SequenceComplete_In, hierarchy)),
1452                             (UINT16) (offsetof(SequenceComplete_Out, validation)) },
1453     /* types */              { TPMI_DH_OBJECT_H_UNMARSHAL,
1454                             TPM2B_MAX_BUFFER_P_UNMARSHAL,
1455                             TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
1456                             END_OF_LIST,
1457                             TPM2B_DIGEST_P_MARSHAL,
1458                             TPMT_TK_HASHCHECK_P_MARSHAL,
1459                             END_OF_LIST };
1460 };
1461 #define _SequenceCompleteDataAddress (&_SequenceCompleteData)
1462 #else
1463 #define _SequenceCompleteDataAddress 0
1464 #endif // CC_SequenceComplete
1465 #if CC_EventSequenceComplete
1466 #include "EventSequenceComplete_fp.h"
1467 typedef TPM_RC (EventSequenceComplete_Entry) (
1468     EventSequenceComplete_In *in,
1469     EventSequenceComplete_Out *out
1470 );
1471 typedef const struct {
1472     EventSequenceComplete_Entry *entry;
1473     UINT16                      inSize;
1474     UINT16                      outSize;
1475     UINT16                      offsetOfTypes;
1476     UINT16                      paramOffsets[2];
1477     BYTE                        types[6];
1478 } EventSequenceComplete_COMMAND_DESCRIPTOR_t;
1479 EventSequenceComplete_COMMAND_DESCRIPTOR_t _EventSequenceCompleteData = {
1480     /* entry */              &TPM2_EventSequenceComplete,
1481     /* inSize */             (UINT16) (sizeof(EventSequenceComplete_In)),
1482     /* outSize */            (UINT16) (sizeof(EventSequenceComplete_Out)),
1483     /* offsetOfTypes */      offsetof(EventSequenceComplete_COMMAND_DESCRIPTOR_t, types),

```



```

1484     /* offsets */                                { (UINT16) (offsetof(EventSequenceComplete_In,
sequenceHandle)),
1485                                                     (UINT16) (offsetof(EventSequenceComplete_In,
buffer))},
1486     /* types */                                {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1487                                                     TPMI_DH_OBJECT_H_UNMARSHAL,
1488                                                     TPM2B_MAX_BUFFER_P_UNMARSHAL,
1489                                                     END_OF_LIST,
1490                                                     TPML_DIGEST_VALUES_P_MARSHAL,
1491                                                     END_OF_LIST}
1492 };
1493 #define _EventSequenceCompleteDataAddress (&_EventSequenceCompleteData)
1494 #else
1495 #define _EventSequenceCompleteDataAddress 0
1496 #endif // CC_EventSequenceComplete
1497 #if CC_Certify
1498 #include "Certify_fp.h"
1499 typedef TPM_RC (Certify_Entry) (
1500     Certify_In                *in,
1501     Certify_Out               *out
1502 );
1503 typedef const struct {
1504     Certify_Entry              *entry;
1505     UINT16                     inSize;
1506     UINT16                     outSize;
1507     UINT16                     offsetOfTypes;
1508     UINT16                     paramOffsets[4];
1509     BYTE                       types[8];
1510 } Certify_COMMAND_DESCRIPTOR_t;
1511 Certify_COMMAND_DESCRIPTOR_t _CertifyData = {
1512     /* entry */                &TPM2_Certify,
1513     /* inSize */               (UINT16) (sizeof(Certify_In)),
1514     /* outSize */              (UINT16) (sizeof(Certify_Out)),
1515     /* offsetOfTypes */        offsetof(Certify_COMMAND_DESCRIPTOR_t, types),
1516     /* offsets */              { (UINT16) (offsetof(Certify_In, signHandle)),
1517                                (UINT16) (offsetof(Certify_In, qualifyingData)),
1518                                (UINT16) (offsetof(Certify_In, inScheme)),
1519                                (UINT16) (offsetof(Certify_Out, signature))},
1520     /* types */                {TPMI_DH_OBJECT_H_UNMARSHAL,
1521                                TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1522                                TPM2B_DATA_P_UNMARSHAL,
1523                                TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1524                                END_OF_LIST,
1525                                TPM2B_ATTEST_P_MARSHAL,
1526                                TPMT_SIGNATURE_P_MARSHAL,
1527                                END_OF_LIST}
1528 };
1529 #define _CertifyDataAddress (&_CertifyData)
1530 #else
1531 #define _CertifyDataAddress 0
1532 #endif // CC_Certify
1533 #if CC_CertifyCreation
1534 #include "CertifyCreation_fp.h"
1535 typedef TPM_RC (CertifyCreation_Entry) (
1536     CertifyCreation_In         *in,
1537     CertifyCreation_Out        *out
1538 );
1539 typedef const struct {
1540     CertifyCreation_Entry       *entry;
1541     UINT16                      inSize;
1542     UINT16                      outSize;
1543     UINT16                      offsetOfTypes;
1544     UINT16                      paramOffsets[6];
1545     BYTE                        types[10];
1546 } CertifyCreation_COMMAND_DESCRIPTOR_t;
1547 CertifyCreation_COMMAND_DESCRIPTOR_t _CertifyCreationData = {

```

```

1548     /* entry */ &TPM2_CertifyCreation,
1549     /* inSize */ (UINT16) (sizeof(CertifyCreation_In)),
1550     /* outSize */ (UINT16) (sizeof(CertifyCreation_Out)),
1551     /* offsetOfTypes */ offsetof(CertifyCreation_COMMAND_DESCRIPTOR_t, types),
1552     /* offsets */ { (UINT16) (offsetof(CertifyCreation_In, objectHandle)),
1553                    (UINT16) (offsetof(CertifyCreation_In, qualifyingData)),
1554                    (UINT16) (offsetof(CertifyCreation_In, creationHash)),
1555                    (UINT16) (offsetof(CertifyCreation_In, inScheme)),
1556                    (UINT16) (offsetof(CertifyCreation_In, creationTicket)),
1557                    (UINT16) (offsetof(CertifyCreation_Out, signature))},
1558     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1559                 TPMI_DH_OBJECT_H_UNMARSHAL,
1560                 TPM2B_DATA_P_UNMARSHAL,
1561                 TPM2B_DIGEST_P_UNMARSHAL,
1562                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1563                 TPMT_TK_CREATION_P_UNMARSHAL,
1564                 END_OF_LIST,
1565                 TPM2B_ATTEST_P_MARSHAL,
1566                 TPMT_SIGNATURE_P_MARSHAL,
1567                 END_OF_LIST};
1568 };
1569 #define _CertifyCreationDataAddress (&CertifyCreationData)
1570 #else
1571 #define _CertifyCreationDataAddress 0
1572 #endif // CC_CertifyCreation
1573 #if CC_Quote
1574 #include "Quote_fp.h"
1575 typedef TPM_RC (Quote_Entry) (
1576     Quote_In *in,
1577     Quote_Out *out
1578 );
1579 typedef const struct {
1580     Quote_Entry *entry;
1581     UINT16 inSize;
1582     UINT16 outSize;
1583     UINT16 offsetOfTypes;
1584     UINT16 paramOffsets[4];
1585     BYTE types[8];
1586 } Quote_COMMAND_DESCRIPTOR_t;
1587 Quote_COMMAND_DESCRIPTOR_t _QuoteData = {
1588     /* entry */ &TPM2_Quote,
1589     /* inSize */ (UINT16) (sizeof(Quote_In)),
1590     /* outSize */ (UINT16) (sizeof(Quote_Out)),
1591     /* offsetOfTypes */ offsetof(Quote_COMMAND_DESCRIPTOR_t, types),
1592     /* offsets */ { (UINT16) (offsetof(Quote_In, qualifyingData)),
1593                   (UINT16) (offsetof(Quote_In, inScheme)),
1594                   (UINT16) (offsetof(Quote_In, PCRselect)),
1595                   (UINT16) (offsetof(Quote_Out, signature))},
1596     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1597                 TPM2B_DATA_P_UNMARSHAL,
1598                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1599                 TPML_PCR_SELECTION_P_UNMARSHAL,
1600                 END_OF_LIST,
1601                 TPM2B_ATTEST_P_MARSHAL,
1602                 TPMT_SIGNATURE_P_MARSHAL,
1603                 END_OF_LIST};
1604 };
1605 #define _QuoteDataAddress (&QuoteData)
1606 #else
1607 #define _QuoteDataAddress 0
1608 #endif // CC_Quote
1609 #if CC_GetSessionAuditDigest
1610 #include "GetSessionAuditDigest_fp.h"
1611 typedef TPM_RC (GetSessionAuditDigest_Entry) (
1612     GetSessionAuditDigest_In *in,
1613     GetSessionAuditDigest_Out *out

```



```

1614 );
1615 typedef const struct {
1616     GetSessionAuditDigest_Entry    *entry;
1617     UINT16                          inSize;
1618     UINT16                          outSize;
1619     UINT16                          offsetOfTypes;
1620     UINT16                          paramOffsets[5];
1621     BYTE                             types[9];
1622 } GetSessionAuditDigest_COMMAND_DESCRIPTOR_t;
1623 GetSessionAuditDigest_COMMAND_DESCRIPTOR_t _GetSessionAuditDigestData = {
1624     /* entry */                                &TPM2_GetSessionAuditDigest,
1625     /* inSize */                              (UINT16) (sizeof(GetSessionAuditDigest_In)),
1626     /* outSize */                             (UINT16) (sizeof(GetSessionAuditDigest_Out)),
1627     /* offsetOfTypes */
offsetof(GetSessionAuditDigest_COMMAND_DESCRIPTOR_t, types),
1628     /* offsets */                             { (UINT16) (offsetof(GetSessionAuditDigest_In,
signHandle)),
1629                                             (UINT16) (offsetof(GetSessionAuditDigest_In,
1630 sessionHandle)),
1631                                             (UINT16) (offsetof(GetSessionAuditDigest_In,
1632 qualifyingData)),
1633                                             (UINT16) (offsetof(GetSessionAuditDigest_In,
1634 inScheme)),
1635                                             (UINT16) (offsetof(GetSessionAuditDigest_Out,
1636 signature))},
1637     /* types */                               {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1638                                             TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1639                                             TPMI_SH_HMAC_H_UNMARSHAL,
1640                                             TPM2B_DATA_P_UNMARSHAL,
1641                                             TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1642                                             END_OF_LIST,
1643                                             TPM2B_ATTEST_P_MARSHAL,
1644                                             TPMT_SIGNATURE_P_MARSHAL,
1645                                             END_OF_LIST};
1646 };
1647 #define _GetSessionAuditDigestDataAddress (&_GetSessionAuditDigestData)
1648 #else
1649 #define _GetSessionAuditDigestDataAddress 0
1650 #endif // CC_GetSessionAuditDigest
1651 #if CC_GetCommandAuditDigest
1652 #include "GetCommandAuditDigest_fp.h"
1653 typedef TPM_RC (GetCommandAuditDigest_Entry) (
1654     GetCommandAuditDigest_In        *in,
1655     GetCommandAuditDigest_Out       *out
1656 );
1657 typedef const struct {
1658     GetCommandAuditDigest_Entry    *entry;
1659     UINT16                          inSize;
1660     UINT16                          outSize;
1661     UINT16                          offsetOfTypes;
1662     UINT16                          paramOffsets[4];
1663     BYTE                             types[8];
1664 } GetCommandAuditDigest_COMMAND_DESCRIPTOR_t;
1665 GetCommandAuditDigest_COMMAND_DESCRIPTOR_t _GetCommandAuditDigestData = {
1666     /* entry */                                &TPM2_GetCommandAuditDigest,
1667     /* inSize */                              (UINT16) (sizeof(GetCommandAuditDigest_In)),
1668     /* outSize */                             (UINT16) (sizeof(GetCommandAuditDigest_Out)),
1669     /* offsetOfTypes */
offsetof(GetCommandAuditDigest_COMMAND_DESCRIPTOR_t, types),
1670     /* offsets */                             { (UINT16) (offsetof(GetCommandAuditDigest_In,
signHandle)),
1671                                             (UINT16) (offsetof(GetCommandAuditDigest_In,
1672 qualifyingData)),
1673                                             (UINT16) (offsetof(GetCommandAuditDigest_In,
1674 inScheme)),

```

```

1669                                     (UINT16) (offsetof(GetCommandAuditDigest_Out,
signature))),
1670     /* types */
1671                                     {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1672                                     TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1673                                     TPM2B_DATA_P_UNMARSHAL,
1674                                     TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1675                                     END_OF_LIST,
1676                                     TPM2B_ATTEST_P_MARSHAL,
1677                                     TPMT_SIGNATURE_P_MARSHAL,
1678                                     END_OF_LIST}
1679 };
1679 #define _GetCommandAuditDigestDataAddress (&_GetCommandAuditDigestData)
1680 #else
1681 #define _GetCommandAuditDigestDataAddress 0
1682 #endif // CC_GetCommandAuditDigest
1683 #if CC_GetTime
1684 #include "GetTime_fp.h"
1685 typedef TPM_RC (GetTime_Entry) (
1686     GetTime_In                *in,
1687     GetTime_Out               *out
1688 );
1689 typedef const struct {
1690     GetTime_Entry              *entry;
1691     UINT16                     inSize;
1692     UINT16                     outSize;
1693     UINT16                     offsetOfTypes;
1694     UINT16                     paramOffsets[4];
1695     BYTE                       types[8];
1696 } GetTime_COMMAND_DESCRIPTOR_t;
1697 GetTime_COMMAND_DESCRIPTOR_t _GetTimeData = {
1698     /* entry */ &TPM2_GetTime,
1699     /* inSize */ (UINT16) (sizeof(GetTime_In)),
1700     /* outSize */ (UINT16) (sizeof(GetTime_Out)),
1701     /* offsetOfTypes */ offsetof(GetTime_COMMAND_DESCRIPTOR_t, types),
1702     /* offsets */ { (UINT16) (offsetof(GetTime_In, signHandle)),
1703                   (UINT16) (offsetof(GetTime_In, qualifyingData)),
1704                   (UINT16) (offsetof(GetTime_In, inScheme)),
1705                   (UINT16) (offsetof(GetTime_Out, signature)) },
1706     /* types */ {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
1707                 TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1708                 TPM2B_DATA_P_UNMARSHAL,
1709                 TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1710                 END_OF_LIST,
1711                 TPM2B_ATTEST_P_MARSHAL,
1712                 TPMT_SIGNATURE_P_MARSHAL,
1713                 END_OF_LIST}
1714 };
1715 #define _GetTimeDataAddress (&_GetTimeData)
1716 #else
1717 #define _GetTimeDataAddress 0
1718 #endif // CC_GetTime
1719 #if CC_CertifyX509
1720 #include "CertifyX509_fp.h"
1721 typedef TPM_RC (CertifyX509_Entry) (
1722     CertifyX509_In            *in,
1723     CertifyX509_Out           *out
1724 );
1725 typedef const struct {
1726     CertifyX509_Entry          *entry;
1727     UINT16                     inSize;
1728     UINT16                     outSize;
1729     UINT16                     offsetOfTypes;
1730     UINT16                     paramOffsets[6];
1731     BYTE                       types[10];
1732 } CertifyX509_COMMAND_DESCRIPTOR_t;
1733 CertifyX509_COMMAND_DESCRIPTOR_t _CertifyX509Data = {

```

```

1734     /* entry */ &TPM2_CertifyX509,
1735     /* inSize */ (UINT16) (sizeof(CertifyX509_In)),
1736     /* outSize */ (UINT16) (sizeof(CertifyX509_Out)),
1737     /* offsetOfTypes */ offsetof(CertifyX509_COMMAND_DESCRIPTOR_t, types),
1738     /* offsets */ { (UINT16) (offsetof(CertifyX509_In, signHandle)),
1739                    (UINT16) (offsetof(CertifyX509_In, qualifyingData)),
1740                    (UINT16) (offsetof(CertifyX509_In, inScheme)),
1741                    (UINT16) (offsetof(CertifyX509_In, partialCertificate)),
1742                    (UINT16) (offsetof(CertifyX509_Out, tbsDigest)),
1743                    (UINT16) (offsetof(CertifyX509_Out, signature)) },
1744     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
1745                  TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
1746                  TPM2B_DATA_P_UNMARSHAL,
1747                  TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1748                  TPM2B_MAX_BUFFER_P_UNMARSHAL,
1749                  END_OF_LIST,
1750                  TPM2B_MAX_BUFFER_P_MARSHAL,
1751                  TPM2B_DIGEST_P_MARSHAL,
1752                  TPMT_SIGNATURE_P_MARSHAL,
1753                  END_OF_LIST }
1754 };
1755 #define _CertifyX509DataAddress (&CertifyX509Data)
1756 #else
1757 #define _CertifyX509DataAddress 0
1758 #endif // CC_CertifyX509
1759 #if CC_Commit
1760 #include "Commit_fp.h"
1761 typedef TPM_RC (Commit_Entry) (
1762     Commit_In *in,
1763     Commit_Out *out
1764 );
1765 typedef const struct {
1766     Commit_Entry *entry;
1767     UINT16 inSize;
1768     UINT16 outSize;
1769     UINT16 offsetOfTypes;
1770     UINT16 paramOffsets[6];
1771     BYTE types[10];
1772 } Commit_COMMAND_DESCRIPTOR_t;
1773 Commit_COMMAND_DESCRIPTOR_t _CommitData = {
1774     /* entry */ &TPM2_Commit,
1775     /* inSize */ (UINT16) (sizeof(Commit_In)),
1776     /* outSize */ (UINT16) (sizeof(Commit_Out)),
1777     /* offsetOfTypes */ offsetof(Commit_COMMAND_DESCRIPTOR_t, types),
1778     /* offsets */ { (UINT16) (offsetof(Commit_In, P1)),
1779                    (UINT16) (offsetof(Commit_In, s2)),
1780                    (UINT16) (offsetof(Commit_In, y2)),
1781                    (UINT16) (offsetof(Commit_Out, L)),
1782                    (UINT16) (offsetof(Commit_Out, E)),
1783                    (UINT16) (offsetof(Commit_Out, counter)) },
1784     /* types */ { TPMI_DH_OBJECT_H_UNMARSHAL,
1785                  TPM2B_ECC_POINT_P_UNMARSHAL,
1786                  TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1787                  TPM2B_ECC_PARAMETER_P_UNMARSHAL,
1788                  END_OF_LIST,
1789                  TPM2B_ECC_POINT_P_MARSHAL,
1790                  TPM2B_ECC_POINT_P_MARSHAL,
1791                  TPM2B_ECC_POINT_P_MARSHAL,
1792                  UINT16_P_MARSHAL,
1793                  END_OF_LIST }
1794 };
1795 #define _CommitDataAddress (&CommitData)
1796 #else
1797 #define _CommitDataAddress 0
1798 #endif // CC_Commit
1799 #if CC_EC_Ephemeral

```

```

1800 #include "EC_Ephemeral_fp.h"
1801 typedef TPM_RC (EC_Ephemeral_Entry) (
1802     EC_Ephemeral_In      *in,
1803     EC_Ephemeral_Out     *out
1804 );
1805 typedef const struct {
1806     EC_Ephemeral_Entry    *entry;
1807     UINT16                inSize;
1808     UINT16                outSize;
1809     UINT16                offsetOfTypes;
1810     UINT16                paramOffsets[1];
1811     BYTE                  types[5];
1812 } EC_Ephemeral_COMMAND_DESCRIPTOR_t;
1813 EC_Ephemeral_COMMAND_DESCRIPTOR_t EC_EphemeralData = {
1814     /* entry */           &TPM2_EC_Ephemeral,
1815     /* inSize */          (UINT16) (sizeof(EC_Ephemeral_In)),
1816     /* outSize */         (UINT16) (sizeof(EC_Ephemeral_Out)),
1817     /* offsetOfTypes */   offsetof(EC_Ephemeral_COMMAND_DESCRIPTOR_t, types),
1818     /* offsets */         { (UINT16) (offsetof(EC_Ephemeral_Out, counter)) },
1819     /* types */           { TPMI_ECC_CURVE_P_UNMARSHAL,
1820                           END_OF_LIST,
1821                           TPM2B_ECC_POINT_P_MARSHAL,
1822                           UINT16_P_MARSHAL,
1823                           END_OF_LIST }
1824 };
1825 #define _EC_EphemeralDataAddress (&EC_EphemeralData)
1826 #else
1827 #define EC_EphemeralDataAddress 0
1828 #endif // CC_EC_Ephemeral
1829 #if CC_VerifySignature
1830 #include "VerifySignature_fp.h"
1831 typedef TPM_RC (VerifySignature_Entry) (
1832     VerifySignature_In     *in,
1833     VerifySignature_Out    *out
1834 );
1835 typedef const struct {
1836     VerifySignature_Entry  *entry;
1837     UINT16                 inSize;
1838     UINT16                 outSize;
1839     UINT16                 offsetOfTypes;
1840     UINT16                 paramOffsets[2];
1841     BYTE                   types[6];
1842 } VerifySignature_COMMAND_DESCRIPTOR_t;
1843 VerifySignature_COMMAND_DESCRIPTOR_t VerifySignatureData = {
1844     /* entry */           &TPM2_VerifySignature,
1845     /* inSize */          (UINT16) (sizeof(VerifySignature_In)),
1846     /* outSize */         (UINT16) (sizeof(VerifySignature_Out)),
1847     /* offsetOfTypes */   offsetof(VerifySignature_COMMAND_DESCRIPTOR_t, types),
1848     /* offsets */         { (UINT16) (offsetof(VerifySignature_In, digest)),
1849                           (UINT16) (offsetof(VerifySignature_In, signature)) },
1850     /* types */           { TPMI_DH_OBJECT_H_UNMARSHAL,
1851                           TPM2B_DIGEST_P_UNMARSHAL,
1852                           TPMT_SIGNATURE_P_UNMARSHAL,
1853                           END_OF_LIST,
1854                           TPMT_TK_VERIFIED_P_MARSHAL,
1855                           END_OF_LIST }
1856 };
1857 #define _VerifySignatureDataAddress (&VerifySignatureData)
1858 #else
1859 #define VerifySignatureDataAddress 0
1860 #endif // CC_VerifySignature
1861 #if CC_Sign
1862 #include "Sign_fp.h"
1863 typedef TPM_RC (Sign_Entry) (
1864     Sign_In                *in,
1865     Sign_Out               *out

```

```

1866 );
1867 typedef const struct {
1868     Sign_Entry          *entry;
1869     UINT16              inSize;
1870     UINT16              outSize;
1871     UINT16              offsetOfTypes;
1872     UINT16              paramOffsets[3];
1873     BYTE               types[7];
1874 } Sign_COMMAND_DESCRIPTOR_t;
1875 Sign_COMMAND_DESCRIPTOR_t _SignData = {
1876     /* entry          */ &TPM2_Sign,
1877     /* inSize         */ (UINT16)(sizeof(Sign_In)),
1878     /* outSize        */ (UINT16)(sizeof(Sign_Out)),
1879     /* offsetOfTypes  */ offsetof(Sign_COMMAND_DESCRIPTOR_t, types),
1880     /* offsets        */ {(UINT16)(offsetof(Sign_In, digest)),
1881                          (UINT16)(offsetof(Sign_In, inScheme)),
1882                          (UINT16)(offsetof(Sign_In, validation))},
1883     /* types          */ {TPMI_DH_OBJECT_H_UNMARSHAL,
1884                          TPM2B_DIGEST_P_UNMARSHAL,
1885                          TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1886                          TPMT_TK_HASHCHECK_P_UNMARSHAL,
1887                          END_OF_LIST,
1888                          TPMT_SIGNATURE_P_MARSHAL,
1889                          END_OF_LIST}
1890 };
1891 #define _SignDataAddress (&_SignData)
1892 #else
1893 #define _SignDataAddress 0
1894 #endif // CC_Sign
1895 #if CC_SetCommandCodeAuditStatus
1896 #include "SetCommandCodeAuditStatus_fp.h"
1897 typedef TPM_RC (SetCommandCodeAuditStatus_Entry)(
1898     SetCommandCodeAuditStatus_In *in
1899 );
1900 typedef const struct {
1901     SetCommandCodeAuditStatus_Entry *entry;
1902     UINT16                          inSize;
1903     UINT16                          outSize;
1904     UINT16                          offsetOfTypes;
1905     UINT16                          paramOffsets[3];
1906     BYTE                           types[6];
1907 } SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t;
1908 SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t _SetCommandCodeAuditStatusData = {
1909     /* entry          */ &TPM2_SetCommandCodeAuditStatus,
1910     /* inSize         */ (UINT16)(sizeof(SetCommandCodeAuditStatus_In)),
1911     /* outSize        */ 0,
1912     /* offsetOfTypes  */ offsetof(SetCommandCodeAuditStatus_COMMAND_DESCRIPTOR_t, types),
1913     /* offsets        */ {(UINT16)(offsetof(SetCommandCodeAuditStatus_In, auditAlg)),
1914                          (UINT16)(offsetof(SetCommandCodeAuditStatus_In, setList)),
1915                          (UINT16)(offsetof(SetCommandCodeAuditStatus_In, clearList))},
1916     /* types          */ {TPMI_RH_PROVISION_H_UNMARSHAL,
1917                          TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
1918                          TPML_CC_P_UNMARSHAL,
1919                          TPML_CC_P_UNMARSHAL,
1920                          END_OF_LIST,
1921                          END_OF_LIST}
1922 };
1923 #define _SetCommandCodeAuditStatusDataAddress (&_SetCommandCodeAuditStatusData)
1924 #else
1925 #define _SetCommandCodeAuditStatusDataAddress 0
1926 #endif // CC_SetCommandCodeAuditStatus

```

```

1927 #if CC_PCR_Extend
1928 #include "PCR_Extend_fp.h"
1929 typedef TPM_RC (PCR_Extend_Entry) (
1930     PCR_Extend_In      *in
1931 );
1932 typedef const struct {
1933     PCR_Extend_Entry    *entry;
1934     UINT16               inSize;
1935     UINT16               outSize;
1936     UINT16               offsetOfTypes;
1937     UINT16               paramOffsets[1];
1938     BYTE                 types[4];
1939 } PCR_Extend_COMMAND_DESCRIPTOR_t;
1940 PCR_Extend_COMMAND_DESCRIPTOR_t _PCR_ExtendData = {
1941     /* entry */           &TPM2_PCR_Extend,
1942     /* inSize */          (UINT16) (sizeof(PCR_Extend_In)),
1943     /* outSize */         0,
1944     /* offsetOfTypes */   offsetof(PCR_Extend_COMMAND_DESCRIPTOR_t, types),
1945     /* offsets */          {(UINT16) (offsetof(PCR_Extend_In, digests))},
1946     /* types */           {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1947                           TPML_DIGEST_VALUES_P_UNMARSHAL,
1948                           END_OF_LIST,
1949                           END_OF_LIST};
1950 };
1951 #define _PCR_ExtendDataAddress (&_PCR_ExtendData)
1952 #else
1953 #define _PCR_ExtendDataAddress 0
1954 #endif // CC_PCR_Extend
1955 #if CC_PCR_Event
1956 #include "PCR_Event_fp.h"
1957 typedef TPM_RC (PCR_Event_Entry) (
1958     PCR_Event_In      *in,
1959     PCR_Event_Out     *out
1960 );
1961 typedef const struct {
1962     PCR_Event_Entry    *entry;
1963     UINT16               inSize;
1964     UINT16               outSize;
1965     UINT16               offsetOfTypes;
1966     UINT16               paramOffsets[1];
1967     BYTE                 types[5];
1968 } PCR_Event_COMMAND_DESCRIPTOR_t;
1969 PCR_Event_COMMAND_DESCRIPTOR_t _PCR_EventData = {
1970     /* entry */           &TPM2_PCR_Event,
1971     /* inSize */          (UINT16) (sizeof(PCR_Event_In)),
1972     /* outSize */         (UINT16) (sizeof(PCR_Event_Out)),
1973     /* offsetOfTypes */   offsetof(PCR_Event_COMMAND_DESCRIPTOR_t, types),
1974     /* offsets */          {(UINT16) (offsetof(PCR_Event_In, eventData))},
1975     /* types */           {TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
1976                           TPM2B_EVENT_P_UNMARSHAL,
1977                           END_OF_LIST,
1978                           TPML_DIGEST_VALUES_P_MARSHAL,
1979                           END_OF_LIST};
1980 };
1981 #define _PCR_EventDataAddress (&_PCR_EventData)
1982 #else
1983 #define _PCR_EventDataAddress 0
1984 #endif // CC_PCR_Event
1985 #if CC_PCR_Read
1986 #include "PCR_Read_fp.h"
1987 typedef TPM_RC (PCR_Read_Entry) (
1988     PCR_Read_In      *in,
1989     PCR_Read_Out     *out
1990 );
1991 typedef const struct {
1992     PCR_Read_Entry    *entry;

```



```

1993     UINT16             inSize;
1994     UINT16             outSize;
1995     UINT16             offsetOfTypes;
1996     UINT16             paramOffsets[2];
1997     BYTE               types[6];
1998 } PCR_Read_COMMAND_DESCRIPTOR_t;
1999 PCR_Read_COMMAND_DESCRIPTOR_t _PCR_ReadData = {
2000     /* entry             */ &TPM2_PCR_Read,
2001     /* inSize            */ (UINT16) (sizeof(PCR_Read_In)),
2002     /* outSize           */ (UINT16) (sizeof(PCR_Read_Out)),
2003     /* offsetOfTypes     */ offsetof(PCR_Read_COMMAND_DESCRIPTOR_t, types),
2004     /* offsets           */ { (UINT16) (offsetof(PCR_Read_Out, pcrSelectionOut)),
2005                             (UINT16) (offsetof(PCR_Read_Out, pcrValues)) },
2006     /* types             */ {TPML_PCR_SELECTION_P_UNMARSHAL,
2007                             END_OF_LIST,
2008                             UINT32_P_MARSHAL,
2009                             TPML_PCR_SELECTION_P_MARSHAL,
2010                             TPML_DIGEST_P_MARSHAL,
2011                             END_OF_LIST};
2012 };
2013 #define _PCR_ReadDataAddress (&_PCR_ReadData)
2014 #else
2015 #define _PCR_ReadDataAddress 0
2016 #endif // CC_PCR_Read
2017 #if CC_PCR_Allocate
2018 #include "PCR_Allocate_fp.h"
2019 typedef TPM_RC (PCR_Allocate_Entry) (
2020     PCR_Allocate_In      *in,
2021     PCR_Allocate_Out     *out
2022 );
2023 typedef const struct {
2024     PCR_Allocate_Entry    *entry;
2025     UINT16                inSize;
2026     UINT16                outSize;
2027     UINT16                offsetOfTypes;
2028     UINT16                paramOffsets[4];
2029     BYTE                  types[8];
2030 } PCR_Allocate_COMMAND_DESCRIPTOR_t;
2031 PCR_Allocate_COMMAND_DESCRIPTOR_t _PCR_AllocateData = {
2032     /* entry             */ &TPM2_PCR_Allocate,
2033     /* inSize            */ (UINT16) (sizeof(PCR_Allocate_In)),
2034     /* outSize           */ (UINT16) (sizeof(PCR_Allocate_Out)),
2035     /* offsetOfTypes     */ offsetof(PCR_Allocate_COMMAND_DESCRIPTOR_t, types),
2036     /* offsets           */ { (UINT16) (offsetof(PCR_Allocate_In, pcrAllocation)),
2037                             (UINT16) (offsetof(PCR_Allocate_Out, maxPCR)),
2038                             (UINT16) (offsetof(PCR_Allocate_Out, sizeNeeded)),
2039                             (UINT16) (offsetof(PCR_Allocate_Out, sizeAvailable)) },
2040     /* types             */ {TPMI_RH_PLATFORM_H_UNMARSHAL,
2041                             TPML_PCR_SELECTION_P_UNMARSHAL,
2042                             END_OF_LIST,
2043                             TPMI_YES_NO_P_MARSHAL,
2044                             UINT32_P_MARSHAL,
2045                             UINT32_P_MARSHAL,
2046                             UINT32_P_MARSHAL,
2047                             END_OF_LIST};
2048 };
2049 #define _PCR_AllocateDataAddress (&_PCR_AllocateData)
2050 #else
2051 #define _PCR_AllocateDataAddress 0
2052 #endif // CC_PCR_Allocate
2053 #if CC_PCR_SetAuthPolicy
2054 #include "PCR_SetAuthPolicy_fp.h"
2055 typedef TPM_RC (PCR_SetAuthPolicy_Entry) (
2056     PCR_SetAuthPolicy_In *in
2057 );
2058 typedef const struct {

```

```

2059     PCR_SetAuthPolicy_Entry    *entry;
2060     UINT16                     inSize;
2061     UINT16                     outSize;
2062     UINT16                     offsetOfTypes;
2063     UINT16                     paramOffsets[3];
2064     BYTE                       types[6];
2065 } PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t;
2066 PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t PCR_SetAuthPolicyData = {
2067     /* entry */                &TPM2_PCR_SetAuthPolicy,
2068     /* inSize */               (UINT16) (sizeof(PCR_SetAuthPolicy_In)),
2069     /* outSize */              0,
2070     /* offsetOfTypes */        offsetof(PCR_SetAuthPolicy_COMMAND_DESCRIPTOR_t,
types),
2071     /* offsets */              { (UINT16) (offsetof(PCR_SetAuthPolicy_In, authPolicy)),
2072                                (UINT16) (offsetof(PCR_SetAuthPolicy_In, hashAlg)),
2073                                (UINT16) (offsetof(PCR_SetAuthPolicy_In, pcrNum)) },
2074     /* types */                { TPMI_RH_PLATFORM_H_UNMARSHAL,
2075                                TPM2B_DIGEST_P_UNMARSHAL,
2076                                TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2077                                TPMI_DH_PCR_P_UNMARSHAL,
2078                                END_OF_LIST,
2079                                END_OF_LIST }
2080 };
2081 #define PCR_SetAuthPolicyDataAddress (&PCR_SetAuthPolicyData)
2082 #else
2083 #define PCR_SetAuthPolicyDataAddress 0
2084 #endif // CC_PCR_SetAuthPolicy
2085 #if CC_PCR_SetAuthValue
2086 #include "PCR_SetAuthValue_fp.h"
2087 typedef TPM_RC (PCR_SetAuthValue_Entry) (
2088     PCR_SetAuthValue_In        *in
2089 );
2090 typedef const struct {
2091     PCR_SetAuthValue_Entry    *entry;
2092     UINT16                     inSize;
2093     UINT16                     outSize;
2094     UINT16                     offsetOfTypes;
2095     UINT16                     paramOffsets[1];
2096     BYTE                       types[4];
2097 } PCR_SetAuthValue_COMMAND_DESCRIPTOR_t;
2098 PCR_SetAuthValue_COMMAND_DESCRIPTOR_t PCR_SetAuthValueData = {
2099     /* entry */                &TPM2_PCR_SetAuthValue,
2100     /* inSize */               (UINT16) (sizeof(PCR_SetAuthValue_In)),
2101     /* outSize */              0,
2102     /* offsetOfTypes */        offsetof(PCR_SetAuthValue_COMMAND_DESCRIPTOR_t, types),
2103     /* offsets */              { (UINT16) (offsetof(PCR_SetAuthValue_In, auth)) },
2104     /* types */                { TPMI_DH_PCR_H_UNMARSHAL,
2105                                TPM2B_DIGEST_P_UNMARSHAL,
2106                                END_OF_LIST,
2107                                END_OF_LIST }
2108 };
2109 #define PCR_SetAuthValueDataAddress (&PCR_SetAuthValueData)
2110 #else
2111 #define PCR_SetAuthValueDataAddress 0
2112 #endif // CC_PCR_SetAuthValue
2113 #if CC_PCR_Reset
2114 #include "PCR_Reset_fp.h"
2115 typedef TPM_RC (PCR_Reset_Entry) (
2116     PCR_Reset_In              *in
2117 );
2118 typedef const struct {
2119     PCR_Reset_Entry           *entry;
2120     UINT16                     inSize;
2121     UINT16                     outSize;
2122     UINT16                     offsetOfTypes;
2123     BYTE                       types[3];

```



```

2124 } PCR_Reset_COMMAND_DESCRIPTOR_t;
2125 PCR_Reset_COMMAND_DESCRIPTOR_t _PCR_ResetData = {
2126     /* entry */ &TPM2_PCR_Reset,
2127     /* inSize */ (UINT16) (sizeof(PCR_Reset_In)),
2128     /* outSize */ 0,
2129     /* offsetOfTypes */ offsetof(PCR_Reset_COMMAND_DESCRIPTOR_t, types),
2130     /* offsets */ // No parameter offsets;
2131     /* types */ {TPMI_DH_PCR_H_UNMARSHAL,
2132                 END_OF_LIST,
2133                 END_OF_LIST}
2134 };
2135 #define _PCR_ResetDataAddress (&_PCR_ResetData)
2136 #else
2137 #define _PCR_ResetDataAddress 0
2138 #endif // CC_PCR_Reset
2139 #if CC_PolicySigned
2140 #include "PolicySigned_fp.h"
2141 typedef TPM_RC (PolicySigned_Entry) (
2142     PolicySigned_In *in,
2143     PolicySigned_Out *out
2144 );
2145 typedef const struct {
2146     PolicySigned_Entry *entry;
2147     UINT16 inSize;
2148     UINT16 outSize;
2149     UINT16 offsetOfTypes;
2150     UINT16 paramOffsets[7];
2151     BYTE types[11];
2152 } PolicySigned_COMMAND_DESCRIPTOR_t;
2153 PolicySigned_COMMAND_DESCRIPTOR_t _PolicySignedData = {
2154     /* entry */ &TPM2_PolicySigned,
2155     /* inSize */ (UINT16) (sizeof(PolicySigned_In)),
2156     /* outSize */ (UINT16) (sizeof(PolicySigned_Out)),
2157     /* offsetOfTypes */ offsetof(PolicySigned_COMMAND_DESCRIPTOR_t, types),
2158     /* offsets */ { (UINT16) (offsetof(PolicySigned_In, policySession)),
2159                   (UINT16) (offsetof(PolicySigned_In, nonceTPM)),
2160                   (UINT16) (offsetof(PolicySigned_In, cpHashA)),
2161                   (UINT16) (offsetof(PolicySigned_In, policyRef)),
2162                   (UINT16) (offsetof(PolicySigned_In, expiration)),
2163                   (UINT16) (offsetof(PolicySigned_In, auth)),
2164                   (UINT16) (offsetof(PolicySigned_Out, policyTicket))},
2165     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
2166                 TPMI_SH_POLICY_H_UNMARSHAL,
2167                 TPM2B_NONCE_P_UNMARSHAL,
2168                 TPM2B_DIGEST_P_UNMARSHAL,
2169                 TPM2B_NONCE_P_UNMARSHAL,
2170                 INT32_P_UNMARSHAL,
2171                 TPMT_SIGNATURE_P_UNMARSHAL,
2172                 END_OF_LIST,
2173                 TPM2B_TIMEOUT_P_MARSHAL,
2174                 TPMT_TK_AUTH_P_MARSHAL,
2175                 END_OF_LIST}
2176 };
2177 #define _PolicySignedDataAddress (&_PolicySignedData)
2178 #else
2179 #define _PolicySignedDataAddress 0
2180 #endif // CC_PolicySigned
2181 #if CC_PolicySecret
2182 #include "PolicySecret_fp.h"
2183 typedef TPM_RC (PolicySecret_Entry) (
2184     PolicySecret_In *in,
2185     PolicySecret_Out *out
2186 );
2187 typedef const struct {
2188     PolicySecret_Entry *entry;
2189     UINT16 inSize;

```

```

2190     UINT16                outSize;
2191     UINT16                offsetOfTypes;
2192     UINT16                paramOffsets[6];
2193     BYTE                  types[10];
2194 } PolicySecret_COMMAND_DESCRIPTOR_t;
2195 PolicySecret_COMMAND_DESCRIPTOR_t _PolicySecretData = {
2196     /* entry */           &TPM2_PolicySecret,
2197     /* inSize */          (UINT16) (sizeof(PolicySecret_In)),
2198     /* outSize */         (UINT16) (sizeof(PolicySecret_Out)),
2199     /* offsetOfTypes */   offsetof(PolicySecret_COMMAND_DESCRIPTOR_t, types),
2200     /* offsets */         { (UINT16) (offsetof(PolicySecret_In, policySession)),
2201                           (UINT16) (offsetof(PolicySecret_In, nonceTPM)),
2202                           (UINT16) (offsetof(PolicySecret_In, cpHashA)),
2203                           (UINT16) (offsetof(PolicySecret_In, policyRef)),
2204                           (UINT16) (offsetof(PolicySecret_In, expiration)),
2205                           (UINT16) (offsetof(PolicySecret_Out, policyTicket))},
2206     /* types */           {TPMI_DH_ENTITY_H_UNMARSHAL,
2207                           TPMI_SH_POLICY_H_UNMARSHAL,
2208                           TPM2B_NONCE_P_UNMARSHAL,
2209                           TPM2B_DIGEST_P_UNMARSHAL,
2210                           TPM2B_NONCE_P_UNMARSHAL,
2211                           INT32_P_UNMARSHAL,
2212                           END_OF_LIST,
2213                           TPM2B_TIMEOUT_P_MARSHAL,
2214                           TPMT_TK_AUTH_P_MARSHAL,
2215                           END_OF_LIST};
2216 };
2217 #define _PolicySecretDataAddress (&_PolicySecretData)
2218 #else
2219 #define _PolicySecretDataAddress 0
2220 #endif // CC_PolicySecret
2221 #if CC_PolicyTicket
2222 #include "PolicyTicket_fp.h"
2223 typedef TPM_RC (PolicyTicket_Entry) (
2224     PolicyTicket_In *in
2225 );
2226 typedef const struct {
2227     PolicyTicket_Entry *entry;
2228     UINT16              inSize;
2229     UINT16              outSize;
2230     UINT16              offsetOfTypes;
2231     UINT16              paramOffsets[5];
2232     BYTE                types[8];
2233 } PolicyTicket_COMMAND_DESCRIPTOR_t;
2234 PolicyTicket_COMMAND_DESCRIPTOR_t _PolicyTicketData = {
2235     /* entry */           &TPM2_PolicyTicket,
2236     /* inSize */          (UINT16) (sizeof(PolicyTicket_In)),
2237     /* outSize */         0,
2238     /* offsetOfTypes */   offsetof(PolicyTicket_COMMAND_DESCRIPTOR_t, types),
2239     /* offsets */         { (UINT16) (offsetof(PolicyTicket_In, timeout)),
2240                           (UINT16) (offsetof(PolicyTicket_In, cpHashA)),
2241                           (UINT16) (offsetof(PolicyTicket_In, policyRef)),
2242                           (UINT16) (offsetof(PolicyTicket_In, authName)),
2243                           (UINT16) (offsetof(PolicyTicket_In, ticket))},
2244     /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2245                           TPM2B_TIMEOUT_P_UNMARSHAL,
2246                           TPM2B_DIGEST_P_UNMARSHAL,
2247                           TPM2B_NONCE_P_UNMARSHAL,
2248                           TPM2B_NAME_P_UNMARSHAL,
2249                           TPMT_TK_AUTH_P_UNMARSHAL,
2250                           END_OF_LIST,
2251                           END_OF_LIST};
2252 };
2253 #define _PolicyTicketDataAddress (&_PolicyTicketData)
2254 #else
2255 #define _PolicyTicketDataAddress 0

```

```

2256 #endif // CC_PolicyTicket
2257 #if CC_PolicyOR
2258 #include "PolicyOR_fp.h"
2259 typedef TPM_RC (PolicyOR_Entry) (
2260     PolicyOR_In          *in
2261 );
2262 typedef const struct {
2263     PolicyOR_Entry        *entry;
2264     UINT16                inSize;
2265     UINT16                outSize;
2266     UINT16                offsetOfTypes;
2267     UINT16                paramOffsets[1];
2268     BYTE                  types[4];
2269 } PolicyOR_COMMAND_DESCRIPTOR_t;
2270 PolicyOR_COMMAND_DESCRIPTOR_t _PolicyORData = {
2271     /* entry */          &TPM2_PolicyOR,
2272     /* inSize */         (UINT16) (sizeof(PolicyOR_In)),
2273     /* outSize */        0,
2274     /* offsetOfTypes */  offsetof(PolicyOR_COMMAND_DESCRIPTOR_t, types),
2275     /* offsets */        {(UINT16) (offsetof(PolicyOR_In, pHashList))},
2276     /* types */          {TPMI_SH_POLICY_H_UNMARSHAL,
2277                          TPML_DIGEST_P_UNMARSHAL,
2278                          END_OF_LIST,
2279                          END_OF_LIST}
2280 };
2281 #define _PolicyORDataAddress (&_PolicyORData)
2282 #else
2283 #define _PolicyORDataAddress 0
2284 #endif // CC_PolicyOR
2285 #if CC_PolicyPCR
2286 #include "PolicyPCR_fp.h"
2287 typedef TPM_RC (PolicyPCR_Entry) (
2288     PolicyPCR_In          *in
2289 );
2290 typedef const struct {
2291     PolicyPCR_Entry        *entry;
2292     UINT16                inSize;
2293     UINT16                outSize;
2294     UINT16                offsetOfTypes;
2295     UINT16                paramOffsets[2];
2296     BYTE                  types[5];
2297 } PolicyPCR_COMMAND_DESCRIPTOR_t;
2298 PolicyPCR_COMMAND_DESCRIPTOR_t _PolicyPCRData = {
2299     /* entry */          &TPM2_PolicyPCR,
2300     /* inSize */         (UINT16) (sizeof(PolicyPCR_In)),
2301     /* outSize */        0,
2302     /* offsetOfTypes */  offsetof(PolicyPCR_COMMAND_DESCRIPTOR_t, types),
2303     /* offsets */        {(UINT16) (offsetof(PolicyPCR_In, pcrDigest)),
2304                          (UINT16) (offsetof(PolicyPCR_In, pcrs))},
2305     /* types */          {TPMI_SH_POLICY_H_UNMARSHAL,
2306                          TPM2B_DIGEST_P_UNMARSHAL,
2307                          TPML_PCR_SELECTION_P_UNMARSHAL,
2308                          END_OF_LIST,
2309                          END_OF_LIST}
2310 };
2311 #define _PolicyPCRDataAddress (&_PolicyPCRData)
2312 #else
2313 #define _PolicyPCRDataAddress 0
2314 #endif // CC_PolicyPCR
2315 #if CC_PolicyLocality
2316 #include "PolicyLocality_fp.h"
2317 typedef TPM_RC (PolicyLocality_Entry) (
2318     PolicyLocality_In      *in
2319 );
2320 typedef const struct {
2321     PolicyLocality_Entry    *entry;

```

```

2322     UINT16             inSize;
2323     UINT16             outSize;
2324     UINT16             offsetOfTypes;
2325     UINT16             paramOffsets[1];
2326     BYTE               types[4];
2327 } PolicyLocality_COMMAND_DESCRIPTOR_t;
2328 PolicyLocality_COMMAND_DESCRIPTOR_t _PolicyLocalityData = {
2329     /* entry */          &TPM2_PolicyLocality,
2330     /* inSize */         (UINT16) (sizeof(PolicyLocality_In)),
2331     /* outSize */        0,
2332     /* offsetOfTypes */  offsetof(PolicyLocality_COMMAND_DESCRIPTOR_t, types),
2333     /* offsets */         { (UINT16) (offsetof(PolicyLocality_In, locality))},
2334     /* types */          {TPMI_SH_POLICY_H_UNMARSHAL,
2335                          TPMA_LOCALITY_P_UNMARSHAL,
2336                          END_OF_LIST,
2337                          END_OF_LIST}
2338 };
2339 #define _PolicyLocalityDataAddress (&_PolicyLocalityData)
2340 #else
2341 #define _PolicyLocalityDataAddress 0
2342 #endif // CC_PolicyLocality
2343 #if CC_PolicyNV
2344 #include "PolicyNV_fp.h"
2345 typedef TPM_RC (PolicyNV_Entry) (
2346     PolicyNV_In          *in
2347 );
2348 typedef const struct {
2349     PolicyNV_Entry        *entry;
2350     UINT16                inSize;
2351     UINT16                outSize;
2352     UINT16                offsetOfTypes;
2353     UINT16                paramOffsets[5];
2354     BYTE                  types[8];
2355 } PolicyNV_COMMAND_DESCRIPTOR_t;
2356 PolicyNV_COMMAND_DESCRIPTOR_t _PolicyNVData = {
2357     /* entry */          &TPM2_PolicyNV,
2358     /* inSize */         (UINT16) (sizeof(PolicyNV_In)),
2359     /* outSize */        0,
2360     /* offsetOfTypes */  offsetof(PolicyNV_COMMAND_DESCRIPTOR_t, types),
2361     /* offsets */         { (UINT16) (offsetof(PolicyNV_In, nvIndex)),
2362                          (UINT16) (offsetof(PolicyNV_In, policySession)),
2363                          (UINT16) (offsetof(PolicyNV_In, operandB)),
2364                          (UINT16) (offsetof(PolicyNV_In, offset)),
2365                          (UINT16) (offsetof(PolicyNV_In, operation))},
2366     /* types */          {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2367                          TPMI_RH_NV_INDEX_H_UNMARSHAL,
2368                          TPMI_SH_POLICY_H_UNMARSHAL,
2369                          TPM2B_OPERAND_P_UNMARSHAL,
2370                          UINT16_P_UNMARSHAL,
2371                          TPM_EO_P_UNMARSHAL,
2372                          END_OF_LIST,
2373                          END_OF_LIST}
2374 };
2375 #define _PolicyNVDataAddress (&_PolicyNVData)
2376 #else
2377 #define _PolicyNVDataAddress 0
2378 #endif // CC_PolicyNV
2379 #if CC_PolicyCounterTimer
2380 #include "PolicyCounterTimer_fp.h"
2381 typedef TPM_RC (PolicyCounterTimer_Entry) (
2382     PolicyCounterTimer_In *in
2383 );
2384 typedef const struct {
2385     PolicyCounterTimer_Entry *entry;
2386     UINT16                   inSize;
2387     UINT16                   outSize;

```

```

2388     UINT16                offsetOfTypes;
2389     UINT16                paramOffsets[3];
2390     BYTE                  types[6];
2391 } PolicyCounterTimer_COMMAND_DESCRIPTOR_t;
2392 PolicyCounterTimer_COMMAND_DESCRIPTOR_t _PolicyCounterTimerData = {
2393     /* entry */            &TPM2_PolicyCounterTimer,
2394     /* inSize */           (UINT16) (sizeof(PolicyCounterTimer_In)),
2395     /* outSize */          0,
2396     /* offsetOfTypes */    offsetof(PolicyCounterTimer_COMMAND_DESCRIPTOR_t,
types),
2397     /* offsets */          { (UINT16) (offsetof(PolicyCounterTimer_In, operandB)),
2398                             (UINT16) (offsetof(PolicyCounterTimer_In, offset)),
2399                             (UINT16) (offsetof(PolicyCounterTimer_In,
operation)) },
2400     /* types */            { TPMI_SH_POLICY_H_UNMARSHAL,
2401                             TPM2B_OPERAND_P_UNMARSHAL,
2402                             UINT16_P_UNMARSHAL,
2403                             TPM_EO_P_UNMARSHAL,
2404                             END_OF_LIST,
2405                             END_OF_LIST }
2406 };
2407 #define _PolicyCounterTimerDataAddress (&_PolicyCounterTimerData)
2408 #else
2409 #define _PolicyCounterTimerDataAddress 0
2410 #endif // CC_PolicyCounterTimer
2411 #if CC_PolicyCommandCode
2412 #include "PolicyCommandCode_fp.h"
2413 typedef TPM_RC (PolicyCommandCode_Entry) (
2414     PolicyCommandCode_In *in
2415 );
2416 typedef const struct {
2417     PolicyCommandCode_Entry *entry;
2418     UINT16 inSize;
2419     UINT16 outSize;
2420     UINT16 offsetOfTypes;
2421     UINT16 paramOffsets[1];
2422     BYTE types[4];
2423 } PolicyCommandCode_COMMAND_DESCRIPTOR_t;
2424 PolicyCommandCode_COMMAND_DESCRIPTOR_t _PolicyCommandCodeData = {
2425     /* entry */            &TPM2_PolicyCommandCode,
2426     /* inSize */           (UINT16) (sizeof(PolicyCommandCode_In)),
2427     /* outSize */          0,
2428     /* offsetOfTypes */    offsetof(PolicyCommandCode_COMMAND_DESCRIPTOR_t,
types),
2429     /* offsets */          { (UINT16) (offsetof(PolicyCommandCode_In, code)) },
2430     /* types */            { TPMI_SH_POLICY_H_UNMARSHAL,
2431                             TPM_CC_P_UNMARSHAL,
2432                             END_OF_LIST,
2433                             END_OF_LIST }
2434 };
2435 #define _PolicyCommandCodeDataAddress (&_PolicyCommandCodeData)
2436 #else
2437 #define _PolicyCommandCodeDataAddress 0
2438 #endif // CC_PolicyCommandCode
2439 #if CC_PolicyPhysicalPresence
2440 #include "PolicyPhysicalPresence_fp.h"
2441 typedef TPM_RC (PolicyPhysicalPresence_Entry) (
2442     PolicyPhysicalPresence_In *in
2443 );
2444 typedef const struct {
2445     PolicyPhysicalPresence_Entry *entry;
2446     UINT16 inSize;
2447     UINT16 outSize;
2448     UINT16 offsetOfTypes;
2449     BYTE types[3];
2450 } PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t;

```



```

2451 PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t _PolicyPhysicalPresenceData = {
2452     /* entry */ &TPM2_PolicyPhysicalPresence,
2453     /* inSize */ (UINT16) (sizeof(PolicyPhysicalPresence_In)),
2454     /* outSize */ 0,
2455     /* offsetOfTypes */
offsetof(PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t, types),
2456     /* offsets */ // No parameter offsets;
2457     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2458                 END_OF_LIST,
2459                 END_OF_LIST}
2460 };
2461 #define _PolicyPhysicalPresenceDataAddress (&_PolicyPhysicalPresenceData)
2462 #else
2463 #define _PolicyPhysicalPresenceDataAddress 0
2464 #endif // CC_PolicyPhysicalPresence
2465 #if CC_PolicyCpHash
2466 #include "PolicyCpHash_fp.h"
2467 typedef TPM_RC (PolicyCpHash_Entry) (
2468     PolicyCpHash_In *in
2469 );
2470 typedef const struct {
2471     PolicyCpHash_Entry *entry;
2472     UINT16 inSize;
2473     UINT16 outSize;
2474     UINT16 offsetOfTypes;
2475     UINT16 paramOffsets[1];
2476     BYTE types[4];
2477 } PolicyCpHash_COMMAND_DESCRIPTOR_t;
2478 PolicyCpHash_COMMAND_DESCRIPTOR_t _PolicyCpHashData = {
2479     /* entry */ &TPM2_PolicyCpHash,
2480     /* inSize */ (UINT16) (sizeof(PolicyCpHash_In)),
2481     /* outSize */ 0,
2482     /* offsetOfTypes */ offsetof(PolicyCpHash_COMMAND_DESCRIPTOR_t, types),
2483     /* offsets */ {(UINT16) (offsetof(PolicyCpHash_In, cpHashA))},
2484     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2485                 TPM2B_DIGEST_P_UNMARSHAL,
2486                 END_OF_LIST,
2487                 END_OF_LIST}
2488 };
2489 #define _PolicyCpHashDataAddress (&_PolicyCpHashData)
2490 #else
2491 #define _PolicyCpHashDataAddress 0
2492 #endif // CC_PolicyCpHash
2493 #if CC_PolicyNameHash
2494 #include "PolicyNameHash_fp.h"
2495 typedef TPM_RC (PolicyNameHash_Entry) (
2496     PolicyNameHash_In *in
2497 );
2498 typedef const struct {
2499     PolicyNameHash_Entry *entry;
2500     UINT16 inSize;
2501     UINT16 outSize;
2502     UINT16 offsetOfTypes;
2503     UINT16 paramOffsets[1];
2504     BYTE types[4];
2505 } PolicyNameHash_COMMAND_DESCRIPTOR_t;
2506 PolicyNameHash_COMMAND_DESCRIPTOR_t _PolicyNameHashData = {
2507     /* entry */ &TPM2_PolicyNameHash,
2508     /* inSize */ (UINT16) (sizeof(PolicyNameHash_In)),
2509     /* outSize */ 0,
2510     /* offsetOfTypes */ offsetof(PolicyNameHash_COMMAND_DESCRIPTOR_t, types),
2511     /* offsets */ {(UINT16) (offsetof(PolicyNameHash_In, nameHash))},
2512     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2513                 TPM2B_DIGEST_P_UNMARSHAL,
2514                 END_OF_LIST,
2515                 END_OF_LIST}

```

```

2516 };
2517 #define _PolicyNameHashDataAddress (&_PolicyNameHashData)
2518 #else
2519 #define _PolicyNameHashDataAddress 0
2520 #endif // CC_PolicyNameHash
2521 #if CC_PolicyDuplicationSelect
2522 #include "PolicyDuplicationSelect_fp.h"
2523 typedef TPM_RC (PolicyDuplicationSelect_Entry) (
2524     PolicyDuplicationSelect_In      *in
2525 );
2526 typedef const struct {
2527     PolicyDuplicationSelect_Entry  *entry;
2528     UINT16                          inSize;
2529     UINT16                          outSize;
2530     UINT16                          offsetOfTypes;
2531     UINT16                          paramOffsets[3];
2532     BYTE                            types[6];
2533 } PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t;
2534 PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t _PolicyDuplicationSelectData = {
2535     /* entry */ &TPM2_PolicyDuplicationSelect,
2536     /* inSize */ (UINT16) (sizeof(PolicyDuplicationSelect_In)),
2537     /* outSize */ 0,
2538     /* offsetOfTypes */
2539     offsetof(PolicyDuplicationSelect_COMMAND_DESCRIPTOR_t, types),
2540     /* offsets */ { (UINT16) (offsetof(PolicyDuplicationSelect_In,
2541     objectName)),
2542     (UINT16) (offsetof(PolicyDuplicationSelect_In,
2543     newParentName)),
2544     (UINT16) (offsetof(PolicyDuplicationSelect_In,
2545     includeObject))},
2546     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2547     TPM2B_NAME_P_UNMARSHAL,
2548     TPM2B_NAME_P_UNMARSHAL,
2549     TPMI_YES_NO_P_UNMARSHAL,
2550     END_OF_LIST,
2551     END_OF_LIST}
2552 };
2553 #define _PolicyDuplicationSelectDataAddress (&_PolicyDuplicationSelectData)
2554 #else
2555 #define _PolicyDuplicationSelectDataAddress 0
2556 #endif // CC_PolicyDuplicationSelect
2557 #if CC_PolicyAuthorize
2558 #include "PolicyAuthorize_fp.h"
2559 typedef TPM_RC (PolicyAuthorize_Entry) (
2560     PolicyAuthorize_In      *in
2561 );
2562 typedef const struct {
2563     PolicyAuthorize_Entry  *entry;
2564     UINT16                  inSize;
2565     UINT16                  outSize;
2566     UINT16                  offsetOfTypes;
2567     UINT16                  paramOffsets[4];
2568     BYTE                    types[7];
2569 } PolicyAuthorize_COMMAND_DESCRIPTOR_t;
2570 PolicyAuthorize_COMMAND_DESCRIPTOR_t _PolicyAuthorizeData = {
2571     /* entry */ &TPM2_PolicyAuthorize,
2572     /* inSize */ (UINT16) (sizeof(PolicyAuthorize_In)),
2573     /* outSize */ 0,
2574     /* offsetOfTypes */
2575     offsetof(PolicyAuthorize_COMMAND_DESCRIPTOR_t, types),
2576     /* offsets */ { (UINT16) (offsetof(PolicyAuthorize_In, approvedPolicy)),
2577     (UINT16) (offsetof(PolicyAuthorize_In, policyRef)),
2578     (UINT16) (offsetof(PolicyAuthorize_In, keySign)),
2579     (UINT16) (offsetof(PolicyAuthorize_In, checkTicket))},
2580     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2581     TPM2B_DIGEST_P_UNMARSHAL,
2582     TPM2B_NONCE_P_UNMARSHAL,

```

```

2578             TPM2B_NAME_P_UNMARSHAL,
2579             TPMT_TK_VERIFIED_P_UNMARSHAL,
2580             END_OF_LIST,
2581             END_OF_LIST}
2582 };
2583 #define _PolicyAuthorizeDataAddress (&_PolicyAuthorizeData)
2584 #else
2585 #define _PolicyAuthorizeDataAddress 0
2586 #endif // CC_PolicyAuthorize
2587 #if CC_PolicyAuthValue
2588 #include "PolicyAuthValue_fp.h"
2589 typedef TPM_RC (PolicyAuthValue_Entry) (
2590     PolicyAuthValue_In      *in
2591 );
2592 typedef const struct {
2593     PolicyAuthValue_Entry    *entry;
2594     UINT16                   inSize;
2595     UINT16                   outSize;
2596     UINT16                   offsetOfTypes;
2597     BYTE                     types[3];
2598 } PolicyAuthValue_COMMAND_DESCRIPTOR_t;
2599 PolicyAuthValue_COMMAND_DESCRIPTOR_t _PolicyAuthValueData = {
2600     /* entry */           &TPM2_PolicyAuthValue,
2601     /* inSize */          (UINT16) (sizeof(PolicyAuthValue_In)),
2602     /* outSize */         0,
2603     /* offsetOfTypes */   offsetof(PolicyAuthValue_COMMAND_DESCRIPTOR_t, types),
2604     /* offsets */         // No parameter offsets;
2605     /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2606                           END_OF_LIST,
2607                           END_OF_LIST}
2608 };
2609 #define _PolicyAuthValueDataAddress (&_PolicyAuthValueData)
2610 #else
2611 #define _PolicyAuthValueDataAddress 0
2612 #endif // CC_PolicyAuthValue
2613 #if CC_PolicyPassword
2614 #include "PolicyPassword_fp.h"
2615 typedef TPM_RC (PolicyPassword_Entry) (
2616     PolicyPassword_In      *in
2617 );
2618 typedef const struct {
2619     PolicyPassword_Entry    *entry;
2620     UINT16                   inSize;
2621     UINT16                   outSize;
2622     UINT16                   offsetOfTypes;
2623     BYTE                     types[3];
2624 } PolicyPassword_COMMAND_DESCRIPTOR_t;
2625 PolicyPassword_COMMAND_DESCRIPTOR_t _PolicyPasswordData = {
2626     /* entry */           &TPM2_PolicyPassword,
2627     /* inSize */          (UINT16) (sizeof(PolicyPassword_In)),
2628     /* outSize */         0,
2629     /* offsetOfTypes */   offsetof(PolicyPassword_COMMAND_DESCRIPTOR_t, types),
2630     /* offsets */         // No parameter offsets;
2631     /* types */           {TPMI_SH_POLICY_H_UNMARSHAL,
2632                           END_OF_LIST,
2633                           END_OF_LIST}
2634 };
2635 #define _PolicyPasswordDataAddress (&_PolicyPasswordData)
2636 #else
2637 #define _PolicyPasswordDataAddress 0
2638 #endif // CC_PolicyPassword
2639 #if CC_PolicyGetDigest
2640 #include "PolicyGetDigest_fp.h"
2641 typedef TPM_RC (PolicyGetDigest_Entry) (
2642     PolicyGetDigest_In      *in,
2643     PolicyGetDigest_Out     *out

```



```

2644 );
2645 typedef const struct {
2646     PolicyGetDigest_Entry    *entry;
2647     UINT16                   inSize;
2648     UINT16                   outSize;
2649     UINT16                   offsetOfTypes;
2650     BYTE                     types[4];
2651 } PolicyGetDigest_COMMAND_DESCRIPTOR_t;
2652 PolicyGetDigest_COMMAND_DESCRIPTOR_t _PolicyGetDigestData = {
2653     /* entry */           &TPM2_PolicyGetDigest,
2654     /* inSize */          (UINT16) (sizeof(PolicyGetDigest_In)),
2655     /* outSize */          (UINT16) (sizeof(PolicyGetDigest_Out)),
2656     /* offsetOfTypes */   offsetof(PolicyGetDigest_COMMAND_DESCRIPTOR_t, types),
2657     /* offsets */          // No parameter offsets;
2658     /* types */            {TPMI_SH_POLICY_H_UNMARSHAL,
2659                             END_OF_LIST,
2660                             TPM2B_DIGEST_P_MARSHAL,
2661                             END_OF_LIST};
2662 };
2663 #define _PolicyGetDigestDataAddress (&_PolicyGetDigestData)
2664 #else
2665 #define _PolicyGetDigestDataAddress 0
2666 #endif // CC_PolicyGetDigest
2667 #if CC_PolicyNvWritten
2668 #include "PolicyNvWritten_fp.h"
2669 typedef TPM_RC (PolicyNvWritten_Entry) (
2670     PolicyNvWritten_In    *in
2671 );
2672 typedef const struct {
2673     PolicyNvWritten_Entry    *entry;
2674     UINT16                   inSize;
2675     UINT16                   outSize;
2676     UINT16                   offsetOfTypes;
2677     UINT16                   paramOffsets[1];
2678     BYTE                     types[4];
2679 } PolicyNvWritten_COMMAND_DESCRIPTOR_t;
2680 PolicyNvWritten_COMMAND_DESCRIPTOR_t _PolicyNvWrittenData = {
2681     /* entry */           &TPM2_PolicyNvWritten,
2682     /* inSize */          (UINT16) (sizeof(PolicyNvWritten_In)),
2683     /* outSize */          0,
2684     /* offsetOfTypes */   offsetof(PolicyNvWritten_COMMAND_DESCRIPTOR_t, types),
2685     /* offsets */          {(UINT16) (offsetof(PolicyNvWritten_In, writtenSet))},
2686     /* types */            {TPMI_SH_POLICY_H_UNMARSHAL,
2687                             TPMI_YES_NO_P_UNMARSHAL,
2688                             END_OF_LIST,
2689                             END_OF_LIST};
2690 };
2691 #define _PolicyNvWrittenDataAddress (&_PolicyNvWrittenData)
2692 #else
2693 #define _PolicyNvWrittenDataAddress 0
2694 #endif // CC_PolicyNvWritten
2695 #if CC_PolicyTemplate
2696 #include "PolicyTemplate_fp.h"
2697 typedef TPM_RC (PolicyTemplate_Entry) (
2698     PolicyTemplate_In    *in
2699 );
2700 typedef const struct {
2701     PolicyTemplate_Entry    *entry;
2702     UINT16                   inSize;
2703     UINT16                   outSize;
2704     UINT16                   offsetOfTypes;
2705     UINT16                   paramOffsets[1];
2706     BYTE                     types[4];
2707 } PolicyTemplate_COMMAND_DESCRIPTOR_t;
2708 PolicyTemplate_COMMAND_DESCRIPTOR_t _PolicyTemplateData = {
2709     /* entry */           &TPM2_PolicyTemplate,

```

```

2710     /* inSize      */ (UINT16) (sizeof(PolicyTemplate_In)),
2711     /* outSize     */ 0,
2712     /* offsetOfTypes */ offsetof(PolicyTemplate_COMMAND_DESCRIPTOR_t, types),
2713     /* offsets      */ {(UINT16) (offsetof(PolicyTemplate_In, templateHash))},
2714     /* types        */ {TPMI_SH_POLICY_H_UNMARSHAL,
2715                        TPM2B_DIGEST_P_UNMARSHAL,
2716                        END_OF_LIST,
2717                        END_OF_LIST};
2718 };
2719 #define _PolicyTemplateDataAddress (&_PolicyTemplateData)
2720 #else
2721 #define _PolicyTemplateDataAddress 0
2722 #endif // CC_PolicyTemplate
2723 #if CC_PolicyAuthorizeNV
2724 #include "PolicyAuthorizeNV_fp.h"
2725 typedef TPM_RC (PolicyAuthorizeNV_Entry) (
2726     PolicyAuthorizeNV_In *in
2727 );
2728 typedef const struct {
2729     PolicyAuthorizeNV_Entry *entry;
2730     UINT16 inSize;
2731     UINT16 outSize;
2732     UINT16 offsetOfTypes;
2733     UINT16 paramOffsets[2];
2734     BYTE types[5];
2735 } PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t;
2736 PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t _PolicyAuthorizeNVData = {
2737     /* entry      */ &TPM2_PolicyAuthorizeNV,
2738     /* inSize     */ (UINT16) (sizeof(PolicyAuthorizeNV_In)),
2739     /* outSize    */ 0,
2740     /* offsetOfTypes */ offsetof(PolicyAuthorizeNV_COMMAND_DESCRIPTOR_t,
2741     types),
2742     /* offsets     */ {(UINT16) (offsetof(PolicyAuthorizeNV_In, nvIndex)),
2743                       (UINT16) (offsetof(PolicyAuthorizeNV_In,
2744     policySession)))},
2745     /* types       */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2746                       TPMI_RH_NV_INDEX_H_UNMARSHAL,
2747                       TPMI_SH_POLICY_H_UNMARSHAL,
2748                       END_OF_LIST,
2749                       END_OF_LIST};
2750 };
2751 #define _PolicyAuthorizeNVDataAddress (&_PolicyAuthorizeNVData)
2752 #else
2753 #define _PolicyAuthorizeNVDataAddress 0
2754 #endif // CC_PolicyAuthorizeNV
2755 #if CC_CreatePrimary
2756 #include "CreatePrimary_fp.h"
2757 typedef TPM_RC (CreatePrimary_Entry) (
2758     CreatePrimary_In *in,
2759     CreatePrimary_Out *out
2760 );
2761 typedef const struct {
2762     CreatePrimary_Entry *entry;
2763     UINT16 inSize;
2764     UINT16 outSize;
2765     UINT16 offsetOfTypes;
2766     UINT16 paramOffsets[9];
2767     BYTE types[13];
2768 } CreatePrimary_COMMAND_DESCRIPTOR_t;
2769 CreatePrimary_COMMAND_DESCRIPTOR_t _CreatePrimaryData = {
2770     /* entry      */ &TPM2_CreatePrimary,
2771     /* inSize     */ (UINT16) (sizeof(CreatePrimary_In)),
2772     /* outSize    */ (UINT16) (sizeof(CreatePrimary_Out)),
2773     /* offsetOfTypes */ offsetof(CreatePrimary_COMMAND_DESCRIPTOR_t, types),
2774     /* offsets     */ {(UINT16) (offsetof(CreatePrimary_In, inSensitive)),
2775                       (UINT16) (offsetof(CreatePrimary_In, inPublic))},

```

```

2774         (UINT16) (offsetof(CreatePrimary_In, outsideInfo)),
2775         (UINT16) (offsetof(CreatePrimary_In, creationPCR)),
2776         (UINT16) (offsetof(CreatePrimary_Out, outPublic)),
2777         (UINT16) (offsetof(CreatePrimary_Out, creationData)),
2778         (UINT16) (offsetof(CreatePrimary_Out, creationHash)),
2779         (UINT16) (offsetof(CreatePrimary_Out, creationTicket)),
2780         (UINT16) (offsetof(CreatePrimary_Out, name))),
2781     /* types */ {TPMI_RH_HIERARCHY_H_UNMARSHAL + ADD_FLAG,
2782     TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
2783     TPM2B_PUBLIC_P_UNMARSHAL,
2784     TPM2B_DATA_P_UNMARSHAL,
2785     TPML_PCR_SELECTION_P_UNMARSHAL,
2786     END_OF_LIST,
2787     TPM_HANDLE_H_MARSHAL,
2788     TPM2B_PUBLIC_P_MARSHAL,
2789     TPM2B_CREATION_DATA_P_MARSHAL,
2790     TPM2B_DIGEST_P_MARSHAL,
2791     TPMT_TK_CREATION_P_MARSHAL,
2792     TPM2B_NAME_P_MARSHAL,
2793     END_OF_LIST};
2794 };
2795 #define _CreatePrimaryDataAddress (&CreatePrimaryData)
2796 #else
2797 #define _CreatePrimaryDataAddress 0
2798 #endif // CC_CreatePrimary
2799 #if CC_HierarchyControl
2800 #include "HierarchyControl_fp.h"
2801 typedef TPM_RC (HierarchyControl_Entry) (
2802     HierarchyControl_In *in
2803 );
2804 typedef const struct {
2805     HierarchyControl_Entry *entry;
2806     UINT16 inSize;
2807     UINT16 outSize;
2808     UINT16 offsetOfTypes;
2809     UINT16 paramOffsets[2];
2810     BYTE types[5];
2811 } HierarchyControl_COMMAND_DESCRIPTOR_t;
2812 HierarchyControl_COMMAND_DESCRIPTOR_t _HierarchyControlData = {
2813     /* entry */ &TPM2_HierarchyControl,
2814     /* inSize */ (UINT16) (sizeof(HierarchyControl_In)),
2815     /* outSize */ 0,
2816     /* offsetOfTypes */ offsetof(HierarchyControl_COMMAND_DESCRIPTOR_t, types),
2817     /* offsets */ {(UINT16) (offsetof(HierarchyControl_In, enable)),
2818     (UINT16) (offsetof(HierarchyControl_In, state))},
2819     /* types */ {TPMI_RH_HIERARCHY_H_UNMARSHAL,
2820     TPMI_RH_ENABLES_P_UNMARSHAL,
2821     TPMI_YES_NO_P_UNMARSHAL,
2822     END_OF_LIST,
2823     END_OF_LIST};
2824 };
2825 #define _HierarchyControlDataAddress (&_HierarchyControlData)
2826 #else
2827 #define _HierarchyControlDataAddress 0
2828 #endif // CC_HierarchyControl
2829 #if CC_SetPrimaryPolicy
2830 #include "SetPrimaryPolicy_fp.h"
2831 typedef TPM_RC (SetPrimaryPolicy_Entry) (
2832     SetPrimaryPolicy_In *in
2833 );
2834 typedef const struct {
2835     SetPrimaryPolicy_Entry *entry;
2836     UINT16 inSize;
2837     UINT16 outSize;
2838     UINT16 offsetOfTypes;
2839     UINT16 paramOffsets[2];

```

```

2840     BYTE                                     types[5];
2841 } SetPrimaryPolicy_COMMAND_DESCRIPTOR_t;
2842 SetPrimaryPolicy_COMMAND_DESCRIPTOR_t _SetPrimaryPolicyData = {
2843     /* entry */                               &TPM2_SetPrimaryPolicy,
2844     /* inSize */                             (UINT16) (sizeof(SetPrimaryPolicy_In)),
2845     /* outSize */                             0,
2846     /* offsetOfTypes */                       offsetof(SetPrimaryPolicy_COMMAND_DESCRIPTOR_t, types),
2847     /* offsets */                             {(UINT16) (offsetof(SetPrimaryPolicy_In, authPolicy)),
2848                                              (UINT16) (offsetof(SetPrimaryPolicy_In, hashAlg))},
2849     /* types */                               {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
2850                                              TPM2B_DIGEST_P_UNMARSHAL,
2851                                              TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2852                                              END_OF_LIST,
2853                                              END_OF_LIST};
2854 };
2855 #define _SetPrimaryPolicyDataAddress (&_SetPrimaryPolicyData)
2856 #else
2857 #define _SetPrimaryPolicyDataAddress 0
2858 #endif // CC_SetPrimaryPolicy
2859 #if CC_ChangePPS
2860 #include "ChangePPS_fp.h"
2861 typedef TPM_RC (ChangePPS_Entry) (
2862     ChangePPS_In *in
2863 );
2864 typedef const struct {
2865     ChangePPS_Entry *entry;
2866     UINT16 inSize;
2867     UINT16 outSize;
2868     UINT16 offsetOfTypes;
2869     BYTE types[3];
2870 } ChangePPS_COMMAND_DESCRIPTOR_t;
2871 ChangePPS_COMMAND_DESCRIPTOR_t _ChangePPSData = {
2872     /* entry */                               &TPM2_ChangePPS,
2873     /* inSize */                             (UINT16) (sizeof(ChangePPS_In)),
2874     /* outSize */                             0,
2875     /* offsetOfTypes */                       offsetof(ChangePPS_COMMAND_DESCRIPTOR_t, types),
2876     /* offsets */                             // No parameter offsets;
2877     /* types */                               {TPMI_RH_PLATFORM_H_UNMARSHAL,
2878                                              END_OF_LIST,
2879                                              END_OF_LIST};
2880 };
2881 #define _ChangePPSDataAddress (&_ChangePPSData)
2882 #else
2883 #define _ChangePPSDataAddress 0
2884 #endif // CC_ChangePPS
2885 #if CC_ChangeEPS
2886 #include "ChangeEPS_fp.h"
2887 typedef TPM_RC (ChangeEPS_Entry) (
2888     ChangeEPS_In *in
2889 );
2890 typedef const struct {
2891     ChangeEPS_Entry *entry;
2892     UINT16 inSize;
2893     UINT16 outSize;
2894     UINT16 offsetOfTypes;
2895     BYTE types[3];
2896 } ChangeEPS_COMMAND_DESCRIPTOR_t;
2897 ChangeEPS_COMMAND_DESCRIPTOR_t _ChangeEPSData = {
2898     /* entry */                               &TPM2_ChangeEPS,
2899     /* inSize */                             (UINT16) (sizeof(ChangeEPS_In)),
2900     /* outSize */                             0,
2901     /* offsetOfTypes */                       offsetof(ChangeEPS_COMMAND_DESCRIPTOR_t, types),
2902     /* offsets */                             // No parameter offsets;
2903     /* types */                               {TPMI_RH_PLATFORM_H_UNMARSHAL,
2904                                              END_OF_LIST,
2905                                              END_OF_LIST};

```

```

2906 };
2907 #define _ChangeEPSDataAddress (&_ChangeEPSData)
2908 #else
2909 #define _ChangeEPSDataAddress 0
2910 #endif // CC_ChangeEPS
2911 #if CC_Clear
2912 #include "Clear_fp.h"
2913 typedef TPM_RC (Clear_Entry) (
2914     Clear_In          *in
2915 );
2916 typedef const struct {
2917     Clear_Entry        *entry;
2918     UINT16             inSize;
2919     UINT16             outSize;
2920     UINT16             offsetOfTypes;
2921     BYTE               types[3];
2922 } Clear_COMMAND_DESCRIPTOR_t;
2923 Clear_COMMAND_DESCRIPTOR_t _ClearData = {
2924     /* entry */        &TPM2_Clear,
2925     /* inSize */       (UINT16) (sizeof(Clear_In)),
2926     /* outSize */      0,
2927     /* offsetOfTypes */ offsetof(Clear_COMMAND_DESCRIPTOR_t, types),
2928     /* offsets */      // No parameter offsets;
2929     /* types */        {TPMI_RH_CLEAR_H_UNMARSHAL,
2930                         END_OF_LIST,
2931                         END_OF_LIST};
2932 };
2933 #define _ClearDataAddress (&_ClearData)
2934 #else
2935 #define _ClearDataAddress 0
2936 #endif // CC_Clear
2937 #if CC_ClearControl
2938 #include "ClearControl_fp.h"
2939 typedef TPM_RC (ClearControl_Entry) (
2940     ClearControl_In    *in
2941 );
2942 typedef const struct {
2943     ClearControl_Entry *entry;
2944     UINT16             inSize;
2945     UINT16             outSize;
2946     UINT16             offsetOfTypes;
2947     UINT16             paramOffsets[1];
2948     BYTE               types[4];
2949 } ClearControl_COMMAND_DESCRIPTOR_t;
2950 ClearControl_COMMAND_DESCRIPTOR_t _ClearControlData = {
2951     /* entry */        &TPM2_ClearControl,
2952     /* inSize */       (UINT16) (sizeof(ClearControl_In)),
2953     /* outSize */      0,
2954     /* offsetOfTypes */ offsetof(ClearControl_COMMAND_DESCRIPTOR_t, types),
2955     /* offsets */      {(UINT16) (offsetof(ClearControl_In, disable))},
2956     /* types */        {TPMI_RH_CLEAR_H_UNMARSHAL,
2957                         TPMI_YES_NO_P_UNMARSHAL,
2958                         END_OF_LIST,
2959                         END_OF_LIST};
2960 };
2961 #define _ClearControlDataAddress (&_ClearControlData)
2962 #else
2963 #define _ClearControlDataAddress 0
2964 #endif // CC_ClearControl
2965 #if CC_HierarchyChangeAuth
2966 #include "HierarchyChangeAuth_fp.h"
2967 typedef TPM_RC (HierarchyChangeAuth_Entry) (
2968     HierarchyChangeAuth_In *in
2969 );
2970 typedef const struct {
2971     HierarchyChangeAuth_Entry *entry;

```



```

2972     UINT16             inSize;
2973     UINT16             outSize;
2974     UINT16             offsetOfTypes;
2975     UINT16             paramOffsets[1];
2976     BYTE               types[4];
2977 } HierarchyChangeAuth_COMMAND_DESCRIPTOR_t;
2978 HierarchyChangeAuth_COMMAND_DESCRIPTOR_t _HierarchyChangeAuthData = {
2979     /* entry          */      &TPM2_HierarchyChangeAuth,
2980     /* inSize         */      (UINT16) (sizeof(HierarchyChangeAuth_In)),
2981     /* outSize        */      0,
2982     /* offsetOfTypes */      offsetof(HierarchyChangeAuth_COMMAND_DESCRIPTOR_t,
types),
2983     /* offsets        */      {(UINT16) (offsetof(HierarchyChangeAuth_In, newAuth))},
2984     /* types          */      {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
TPM2B_AUTH_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST}
2985 };
2986 #define _HierarchyChangeAuthDataAddress (&_HierarchyChangeAuthData)
2987 #else
2988 #define _HierarchyChangeAuthDataAddress 0
2989 #endif // CC_HierarchyChangeAuth
2990 #if CC_DictionaryAttackLockReset
2991 #include "DictionaryAttackLockReset_fp.h"
2992 typedef TPM_RC (DictionaryAttackLockReset_Entry) (
2993     DictionaryAttackLockReset_In *in
2994 );
2995 typedef const struct {
2996     DictionaryAttackLockReset_Entry *entry;
2997     UINT16 inSize;
2998     UINT16 outSize;
2999     UINT16 offsetOfTypes;
3000     BYTE types[3];
3001 } DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t;
3002 DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t _DictionaryAttackLockResetData = {
3003     /* entry          */      &TPM2_DictionaryAttackLockReset,
3004     /* inSize         */      (UINT16) (sizeof(DictionaryAttackLockReset_In)),
3005     /* outSize        */      0,
3006     /* offsetOfTypes */      offsetof(DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t, types),
3007     /* offsets        */      // No parameter offsets;
3008     /* types          */      {TPMI_RH_LOCKOUT_H_UNMARSHAL,
END_OF_LIST,
END_OF_LIST}
3009 };
3010 #define _DictionaryAttackLockResetDataAddress (&_DictionaryAttackLockResetData)
3011 #else
3012 #define _DictionaryAttackLockResetDataAddress 0
3013 #endif // CC_DictionaryAttackLockReset
3014 #if CC_DictionaryAttackParameters
3015 #include "DictionaryAttackParameters_fp.h"
3016 typedef TPM_RC (DictionaryAttackParameters_Entry) (
3017     DictionaryAttackParameters_In *in
3018 );
3019 typedef const struct {
3020     DictionaryAttackParameters_Entry *entry;
3021     UINT16 inSize;
3022     UINT16 outSize;
3023     UINT16 offsetOfTypes;
3024     UINT16 paramOffsets[3];
3025     BYTE types[6];
3026 } DictionaryAttackParameters_COMMAND_DESCRIPTOR_t;
3027 DictionaryAttackParameters_COMMAND_DESCRIPTOR_t _DictionaryAttackParametersData = {
3028     /* entry          */      &TPM2_DictionaryAttackParameters,

```

```

3034     /* inSize */
3035     (UINT16) (sizeof(DictionaryAttackParameters_In)),
3036     /* outSize */
3037     /* offsetOfTypes */
offsetof(DictionaryAttackParameters_COMMAND_DESCRIPTOR_t, types),
3038     /* offsets */
3039     {(UINT16) (offsetof(DictionaryAttackParameters_In, newMaxTries)),
3040      (UINT16) (offsetof(DictionaryAttackParameters_In, newRecoveryTime)),
3041      (UINT16) (offsetof(DictionaryAttackParameters_In, lockoutRecovery))},
3042     /* types */
3043     {TPMI_RH_LOCKOUT_H_UNMARSHAL,
3044      UINT32_P_UNMARSHAL,
3045      UINT32_P_UNMARSHAL,
3046      UINT32_P_UNMARSHAL,
3047      END_OF_LIST,
3048      END_OF_LIST}
3049 };
3050 #define DictionaryAttackParametersDataAddress (&DictionaryAttackParametersData)
3051 #else
3052 #define DictionaryAttackParametersDataAddress 0
3053 #endif // CC_DictionaryAttackParameters
3054 #if CC_PP_Commands
3055 #include "PP_Commands_fp.h"
3056 typedef TPM_RC (PP_Commands_Entry) (
3057     PP_Commands_In *in
3058 );
3059 typedef const struct {
3060     PP_Commands_Entry *entry;
3061     UINT16 inSize;
3062     UINT16 outSize;
3063     UINT16 offsetOfTypes;
3064     UINT16 paramOffsets[2];
3065     BYTE types[5];
3066 } PP_Commands_COMMAND_DESCRIPTOR_t;
3067 PP_Commands_COMMAND_DESCRIPTOR_t PP_CommandsData = {
3068     /* entry */
3069     /* inSize */
3070     /* outSize */
3071     /* offsetOfTypes */
3072     /* offsets */
3073     /* types */
3074     {&TPM2_PP_Commands,
3075      (UINT16) (sizeof(PP_Commands_In)),
3076      0,
3077      offsetof(PP_Commands_COMMAND_DESCRIPTOR_t, types),
3078      {(UINT16) (offsetof(PP_Commands_In, setList)),
3079       (UINT16) (offsetof(PP_Commands_In, clearList))},
3080      {TPMI_RH_PLATFORM_H_UNMARSHAL,
3081       TPML_CC_P_UNMARSHAL,
3082       TPML_CC_P_UNMARSHAL,
3083       END_OF_LIST,
3084       END_OF_LIST}
3085     };
3086 #define PP_CommandsDataAddress (&PP_CommandsData)
3087 #else
3088 #define PP_CommandsDataAddress 0
3089 #endif // CC_PP_Commands
3090 #if CC_SetAlgorithmSet
3091 #include "SetAlgorithmSet_fp.h"
3092 typedef TPM_RC (SetAlgorithmSet_Entry) (
3093     SetAlgorithmSet_In *in
3094 );
3095 typedef const struct {
3096     SetAlgorithmSet_Entry *entry;
3097     UINT16 inSize;
3098     UINT16 outSize;
3099     UINT16 offsetOfTypes;
3100     UINT16 paramOffsets[1];
3101     BYTE types[4];
3102 } SetAlgorithmSet_COMMAND_DESCRIPTOR_t;
3103 SetAlgorithmSet_COMMAND_DESCRIPTOR_t SetAlgorithmSetData = {

```

```

3095     /* entry          */      &TPM2_SetAlgorithmSet,
3096     /* inSize         */      (UINT16) (sizeof(SetAlgorithmSet_In)),
3097     /* outSize        */      0,
3098     /* offsetOfTypes  */      offsetof(SetAlgorithmSet_COMMAND_DESCRIPTOR_t, types),
3099     /* offsets         */      {(UINT16) (offsetof(SetAlgorithmSet_In, algorithmSet))},
3100     /* types          */      {TPMI_RH_PLATFORM_H_UNMARSHAL,
3101                               UINT32_P_UNMARSHAL,
3102                               END_OF_LIST,
3103                               END_OF_LIST}
3104 };
3105 #define _SetAlgorithmSetDataAddress (&_SetAlgorithmSetData)
3106 #else
3107 #define _SetAlgorithmSetDataAddress 0
3108 #endif // CC_SetAlgorithmSet
3109 #if CC_FieldUpgradeStart
3110 #include "FieldUpgradeStart_fp.h"
3111 typedef TPM_RC (FieldUpgradeStart_Entry) (
3112     FieldUpgradeStart_In      *in
3113 );
3114 typedef const struct {
3115     FieldUpgradeStart_Entry    *entry;
3116     UINT16                     inSize;
3117     UINT16                     outSize;
3118     UINT16                     offsetOfTypes;
3119     UINT16                     paramOffsets[3];
3120     BYTE                       types[6];
3121 } FieldUpgradeStart_COMMAND_DESCRIPTOR_t;
3122 FieldUpgradeStart_COMMAND_DESCRIPTOR_t _FieldUpgradeStartData = {
3123     /* entry          */      &TPM2_FieldUpgradeStart,
3124     /* inSize         */      (UINT16) (sizeof(FieldUpgradeStart_In)),
3125     /* outSize        */      0,
3126     /* offsetOfTypes  */      offsetof(FieldUpgradeStart_COMMAND_DESCRIPTOR_t,
3127     types),
3128     /* offsets         */      {(UINT16) (offsetof(FieldUpgradeStart_In, keyHandle)),
3129                               (UINT16) (offsetof(FieldUpgradeStart_In, fuDigest)),
3130                               (UINT16) (offsetof(FieldUpgradeStart_In,
3131 manifestSignature)))},
3132     /* types          */      {TPMI_RH_PLATFORM_H_UNMARSHAL,
3133                               TPMI_DH_OBJECT_H_UNMARSHAL,
3134                               TPM2B_DIGEST_P_UNMARSHAL,
3135                               TPMT_SIGNATURE_P_UNMARSHAL,
3136                               END_OF_LIST,
3137                               END_OF_LIST}
3138 };
3139 #define _FieldUpgradeStartDataAddress (&_FieldUpgradeStartData)
3140 #else
3141 #define _FieldUpgradeStartDataAddress 0
3142 #endif // CC_FieldUpgradeStart
3143 #if CC_FieldUpgradeData
3144 #include "FieldUpgradeData_fp.h"
3145 typedef TPM_RC (FieldUpgradeData_Entry) (
3146     FieldUpgradeData_In      *in,
3147     FieldUpgradeData_Out     *out
3148 );
3149 typedef const struct {
3150     FieldUpgradeData_Entry    *entry;
3151     UINT16                     inSize;
3152     UINT16                     outSize;
3153     UINT16                     offsetOfTypes;
3154     UINT16                     paramOffsets[1];
3155     BYTE                       types[5];
3156 } FieldUpgradeData_COMMAND_DESCRIPTOR_t;
3157 FieldUpgradeData_COMMAND_DESCRIPTOR_t _FieldUpgradeDataData = {
3158     /* entry          */      &TPM2_FieldUpgradeData,
3159     /* inSize         */      (UINT16) (sizeof(FieldUpgradeData_In)),
3160     /* outSize        */      (UINT16) (sizeof(FieldUpgradeData_Out)),

```



```

3159     /* offsetOfTypes */    offsetof(FieldUpgradeData_COMMAND_DESCRIPTOR_t, types),
3160     /* offsets */          {(UINT16) (offsetof(FieldUpgradeData_Out, firstDigest))},
3161     /* types */            {TPM2B_MAX_BUFFER_P_UNMARSHAL,
3162                             END_OF_LIST,
3163                             TPMT_HA_P_MARSHAL,
3164                             TPMT_HA_P_MARSHAL,
3165                             END_OF_LIST};
3166 };
3167 #define _FieldUpgradeDataDataAddress (&_FieldUpgradeDataData)
3168 #else
3169 #define _FieldUpgradeDataDataAddress 0
3170 #endif // CC_FieldUpgradeData
3171 #if CC_FirmwareRead
3172 #include "FirmwareRead_fp.h"
3173 typedef TPM_RC (FirmwareRead_Entry) (
3174     FirmwareRead_In          *in,
3175     FirmwareRead_Out         *out
3176 );
3177 typedef const struct {
3178     FirmwareRead_Entry      *entry;
3179     UINT16                  inSize;
3180     UINT16                  outSize;
3181     UINT16                  offsetOfTypes;
3182     BYTE                    types[4];
3183 } FirmwareRead_COMMAND_DESCRIPTOR_t;
3184 FirmwareRead_COMMAND_DESCRIPTOR_t _FirmwareReadData = {
3185     /* entry */              &TPM2_FirmwareRead,
3186     /* inSize */             (UINT16) (sizeof(FirmwareRead_In)),
3187     /* outSize */            (UINT16) (sizeof(FirmwareRead_Out)),
3188     /* offsetOfTypes */      offsetof(FirmwareRead_COMMAND_DESCRIPTOR_t, types),
3189     /* offsets */            // No parameter offsets;
3190     /* types */              {UINT32_P_UNMARSHAL,
3191                             END_OF_LIST,
3192                             TPM2B_MAX_BUFFER_P_MARSHAL,
3193                             END_OF_LIST};
3194 };
3195 #define _FirmwareReadDataAddress (&_FirmwareReadData)
3196 #else
3197 #define _FirmwareReadDataAddress 0
3198 #endif // CC_FirmwareRead
3199 #if CC_ContextSave
3200 #include "ContextSave_fp.h"
3201 typedef TPM_RC (ContextSave_Entry) (
3202     ContextSave_In          *in,
3203     ContextSave_Out         *out
3204 );
3205 typedef const struct {
3206     ContextSave_Entry      *entry;
3207     UINT16                  inSize;
3208     UINT16                  outSize;
3209     UINT16                  offsetOfTypes;
3210     BYTE                    types[4];
3211 } ContextSave_COMMAND_DESCRIPTOR_t;
3212 ContextSave_COMMAND_DESCRIPTOR_t _ContextSaveData = {
3213     /* entry */              &TPM2_ContextSave,
3214     /* inSize */             (UINT16) (sizeof(ContextSave_In)),
3215     /* outSize */            (UINT16) (sizeof(ContextSave_Out)),
3216     /* offsetOfTypes */      offsetof(ContextSave_COMMAND_DESCRIPTOR_t, types),
3217     /* offsets */            // No parameter offsets;
3218     /* types */              {TPMI_DH_CONTEXT_H_UNMARSHAL,
3219                             END_OF_LIST,
3220                             TPMS_CONTEXT_P_MARSHAL,
3221                             END_OF_LIST};
3222 };
3223 #define _ContextSaveDataAddress (&_ContextSaveData)
3224 #else

```

```

3225 #define _ContextSaveDataAddress 0
3226 #endif // CC_ContextSave
3227 #if CC_ContextLoad
3228 #include "ContextLoad_fp.h"
3229 typedef TPM_RC (ContextLoad_Entry) (
3230     ContextLoad_In      *in,
3231     ContextLoad_Out     *out
3232 );
3233 typedef const struct {
3234     ContextLoad_Entry    *entry;
3235     UINT16               inSize;
3236     UINT16               outSize;
3237     UINT16               offsetOfTypes;
3238     BYTE                 types[4];
3239 } ContextLoad_COMMAND_DESCRIPTOR_t;
3240 ContextLoad_COMMAND_DESCRIPTOR_t _ContextLoadData = {
3241     /* entry */           &TPM2_ContextLoad,
3242     /* inSize */          (UINT16) (sizeof(ContextLoad_In)),
3243     /* outSize */         (UINT16) (sizeof(ContextLoad_Out)),
3244     /* offsetOfTypes */   offsetof(ContextLoad_COMMAND_DESCRIPTOR_t, types),
3245     /* offsets */         // No parameter offsets;
3246     /* types */           {TPMS_CONTEXT_P_UNMARSHAL,
3247                           END_OF_LIST,
3248                           TPMI_DH_CONTEXT_H_MARSHAL,
3249                           END_OF_LIST}
3250 };
3251 #define _ContextLoadDataAddress (&_ContextLoadData)
3252 #else
3253 #define _ContextLoadDataAddress 0
3254 #endif // CC_ContextLoad
3255 #if CC_FlushContext
3256 #include "FlushContext_fp.h"
3257 typedef TPM_RC (FlushContext_Entry) (
3258     FlushContext_In      *in
3259 );
3260 typedef const struct {
3261     FlushContext_Entry    *entry;
3262     UINT16               inSize;
3263     UINT16               outSize;
3264     UINT16               offsetOfTypes;
3265     BYTE                 types[3];
3266 } FlushContext_COMMAND_DESCRIPTOR_t;
3267 FlushContext_COMMAND_DESCRIPTOR_t _FlushContextData = {
3268     /* entry */           &TPM2_FlushContext,
3269     /* inSize */          (UINT16) (sizeof(FlushContext_In)),
3270     /* outSize */         0,
3271     /* offsetOfTypes */   offsetof(FlushContext_COMMAND_DESCRIPTOR_t, types),
3272     /* offsets */         // No parameter offsets;
3273     /* types */           {TPMI_DH_CONTEXT_P_UNMARSHAL,
3274                           END_OF_LIST,
3275                           END_OF_LIST}
3276 };
3277 #define _FlushContextDataAddress (&_FlushContextData)
3278 #else
3279 #define _FlushContextDataAddress 0
3280 #endif // CC_FlushContext
3281 #if CC_EvictControl
3282 #include "EvictControl_fp.h"
3283 typedef TPM_RC (EvictControl_Entry) (
3284     EvictControl_In      *in
3285 );
3286 typedef const struct {
3287     EvictControl_Entry    *entry;
3288     UINT16               inSize;
3289     UINT16               outSize;
3290     UINT16               offsetOfTypes;

```

```

3291     UINT16                paramOffsets[2];
3292     BYTE                  types[5];
3293 } EvictControl_COMMAND_DESCRIPTOR_t;
3294 EvictControl_COMMAND_DESCRIPTOR_t _EvictControlData = {
3295     /* entry */            &TPM2_EvictControl,
3296     /* inSize */           (UINT16) (sizeof(EvictControl_In)),
3297     /* outSize */          0,
3298     /* offsetOfTypes */    offsetof(EvictControl_COMMAND_DESCRIPTOR_t, types),
3299     /* offsets */          {(UINT16) (offsetof(EvictControl_In, objectHandle)),
3300                          (UINT16) (offsetof(EvictControl_In, persistentHandle))},
3301     /* types */            {TPMI_RH_PROVISION_H_UNMARSHAL,
3302                          TPMI_DH_OBJECT_H_UNMARSHAL,
3303                          TPMI_DH_PERSISTENT_P_UNMARSHAL,
3304                          END_OF_LIST,
3305                          END_OF_LIST};
3306 };
3307 #define _EvictControlDataAddress (&_EvictControlData)
3308 #else
3309 #define _EvictControlDataAddress 0
3310 #endif // CC_EvictControl
3311 #if CC_ReadClock
3312 #include "ReadClock_fp.h"
3313 typedef TPM_RC (ReadClock_Entry) (
3314     ReadClock_Out          *out
3315 );
3316 typedef const struct {
3317     ReadClock_Entry         *entry;
3318     UINT16                  inSize;
3319     UINT16                  outSize;
3320     UINT16                  offsetOfTypes;
3321     BYTE                    types[3];
3322 } ReadClock_COMMAND_DESCRIPTOR_t;
3323 ReadClock_COMMAND_DESCRIPTOR_t _ReadClockData = {
3324     /* entry */            &TPM2_ReadClock,
3325     /* inSize */           0,
3326     /* outSize */          (UINT16) (sizeof(ReadClock_Out)),
3327     /* offsetOfTypes */    offsetof(ReadClock_COMMAND_DESCRIPTOR_t, types),
3328     /* offsets */          // No parameter offsets;
3329     /* types */            {END_OF_LIST,
3330                          TPMS_TIME_INFO_P_MARSHAL,
3331                          END_OF_LIST};
3332 };
3333 #define _ReadClockDataAddress (&_ReadClockData)
3334 #else
3335 #define _ReadClockDataAddress 0
3336 #endif // CC_ReadClock
3337 #if CC_ClockSet
3338 #include "ClockSet_fp.h"
3339 typedef TPM_RC (ClockSet_Entry) (
3340     ClockSet_In            *in
3341 );
3342 typedef const struct {
3343     ClockSet_Entry         *entry;
3344     UINT16                  inSize;
3345     UINT16                  outSize;
3346     UINT16                  offsetOfTypes;
3347     UINT16                  paramOffsets[1];
3348     BYTE                    types[4];
3349 } ClockSet_COMMAND_DESCRIPTOR_t;
3350 ClockSet_COMMAND_DESCRIPTOR_t _ClockSetData = {
3351     /* entry */            &TPM2_ClockSet,
3352     /* inSize */           (UINT16) (sizeof(ClockSet_In)),
3353     /* outSize */          0,
3354     /* offsetOfTypes */    offsetof(ClockSet_COMMAND_DESCRIPTOR_t, types),
3355     /* offsets */          {(UINT16) (offsetof(ClockSet_In, newTime))},
3356     /* types */            {TPMI_RH_PROVISION_H_UNMARSHAL,

```

```

3357             UINT64_P_UNMARSHAL,
3358             END_OF_LIST,
3359             END_OF_LIST}
3360 };
3361 #define _ClockSetDataAddress (&_ClockSetData)
3362 #else
3363 #define _ClockSetDataAddress 0
3364 #endif // CC_ClockSet
3365 #if CC_ClockRateAdjust
3366 #include "ClockRateAdjust_fp.h"
3367 typedef TPM_RC (ClockRateAdjust_Entry) (
3368     ClockRateAdjust_In      *in
3369 );
3370 typedef const struct {
3371     ClockRateAdjust_Entry    *entry;
3372     UINT16                   inSize;
3373     UINT16                   outSize;
3374     UINT16                   offsetOfTypes;
3375     UINT16                   paramOffsets[1];
3376     BYTE                     types[4];
3377 } ClockRateAdjust_COMMAND_DESCRIPTOR_t;
3378 ClockRateAdjust_COMMAND_DESCRIPTOR_t _ClockRateAdjustData = {
3379     /* entry */           &TPM2_ClockRateAdjust,
3380     /* inSize */          (UINT16) (sizeof(ClockRateAdjust_In)),
3381     /* outSize */         0,
3382     /* offsetOfTypes */   offsetof(ClockRateAdjust_COMMAND_DESCRIPTOR_t, types),
3383     /* offsets */         {(UINT16) (offsetof(ClockRateAdjust_In, rateAdjust))},
3384     /* types */           {TPMI_RH_PROVISION_H_UNMARSHAL,
3385                           TPM_CLOCK_ADJUST_P_UNMARSHAL,
3386                           END_OF_LIST,
3387                           END_OF_LIST}
3388 };
3389 #define _ClockRateAdjustDataAddress (&_ClockRateAdjustData)
3390 #else
3391 #define _ClockRateAdjustDataAddress 0
3392 #endif // CC_ClockRateAdjust
3393 #if CC_GetCapability
3394 #include "GetCapability_fp.h"
3395 typedef TPM_RC (GetCapability_Entry) (
3396     GetCapability_In      *in,
3397     GetCapability_Out     *out
3398 );
3399 typedef const struct {
3400     GetCapability_Entry    *entry;
3401     UINT16                   inSize;
3402     UINT16                   outSize;
3403     UINT16                   offsetOfTypes;
3404     UINT16                   paramOffsets[3];
3405     BYTE                     types[7];
3406 } GetCapability_COMMAND_DESCRIPTOR_t;
3407 GetCapability_COMMAND_DESCRIPTOR_t _GetCapabilityData = {
3408     /* entry */           &TPM2_GetCapability,
3409     /* inSize */          (UINT16) (sizeof(GetCapability_In)),
3410     /* outSize */         (UINT16) (sizeof(GetCapability_Out)),
3411     /* offsetOfTypes */   offsetof(GetCapability_COMMAND_DESCRIPTOR_t, types),
3412     /* offsets */         {(UINT16) (offsetof(GetCapability_In, property)),
3413                           (UINT16) (offsetof(GetCapability_In, propertyCount)),
3414                           (UINT16) (offsetof(GetCapability_Out, capabilityData))},
3415     /* types */           {TPM_CAP_P_UNMARSHAL,
3416                           UINT32_P_UNMARSHAL,
3417                           UINT32_P_UNMARSHAL,
3418                           END_OF_LIST,
3419                           TPMI_YES_NO_P_MARSHAL,
3420                           TPMS_CAPABILITY_DATA_P_MARSHAL,
3421                           END_OF_LIST}
3422 };

```

```

3423 #define _GetCapabilityDataAddress (&_GetCapabilityData)
3424 #else
3425 #define _GetCapabilityDataAddress 0
3426 #endif // CC_GetCapability
3427 #if CC_TestParms
3428 #include "TestParms_fp.h"
3429 typedef TPM_RC (TestParms_Entry) (
3430     TestParms_In          *in
3431 );
3432 typedef const struct {
3433     TestParms_Entry        *entry;
3434     UINT16                 inSize;
3435     UINT16                 outSize;
3436     UINT16                 offsetOfTypes;
3437     BYTE                   types[3];
3438 } TestParms_COMMAND_DESCRIPTOR_t;
3439 TestParms_COMMAND_DESCRIPTOR_t _TestParmsData = {
3440     /* entry          */ &TPM2_TestParms,
3441     /* inSize         */ (UINT16) (sizeof(TestParms_In)),
3442     /* outSize        */ 0,
3443     /* offsetOfTypes  */ offsetof(TestParms_COMMAND_DESCRIPTOR_t, types),
3444     /* offsets        */ // No parameter offsets;
3445     /* types          */ {TPMT_PUBLIC_PARMS_P_UNMARSHAL,
3446                          END_OF_LIST,
3447                          END_OF_LIST}
3448 };
3449 #define _TestParmsDataAddress (&_TestParmsData)
3450 #else
3451 #define _TestParmsDataAddress 0
3452 #endif // CC_TestParms
3453 #if CC_NV_DefineSpace
3454 #include "NV_DefineSpace_fp.h"
3455 typedef TPM_RC (NV_DefineSpace_Entry) (
3456     NV_DefineSpace_In      *in
3457 );
3458 typedef const struct {
3459     NV_DefineSpace_Entry    *entry;
3460     UINT16                 inSize;
3461     UINT16                 outSize;
3462     UINT16                 offsetOfTypes;
3463     UINT16                 paramOffsets[2];
3464     BYTE                   types[5];
3465 } NV_DefineSpace_COMMAND_DESCRIPTOR_t;
3466 NV_DefineSpace_COMMAND_DESCRIPTOR_t _NV_DefineSpaceData = {
3467     /* entry          */ &TPM2_NV_DefineSpace,
3468     /* inSize         */ (UINT16) (sizeof(NV_DefineSpace_In)),
3469     /* outSize        */ 0,
3470     /* offsetOfTypes  */ offsetof(NV_DefineSpace_COMMAND_DESCRIPTOR_t, types),
3471     /* offsets        */ { (UINT16) (offsetof(NV_DefineSpace_In, auth)),
3472                          (UINT16) (offsetof(NV_DefineSpace_In, publicInfo)) },
3473     /* types          */ {TPMI_RH_PROVISION_H_UNMARSHAL,
3474                          TPM2B_AUTH_P_UNMARSHAL,
3475                          TPM2B_NV_PUBLIC_P_UNMARSHAL,
3476                          END_OF_LIST,
3477                          END_OF_LIST}
3478 };
3479 #define _NV_DefineSpaceDataAddress (&_NV_DefineSpaceData)
3480 #else
3481 #define _NV_DefineSpaceDataAddress 0
3482 #endif // CC_NV_DefineSpace
3483 #if CC_NV_UndefineSpace
3484 #include "NV_UndefineSpace_fp.h"
3485 typedef TPM_RC (NV_UndefineSpace_Entry) (
3486     NV_UndefineSpace_In     *in
3487 );
3488 typedef const struct {

```

```

3489     NV_UndefineSpace_Entry  *entry;
3490     UINT16                  inSize;
3491     UINT16                  outSize;
3492     UINT16                  offsetOfTypes;
3493     UINT16                  paramOffsets[1];
3494     BYTE                    types[4];
3495 } NV_UndefineSpace_COMMAND_DESCRIPTOR_t;
3496 NV_UndefineSpace_COMMAND_DESCRIPTOR_t NV_UndefineSpaceData = {
3497     /* entry */ &TPM2_NV_UndefineSpace,
3498     /* inSize */ (UINT16)(sizeof(NV_UndefineSpace_In)),
3499     /* outSize */ 0,
3500     /* offsetOfTypes */ offsetof(NV_UndefineSpace_COMMAND_DESCRIPTOR_t, types),
3501     /* offsets */ {(UINT16)(offsetof(NV_UndefineSpace_In, nvIndex))},
3502     /* types */ {TPMI_RH_PROVISION_H_UNMARSHAL,
3503                 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3504                 END_OF_LIST,
3505                 END_OF_LIST}
3506 };
3507 #define NV_UndefineSpaceDataAddress (&NV_UndefineSpaceData)
3508 #else
3509 #define NV_UndefineSpaceDataAddress 0
3510 #endif // CC_NV_UndefineSpace
3511 #if CC_NV_UndefineSpaceSpecial
3512 #include "NV_UndefineSpaceSpecial_fp.h"
3513 typedef TPM_RC (NV_UndefineSpaceSpecial_Entry)(
3514     NV_UndefineSpaceSpecial_In *in
3515 );
3516 typedef const struct {
3517     NV_UndefineSpaceSpecial_Entry *entry;
3518     UINT16 inSize;
3519     UINT16 outSize;
3520     UINT16 offsetOfTypes;
3521     UINT16 paramOffsets[1];
3522     BYTE types[4];
3523 } NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t;
3524 NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t NV_UndefineSpaceSpecialData = {
3525     /* entry */ &TPM2_NV_UndefineSpaceSpecial,
3526     /* inSize */ (UINT16)(sizeof(NV_UndefineSpaceSpecial_In)),
3527     /* outSize */ 0,
3528     /* offsetOfTypes */
3529     offsetof(NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t, types),
3530     /* offsets */ {(UINT16)(offsetof(NV_UndefineSpaceSpecial_In,
3531 platform))},
3532     /* types */ {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3533                 TPMI_RH_PLATFORM_H_UNMARSHAL,
3534                 END_OF_LIST,
3535                 END_OF_LIST}
3536 };
3537 #define NV_UndefineSpaceSpecialDataAddress (&NV_UndefineSpaceSpecialData)
3538 #else
3539 #define NV_UndefineSpaceSpecialDataAddress 0
3540 #endif // CC_NV_UndefineSpaceSpecial
3541 #if CC_NV_ReadPublic
3542 #include "NV_ReadPublic_fp.h"
3543 typedef TPM_RC (NV_ReadPublic_Entry)(
3544     NV_ReadPublic_In *in,
3545     NV_ReadPublic_Out *out
3546 );
3547 typedef const struct {
3548     NV_ReadPublic_Entry *entry;
3549     UINT16 inSize;
3550     UINT16 outSize;
3551     UINT16 offsetOfTypes;
3552     UINT16 paramOffsets[1];
3553     BYTE types[5];
3554 } NV_ReadPublic_COMMAND_DESCRIPTOR_t;

```



```

3553 NV_ReadPublic_COMMAND_DESCRIPTOR t_NV_ReadPublicData = {
3554     /* entry */ &TPM2_NV_ReadPublic,
3555     /* inSize */ (UINT16) (sizeof(NV_ReadPublic_In)),
3556     /* outSize */ (UINT16) (sizeof(NV_ReadPublic_Out)),
3557     /* offsetOfTypes */ offsetof(NV_ReadPublic_COMMAND_DESCRIPTOR_t, types),
3558     /* offsets */ { (UINT16) (offsetof(NV_ReadPublic_Out, nvName))},
3559     /* types */ {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3560                  END_OF_LIST,
3561                  TPM2B_NV_PUBLIC_P_MARSHAL,
3562                  TPM2B_NAME_P_MARSHAL,
3563                  END_OF_LIST}
3564 };
3565 #define _NV_ReadPublicDataAddress (&_NV_ReadPublicData)
3566 #else
3567 #define _NV_ReadPublicDataAddress 0
3568 #endif // CC_NV_ReadPublic
3569 #if CC_NV_Write
3570 #include "NV_Write_fp.h"
3571 typedef TPM_RC (NV_Write_Entry) (
3572     NV_Write_In *in
3573 );
3574 typedef const struct {
3575     NV_Write_Entry *entry;
3576     UINT16 inSize;
3577     UINT16 outSize;
3578     UINT16 offsetOfTypes;
3579     UINT16 paramOffsets[3];
3580     BYTE types[6];
3581 } NV_Write_COMMAND_DESCRIPTOR_t;
3582 NV_Write_COMMAND_DESCRIPTOR_t _NV_WriteData = {
3583     /* entry */ &TPM2_NV_Write,
3584     /* inSize */ (UINT16) (sizeof(NV_Write_In)),
3585     /* outSize */ 0,
3586     /* offsetOfTypes */ offsetof(NV_Write_COMMAND_DESCRIPTOR_t, types),
3587     /* offsets */ { (UINT16) (offsetof(NV_Write_In, nvIndex)),
3588                   (UINT16) (offsetof(NV_Write_In, data)),
3589                   (UINT16) (offsetof(NV_Write_In, offset))},
3590     /* types */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3591                  TPMI_RH_NV_INDEX_H_UNMARSHAL,
3592                  TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3593                  UINT16_P_UNMARSHAL,
3594                  END_OF_LIST,
3595                  END_OF_LIST}
3596 };
3597 #define _NV_WriteDataAddress (&_NV_WriteData)
3598 #else
3599 #define _NV_WriteDataAddress 0
3600 #endif // CC_NV_Write
3601 #if CC_NV_Increment
3602 #include "NV_Increment_fp.h"
3603 typedef TPM_RC (NV_Increment_Entry) (
3604     NV_Increment_In *in
3605 );
3606 typedef const struct {
3607     NV_Increment_Entry *entry;
3608     UINT16 inSize;
3609     UINT16 outSize;
3610     UINT16 offsetOfTypes;
3611     UINT16 paramOffsets[1];
3612     BYTE types[4];
3613 } NV_Increment_COMMAND_DESCRIPTOR_t;
3614 NV_Increment_COMMAND_DESCRIPTOR_t _NV_IncrementData = {
3615     /* entry */ &TPM2_NV_Increment,
3616     /* inSize */ (UINT16) (sizeof(NV_Increment_In)),
3617     /* outSize */ 0,
3618     /* offsetOfTypes */ offsetof(NV_Increment_COMMAND_DESCRIPTOR_t, types),

```



```

3619     /* offsets */      {(UINT16) (offsetof(NV_Increment_In, nvIndex))},
3620     /* types */        {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3621                        TPMI_RH_NV_INDEX_H_UNMARSHAL,
3622                        END_OF_LIST,
3623                        END_OF_LIST}
3624 };
3625 #define _NV_IncrementDataAddress (&_NV_IncrementData)
3626 #else
3627 #define _NV_IncrementDataAddress 0
3628 #endif // CC_NV_Increment
3629 #if CC_NV_Extend
3630 #include "NV_Extend_fp.h"
3631 typedef TPM_RC (NV_Extend_Entry) (
3632     NV_Extend_In          *in
3633 );
3634 typedef const struct {
3635     NV_Extend_Entry        *entry;
3636     UINT16                 inSize;
3637     UINT16                 outSize;
3638     UINT16                 offsetOfTypes;
3639     UINT16                 paramOffsets[2];
3640     BYTE                   types[5];
3641 } NV_Extend_COMMAND_DESCRIPTOR_t;
3642 NV_Extend_COMMAND_DESCRIPTOR_t _NV_ExtendData = {
3643     /* entry */          &TPM2_NV_Extend,
3644     /* inSize */         (UINT16) (sizeof(NV_Extend_In)),
3645     /* outSize */        0,
3646     /* offsetOfTypes */  offsetof(NV_Extend_COMMAND_DESCRIPTOR_t, types),
3647     /* offsets */        {(UINT16) (offsetof(NV_Extend_In, nvIndex)),
3648                          (UINT16) (offsetof(NV_Extend_In, data))},
3649     /* types */          {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3650                          TPMI_RH_NV_INDEX_H_UNMARSHAL,
3651                          TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3652                          END_OF_LIST,
3653                          END_OF_LIST}
3654 };
3655 #define _NV_ExtendDataAddress (&_NV_ExtendData)
3656 #else
3657 #define _NV_ExtendDataAddress 0
3658 #endif // CC_NV_Extend
3659 #if CC_NV_SetBits
3660 #include "NV_SetBits_fp.h"
3661 typedef TPM_RC (NV_SetBits_Entry) (
3662     NV_SetBits_In          *in
3663 );
3664 typedef const struct {
3665     NV_SetBits_Entry        *entry;
3666     UINT16                 inSize;
3667     UINT16                 outSize;
3668     UINT16                 offsetOfTypes;
3669     UINT16                 paramOffsets[2];
3670     BYTE                   types[5];
3671 } NV_SetBits_COMMAND_DESCRIPTOR_t;
3672 NV_SetBits_COMMAND_DESCRIPTOR_t _NV_SetBitsData = {
3673     /* entry */          &TPM2_NV_SetBits,
3674     /* inSize */         (UINT16) (sizeof(NV_SetBits_In)),
3675     /* outSize */        0,
3676     /* offsetOfTypes */  offsetof(NV_SetBits_COMMAND_DESCRIPTOR_t, types),
3677     /* offsets */        {(UINT16) (offsetof(NV_SetBits_In, nvIndex)),
3678                          (UINT16) (offsetof(NV_SetBits_In, bits))},
3679     /* types */          {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3680                          TPMI_RH_NV_INDEX_H_UNMARSHAL,
3681                          UINT64_P_UNMARSHAL,
3682                          END_OF_LIST,
3683                          END_OF_LIST}
3684 };

```

```

3685 #define _NV_SetBitsDataAddress (&_NV_SetBitsData)
3686 #else
3687 #define _NV_SetBitsDataAddress 0
3688 #endif // CC_NV_SetBits
3689 #if CC_NV_WriteLock
3690 #include "NV_WriteLock_fp.h"
3691 typedef TPM_RC (NV_WriteLock_Entry) (
3692     NV_WriteLock_In          *in
3693 );
3694 typedef const struct {
3695     NV_WriteLock_Entry      *entry;
3696     UINT16                  inSize;
3697     UINT16                  outSize;
3698     UINT16                  offsetOfTypes;
3699     UINT16                  paramOffsets[1];
3700     BYTE                    types[4];
3701 } NV_WriteLock_COMMAND_DESCRIPTOR_t;
3702 NV_WriteLock_COMMAND_DESCRIPTOR_t _NV_WriteLockData = {
3703     /* entry */          &TPM2_NV_WriteLock,
3704     /* inSize */         (UINT16) (sizeof(NV_WriteLock_In)),
3705     /* outSize */        0,
3706     /* offsetOfTypes */  offsetof(NV_WriteLock_COMMAND_DESCRIPTOR_t, types),
3707     /* offsets */         {(UINT16) (offsetof(NV_WriteLock_In, nvIndex))},
3708     /* types */          {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3709                          TPMI_RH_NV_INDEX_H_UNMARSHAL,
3710                          END_OF_LIST,
3711                          END_OF_LIST}
3712 };
3713 #define _NV_WriteLockDataAddress (&_NV_WriteLockData)
3714 #else
3715 #define _NV_WriteLockDataAddress 0
3716 #endif // CC_NV_WriteLock
3717 #if CC_NV_GlobalWriteLock
3718 #include "NV_GlobalWriteLock_fp.h"
3719 typedef TPM_RC (NV_GlobalWriteLock_Entry) (
3720     NV_GlobalWriteLock_In    *in
3721 );
3722 typedef const struct {
3723     NV_GlobalWriteLock_Entry *entry;
3724     UINT16                  inSize;
3725     UINT16                  outSize;
3726     UINT16                  offsetOfTypes;
3727     BYTE                    types[3];
3728 } NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t;
3729 NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t _NV_GlobalWriteLockData = {
3730     /* entry */          &TPM2_NV_GlobalWriteLock,
3731     /* inSize */         (UINT16) (sizeof(NV_GlobalWriteLock_In)),
3732     /* outSize */        0,
3733     /* offsetOfTypes */  offsetof(NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t,
3734     types),
3735     /* offsets */         // No parameter offsets;
3736     /* types */          {TPMI_RH_PROVISION_H_UNMARSHAL,
3737                          END_OF_LIST,
3738                          END_OF_LIST}
3739 };
3740 #define _NV_GlobalWriteLockDataAddress (&_NV_GlobalWriteLockData)
3741 #else
3742 #define _NV_GlobalWriteLockDataAddress 0
3743 #endif // CC_NV_GlobalWriteLock
3744 #if CC_NV_Read
3745 #include "NV_Read_fp.h"
3746 typedef TPM_RC (NV_Read_Entry) (
3747     NV_Read_In             *in,
3748     NV_Read_Out            *out
3749 );
3750 typedef const struct {

```

```

3750     NV_Read_Entry          *entry;
3751     UINT16                  inSize;
3752     UINT16                  outSize;
3753     UINT16                  offsetOfTypes;
3754     UINT16                  paramOffsets[3];
3755     BYTE                     types[7];
3756 } NV_Read_COMMAND_DESCRIPTOR_t;
3757 NV_Read_COMMAND_DESCRIPTOR_t NV_ReadData = {
3758     /* entry */              &TPM2_NV_Read,
3759     /* inSize */             (UINT16) (sizeof(NV_Read_In)),
3760     /* outSize */            (UINT16) (sizeof(NV_Read_Out)),
3761     /* offsetOfTypes */      offsetof(NV_Read_COMMAND_DESCRIPTOR_t, types),
3762     /* offsets */            { (UINT16) (offsetof(NV_Read_In, nvIndex)),
3763                             (UINT16) (offsetof(NV_Read_In, size)),
3764                             (UINT16) (offsetof(NV_Read_In, offset)) },
3765     /* types */              { TPMI_RH_NV_AUTH_H_UNMARSHAL,
3766                             TPMI_RH_NV_INDEX_H_UNMARSHAL,
3767                             UINT16_P_UNMARSHAL,
3768                             UINT16_P_UNMARSHAL,
3769                             END_OF_LIST,
3770                             TPM2B_MAX_NV_BUFFER_P_MARSHAL,
3771                             END_OF_LIST }
3772 };
3773 #define _NV_ReadDataAddress (&NV_ReadData)
3774 #else
3775 #define _NV_ReadDataAddress 0
3776 #endif // CC_NV_Read
3777 #if CC_NV_ReadLock
3778 #include "NV_ReadLock_fp.h"
3779 typedef TPM_RC (NV_ReadLock_Entry) (
3780     NV_ReadLock_In          *in
3781 );
3782 typedef const struct {
3783     NV_ReadLock_Entry        *entry;
3784     UINT16                   inSize;
3785     UINT16                   outSize;
3786     UINT16                   offsetOfTypes;
3787     UINT16                   paramOffsets[1];
3788     BYTE                     types[4];
3789 } NV_ReadLock_COMMAND_DESCRIPTOR_t;
3790 NV_ReadLock_COMMAND_DESCRIPTOR_t NV_ReadLockData = {
3791     /* entry */              &TPM2_NV_ReadLock,
3792     /* inSize */             (UINT16) (sizeof(NV_ReadLock_In)),
3793     /* outSize */            0,
3794     /* offsetOfTypes */      offsetof(NV_ReadLock_COMMAND_DESCRIPTOR_t, types),
3795     /* offsets */            { (UINT16) (offsetof(NV_ReadLock_In, nvIndex)) },
3796     /* types */              { TPMI_RH_NV_AUTH_H_UNMARSHAL,
3797                             TPMI_RH_NV_INDEX_H_UNMARSHAL,
3798                             END_OF_LIST,
3799                             END_OF_LIST }
3800 };
3801 #define _NV_ReadLockDataAddress (&NV_ReadLockData)
3802 #else
3803 #define _NV_ReadLockDataAddress 0
3804 #endif // CC_NV_ReadLock
3805 #if CC_NV_ChangeAuth
3806 #include "NV_ChangeAuth_fp.h"
3807 typedef TPM_RC (NV_ChangeAuth_Entry) (
3808     NV_ChangeAuth_In         *in
3809 );
3810 typedef const struct {
3811     NV_ChangeAuth_Entry      *entry;
3812     UINT16                   inSize;
3813     UINT16                   outSize;
3814     UINT16                   offsetOfTypes;
3815     UINT16                   paramOffsets[1];

```

```

3816     BYTE                types[4];
3817 } NV_ChangeAuth_COMMAND_DESCRIPTOR_t;
3818 NV_ChangeAuth_COMMAND_DESCRIPTOR_t NV_ChangeAuthData = {
3819     /* entry */          &TPM2_NV_ChangeAuth,
3820     /* inSize */         (UINT16) (sizeof(NV_ChangeAuth_In)),
3821     /* outSize */        0,
3822     /* offsetOfTypes */  offsetof(NV_ChangeAuth_COMMAND_DESCRIPTOR_t, types),
3823     /* offsets */         {(UINT16) (offsetof(NV_ChangeAuth_In, newAuth))},
3824     /* types */           {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3825                          TPM2B_AUTH_P_UNMARSHAL,
3826                          END_OF_LIST,
3827                          END_OF_LIST};
3828 };
3829 #define _NV_ChangeAuthDataAddress (&_NV_ChangeAuthData)
3830 #else
3831 #define _NV_ChangeAuthDataAddress 0
3832 #endif // CC_NV_ChangeAuth
3833 #if CC_NV_Certify
3834 #include "NV_Certify_fp.h"
3835 typedef TPM_RC (NV_Certify_Entry) (
3836     NV_Certify_In          *in,
3837     NV_Certify_Out         *out
3838 );
3839 typedef const struct {
3840     NV_Certify_Entry        *entry;
3841     UINT16                  inSize;
3842     UINT16                  outSize;
3843     UINT16                  offsetOfTypes;
3844     UINT16                  paramOffsets[7];
3845     BYTE                    types[11];
3846 } NV_Certify_COMMAND_DESCRIPTOR_t;
3847 NV_Certify_COMMAND_DESCRIPTOR_t NV_CertifyData = {
3848     /* entry */          &TPM2_NV_Certify,
3849     /* inSize */         (UINT16) (sizeof(NV_Certify_In)),
3850     /* outSize */        (UINT16) (sizeof(NV_Certify_Out)),
3851     /* offsetOfTypes */  offsetof(NV_Certify_COMMAND_DESCRIPTOR_t, types),
3852     /* offsets */         {(UINT16) (offsetof(NV_Certify_In, authHandle)),
3853                          (UINT16) (offsetof(NV_Certify_In, nvIndex)),
3854                          (UINT16) (offsetof(NV_Certify_In, qualifyingData)),
3855                          (UINT16) (offsetof(NV_Certify_In, inScheme)),
3856                          (UINT16) (offsetof(NV_Certify_In, size)),
3857                          (UINT16) (offsetof(NV_Certify_In, offset)),
3858                          (UINT16) (offsetof(NV_Certify_Out, signature))},
3859     /* types */           {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
3860                          TPMI_RH_NV_AUTH_H_UNMARSHAL,
3861                          TPMI_RH_NV_INDEX_H_UNMARSHAL,
3862                          TPM2B_DATA_P_UNMARSHAL,
3863                          TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
3864                          UINT16_P_UNMARSHAL,
3865                          UINT16_P_UNMARSHAL,
3866                          END_OF_LIST,
3867                          TPM2B_ATTEST_P_MARSHAL,
3868                          TPMT_SIGNATURE_P_MARSHAL,
3869                          END_OF_LIST};
3870 };
3871 #define _NV_CertifyDataAddress (&_NV_CertifyData)
3872 #else
3873 #define _NV_CertifyDataAddress 0
3874 #endif // CC_NV_Certify
3875 #if CC_AC_GetCapability
3876 #include "AC_GetCapability_fp.h"
3877 typedef TPM_RC (AC_GetCapability_Entry) (
3878     AC_GetCapability_In     *in,
3879     AC_GetCapability_Out    *out
3880 );
3881 typedef const struct {

```

```

3882     AC_GetCapability_Entry  *entry;
3883     UINT16                  inSize;
3884     UINT16                  outSize;
3885     UINT16                  offsetOfTypes;
3886     UINT16                  paramOffsets[3];
3887     BYTE                    types[7];
3888 } AC_GetCapability_COMMAND_DESCRIPTOR_t;
3889 AC_GetCapability_COMMAND_DESCRIPTOR_t _AC_GetCapabilityData = {
3890     /* entry */ &TPM2_AC_GetCapability,
3891     /* inSize */ (UINT16)(sizeof(AC_GetCapability_In)),
3892     /* outSize */ (UINT16)(sizeof(AC_GetCapability_Out)),
3893     /* offsetOfTypes */ offsetof(AC_GetCapability_COMMAND_DESCRIPTOR_t, types),
3894     /* offsets */ { (UINT16)(offsetof(AC_GetCapability_In, capability)),
3895                   (UINT16)(offsetof(AC_GetCapability_In, count)),
3896                   (UINT16)(offsetof(AC_GetCapability_Out,
3897 capabilitiesData)))},
3897     /* types */ {TPMI_RH_AC_H_UNMARSHAL,
3898                 TPM_AT_P_UNMARSHAL,
3899                 UINT32_P_UNMARSHAL,
3900                 END_OF_LIST,
3901                 TPMI_YES_NO_P_MARSHAL,
3902                 TPML_AC_CAPABILITIES_P_MARSHAL,
3903                 END_OF_LIST}
3904 };
3905 #define _AC_GetCapabilityDataAddress (&_AC_GetCapabilityData)
3906 #else
3907 #define _AC_GetCapabilityDataAddress 0
3908 #endif // CC_AC_GetCapability
3909 #if CC_AC_Send
3910 #include "AC_Send_fp.h"
3911 typedef TPM_RC (AC_Send_Entry) (
3912     AC_Send_In             *in,
3913     AC_Send_Out            *out
3914 );
3915 typedef const struct {
3916     AC_Send_Entry          *entry;
3917     UINT16                  inSize;
3918     UINT16                  outSize;
3919     UINT16                  offsetOfTypes;
3920     UINT16                  paramOffsets[3];
3921     BYTE                    types[7];
3922 } AC_Send_COMMAND_DESCRIPTOR_t;
3923 AC_Send_COMMAND_DESCRIPTOR_t _AC_SendData = {
3924     /* entry */ &TPM2_AC_Send,
3925     /* inSize */ (UINT16)(sizeof(AC_Send_In)),
3926     /* outSize */ (UINT16)(sizeof(AC_Send_Out)),
3927     /* offsetOfTypes */ offsetof(AC_Send_COMMAND_DESCRIPTOR_t, types),
3928     /* offsets */ { (UINT16)(offsetof(AC_Send_In, authHandle)),
3929                   (UINT16)(offsetof(AC_Send_In, ac)),
3930                   (UINT16)(offsetof(AC_Send_In, acDataIn))},
3931     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
3932                 TPMI_RH_NV_AUTH_H_UNMARSHAL,
3933                 TPMI_RH_AC_H_UNMARSHAL,
3934                 TPM2B_MAX_BUFFER_P_UNMARSHAL,
3935                 END_OF_LIST,
3936                 TPMS_AC_OUTPUT_P_MARSHAL,
3937                 END_OF_LIST}
3938 };
3939 #define _AC_SendDataAddress (&_AC_SendData)
3940 #else
3941 #define _AC_SendDataAddress 0
3942 #endif // CC_AC_Send
3943 #if CC_Policy_AC_SendSelect
3944 #include "Policy_AC_SendSelect_fp.h"
3945 typedef TPM_RC (Policy_AC_SendSelect_Entry) (
3946     Policy_AC_SendSelect_In *in

```



```

3947 );
3948 typedef const struct {
3949     Policy_AC_SendSelect_Entry *entry;
3950     UINT16 inSize;
3951     UINT16 outSize;
3952     UINT16 offsetOfTypes;
3953     UINT16 paramOffsets[4];
3954     BYTE types[7];
3955 } Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t;
3956 Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t _Policy_AC_SendSelectData = {
3957     /* entry */ &TPM2_Policy_AC_SendSelect,
3958     /* inSize */ (UINT16) (sizeof(Policy_AC_SendSelect_In)),
3959     /* outSize */ 0,
3960     /* offsetOfTypes */ offsetof(Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t,
types),
3961     /* offsets */ { (UINT16) (offsetof(Policy_AC_SendSelect_In,
objectName)),
3962                     (UINT16) (offsetof(Policy_AC_SendSelect_In,
authHandleName)),
3963                     (UINT16) (offsetof(Policy_AC_SendSelect_In, acName)),
3964                     (UINT16) (offsetof(Policy_AC_SendSelect_In,
includeObject))},
3965     /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
3966                 TPM2B_NAME_P_UNMARSHAL,
3967                 TPM2B_NAME_P_UNMARSHAL,
3968                 TPM2B_NAME_P_UNMARSHAL,
3969                 TPMI_YES_NO_P_UNMARSHAL,
3970                 END_OF_LIST,
3971                 END_OF_LIST}
3972 };
3973 #define _Policy_AC_SendSelectDataAddress (&_Policy_AC_SendSelectData)
3974 #else
3975 #define _Policy_AC_SendSelectDataAddress 0
3976 #endif // CC_Policy_AC_SendSelect
3977 #if CC_Vendor_TCG_Test
3978 #include "Vendor_TCG_Test_fp.h"
3979 typedef TPM_RC (Vendor_TCG_Test_Entry) (
3980     Vendor_TCG_Test_In *in,
3981     Vendor_TCG_Test_Out *out
3982 );
3983 typedef const struct {
3984     Vendor_TCG_Test_Entry *entry;
3985     UINT16 inSize;
3986     UINT16 outSize;
3987     UINT16 offsetOfTypes;
3988     BYTE types[4];
3989 } Vendor_TCG_Test_COMMAND_DESCRIPTOR_t;
3990 Vendor_TCG_Test_COMMAND_DESCRIPTOR_t _Vendor_TCG_TestData = {
3991     /* entry */ &TPM2_Vendor_TCG_Test,
3992     /* inSize */ (UINT16) (sizeof(Vendor_TCG_Test_In)),
3993     /* outSize */ (UINT16) (sizeof(Vendor_TCG_Test_Out)),
3994     /* offsetOfTypes */ offsetof(Vendor_TCG_Test_COMMAND_DESCRIPTOR_t, types),
3995     /* offsets */ // No parameter offsets;
3996     /* types */ {TPM2B_DATA_P_UNMARSHAL,
3997                 END_OF_LIST,
3998                 TPM2B_DATA_P_MARSHAL,
3999                 END_OF_LIST}
4000 };
4001 #define _Vendor_TCG_TestDataAddress (&_Vendor_TCG_TestData)
4002 #else
4003 #define _Vendor_TCG_TestDataAddress 0
4004 #endif // CC_Vendor_TCG_Test
4005 COMMAND_DESCRIPTOR_t *s_CommandDataArray[] = {
4006 #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
4007     (COMMAND_DESCRIPTOR_t *) NV_UndefineSpaceSpecialDataAddress,
4008 #endif // CC_NV_UndefineSpaceSpecial

```

```

4009 #if (PAD_LIST || CC_EvictControl)
4010     (COMMAND_DESCRIPTOR_t *)_EvictControlDataAddress,
4011 #endif // CC_EvictControl
4012 #if (PAD_LIST || CC_HierarchyControl)
4013     (COMMAND_DESCRIPTOR_t *)_HierarchyControlDataAddress,
4014 #endif // CC_HierarchyControl
4015 #if (PAD_LIST || CC_NV_UndefineSpace)
4016     (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceDataAddress,
4017 #endif // CC_NV_UndefineSpace
4018 #if (PAD_LIST)
4019     (COMMAND_DESCRIPTOR_t *)0,
4020 #endif //
4021 #if (PAD_LIST || CC_ChangeEPS)
4022     (COMMAND_DESCRIPTOR_t *)_ChangeEPSDataAddress,
4023 #endif // CC_ChangeEPS
4024 #if (PAD_LIST || CC_ChangePPS)
4025     (COMMAND_DESCRIPTOR_t *)_ChangePPSDataAddress,
4026 #endif // CC_ChangePPS
4027 #if (PAD_LIST || CC_Clear)
4028     (COMMAND_DESCRIPTOR_t *)_ClearDataAddress,
4029 #endif // CC_Clear
4030 #if (PAD_LIST || CC_ClearControl)
4031     (COMMAND_DESCRIPTOR_t *)_ClearControlDataAddress,
4032 #endif // CC_ClearControl
4033 #if (PAD_LIST || CC_ClockSet)
4034     (COMMAND_DESCRIPTOR_t *)_ClockSetDataAddress,
4035 #endif // CC_ClockSet
4036 #if (PAD_LIST || CC_HierarchyChangeAuth)
4037     (COMMAND_DESCRIPTOR_t *)_HierarchyChangeAuthDataAddress,
4038 #endif // CC_HierarchyChangeAuth
4039 #if (PAD_LIST || CC_NV_DefineSpace)
4040     (COMMAND_DESCRIPTOR_t *)_NV_DefineSpaceDataAddress,
4041 #endif // CC_NV_DefineSpace
4042 #if (PAD_LIST || CC_PCR_Allocate)
4043     (COMMAND_DESCRIPTOR_t *)_PCR_AllocateDataAddress,
4044 #endif // CC_PCR_Allocate
4045 #if (PAD_LIST || CC_PCR_SetAuthPolicy)
4046     (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthPolicyDataAddress,
4047 #endif // CC_PCR_SetAuthPolicy
4048 #if (PAD_LIST || CC_PP_Commands)
4049     (COMMAND_DESCRIPTOR_t *)_PP_CommandsDataAddress,
4050 #endif // CC_PP_Commands
4051 #if (PAD_LIST || CC_SetPrimaryPolicy)
4052     (COMMAND_DESCRIPTOR_t *)_SetPrimaryPolicyDataAddress,
4053 #endif // CC_SetPrimaryPolicy
4054 #if (PAD_LIST || CC_FieldUpgradeStart)
4055     (COMMAND_DESCRIPTOR_t *)_FieldUpgradeStartDataAddress,
4056 #endif // CC_FieldUpgradeStart
4057 #if (PAD_LIST || CC_ClockRateAdjust)
4058     (COMMAND_DESCRIPTOR_t *)_ClockRateAdjustDataAddress,
4059 #endif // CC_ClockRateAdjust
4060 #if (PAD_LIST || CC_CreatePrimary)
4061     (COMMAND_DESCRIPTOR_t *)_CreatePrimaryDataAddress,
4062 #endif // CC_CreatePrimary
4063 #if (PAD_LIST || CC_NV_GlobalWriteLock)
4064     (COMMAND_DESCRIPTOR_t *)_NV_GlobalWriteLockDataAddress,
4065 #endif // CC_NV_GlobalWriteLock
4066 #if (PAD_LIST || CC_GetCommandAuditDigest)
4067     (COMMAND_DESCRIPTOR_t *)_GetCommandAuditDigestDataAddress,
4068 #endif // CC_GetCommandAuditDigest
4069 #if (PAD_LIST || CC_NV_Increment)
4070     (COMMAND_DESCRIPTOR_t *)_NV_IncrementDataAddress,
4071 #endif // CC_NV_Increment
4072 #if (PAD_LIST || CC_NV_SetBits)
4073     (COMMAND_DESCRIPTOR_t *)_NV_SetBitsDataAddress,
4074 #endif // CC_NV_SetBits

```



```

4075 #if (PAD_LIST || CC_NV_Extend)
4076     (COMMAND_DESCRIPTOR_t *)_NV_ExtendDataAddress,
4077 #endif // CC_NV_Extend
4078 #if (PAD_LIST || CC_NV_Write)
4079     (COMMAND_DESCRIPTOR_t *)_NV_WriteDataAddress,
4080 #endif // CC_NV_Write
4081 #if (PAD_LIST || CC_NV_WriteLock)
4082     (COMMAND_DESCRIPTOR_t *)_NV_WriteLockDataAddress,
4083 #endif // CC_NV_WriteLock
4084 #if (PAD_LIST || CC_DictionaryAttackLockReset)
4085     (COMMAND_DESCRIPTOR_t *)_DictionaryAttackLockResetDataAddress,
4086 #endif // CC_DictionaryAttackLockReset
4087 #if (PAD_LIST || CC_DictionaryAttackParameters)
4088     (COMMAND_DESCRIPTOR_t *)_DictionaryAttackParametersDataAddress,
4089 #endif // CC_DictionaryAttackParameters
4090 #if (PAD_LIST || CC_NV_ChangeAuth)
4091     (COMMAND_DESCRIPTOR_t *)_NV_ChangeAuthDataAddress,
4092 #endif // CC_NV_ChangeAuth
4093 #if (PAD_LIST || CC_PCR_Event)
4094     (COMMAND_DESCRIPTOR_t *)_PCR_EventDataAddress,
4095 #endif // CC_PCR_Event
4096 #if (PAD_LIST || CC_PCR_Reset)
4097     (COMMAND_DESCRIPTOR_t *)_PCR_ResetDataAddress,
4098 #endif // CC_PCR_Reset
4099 #if (PAD_LIST || CC_SequenceComplete)
4100     (COMMAND_DESCRIPTOR_t *)_SequenceCompleteDataAddress,
4101 #endif // CC_SequenceComplete
4102 #if (PAD_LIST || CC_SetAlgorithmSet)
4103     (COMMAND_DESCRIPTOR_t *)_SetAlgorithmSetDataAddress,
4104 #endif // CC_SetAlgorithmSet
4105 #if (PAD_LIST || CC_SetCommandCodeAuditStatus)
4106     (COMMAND_DESCRIPTOR_t *)_SetCommandCodeAuditStatusDataAddress,
4107 #endif // CC_SetCommandCodeAuditStatus
4108 #if (PAD_LIST || CC_FieldUpgradeData)
4109     (COMMAND_DESCRIPTOR_t *)_FieldUpgradeDataDataAddress,
4110 #endif // CC_FieldUpgradeData
4111 #if (PAD_LIST || CC_IncrementalSelfTest)
4112     (COMMAND_DESCRIPTOR_t *)_IncrementalSelfTestDataAddress,
4113 #endif // CC_IncrementalSelfTest
4114 #if (PAD_LIST || CC_SelfTest)
4115     (COMMAND_DESCRIPTOR_t *)_SelfTestDataAddress,
4116 #endif // CC_SelfTest
4117 #if (PAD_LIST || CC_Startup)
4118     (COMMAND_DESCRIPTOR_t *)_StartupDataAddress,
4119 #endif // CC_Startup
4120 #if (PAD_LIST || CC_Shutdown)
4121     (COMMAND_DESCRIPTOR_t *)_ShutdownDataAddress,
4122 #endif // CC_Shutdown
4123 #if (PAD_LIST || CC_StirRandom)
4124     (COMMAND_DESCRIPTOR_t *)_StirRandomDataAddress,
4125 #endif // CC_StirRandom
4126 #if (PAD_LIST || CC_ActivateCredential)
4127     (COMMAND_DESCRIPTOR_t *)_ActivateCredentialDataAddress,
4128 #endif // CC_ActivateCredential
4129 #if (PAD_LIST || CC_Certify)
4130     (COMMAND_DESCRIPTOR_t *)_CertifyDataAddress,
4131 #endif // CC_Certify
4132 #if (PAD_LIST || CC_PolicyNV)
4133     (COMMAND_DESCRIPTOR_t *)_PolicyNVDataAddress,
4134 #endif // CC_PolicyNV
4135 #if (PAD_LIST || CC_CertifyCreation)
4136     (COMMAND_DESCRIPTOR_t *)_CertifyCreationDataAddress,
4137 #endif // CC_CertifyCreation
4138 #if (PAD_LIST || CC_Duplicate)
4139     (COMMAND_DESCRIPTOR_t *)_DuplicateDataAddress,
4140 #endif // CC_Duplicate

```

```

4141 #if (PAD_LIST || CC_GetTime)
4142     (COMMAND_DESCRIPTOR_t *)_GetTimeDataAddress,
4143 #endif // CC_GetTime
4144 #if (PAD_LIST || CC_GetSessionAuditDigest)
4145     (COMMAND_DESCRIPTOR_t *)_GetSessionAuditDigestDataAddress,
4146 #endif // CC_GetSessionAuditDigest
4147 #if (PAD_LIST || CC_NV_Read)
4148     (COMMAND_DESCRIPTOR_t *)_NV_ReadDataAddress,
4149 #endif // CC_NV_Read
4150 #if (PAD_LIST || CC_NV_ReadLock)
4151     (COMMAND_DESCRIPTOR_t *)_NV_ReadLockDataAddress,
4152 #endif // CC_NV_ReadLock
4153 #if (PAD_LIST || CC_ObjectChangeAuth)
4154     (COMMAND_DESCRIPTOR_t *)_ObjectChangeAuthDataAddress,
4155 #endif // CC_ObjectChangeAuth
4156 #if (PAD_LIST || CC_PolicySecret)
4157     (COMMAND_DESCRIPTOR_t *)_PolicySecretDataAddress,
4158 #endif // CC_PolicySecret
4159 #if (PAD_LIST || CC_Rewrap)
4160     (COMMAND_DESCRIPTOR_t *)_RewrapDataAddress,
4161 #endif // CC_Rewrap
4162 #if (PAD_LIST || CC_Create)
4163     (COMMAND_DESCRIPTOR_t *)_CreateDataAddress,
4164 #endif // CC_Create
4165 #if (PAD_LIST || CC_ECDH_ZGen)
4166     (COMMAND_DESCRIPTOR_t *)_ECDH_ZGenDataAddress,
4167 #endif // CC_ECDH_ZGen
4168 #if (PAD_LIST || (CC_HMAC || CC_MAC))
4169     # if CC_HMAC
4170         (COMMAND_DESCRIPTOR_t *)_HMACDataAddress,
4171     # endif
4172     # if CC_MAC
4173         (COMMAND_DESCRIPTOR_t *)_MACDataAddress,
4174     # endif
4175     # if (CC_HMAC || CC_MAC) > 1
4176         # error "More than one aliased command defined"
4177     # endif
4178 #endif // CC_HMAC CC_MAC
4179 #if (PAD_LIST || CC_Import)
4180     (COMMAND_DESCRIPTOR_t *)_ImportDataAddress,
4181 #endif // CC_Import
4182 #if (PAD_LIST || CC_Load)
4183     (COMMAND_DESCRIPTOR_t *)_LoadDataAddress,
4184 #endif // CC_Load
4185 #if (PAD_LIST || CC_Quote)
4186     (COMMAND_DESCRIPTOR_t *)_QuoteDataAddress,
4187 #endif // CC_Quote
4188 #if (PAD_LIST || CC_RSA_Decrypt)
4189     (COMMAND_DESCRIPTOR_t *)_RSA_DecryptDataAddress,
4190 #endif // CC_RSA_Decrypt
4191 #if (PAD_LIST)
4192     (COMMAND_DESCRIPTOR_t *)0,
4193 #endif //
4194 #if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
4195     # if CC_HMAC_Start
4196         (COMMAND_DESCRIPTOR_t *)_HMAC_StartDataAddress,
4197     # endif
4198     # if CC_MAC_Start
4199         (COMMAND_DESCRIPTOR_t *)_MAC_StartDataAddress,
4200     # endif
4201     # if (CC_HMAC_Start || CC_MAC_Start) > 1
4202         # error "More than one aliased command defined"
4203     # endif
4204 #endif // CC_HMAC_Start CC_MAC_Start
4205 #if (PAD_LIST || CC_SequenceUpdate)
4206     (COMMAND_DESCRIPTOR_t *)_SequenceUpdateDataAddress,

```

```

4207 #endif // CC_SequenceUpdate
4208 #if (PAD_LIST || CC_Sign)
4209     (COMMAND_DESCRIPTOR_t *)_SignDataAddress,
4210 #endif // CC_Sign
4211 #if (PAD_LIST || CC_Unseal)
4212     (COMMAND_DESCRIPTOR_t *)_UnsealDataAddress,
4213 #endif // CC_Unseal
4214 #if (PAD_LIST)
4215     (COMMAND_DESCRIPTOR_t *)0,
4216 #endif //
4217 #if (PAD_LIST || CC_PolicySigned)
4218     (COMMAND_DESCRIPTOR_t *)_PolicySignedDataAddress,
4219 #endif // CC_PolicySigned
4220 #if (PAD_LIST || CC_ContextLoad)
4221     (COMMAND_DESCRIPTOR_t *)_ContextLoadDataAddress,
4222 #endif // CC_ContextLoad
4223 #if (PAD_LIST || CC_ContextSave)
4224     (COMMAND_DESCRIPTOR_t *)_ContextSaveDataAddress,
4225 #endif // CC_ContextSave
4226 #if (PAD_LIST || CC_ECDH_KeyGen)
4227     (COMMAND_DESCRIPTOR_t *)_ECDH_KeyGenDataAddress,
4228 #endif // CC_ECDH_KeyGen
4229 #if (PAD_LIST || CC_EncryptDecrypt)
4230     (COMMAND_DESCRIPTOR_t *)_EncryptDecryptDataAddress,
4231 #endif // CC_EncryptDecrypt
4232 #if (PAD_LIST || CC_FlushContext)
4233     (COMMAND_DESCRIPTOR_t *)_FlushContextDataAddress,
4234 #endif // CC_FlushContext
4235 #if (PAD_LIST)
4236     (COMMAND_DESCRIPTOR_t *)0,
4237 #endif //
4238 #if (PAD_LIST || CC_LoadExternal)
4239     (COMMAND_DESCRIPTOR_t *)_LoadExternalDataAddress,
4240 #endif // CC_LoadExternal
4241 #if (PAD_LIST || CC_MakeCredential)
4242     (COMMAND_DESCRIPTOR_t *)_MakeCredentialDataAddress,
4243 #endif // CC_MakeCredential
4244 #if (PAD_LIST || CC_NV_ReadPublic)
4245     (COMMAND_DESCRIPTOR_t *)_NV_ReadPublicDataAddress,
4246 #endif // CC_NV_ReadPublic
4247 #if (PAD_LIST || CC_PolicyAuthorize)
4248     (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeDataAddress,
4249 #endif // CC_PolicyAuthorize
4250 #if (PAD_LIST || CC_PolicyAuthValue)
4251     (COMMAND_DESCRIPTOR_t *)_PolicyAuthValueDataAddress,
4252 #endif // CC_PolicyAuthValue
4253 #if (PAD_LIST || CC_PolicyCommandCode)
4254     (COMMAND_DESCRIPTOR_t *)_PolicyCommandCodeDataAddress,
4255 #endif // CC_PolicyCommandCode
4256 #if (PAD_LIST || CC_PolicyCounterTimer)
4257     (COMMAND_DESCRIPTOR_t *)_PolicyCounterTimerDataAddress,
4258 #endif // CC_PolicyCounterTimer
4259 #if (PAD_LIST || CC_PolicyCpHash)
4260     (COMMAND_DESCRIPTOR_t *)_PolicyCpHashDataAddress,
4261 #endif // CC_PolicyCpHash
4262 #if (PAD_LIST || CC_PolicyLocality)
4263     (COMMAND_DESCRIPTOR_t *)_PolicyLocalityDataAddress,
4264 #endif // CC_PolicyLocality
4265 #if (PAD_LIST || CC_PolicyNameHash)
4266     (COMMAND_DESCRIPTOR_t *)_PolicyNameHashDataAddress,
4267 #endif // CC_PolicyNameHash
4268 #if (PAD_LIST || CC_PolicyOR)
4269     (COMMAND_DESCRIPTOR_t *)_PolicyORDataAddress,
4270 #endif // CC_PolicyOR
4271 #if (PAD_LIST || CC_PolicyTicket)
4272     (COMMAND_DESCRIPTOR_t *)_PolicyTicketDataAddress,

```

```
4273 #endif // CC_PolicyTicket
4274 #if (PAD_LIST || CC_ReadPublic)
4275     (COMMAND_DESCRIPTOR_t *)_ReadPublicDataAddress,
4276 #endif // CC_ReadPublic
4277 #if (PAD_LIST || CC_RSA_Encrypt)
4278     (COMMAND_DESCRIPTOR_t *)_RSA_EncryptDataAddress,
4279 #endif // CC_RSA_Encrypt
4280 #if (PAD_LIST)
4281     (COMMAND_DESCRIPTOR_t *)0,
4282 #endif //
4283 #if (PAD_LIST || CC_StartAuthSession)
4284     (COMMAND_DESCRIPTOR_t *)_StartAuthSessionDataAddress,
4285 #endif // CC_StartAuthSession
4286 #if (PAD_LIST || CC_VerifySignature)
4287     (COMMAND_DESCRIPTOR_t *)_VerifySignatureDataAddress,
4288 #endif // CC_VerifySignature
4289 #if (PAD_LIST || CC_ECC_Parameters)
4290     (COMMAND_DESCRIPTOR_t *)_ECC_ParametersDataAddress,
4291 #endif // CC_ECC_Parameters
4292 #if (PAD_LIST || CC_FirmwareRead)
4293     (COMMAND_DESCRIPTOR_t *)_FirmwareReadDataAddress,
4294 #endif // CC_FirmwareRead
4295 #if (PAD_LIST || CC_GetCapability)
4296     (COMMAND_DESCRIPTOR_t *)_GetCapabilityDataAddress,
4297 #endif // CC_GetCapability
4298 #if (PAD_LIST || CC_GetRandom)
4299     (COMMAND_DESCRIPTOR_t *)_GetRandomDataAddress,
4300 #endif // CC_GetRandom
4301 #if (PAD_LIST || CC_GetTestResult)
4302     (COMMAND_DESCRIPTOR_t *)_GetTestResultDataAddress,
4303 #endif // CC_GetTestResult
4304 #if (PAD_LIST || CC_Hash)
4305     (COMMAND_DESCRIPTOR_t *)_HashDataAddress,
4306 #endif // CC_Hash
4307 #if (PAD_LIST || CC_PCR_Read)
4308     (COMMAND_DESCRIPTOR_t *)_PCR_ReadDataAddress,
4309 #endif // CC_PCR_Read
4310 #if (PAD_LIST || CC_PolicyPCR)
4311     (COMMAND_DESCRIPTOR_t *)_PolicyPCRDataAddress,
4312 #endif // CC_PolicyPCR
4313 #if (PAD_LIST || CC_PolicyRestart)
4314     (COMMAND_DESCRIPTOR_t *)_PolicyRestartDataAddress,
4315 #endif // CC_PolicyRestart
4316 #if (PAD_LIST || CC_ReadClock)
4317     (COMMAND_DESCRIPTOR_t *)_ReadClockDataAddress,
4318 #endif // CC_ReadClock
4319 #if (PAD_LIST || CC_PCR_Extend)
4320     (COMMAND_DESCRIPTOR_t *)_PCR_ExtendDataAddress,
4321 #endif // CC_PCR_Extend
4322 #if (PAD_LIST || CC_PCR_SetAuthValue)
4323     (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthValueDataAddress,
4324 #endif // CC_PCR_SetAuthValue
4325 #if (PAD_LIST || CC_NV_Certify)
4326     (COMMAND_DESCRIPTOR_t *)_NV_CertifyDataAddress,
4327 #endif // CC_NV_Certify
4328 #if (PAD_LIST || CC_EventSequenceComplete)
4329     (COMMAND_DESCRIPTOR_t *)_EventSequenceCompleteDataAddress,
4330 #endif // CC_EventSequenceComplete
4331 #if (PAD_LIST || CC_HashSequenceStart)
4332     (COMMAND_DESCRIPTOR_t *)_HashSequenceStartDataAddress,
4333 #endif // CC_HashSequenceStart
4334 #if (PAD_LIST || CC_PolicyPhysicalPresence)
4335     (COMMAND_DESCRIPTOR_t *)_PolicyPhysicalPresenceDataAddress,
4336 #endif // CC_PolicyPhysicalPresence
4337 #if (PAD_LIST || CC_PolicyDuplicationSelect)
4338     (COMMAND_DESCRIPTOR_t *)_PolicyDuplicationSelectDataAddress,
```

```
4339 #endif // CC_PolicyDuplicationSelect
4340 #if (PAD_LIST || CC_PolicyGetDigest)
4341     (COMMAND_DESCRIPTOR_t *)_PolicyGetDigestDataAddress,
4342 #endif // CC_PolicyGetDigest
4343 #if (PAD_LIST || CC_TestParms)
4344     (COMMAND_DESCRIPTOR_t *)_TestParmsDataAddress,
4345 #endif // CC_TestParms
4346 #if (PAD_LIST || CC_Commit)
4347     (COMMAND_DESCRIPTOR_t *)_CommitDataAddress,
4348 #endif // CC_Commit
4349 #if (PAD_LIST || CC_PolicyPassword)
4350     (COMMAND_DESCRIPTOR_t *)_PolicyPasswordDataAddress,
4351 #endif // CC_PolicyPassword
4352 #if (PAD_LIST || CC_ZGen_2Phase)
4353     (COMMAND_DESCRIPTOR_t *)_ZGen_2PhaseDataAddress,
4354 #endif // CC_ZGen_2Phase
4355 #if (PAD_LIST || CC_EC_Ephemeral)
4356     (COMMAND_DESCRIPTOR_t *)_EC_EphemeralDataAddress,
4357 #endif // CC_EC_Ephemeral
4358 #if (PAD_LIST || CC_PolicyNvWritten)
4359     (COMMAND_DESCRIPTOR_t *)_PolicyNvWrittenDataAddress,
4360 #endif // CC_PolicyNvWritten
4361 #if (PAD_LIST || CC_PolicyTemplate)
4362     (COMMAND_DESCRIPTOR_t *)_PolicyTemplateDataAddress,
4363 #endif // CC_PolicyTemplate
4364 #if (PAD_LIST || CC_CreateLoaded)
4365     (COMMAND_DESCRIPTOR_t *)_CreateLoadedDataAddress,
4366 #endif // CC_CreateLoaded
4367 #if (PAD_LIST || CC_PolicyAuthorizeNV)
4368     (COMMAND_DESCRIPTOR_t *)_PolicyAuthorizeNVDataAddress,
4369 #endif // CC_PolicyAuthorizeNV
4370 #if (PAD_LIST || CC_EncryptDecrypt2)
4371     (COMMAND_DESCRIPTOR_t *)_EncryptDecrypt2DataAddress,
4372 #endif // CC_EncryptDecrypt2
4373 #if (PAD_LIST || CC_AC_GetCapability)
4374     (COMMAND_DESCRIPTOR_t *)_AC_GetCapabilityDataAddress,
4375 #endif // CC_AC_GetCapability
4376 #if (PAD_LIST || CC_AC_Send)
4377     (COMMAND_DESCRIPTOR_t *)_AC_SendDataAddress,
4378 #endif // CC_AC_Send
4379 #if (PAD_LIST || CC_Policy_AC_SendSelect)
4380     (COMMAND_DESCRIPTOR_t *)_Policy_AC_SendSelectDataAddress,
4381 #endif // CC_Policy_AC_SendSelect
4382 #if (PAD_LIST || CC_CertifyX509)
4383     (COMMAND_DESCRIPTOR_t *)_CertifyX509DataAddress,
4384 #endif // CC_CertifyX509
4385 #if (PAD_LIST || CC_Vendor_TCG_Test)
4386     (COMMAND_DESCRIPTOR_t *)_Vendor_TCG_TestDataAddress,
4387 #endif // CC_Vendor_TCG_Test
4388     0
4389 };
4390 #endif // _COMMAND_TABLE_DISPATCH_
```


5.7 Commands.h

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

Start-up

```
3  #ifndef TPM_CC_Startup
4  #include "Startup_fp.h"
5  #endif
6  #ifndef TPM_CC_Shutdown
7  #include "Shutdown_fp.h"
8  #endif
```

Testing

```
9  #ifndef TPM_CC_SelfTest
10 #include "SelfTest_fp.h"
11 #endif
12 #ifndef TPM_CC_IncrementalSelfTest
13 #include "IncrementalSelfTest_fp.h"
14 #endif
15 #ifndef TPM_CC_GetTestResult
16 #include "GetTestResult_fp.h"
17 #endif
```

Session Commands

```
18 #ifndef TPM_CC_StartAuthSession
19 #include "StartAuthSession_fp.h"
20 #endif
21 #ifndef TPM_CC_PolicyRestart
22 #include "PolicyRestart_fp.h"
23 #endif
```

Object Commands

```
24 #ifndef TPM_CC_Create
25 #include "Create_fp.h"
26 #endif
27 #ifndef TPM_CC_Load
28 #include "Load_fp.h"
29 #endif
30 #ifndef TPM_CC_LoadExternal
31 #include "LoadExternal_fp.h"
32 #endif
33 #ifndef TPM_CC_ReadPublic
34 #include "ReadPublic_fp.h"
35 #endif
36 #ifndef TPM_CC_ActivateCredential
37 #include "ActivateCredential_fp.h"
38 #endif
39 #ifndef TPM_CC_MakeCredential
40 #include "MakeCredential_fp.h"
41 #endif
42 #ifndef TPM_CC_Unseal
43 #include "Unseal_fp.h"
44 #endif
45 #ifndef TPM_CC_ObjectChangeAuth
46 #include "ObjectChangeAuth_fp.h"
47 #endif
48 #ifndef TPM_CC_CreateLoaded
49 #include "CreateLoaded_fp.h"
```

```
50 #endif
```

Duplication Commands

```
51 #ifndef TPM_CC_Duplicate
52 #include "Duplicate_fp.h"
53 #endif
54 #ifndef TPM_CC_Rewrap
55 #include "Rewrap_fp.h"
56 #endif
57 #ifndef TPM_CC_Import
58 #include "Import_fp.h"
59 #endif
```

Asymmetric Primitives

```
60 #ifndef TPM_CC_RSA_Encrypt
61 #include "RSA_Encrypt_fp.h"
62 #endif
63 #ifndef TPM_CC_RSA_Decrypt
64 #include "RSA_Decrypt_fp.h"
65 #endif
66 #ifndef TPM_CC_ECDH_KeyGen
67 #include "ECDH_KeyGen_fp.h"
68 #endif
69 #ifndef TPM_CC_ECDH_ZGen
70 #include "ECDH_ZGen_fp.h"
71 #endif
72 #ifndef TPM_CC_ECC_Parameters
73 #include "ECC_Parameters_fp.h"
74 #endif
75 #ifndef TPM_CC_ZGen_2Phase
76 #include "ZGen_2Phase_fp.h"
77 #endif
```

Symmetric Primitives

```
78 #ifndef TPM_CC_EncryptDecrypt
79 #include "EncryptDecrypt_fp.h"
80 #endif
81 #ifndef TPM_CC_EncryptDecrypt2
82 #include "EncryptDecrypt2_fp.h"
83 #endif
84 #ifndef TPM_CC_Hash
85 #include "Hash_fp.h"
86 #endif
87 #ifndef TPM_CC_HMAC
88 #include "HMAC_fp.h"
89 #endif
90 #ifndef TPM_CC_MAC
91 #include "MAC_fp.h"
92 #endif
```

Random Number Generator

```
93 #ifndef TPM_CC_GetRandom
94 #include "GetRandom_fp.h"
95 #endif
96 #ifndef TPM_CC_StirRandom
97 #include "StirRandom_fp.h"
98 #endif
```

Hash/HMAC/Event Sequences


```
99  #ifndef TPM_CC_HMAC_Start
100  #include "HMAC_Start_fp.h"
101  #endif
102  #ifndef TPM_CC_MAC_Start
103  #include "MAC_Start_fp.h"
104  #endif
105  #ifndef TPM_CC_HashSequenceStart
106  #include "HashSequenceStart_fp.h"
107  #endif
108  #ifndef TPM_CC_SequenceUpdate
109  #include "SequenceUpdate_fp.h"
110  #endif
111  #ifndef TPM_CC_SequenceComplete
112  #include "SequenceComplete_fp.h"
113  #endif
114  #ifndef TPM_CC_EventSequenceComplete
115  #include "EventSequenceComplete_fp.h"
116  #endif
```

Attestation Commands

```
117  #ifndef TPM_CC_Certify
118  #include "Certify_fp.h"
119  #endif
120  #ifndef TPM_CC_CertifyCreation
121  #include "CertifyCreation_fp.h"
122  #endif
123  #ifndef TPM_CC_Quote
124  #include "Quote_fp.h"
125  #endif
126  #ifndef TPM_CC_GetSessionAuditDigest
127  #include "GetSessionAuditDigest_fp.h"
128  #endif
129  #ifndef TPM_CC_GetCommandAuditDigest
130  #include "GetCommandAuditDigest_fp.h"
131  #endif
132  #ifndef TPM_CC_GetTime
133  #include "GetTime_fp.h"
134  #endif
135  #ifndef TPM_CC_CertifyX509
136  #include "CertifyX509_fp.h"
137  #endif
```

Ephemeral EC Keys

```
138  #ifndef TPM_CC_Commit
139  #include "Commit_fp.h"
140  #endif
141  #ifndef TPM_CC_EC_Ephemeral
142  #include "EC_Ephemeral_fp.h"
143  #endif
```

Signing and Signature Verification

```
144  #ifndef TPM_CC_VerifySignature
145  #include "VerifySignature_fp.h"
146  #endif
147  #ifndef TPM_CC_Sign
148  #include "Sign_fp.h"
149  #endif
```

Command Audit

```
150  #ifndef TPM_CC_SetCommandCodeAuditStatus
```

```
151 #include "SetCommandCodeAuditStatus_fp.h"
152 #endif
```

Integrity Collection (PCR)

```
153 #ifndef TPM_CC_PCR_Extend
154 #include "PCR_Extend_fp.h"
155 #endif
156 #ifndef TPM_CC_PCR_Event
157 #include "PCR_Event_fp.h"
158 #endif
159 #ifndef TPM_CC_PCR_Read
160 #include "PCR_Read_fp.h"
161 #endif
162 #ifndef TPM_CC_PCR_Allocate
163 #include "PCR_Allocate_fp.h"
164 #endif
165 #ifndef TPM_CC_PCR_SetAuthPolicy
166 #include "PCR_SetAuthPolicy_fp.h"
167 #endif
168 #ifndef TPM_CC_PCR_SetAuthValue
169 #include "PCR_SetAuthValue_fp.h"
170 #endif
171 #ifndef TPM_CC_PCR_Reset
172 #include "PCR_Reset_fp.h"
173 #endif
```

Enhanced Authorization (EA) Commands

```
174 #ifndef TPM_CC_PolicySigned
175 #include "PolicySigned_fp.h"
176 #endif
177 #ifndef TPM_CC_PolicySecret
178 #include "PolicySecret_fp.h"
179 #endif
180 #ifndef TPM_CC_PolicyTicket
181 #include "PolicyTicket_fp.h"
182 #endif
183 #ifndef TPM_CC_PolicyOR
184 #include "PolicyOR_fp.h"
185 #endif
186 #ifndef TPM_CC_PolicyPCR
187 #include "PolicyPCR_fp.h"
188 #endif
189 #ifndef TPM_CC_PolicyLocality
190 #include "PolicyLocality_fp.h"
191 #endif
192 #ifndef TPM_CC_PolicyNV
193 #include "PolicyNV_fp.h"
194 #endif
195 #ifndef TPM_CC_PolicyCounterTimer
196 #include "PolicyCounterTimer_fp.h"
197 #endif
198 #ifndef TPM_CC_PolicyCommandCode
199 #include "PolicyCommandCode_fp.h"
200 #endif
201 #ifndef TPM_CC_PolicyPhysicalPresence
202 #include "PolicyPhysicalPresence_fp.h"
203 #endif
204 #ifndef TPM_CC_PolicyCpHash
205 #include "PolicyCpHash_fp.h"
206 #endif
207 #ifndef TPM_CC_PolicyNameHash
208 #include "PolicyNameHash_fp.h"
209 #endif
```

```
210 #ifndef TPM_CC_PolicyDuplicationSelect
211 #include "PolicyDuplicationSelect_fp.h"
212 #endif
213 #ifndef TPM_CC_PolicyAuthorize
214 #include "PolicyAuthorize_fp.h"
215 #endif
216 #ifndef TPM_CC_PolicyAuthValue
217 #include "PolicyAuthValue_fp.h"
218 #endif
219 #ifndef TPM_CC_PolicyPassword
220 #include "PolicyPassword_fp.h"
221 #endif
222 #ifndef TPM_CC_PolicyGetDigest
223 #include "PolicyGetDigest_fp.h"
224 #endif
225 #ifndef TPM_CC_PolicyNvWritten
226 #include "PolicyNvWritten_fp.h"
227 #endif
228 #ifndef TPM_CC_PolicyTemplate
229 #include "PolicyTemplate_fp.h"
230 #endif
231 #ifndef TPM_CC_PolicyAuthorizeNV
232 #include "PolicyAuthorizeNV_fp.h"
233 #endif
```

Hierarchy Commands

```
234 #ifndef TPM_CC_CreatePrimary
235 #include "CreatePrimary_fp.h"
236 #endif
237 #ifndef TPM_CC_HierarchyControl
238 #include "HierarchyControl_fp.h"
239 #endif
240 #ifndef TPM_CC_SetPrimaryPolicy
241 #include "SetPrimaryPolicy_fp.h"
242 #endif
243 #ifndef TPM_CC_ChangePPS
244 #include "ChangePPS_fp.h"
245 #endif
246 #ifndef TPM_CC_ChangeEPS
247 #include "ChangeEPS_fp.h"
248 #endif
249 #ifndef TPM_CC_Clear
250 #include "Clear_fp.h"
251 #endif
252 #ifndef TPM_CC_ClearControl
253 #include "ClearControl_fp.h"
254 #endif
255 #ifndef TPM_CC_HierarchyChangeAuth
256 #include "HierarchyChangeAuth_fp.h"
257 #endif
```

Dictionary Attack Functions

```
258 #ifndef TPM_CC_DictionaryAttackLockReset
259 #include "DictionaryAttackLockReset_fp.h"
260 #endif
261 #ifndef TPM_CC_DictionaryAttackParameters
262 #include "DictionaryAttackParameters_fp.h"
263 #endif
```

Miscellaneous Management Functions

```
264 #ifndef TPM_CC_PP_Commands
```

```
265 #include "PP_Commands_fp.h"
266 #endif
267 #ifdef TPM_CC_SetAlgorithmSet
268 #include "SetAlgorithmSet_fp.h"
269 #endif
```

Field Upgrade

```
270 #ifdef TPM_CC_FieldUpgradeStart
271 #include "FieldUpgradeStart_fp.h"
272 #endif
273 #ifdef TPM_CC_FieldUpgradeData
274 #include "FieldUpgradeData_fp.h"
275 #endif
276 #ifdef TPM_CC_FirmwareRead
277 #include "FirmwareRead_fp.h"
278 #endif
```

Context Management

```
279 #ifdef TPM_CC_ContextSave
280 #include "ContextSave_fp.h"
281 #endif
282 #ifdef TPM_CC_ContextLoad
283 #include "ContextLoad_fp.h"
284 #endif
285 #ifdef TPM_CC_FlushContext
286 #include "FlushContext_fp.h"
287 #endif
288 #ifdef TPM_CC_EvictControl
289 #include "EvictControl_fp.h"
290 #endif
```

Clocks and Timers

```
291 #ifdef TPM_CC_ReadClock
292 #include "ReadClock_fp.h"
293 #endif
294 #ifdef TPM_CC_ClockSet
295 #include "ClockSet_fp.h"
296 #endif
297 #ifdef TPM_CC_ClockRateAdjust
298 #include "ClockRateAdjust_fp.h"
299 #endif
```

Capability Commands

```
300 #ifdef TPM_CC_GetCapability
301 #include "GetCapability_fp.h"
302 #endif
303 #ifdef TPM_CC_TestParms
304 #include "TestParms_fp.h"
305 #endif
```

Non-volatile Storage

```
306 #ifdef TPM_CC_NV_DefineSpace
307 #include "NV_DefineSpace_fp.h"
308 #endif
309 #ifdef TPM_CC_NV_UndefineSpace
310 #include "NV_UndefineSpace_fp.h"
311 #endif
312 #ifdef TPM_CC_NV_UndefineSpaceSpecial
```

```
313 #include "NV_UndefineSpaceSpecial_fp.h"
314 #endif
315 #ifdef TPM_CC_NV_ReadPublic
316 #include "NV_ReadPublic_fp.h"
317 #endif
318 #ifdef TPM_CC_NV_Write
319 #include "NV_Write_fp.h"
320 #endif
321 #ifdef TPM_CC_NV_Increment
322 #include "NV_Increment_fp.h"
323 #endif
324 #ifdef TPM_CC_NV_Extend
325 #include "NV_Extend_fp.h"
326 #endif
327 #ifdef TPM_CC_NV_SetBits
328 #include "NV_SetBits_fp.h"
329 #endif
330 #ifdef TPM_CC_NV_WriteLock
331 #include "NV_WriteLock_fp.h"
332 #endif
333 #ifdef TPM_CC_NV_GlobalWriteLock
334 #include "NV_GlobalWriteLock_fp.h"
335 #endif
336 #ifdef TPM_CC_NV_Read
337 #include "NV_Read_fp.h"
338 #endif
339 #ifdef TPM_CC_NV_ReadLock
340 #include "NV_ReadLock_fp.h"
341 #endif
342 #ifdef TPM_CC_NV_ChangeAuth
343 #include "NV_ChangeAuth_fp.h"
344 #endif
345 #ifdef TPM_CC_NV_Certify
346 #include "NV_Certify_fp.h"
347 #endif
```

Attached Components

```
348 #ifdef TPM_CC_AC_GetCapability
349 #include "AC_GetCapability_fp.h"
350 #endif
351 #ifdef TPM_CC_AC_Send
352 #include "AC_Send_fp.h"
353 #endif
354 #ifdef TPM_CC_Policy_AC_SendSelect
355 #include "Policy_AC_SendSelect_fp.h"
356 #endif
```

Vendor Specific

```
357 #ifdef TPM_CC_Vendor_TCG_Test
358 #include "Vendor_TCG_Test_fp.h"
359 #endif
360 #endif
```

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  #ifdef GCC
4  #   undef _MSC_VER
5  #   undef WIN32
6  #endif
7  #ifdef _MSC_VER

```

These definitions are for the Microsoft compiler Endian conversion for aligned structures

```

8  #   define REVERSE_ENDIAN_16(_Number) _byteswap_ushort(_Number)
9  #   define REVERSE_ENDIAN_32(_Number) _byteswap_ulong(_Number)
10 #   define REVERSE_ENDIAN_64(_Number) _byteswap_uint64(_Number)

```

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

```

11 // #define _NO_CRT_STDIO_INLINE

```

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.

```

12 #   define LIB_EXPORT __declspec(dllexport) // VS compatible version
13 #   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.

```

14 #   define NORETURN __declspec(noreturn)
15 #   if _MSC_VER >= 1400 // SAL processing when needed
16 #       include <sal.h>
17 #   endif
18 #   ifdef _WIN64
19 #       define _INTPTR 2
20 #   else
21 #       define _INTPTR 1
22 #   endif
23 #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

```

24 #   define _REDUCE_WARNING_LEVEL_(n) \
25 __pragma(warning(push, n))

```

Restore the compiler warning level

```

26 #   define _NORMAL_WARNING_LEVEL_ \
27 __pragma(warning(pop))
28 #   include <stdint.h>
29 #endif
30 #ifndef _MSC_VER
31 #ifndef WINAPI
32 #   define WINAPI
33 #endif
34 #   define __pragma(x)
35 #   define REVERSE_ENDIAN_16(_Number) __builtin_bswap16(_Number)

```

```
36 #   define REVERSE_ENDIAN_32(_Number) __builtin_bswap32(_Number)
37 #   define REVERSE_ENDIAN_64(_Number) __builtin_bswap64(_Number)
38 #endif
39 #if defined(__GNUC__)
40 #   define NORETURN                                __attribute__((noreturn))
41 #   include <stdint.h>
42 #endif
```

Things that are not defined should be defined as NULL

```
43 #ifndef NORETURN
44 #   define NORETURN
45 #endif
46 #ifndef LIB_EXPORT
47 #   define LIB_EXPORT
48 #endif
49 #ifndef LIB_IMPORT
50 #   define LIB_IMPORT
51 #endif
52 #ifndef REDUCE_WARNING_LEVEL_
53 #   define REDUCE_WARNING_LEVEL_(n)
54 #endif
55 #ifndef NORMAL_WARNING_LEVEL_
56 #   define NORMAL_WARNING_LEVEL_
57 #endif
58 #ifndef NOT_REFERENCED
59 #   define NOT_REFERENCED(x) (x = x)
60 #endif
61 #ifdef _POSIX_
62 typedef int SOCKET;
63 #endif
64 #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  #ifdef GLOBAL_C
7  #define EXTERN
8  #define INITIALIZER(_value_) = _value_
9  #else
10 #define EXTERN extern
11 #define INITIALIZER(_name_)
12 #endif
13 _REDUCE_WARNING_LEVEL_(2)
14 #include <string.h>
15 #include <stddef.h>
16 _NORMAL_WARNING_LEVEL_
17
18 #if SIMULATION
19 #undef CONTEXT_SLOT
20 # define CONTEXT_SLOT  UINT8
21 #endif
22 #include "Capabilities.h"
23 #include "TpmTypes.h"
24 #include "CommandAttributes.h"
25 #include "CryptTest.h"
26 #include "BnValues.h"
27 #include "CryptHash.h"
28 #include "CryptSym.h"
29 #include "CryptRand.h"
30 #include "CryptEcc.h"
31 #include "CryptRsa.h"
32 #include "CryptTest.h"
33 #include "TpmError.h"
34 #include "NV.h"
35
36 /** Defines and Types
37
38 **** Size Types
39 // These types are used to differentiate the two different size values used.
40 //
41 // NUMBYTES is used when a size is a number of bytes (usually a TPM2B)
42 typedef UINT16  NUMBYTES;
43
44 **** Other Types
45 // An AUTH_VALUE is a BYTE array containing a digest (TPMU_HA)
```

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

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

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

A NAME is a BYTE array that can contain a TPMU_NAME

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

Definition for a PROOF value

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

Definition for a Primary Seed value

```
50  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.

```
51  #if CLOCK_STOPS
52  typedef UINT64    CLOCK_NONCE;
53  #else
54  typedef UINT32    CLOCK_NONCE;
55  #endif
```

5.9.3 Loaded Object Structures

5.9.3.1 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.

5.9.3.2 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.

```
56  typedef struct
57  {
58      unsigned    publicOnly : 1;    //0) SET if only the public portion of
59                                     // an object is loaded
60      unsigned    epsHierarchy : 1;  //1) SET if the object belongs to EPS
61                                     // Hierarchy
62      unsigned    ppsHierarchy : 1;  //2) SET if the object belongs to PPS
63                                     // Hierarchy
64      unsigned    spsHierarchy : 1;  //3) SET if the object belongs to SPS
65                                     // Hierarchy
66      unsigned    evict : 1;         //4) SET if the object is a platform or
67                                     // owner evict object. Platform-
68                                     // evict object belongs to PPS
69                                     // hierarchy, owner-evict object
70                                     // belongs to SPS or EPS hierarchy.
```

```

71                                     // This bit is also used to mark a
72                                     // completed sequence object so it
73                                     // will be flush when the
74                                     // SequenceComplete command succeeds.
75     unsigned          primary : 1;    //5) SET for a primary object
76     unsigned          temporary : 1;  //6) SET for a temporary object
77     unsigned          stClear : 1;    //7) SET for an stClear object
78     unsigned          hmacSeq : 1;    //8) SET for an HMAC or MAC sequence
79                                     // object
80     unsigned          hashSeq : 1;    //9) SET for a hash sequence object
81     unsigned          eventSeq : 1;   //10) SET for an event sequence object
82     unsigned          ticketSafe : 1; //11) SET if a ticket is safe to create
83                                     // for hash sequence object
84     unsigned          firstBlock : 1; //12) SET if the first block of hash
85                                     // data has been received. It
86                                     // works with ticketSafe bit
87     unsigned          isParent : 1;   //13) SET if the key has the proper
88                                     // attributes to be a parent key
89     // unsigned       privateExp : 1; //14) SET when the private exponent
90     //                                     // of an RSA key has been validated.
91     unsigned          not_used_14 : 1;
92     unsigned          occupied : 1;   //15) SET when the slot is occupied.
93     unsigned          derivation : 1; //16) SET when the key is a derivation
94                                     // parent
95     unsigned          external : 1;   //17) SET when the object is loaded with
96                                     // TPM2_LoadExternal();
97 } OBJECT_ATTRIBUTES;
98 #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.

```

99 #define      RSA_prime_flag      0x8000
100 #endif

```

5.9.3.3 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.

```

101 typedef struct OBJECT
102 {
103     // The attributes field is required to be first followed by the publicArea.
104     // This allows the overlay of the object structure and a sequence structure
105     OBJECT_ATTRIBUTES  attributes;    // object attributes
106     TPMT_PUBLIC        publicArea;    // public area of an object
107     TPMT_SENSITIVE     sensitive;     // sensitive area of an object
108     TPM2B_NAME         qualifiedName;  // object qualified name
109     TPMI_DH_OBJECT     evictHandle;   // if the object is an evict object,
110                                     // the original handle is kept here.
111                                     // The 'working' handle will be the
112                                     // handle of an object slot.
113     TPM2B_NAME         name;          // Name of the object name. Kept here
114                                     // to avoid repeatedly computing it.
115 } OBJECT;

```

5.9.3.4 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

```

116 typedef struct HASH_OBJECT
117 {
118     OBJECT_ATTRIBUTES    attributes;           // The attributes of the HASH object
119     TPMI_ALG_PUBLIC      type;                // algorithm
120     TPMI_ALG_HASH        nameAlg;            // name algorithm
121     TPMA_OBJECT          objectAttributes;    // object attributes
122
123     // The data below is unique to a sequence object
124     TPM2B_AUTH           auth;                // authorization for use of sequence
125     union
126     {
127         HASH_STATE       hashState[HASH_COUNT];
128         HMAC_STATE       hmacState;
129         state;
130     } HASH_OBJECT;
131     typedef BYTE HASH_OBJECT_BUFFER[sizeof(HASH_OBJECT)];

```

5.9.3.5 ANY_OBJECT

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

```

132 typedef union ANY_OBJECT
133 {
134     OBJECT          entity;
135     HASH_OBJECT     hash;
136 } ANY_OBJECT;
137 typedef BYTE ANY_OBJECT_BUFFER[sizeof(ANY_OBJECT)];

```

5.9.4 AUTH_DUP Types

These values are used in the authorization processing.

```

138 typedef UINT32 AUTH_ROLE;
139 #define AUTH_NONE ((AUTH_ROLE) (0))
140 #define AUTH_USER ((AUTH_ROLE) (1))
141 #define AUTH_ADMIN ((AUTH_ROLE) (2))
142 #define AUTH_DUP ((AUTH_ROLE) (3))

```

5.9.5 Active Session Context

5.9.5.1 Description

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

5.9.5.2 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.

```

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

```

5.9.5.3 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.

```

193 typedef struct SESSION
194 {
195     SESSION_ATTRIBUTES attributes;           // session attributes
196     UINT32 pcrCounter;                      // PCR counter value when PCR is
197                                           // included (policy session)
198                                           // If no PCR is included, this
199                                           // value is 0.
200     UINT64 startTime;                      // The value in g_time when the session
201                                           // was started (policy session)
202     UINT64 timeout;                        // The timeout relative to g_time
203                                           // There is no timeout if this value
204                                           // is 0.
205     CLOCK_NONCE epoch;                    // The g_clockEpoch value when the
206                                           // session was started. If g_clockEpoch
207                                           // does not match this value when the
208                                           // timeout is used, then
209                                           // then the command will fail.
210     TPM_CC commandCode;                   // command code (policy session)
211     TPM_ALG_ID authHashAlg;               // session hash algorithm
212     TPMA_LOCALITY commandLocality;         // command locality (policy session)
213     TPMT_SYM_DEF symmetric;                // session symmetric algorithm (if any)
214     TPM2B_AUTH sessionKey;                // session secret value used for
215                                           // this session
216     TPM2B_NONCE nonceTPM;                 // last TPM-generated nonce for
217                                           // generating HMAC and encryption keys
218     union
219     {
220         TPM2B_NAME boundEntity;           // value used to track the entity to
221                                           // which the session is bound
222
223         TPM2B_DIGEST cpHash;              // the required cpHash value for the
224                                           // command being authorized
225         TPM2B_DIGEST nameHash;            // the required nameHash
226         TPM2B_DIGEST templateHash;        // the required template for creation
227     } u1;
228
229     union
230     {
231         TPM2B_DIGEST auditDigest;          // audit session digest
232         TPM2B_DIGEST policyDigest;         // policyHash
233     } u2;
234                                           // audit log and policyHash may
235                                           // share space to save memory
236 } SESSION;
237
238 #define EXPIRES_ON_RESET INT32_MIN
239 #define TIMEOUT_ON_RESET UINT64_MAX
240 #define EXPIRES_ON_RESTART (INT32_MIN + 1)
241 #define TIMEOUT_ON_RESTART (UINT64_MAX - 1)
242
243 typedef BYTE SESSION_BUF[sizeof(SESSION)];

```

5.9.6 PCR

5.9.6.1 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.

```

241 typedef struct PCR_SAVE
242 {
243     #if ALG_SHA1

```

```

244     BYTE          sha1[NUM_STATIC_PCR][SHA1_DIGEST_SIZE];
245 #endif
246 #if ALG_SHA256
247     BYTE          sha256[NUM_STATIC_PCR][SHA256_DIGEST_SIZE];
248 #endif
249 #if ALG_SHA384
250     BYTE          sha384[NUM_STATIC_PCR][SHA384_DIGEST_SIZE];
251 #endif
252 #if ALG_SHA512
253     BYTE          sha512[NUM_STATIC_PCR][SHA512_DIGEST_SIZE];
254 #endif
255 #if ALG_SM3_256
256     BYTE          sm3_256[NUM_STATIC_PCR][SM3_256_DIGEST_SIZE];
257 #endif
258
259     // This counter increments whenever the PCR are updated.
260     // NOTE: A platform-specific specification may designate
261     //       certain PCR changes as not causing this counter
262     //       to increment.
263     UINT32         pcrCounter;
264 } PCR_SAVE;

```

5.9.6.2 PCR_POLICY

```

265 #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.

```

266 typedef struct PCR_POLICY
267 {
268     TPMI_ALG_HASH    hashAlg[NUM_POLICY_PCR_GROUP];
269     TPM2B_DIGEST     a;
270     TPM2B_DIGEST     policy[NUM_POLICY_PCR_GROUP];
271 } PCR_POLICY;
272 #endif

```

5.9.6.3 PCR_AUTHVALUE

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

```

273 typedef struct PCR_AUTH_VALUE
274 {
275     TPM2B_DIGEST     auth[NUM_AUTHVALUE_PCR_GROUP];
276 } PCR_AUTHVALUE;

```

5.9.7 STARTUP_TYPE

This enumeration is the possible startup types. The type is determined by the combination of TPM2_Shutdown() and TPM2_Startup().

```

277 typedef enum
278 {
279     SU_RESET,
280     SU_RESTART,
281     SU_RESUME
282 } STARTUP_TYPE;

```


5.9.8 NV

5.9.8.1 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.

```

283 typedef struct NV_INDEX
284 {
285     TPMS_NV_PUBLIC    publicArea;
286     TPM2B_AUTH        authValue;
287 } NV_INDEX;

```

5.9.8.2 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.

```

288 typedef UINT32      NV_REF;
289 typedef BYTE        *NV_RAM_REF;

```

5.9.8.3 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.

```

290 #if BIG_ENDIAN_TPM
291 typedef struct
292 {
293     UINT32    pinCount;
294     UINT32    pinLimit;
295 } PIN_DATA;
296 #else
297 typedef struct
298 {
299     UINT32    pinLimit;
300     UINT32    pinCount;
301 } PIN_DATA;
302 #endif
303 typedef union
304 {
305     UINT64    intVal;
306     PIN_DATA  pin;
307 } NV_PIN;

```

5.9.9 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.

```

308 #if ALG_ECC

```

```

309 #define COMMIT_INDEX_MASK ((UINT16)((sizeof(gr.commitArray)*8)-1))
310 #endif

```

5.9.10 RAM Global Values

5.9.10.1 Description

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

5.9.10.2 Crypto Self-Test Values

```

311 EXTERN ALGORITHM_VECTOR      g_implementedAlgorithms;
312 EXTERN ALGORITHM_VECTOR      g_toTest;

```

5.9.10.3 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.

```

313 #define g_rcIndexInitializer { TPM_RC_1, TPM_RC_2, TPM_RC_3, TPM_RC_4,      \
314                               TPM_RC_5, TPM_RC_6, TPM_RC_7, TPM_RC_8,      \
315                               TPM_RC_9, TPM_RC_A, TPM_RC_B, TPM_RC_C,      \
316                               TPM_RC_D, TPM_RC_E, TPM_RC_F }
317 EXTERN const UINT16          g_rcIndex[15] INITIALIZER(g_rcIndexInitializer);

```

5.9.10.4 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.

```

318 EXTERN TPM_HANDLE           g_exclusiveAuditSession;

```

5.9.10.5 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.

```

319 EXTERN UINT64               g_time;

```

5.9.10.6 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.

```

320 #if CLOCK_STOPS
321 EXTERN CLOCK_NONCE          g_timeEpoch;
322 #else
323 #define g_timeEpoch          gp.timeEpoch
324 #endif

```

5.9.10.7 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.

```
325  EXTERN  BOOL          g_phEnable;
```

5.9.10.8 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).

```
326  EXTERN  BOOL          g_pcrReConfig;
```

5.9.10.9 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().

```
327  EXTERN  TPMI_DH_OBJECT  g_DRTMHandle;
```

5.9.10.10 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.

```
328  EXTERN  BOOL          g_DrtmPreStartup;
```

5.9.10.11 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.

```
329  EXTERN  BOOL          g_StartupLocality3;
```

5.9.10.12 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.

```
330  #define  SU_NONE_VALUE          (0xFFFF)
331  #define  TPM_SU_NONE            (TPM_SU) (SU_NONE_VALUE)
```

5.9.10.13 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.

```
332  #define  SU_DA_USED_VALUE        (SU_NONE_VALUE - 1)
333  #define  TPM_SU_DA_USED          (TPM_SU) (SU_DA_USED_VALUE)
```

5.9.10.14 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)*.

```
334 #define PRE_STARTUP_FLAG      0x8000
335 #define STARTUP_LOCALITY_3    0x4000
336 #if USE_DA_USED
```

5.9.10.15 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 that *gp.orderly* is set to *SU_NONE_VALUE*

```
337 EXTERN  BOOL                g_daUsed;
338 #endif
```

5.9.10.16 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.

```
339 typedef BYTE                UPDATE_TYPE;
340 #define UT_NONE              (UPDATE_TYPE) 0
341 #define UT_NV                 (UPDATE_TYPE) 1
342 #define UT_ORDERLY           (UPDATE_TYPE) (UT_NV + 2)
343 EXTERN UPDATE_TYPE          g_updateNV;
```

5.9.10.17 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.

```
344 EXTERN  BOOL                g_powerWasLost;
```

5.9.10.18 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*.

```
345 EXTERN  BOOL                g_clearOrderly;
```

5.9.10.19 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.

```
346  EXTERN TPM_SU          g_prevOrderlyState;
```

5.9.10.20 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.

```
347  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

```
348  EXTERN TPM_RC        g_NvStatus;
```

5.9.10.21 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.

```
349  EXTERN TPM2B_AUTH     g_platformUniqueAuthorities; // Reserved for RNG
350  EXTERN TPM2B_AUTH     g_platformUniqueDetails;    // referenced by VENDOR_PERMANENT
```

5.9.11 Persistent Global Values

5.9.11.1 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.

5.9.11.2 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()).

```
351  typedef struct
352  {
353  //*****
354  //      Hierarchy
355  //*****
356  // The values in this section are related to the hierarchies.
357
358      BOOL          disableClear;          // TRUE if TPM2_Clear() using
359                                          // lockoutAuth is disabled
```

```

360
361 // Hierarchy authPolicies
362 TPMI_ALG_HASH      ownerAlg;
363 TPMI_ALG_HASH      endorsementAlg;
364 TPMI_ALG_HASH      lockoutAlg;
365 TPM2B_DIGEST       ownerPolicy;
366 TPM2B_DIGEST       endorsementPolicy;
367 TPM2B_DIGEST       lockoutPolicy;
368
369 // Hierarchy authValues
370 TPM2B_AUTH          ownerAuth;
371 TPM2B_AUTH          endorsementAuth;
372 TPM2B_AUTH          lockoutAuth;
373
374 // Primary Seeds
375 TPM2B_SEED          EPSeed;
376 TPM2B_SEED          SPSeed;
377 TPM2B_SEED          PPSeed;
378 // Note there is a nullSeed in the state_reset memory.
379
380 // Hierarchy proofs
381 TPM2B_PROOF         phProof;
382 TPM2B_PROOF         shProof;
383 TPM2B_PROOF         ehProof;
384 // Note there is a nullProof in the state_reset memory.
385
386 //*****
387 //      Reset Events
388 //*****
389 // A count that increments at each TPM reset and never get reset during the life
390 // time of TPM. The value of this counter is initialized to 1 during TPM
391 // manufacture process. It is used to invalidate all saved contexts after a TPM
392 // Reset.
393     UINT64          totalResetCount;
394
395 // This counter increments on each TPM Reset. The counter is reset by
396 // TPM2_Clear().
397     UINT32          resetCount;
398
399 //*****
400 //      PCR
401 //*****
402 // This structure hold the policies for those PCR that have an update policy.
403 // This implementation only supports a single group of PCR controlled by
404 // policy. If more are required, then this structure would be changed to
405 // an array.
406 #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
407     PCR_POLICY      pcrPolicies;
408 #endif
409
410 // This structure indicates the allocation of PCR. The structure contains a
411 // list of PCR allocations for each implemented algorithm. If no PCR are
412 // allocated for an algorithm, a list entry still exists but the bit map
413 // will contain no SET bits.
414     TPML_PCR_SELECTION pcrAllocated;
415
416 //*****
417 //      Physical Presence
418 //*****
419 // The PP_LIST type contains a bit map of the commands that require physical
420 // to be asserted when the authorization is evaluated. Physical presence will be
421 // checked if the corresponding bit in the array is SET and if the authorization
422 // handle is TPM_RH_PLATFORM.
423 //
424 // These bits may be changed with TPM2_PP_Commands().
425     BYTE            ppList[(COMMAND_COUNT + 7) / 8];

```

```

426
427 //*****
428 //      Dictionary attack values
429 //*****
430 // These values are used for dictionary attack tracking and control.
431     UINT32          failedTries;          // the current count of unexpired
432                                           // authorization failures
433
434     UINT32          maxTries;             // number of unexpired authorization
435                                           // failures before the TPM is in
436                                           // lockout
437
438     UINT32          recoveryTime;         // time between authorization failures
439                                           // before failedTries is decremented
440
441     UINT32          lockoutRecovery;      // time that must expire between
442                                           // authorization failures associated
443                                           // with lockoutAuth
444
445     BOOL            lockOutAuthEnabled;   // TRUE if use of lockoutAuth is
446                                           // allowed
447
448 //*****
449 //      Orderly State
450 //*****
451 // The orderly state for current cycle
452     TPM_SU          orderlyState;
453
454 //*****
455 //      Command audit values.
456 //*****
457     BYTE            auditCommands[ ((COMMAND_COUNT + 1) + 7) / 8];
458     TPMI_ALG_HASH   auditHashAlg;
459     UINT64           auditCounter;
460
461 //*****
462 //      Algorithm selection
463 //*****
464 //
465 // The 'algorithmSet' value indicates the collection of algorithms that are
466 // currently in used on the TPM. The interpretation of value is vendor dependent.
467     UINT32          algorithmSet;
468
469 //*****
470 //      Firmware version
471 //*****
472 // The firmwareV1 and firmwareV2 values are instantiated in TimeStamp.c. This is
473 // a scheme used in development to allow determination of the linker build time
474 // of the TPM. An actual implementation would implement these values in a way that
475 // is consistent with vendor needs. The values are maintained in RAM for simplified
476 // access with a master version in NV. These values are modified in a
477 // vendor-specific way.
478
479 // g_firmwareV1 contains the more significant 32-bits of the vendor version number.
480 // In the reference implementation, if this value is printed as a hex
481 // value, it will have the format of YYYYMMDD
482     UINT32          firmwareV1;
483
484 // g_firmwareV1 contains the less significant 32-bits of the vendor version number.
485 // In the reference implementation, if this value is printed as a hex
486 // value, it will have the format of 00 HH MM SS
487     UINT32          firmwareV2;
488 //*****
489 //      Timer Epoch
490 //*****
491 // timeEpoch contains a nonce that has a vendor-specific size (should not be

```



```

492 // less than 8 bytes. This nonce changes when the clock epoch changes. The clock
493 // epoch changes when there is a discontinuity in the timing of the TPM.
494 #if !CLOCK_STOPS
495     CLOCK_NONCE          timeEpoch;
496 #endif
497
498 } PERSISTENT_DATA;
499 EXTERN PERSISTENT_DATA gp;

```

5.9.11.3 ORDERLY_DATA

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

```

500 typedef struct orderly_data
501 {
502 //*****
503 //          TIME
504 //*****
505
506 // Clock has two parts. One is the state save part and one is the NV part. The
507 // state save version is updated on each command. When the clock rolls over, the
508 // NV version is updated. When the TPM starts up, if the TPM was shutdown in and
509 // orderly way, then the sClock value is used to initialize the clock. If the
510 // TPM shutdown was not orderly, then the persistent value is used and the safe
511 // attribute is clear.
512
513     UINT64          clock;          // The orderly version of clock
514     TPMI_YES_NO     clockSafe;      // Indicates if the clock value is
515                                     // safe.
516
517     // In many implementations, the quality of the entropy available is not that
518     // high. To compensate, the current value of the drbgState can be saved and
519     // restored on each power cycle. This prevents the internal state from reverting
520     // to the initial state on each power cycle and starting with a limited amount
521     // of entropy. By keeping the old state and adding entropy, the entropy will
522     // accumulate.
523     DRBG_STATE      drbgState;
524
525     // These values allow the accumulation of self-healing time across orderly shutdown
526     // of the TPM.
527     #if ACCUMULATE_SELF_HEAL_TIMER
528         UINT64      selfHealTimer; // current value of s_selfHealTimer
529         UINT64      lockoutTimer;  // current value of s_lockoutTimer
530         UINT64      time;          // current value of g_time at shutdown
531     #endif // ACCUMULATE_SELF_HEAL_TIMER
532
533 } ORDERLY_DATA;
534 #if ACCUMULATE_SELF_HEAL_TIMER
535 #define s_selfHealTimer go.selfHealTimer
536 #define s_lockoutTimer go.lockoutTimer
537 #endif // ACCUMULATE_SELF_HEAL_TIMER
538 # define drbgDefault go.drbgState
539 EXTERN ORDERLY_DATA gp;

```

5.9.11.4 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).

```

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

```

5.9.11.5 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.

```

566 typedef struct state_reset_data
567 {
568     /*******
569     //          Hierarchy Control
570     /*******
571     TPM2B_PROOF    nullProof;          // The proof value associated with
572                                         // the TPM_RH_NULL hierarchy. The
573                                         // default reset value is from the RNG.
574
575     TPM2B_SEED     nullSeed;           // The seed value for the TPM_RN_NULL
576                                         // hierarchy. The default reset value
577                                         // is from the RNG.
578
579     /*******
580     //          Context
581     /*******
582     // The 'clearCount' counter is incremented each time the TPM successfully executes
583     // a TPM Resume. The counter is included in each saved context that has 'stClear'
584     // SET (including descendants of keys that have 'stClear' SET). This prevents these
585     // objects from being loaded after a TPM Resume.
586     // If 'clearCount' is at its maximum value when the TPM receives a Shutdown(STATE),
587     // the TPM will return TPM_RC_RANGE and the TPM will only accept Shutdown(CLEAR).
588     UINT32         clearCount;         // The default reset value is 0.
589
590     UINT64         objectContextID;    // This is the context ID for a saved
591                                         // object context. The default reset
592                                         // value is 0.
593
594 #ifndef NDEBUG
595 #define CONTEXT_SLOT
596 #endif

```

```

597
598     CONTEXT_SLOT      contextArray[MAX_ACTIVE_SESSIONS];    // This array contains
599                       // contains the values used to track
600                       // the version numbers of saved
601                       // contexts (see
602                       // Session.c in for details). The
603                       // default reset value is {0}.
604
605     CONTEXT_COUNTER    contextCounter;    // This is the value from which the
606                                           // 'contextID' is derived. The
607                                           // default reset value is {0}.
608
609 //*****
610 //      Command Audit
611 //*****
612 // When an audited command completes, ExecuteCommand() checks the return
613 // value. If it is TPM_RC_SUCCESS, and the command is an audited command, the
614 // TPM will extend the cpHash and rpHash for the command to this value. If this
615 // digest was the Zero Digest before the cpHash was extended, the audit counter
616 // is incremented.
617
618     TPM2B_DIGEST       commandAuditDigest; // This value is set to an Empty Digest
619                                           // by TPM2_GetCommandAuditDigest() or a
620                                           // TPM Reset.
621
622 //*****
623 //      Boot counter
624 //*****
625
626     UINT32             restartCount;    // This counter counts TPM Restarts.
627                                           // The default reset value is 0.
628
629 //*****
630 //      PCR
631 //*****
632 // This counter increments whenever the PCR are updated. This counter is preserved
633 // across TPM Resume even though the PCR are not preserved. This is because
634 // sessions remain active across TPM Restart and the count value in the session
635 // is compared to this counter so this counter must have values that are unique
636 // as long as the sessions are active.
637 // NOTE: A platform-specific specification may designate that certain PCR changes
638 // do not increment this counter to increment.
639
640     UINT32             pcrCounter;    // The default reset value is 0.
641
642 #if ALG_ECC
643 //*****
644 //      ECDA
645 //*****
646     UINT64             commitCounter;    // This counter increments each time
647                                           // TPM2_Commit() returns
648                                           // TPM_RC_SUCCESS. The default reset
649                                           // value is 0.
650
651     TPM2B_NONCE        commitNonce;    // This random value is used to compute
652                                           // the commit values. The default reset
653                                           // value is from the RNG.
654
655 // This implementation relies on the number of bits in g_commitArray being a
656 // power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
657     BYTE               commitArray[16]; // The default reset value is {0}.
658
659 #endif // ALG_ECC
660 } STATE_RESET_DATA;
661 EXTERN STATE_RESET_DATA gr;

```

5.9.12 NV Layout

The NV data organization is

- a) a PERSISTENT_DATA structure
- b) a STATE_RESET_DATA structure
- c) a STATE_CLEAR_DATA structure
- d) an ORDERLY_DATA structure
- e) the user defined NV index space

```

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

```

5.9.13 Global Macro Definitions

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

```

669 #define NV_READ_PERSISTENT(to, from) \
670     NvRead(&to, offsetof(PERSISTENT_DATA, from), sizeof(to))
671 #define NV_WRITE_PERSISTENT(to, from) \
672     NvWrite(offsetof(PERSISTENT_DATA, to), sizeof(gp.to), &from)
673 #define CLEAR_PERSISTENT(item) \
674     NvClearPersistent(offsetof(PERSISTENT_DATA, item), sizeof(gp.item))
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 typedef struct _COMMAND_FLAGS_
679 {
680     unsigned trialPolicy : 1; //1) If SET, one of the handles references a
681                               // trial policy and authorization may be
682                               // skipped. This is only allowed for a policy
683                               // command.
684 } 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.

```

685 typedef struct _COMMAND_
686 {
687     TPM_ST tag; // the parsed command tag
688     TPM_CC code; // the parsed command code
689     COMMAND_INDEX index; // the computed command index
690     UINT32 handleNum; // the number of entity handles in the
691                      // handle area of the command
692     TPM_HANDLE handles[MAX_HANDLE_NUM]; // the parsed handle values
693     UINT32 sessionNum; // the number of sessions found
694     INT32 parameterSize; // starts out with the parsed command size
695                          // and is reduced and values are

```

```

696                                     // unmarshaled. Just before calling the
697                                     // command actions, this should be zero.
698                                     // After the command actions, this number
699                                     // should grow as values are marshaled
700                                     // in to the response buffer.
701     INT32          authSize;         // this is initialized with the parsed size
702                                     // of authorizationSize field and should
703                                     // be zero when the authorizations are
704                                     // parsed.
705     BYTE           *parameterBuffer; // input to ExecuteCommand
706     BYTE           *responseBuffer;  // input to ExecuteCommand
707 #if ALG_SHA1
708     TPM2B_SHA1_DIGEST sha1CpHash;
709     TPM2B_SHA1_DIGEST sha1RpHash;
710 #endif
711 #if ALG_SHA256
712     TPM2B_SHA256_DIGEST sha256CpHash;
713     TPM2B_SHA256_DIGEST sha256RpHash;
714 #endif
715 #if ALG_SHA384
716     TPM2B_SHA384_DIGEST sha384CpHash;
717     TPM2B_SHA384_DIGEST sha384RpHash;
718 #endif
719 #if ALG_SHA512
720     TPM2B_SHA512_DIGEST sha512CpHash;
721     TPM2B_SHA512_DIGEST sha512RpHash;
722 #endif
723 #if ALG_SM3_256
724     TPM2B_SM3_256_DIGEST sm3_256CpHash;
725     TPM2B_SM3_256_DIGEST sm3_256RpHash;
726 #endif
727 } COMMAND;

```

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

```

728 #define STRING_INITIALIZER(value)    {{sizeof(value), {value}}}
729 #define TPM2B_STRING(name, value)   \
730 typedef union name##_ {             \
731     struct {                        \
732         UINT16  size;               \
733         BYTE    buffer[sizeof(value)]; \
734     } t;                             \
735     TPM2B      b;                   \
736 } TPM2B_##name##_;                  \
737 EXTERN const TPM2B_##name##_ name##_ INITIALIZER(STRING_INITIALIZER(value)); \
738 EXTERN const TPM2B      *name INITIALIZER(&name##_b)
739 TPM2B_STRING(PRIMARY_OBJECT_CREATION, "Primary Object Creation");
740 TPM2B_STRING(CFB_KEY, "CFB");
741 TPM2B_STRING(CONTEXT_KEY, "CONTEXT");
742 TPM2B_STRING(INTEGRITY_KEY, "INTEGRITY");
743 TPM2B_STRING(SECRET_KEY, "SECRET");
744 TPM2B_STRING(SESSION_KEY, "ATH");
745 TPM2B_STRING(STORAGE_KEY, "STORAGE");
746 TPM2B_STRING(XOR_KEY, "XOR");
747 TPM2B_STRING(COMMIT_STRING, "ECDA Commit");
748 TPM2B_STRING(DUPLICATE_STRING, "DUPLICATE");
749 TPM2B_STRING(IDENTITY_STRING, "IDENTITY");
750 TPM2B_STRING(OBFUSCATE_STRING, "OBFUSCATE");
751 #if SELF_TEST
752 TPM2B_STRING(OAEP_TEST_STRING, "OAEP Test Value");
753 #endif // SELF_TEST

```

5.9.14 From CryptTest.c

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

```
754 EXTERN CRYPTO_SELF_TEST_STATE    g_cryptoSelfTestState;
```

5.9.15 From Manufacture.c

```
755 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.

```
756 EXTERN BOOL                      g_initialized;
```

5.9.16 Private data

5.9.16.1 From SessionProcess.c

```
757 #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

```
758 EXTERN TPM_HANDLE                s_sessionHandles[MAX_SESSION_NUM];
```

Array of authorization session attributes

```
759 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

```
760 EXTERN TPM_HANDLE                s_associatedHandles[MAX_SESSION_NUM];
```

Array of nonces provided by the caller for the corresponding sessions

```
761 EXTERN TPM2B_NONCE               s_nonceCaller[MAX_SESSION_NUM];
```

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

```
762 EXTERN TPM2B_AUTH                 s_inputAuthValues[MAX_SESSION_NUM];
```

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

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

Special value to indicate an undefined session index

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

Index of the session used for encryption of a response parameter


```
765  EXTERN UINT32          s_encryptSessionIndex;
```

Index of the session used for decryption of a command parameter

```
766  EXTERN UINT32          s_decryptSessionIndex;
```

Index of a session used for audit

```
767  EXTERN UINT32          s_auditSessionIndex;
```

The *cpHash* for command audit

```
768  #ifdef  TPM_CC_GetCommandAuditDigest
769  EXTERN TPM2B_DIGEST      s_cpHashForCommandAudit;
770  #endif
```

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

```
771  EXTERN BOOL             s_DAPendingOnNV;
772  #endif // SESSION_PROCESS_C
```

5.9.16.2 From DA.c

```
773  #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.

```
774  #if !ACCUMULATE_SELF_HEAL_TIMER
775  EXTERN UINT64           s_selfHealTimer;
```

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

```
776  EXTERN UINT64           s_lockoutTimer;
777  #endif // ACCUMULATE_SELF_HEAL_TIMER
778  #endif // DA_C
```

5.9.16.3 From NV.c

```
779  #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.

```
780  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.

```
781  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.

```
782  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.

```
783 EXTERN    NV_INDEX    s_cachedNvIndex;
784 EXTERN    NV_REF      s_cachedNvRef;
785 EXTERN    BYTE         *s_cachedNvRamRef;
```

Initial NV Index/evict object iterator value

```
786 #define    NV_REF_INIT    (NV_REF) 0xFFFFFFFF
787 #endif
```

5.9.16.4 From Object.c

```
788 #if defined OBJECT_C || defined GLOBAL_C
```

This type is the container for an object.

```
789 EXTERN OBJECT    s_objects[MAX_LOADED_OBJECTS];
790 #endif // OBJECT_C
```

5.9.16.5 From PCR.c

```
791 #if defined PCR_C || defined GLOBAL_C
792 typedef struct
793 {
794     #if    ALG_SHA1
795         // SHA1 PCR
796         BYTE    sha1Pcr[SHA1_DIGEST_SIZE];
797     #endif
798     #if    ALG_SHA256
799         // SHA256 PCR
800         BYTE    sha256Pcr[SHA256_DIGEST_SIZE];
801     #endif
802     #if    ALG_SHA384
803         // SHA384 PCR
804         BYTE    sha384Pcr[SHA384_DIGEST_SIZE];
805     #endif
806     #if    ALG_SHA512
807         // SHA512 PCR
808         BYTE    sha512Pcr[SHA512_DIGEST_SIZE];
809     #endif
810     #if    ALG_SM3_256
811         // SHA256 PCR
812         BYTE    sm3_256Pcr[SM3_256_DIGEST_SIZE];
813     #endif
814 } PCR;
815 typedef struct
816 {
817     unsigned int    stateSave : 1;           // if the PCR value should be
818                                           // saved in state save
819     unsigned int    resetLocality : 5;       // The locality that the PCR
820                                           // can be reset
821     unsigned int    extendLocality : 5;      // The locality that the PCR
822                                           // can be extend
823 } PCR_Attributes;
824 EXTERN PCR         s_pcrs[IMPLEMENTATION_PCR];
```

```
825 #endif // PCR_C
```

5.9.16.6 From Session.c

```
826 #if defined SESSION_C || defined GLOBAL_C
```

Container for HMAC or policy session tracking information

```
827 typedef struct
828 {
829     BOOL            occupied;
830     SESSION         session;           // session structure
831 } SESSION_SLOT;
832 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.

```
833 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)

```
834 EXTERN int         s_freeSessionSlots;
835 #endif // SESSION_C
```

5.9.16.7 From IoBuffers.c

```
836 #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.

```
837 EXTERN UINT64      s_actionIoBuffer[768];           // action I/O buffer
838 EXTERN UINT32      s_actionIoAllocation;           // number of UIN64 allocated for the
839                                                         // action input structure
840 #endif // IO_BUFFER_C
```

5.9.16.8 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 isn't useful for anything other than helping the vendor to know in which file the failure occurred.

```
841 EXTERN BOOL        g_inFailureMode;                // Indicates that the TPM is in failure mode
842 #if SIMULATION
843 EXTERN BOOL        g_forceFailureMode;            // flag to force failure mode during test
844 #endif
845 typedef void (FailFunction)(const char *function, int line, int code);
846 #if defined TPM_FAIL_C || defined GLOBAL_C
847 EXTERN UINT32      s_failFunction;
848 EXTERN UINT32      s_failLine;                    // the line in the file at which
849                                                         // the error was signaled
850 EXTERN UINT32      s_failCode;                    // the error code used
851 EXTERN FailFunction *LibFailCallback;
```

```
852  #endif // TPM_FAIL_C
```

5.9.16.9 From CommandCodeAttributes.c

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

```
853  extern const TPMA_CC      s_ccAttr[];
854  extern const COMMAND_ATTRIBUTES s_commandAttributes[];
855  #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  #ifndef NULL
4  #define NULL 0
5  #endif
6  #include "swap.h"
7  #include "VendorString.h"
```

5.10.2 For Self-test

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

```
8  #if SELF_TEST
9  #   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.

```
10 #   define    TEST_HASH(alg)                                     \
11               if(TEST_BIT(alg, g_toTest)                        \
12                   && (alg != ALG_NULL_VALUE))                  \
13                   CryptTestAlgorithm(alg, NULL)                \
14 #else
15 #   define TEST(alg)
16 #   define TEST_HASH(alg)
17 #endif // SELF_TEST
```

5.10.3 For Failures

```
18 #if defined _POSIX_
19 #   define FUNCTION_NAME    0
20 #else
21 #   define FUNCTION_NAME    __FUNCTION__
22 #endif
23 #if !FAIL_TRACE
24 #   define FAIL(errorCode) (TpmFail(errorCode))
25 #   define LOG_FAILURE(errorCode) (TpmLogFailure(errorCode))
26 #else
27 #   define FAIL(errorCode)    TpmFail(FUNCTION_NAME, __LINE__, errorCode)
28 #   define LOG_FAILURE(errorCode) TpmLogFailure(FUNCTION_NAME, __LINE__, errorCode)
29 #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.

```
30 #ifndef NO_LONGJMP
31 #   define FAIL_RETURN(returnCode)
32 #   define TPM_FAIL_RETURN    NORETURN void
33 #else
34 #   define FAIL_RETURN(returnCode) return (returnCode)
35 #   define TPM_FAIL_RETURN    void
36 #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.

```

37 #define REQUIRE(condition, errorCode, returnCode) \
38 { \
39     if(!(condition)) \
40     { \
41         FAIL(FATAL_ERROR_errorCode); \
42         FAIL_RETURN(returnCode); \
43     } \
44 } \
45 #define PARAMETER_CHECK(condition, returnCode) \
46     REQUIRE((condition), PARAMETER, returnCode)
47 #if (defined EMPTY_ASSERT) && (EMPTY_ASSERT != NO)
48 # define pAssert(a) ((void)0)
49 #else
50 # define pAssert(a) {if(!(a)) FAIL(FATAL_ERROR_PARAMETER);}
51 #endif

```

5.10.4 Derived from Vendor-specific values

Values derived from vendor specific settings in Implementation.h

```

52 #define PCR_SELECT_MIN ((PLATFORM_PCR+7)/8)
53 #define PCR_SELECT_MAX ((IMPLEMENTATION_PCR+7)/8)
54 #define MAX_ORDERLY_COUNT ((1 << ORDERLY_BITS) - 1)
55 #ifndef PRIVATE_VENDOR_SPECIFIC_BYTES
56 # define PRIVATE_VENDOR_SPECIFIC_BYTES \
57     ((MAX_RSA_KEY_BYTES/2) * (3 + CRT_FORMAT_RSA * 2))
58 #endif

```

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 isn't 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

```

59 #if COMPILER_CHECKS
60 # define cAssert pAssert
61 #else
62 # define cAssert(value)
63 #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.

```

64 #define ERROR_RETURN(returnCode) \
65 { \
66     retVal = returnCode; \
67     goto Exit; \
68 }
69 #ifndef MAX
70 # define MAX(a, b) ((a) > (b) ? (a) : (b))
71 #endif
72 #ifndef MIN
73 # define MIN(a, b) ((a) < (b) ? (a) : (b))

```

```

74 #endif
75 #ifndef IsOdd
76 # define IsOdd(a)          (((a) & 1) != 0)
77 #endif
78 #ifndef BITS_TO_BYTES
79 # define BITS_TO_BYTES(bits) (((bits) + 7) >> 3)
80 #endif

```

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

```

81 #define TEST_BIT(bit, vector)    TestBit((bit), (BYTE *)&(vector), sizeof(vector))
82 #define SET_BIT(bit, vector)    SetBit((bit), (BYTE *)&(vector), sizeof(vector))
83 #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)

```

84 #ifndef LIB_EXPORT
85 # define LIB_EXPORT
86 # define LIB_IMPORT
87 #endif
88 #ifndef NORETURN
89 # define NORETURN _Noreturn
90 #endif
91 #ifndef NOT_REFERENCED
92 # define NOT_REFERENCED(x = x) ((void) (x))
93 #endif
94 #define STD_RESPONSE_HEADER (sizeof(TPM_ST) + sizeof(UINT32) + sizeof(TPM_RC))
95 #define JOIN(x, y) x##y
96 #define JOIN3(x, y, z) x##y##z
97 #define CONCAT(x, y) JOIN(x, y)
98 #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.

```

99 #ifndef CONTEXT_HASH_ALGORITHM
100 # if defined ALG_SHA512 && ALG_SHA512 == YES
101 #   define CONTEXT_HASH_ALGORITHM    SHA512
102 # elif defined ALG_SHA384 && ALG_SHA384 == YES
103 #   define CONTEXT_HASH_ALGORITHM    SHA384
104 # elif defined ALG_SHA256 && ALG_SHA256 == YES
105 #   define CONTEXT_HASH_ALGORITHM    SHA256
106 # elif defined ALG_SM3_256 && ALG_SM3_256 == YES
107 #   define CONTEXT_HASH_ALGORITHM    SM3_256
108 # elif defined ALG_SHA1 && ALG_SHA1 == YES
109 #   define CONTEXT_HASH_ALGORITHM    SHA1
110 # endif
111 # define CONTEXT_INTEGRITY_HASH_ALG  CONCAT(TPM_ALG_, CONTEXT_HASH_ALGORITHM)
112 #endif
113 #ifndef CONTEXT_INTEGRITY_HASH_SIZE
114 #define CONTEXT_INTEGRITY_HASH_SIZE CONCAT(CONTEXT_HASH_ALGORITHM, _DIGEST_SIZE)
115 #endif
116 #if ALG_RSA
117 #define RSA_SECURITY_STRENGTH (MAX_RSA_KEY_BITS >= 15360 ? 256 : \
118                               (MAX_RSA_KEY_BITS >= 7680 ? 192 : \
119                               (MAX_RSA_KEY_BITS >= 3072 ? 128 : \
120                               (MAX_RSA_KEY_BITS >= 2048 ? 112 : \
121                               (MAX_RSA_KEY_BITS >= 1024 ? 80 : 0))))
122 #else
123 #define RSA_SECURITY_STRENGTH 0
124 #endif // ALG_RSA
125 #if ALG_ECC
126 #define ECC_SECURITY_STRENGTH (MAX_ECC_KEY_BITS >= 521 ? 256 : \

```

```

127                                     (MAX_ECC_KEY_BITS >= 384 ? 192 :
128                                     (MAX_ECC_KEY_BITS >= 256 ? 128 : 0)))
129 #else
130 #define ECC_SECURITY_STRENGTH 0
131 #endif // ALG_ECC
132 #define MAX_ASYM_SECURITY_STRENGTH
133                                     MAX(RSA_SECURITY_STRENGTH, ECC_SECURITY_STRENGTH)
134 #define MAX_HASH_SECURITY_STRENGTH ((CONTEXT_INTEGRITY_HASH_SIZE * 8) / 2)

```

Unless some algorithm is broken...

```

135 #define MAX_SYM_SECURITY_STRENGTH MAX_SYM_KEY_BITS
136 #define MAX_SECURITY_STRENGTH_BITS
137                                     MAX(MAX_ASYM_SECURITY_STRENGTH,
138                                     MAX(MAX_SYM_SECURITY_STRENGTH,
139                                     MAX_HASH_SECURITY_STRENGTH))

```

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

```

140 #define PROOF_SIZE CONTEXT_INTEGRITY_HASH_SIZE

```

As required by P1.14.4

```

141 #define COMPLIANT_PROOF_SIZE
142                                     (MAX(CONTEXT_INTEGRITY_HASH_SIZE, (2 * MAX_SYM_KEY_BYTES)))

```

As required by P1.14.3.1

```

143 #define COMPLIANT_PRIMARY_SEED_SIZE
144                                     BITS_TO_BYTES(MAX_SECURITY_STRENGTH_BITS * 2)

```

This is the pre-errata version

```

145 #ifndef PRIMARY_SEED_SIZE
146 #   define PRIMARY_SEED_SIZE PROOF_SIZE
147 #endif
148 #if USE_SPEC_COMPLIANT_PROOFS
149 #   undef PROOF_SIZE
150 #   define PROOF_SIZE COMPLIANT_PROOF_SIZE
151 #   undef PRIMARY_SEED_SIZE
152 #   define PRIMARY_SEED_SIZE COMPLIANT_PRIMARY_SEED_SIZE
153 #endif // USE_SPEC_COMPLIANT_PROOFS
154 #if !SKIP_PROOF_ERRORS
155 #   if PROOF_SIZE < COMPLIANT_PROOF_SIZE
156 #       error "PROOF_SIZE is not compliant with TPM specification"
157 #   endif
158 #   if PRIMARY_SEED_SIZE < COMPLIANT_PRIMARY_SEED_SIZE
159 #       error "Implementation.h specifies a non-compliant PRIMARY_SEED_SIZE"
160 #   endif
161 #endif // !SKIP_PROOF_ERRORS

```

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

```

162 #if defined CONTEXT_ENCRYPT_ALG
163 #   undef CONTEXT_ENCRYPT_ALGORITHM
164 #   if CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
165 #       define CONTEXT_ENCRYPT_ALGORITHM AES
166 #   elif CONTEXT_ENCRYPT_ALG == ALG_SM4_VALUE
167 #       define CONTEXT_ENCRYPT_ALGORITHM SM4
168 #   elif CONTEXT_ENCRYPT_ALG == ALG_CAMELLIA_VALUE
169 #       define CONTEXT_ENCRYPT_ALGORITHM CAMELLIA
170 #   elif CONTEXT_ENCRYPT_ALG == ALG_TDES_VALUE
171 #       error Are you kidding?

```



```

172 # else
173 #     error Unknown value for CONTEXT_ENCRYPT_ALG
174 # endif // CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
175 #else
176 # define CONTEXT_ENCRYPT_ALG
177     CONCAT3(ALG_, CONTEXT_ENCRYPT_ALGORITHM, _VALUE)
178 #endif // CONTEXT_ENCRYPT_ALG
179 #define CONTEXT_ENCRYPT_KEY_BITS
180     CONCAT(CONTEXT_ENCRYPT_ALGORITHM, _MAX_KEY_SIZE_BITS)
181 #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.

```

182 #ifndef LABEL_MAX_BUFFER
183 #define LABEL_MAX_BUFFER MIN(32, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE))
184 #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. Sealing 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.

```

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

```

Check for consistency of the bit ordering of bit fields

```

186 #if BIG_ENDIAN_TPM && MOST_SIGNIFICANT_BIT_0 && USE_BIT_FIELD_STRUCTURES
187 # error "Settings not consistent"
188 #endif

```

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

```

189 #if USE_BIT_FIELD_STRUCTURES // The default, old version, with bit fields
190 # define IS_ATTRIBUTE(a, type, b) ((a.b != 0))
191 # define SET_ATTRIBUTE(a, type, b) (a.b = SET)
192 # define CLEAR_ATTRIBUTE(a, type, b) (a.b = CLEAR)
193 # define GET_ATTRIBUTE(a, type, b) (a.b)
194 #else
195 # define IS_ATTRIBUTE(a, type, b) ((a & type##_##b) != 0)
196 # define SET_ATTRIBUTE(a, type, b) (a |= type##_##b)
197 # define CLEAR_ATTRIBUTE(a, type, b) (a &= ~type##_##b)
198 # define GET_ATTRIBUTE(a, type, b) \
199     (type)((a & type##_##b) >> type##_##b##_SHIFT)
200 #endif
201 #define VERIFY(_X) if(!(_X)) goto Error
202 #endif // GP_MACROS_H

```

5.11 InternalRoutines.h

```

1  #ifndef      INTERNAL_ROUTINES_H
2  #define      INTERNAL_ROUTINES_H
3  #if !defined _LIB_SUPPORT_H_ && !defined _TPM_H_
4  #error "Should not be called"
5  #endif

```

DRTM functions

```

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

```

Internal subsystem functions

```

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

```

Internal support functions

```

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

```

Miscellaneous

```

36 #include "Bits_fp.h"
37 #include "AlgorithmCap_fp.h"
38 #include "PropertyCap_fp.h"
39 #include "IoBuffers_fp.h"
40 #include "Memory_fp.h"
41 #include "ResponseCodeProcessing_fp.h"

```

Internal cryptographic functions

```

42 #include "BnConvert_fp.h"
43 #include "BnMath_fp.h"
44 #include "BnMemory_fp.h"
45 #include "Ticket_fp.h"

```

```
46 #include "CryptUtil_fp.h"
47 #include "CryptHash_fp.h"
48 #include "CryptSym_fp.h"
49 #include "CryptDes_fp.h"
50 #include "CryptPrime_fp.h"
51 #include "CryptRand_fp.h"
52 #include "CryptSelfTest_fp.h"
53 #include "MathOnByteBuffers_fp.h"
54 #include "CryptSym_fp.h"
55 #include "AlgorithmTests_fp.h"
56 #if ALG_RSA
57 #include "CryptRsa_fp.h"
58 #include "CryptPrimeSieve_fp.h"
59 #endif
60 #if ALG_ECC
61 #include "CryptEccMain_fp.h"
62 #include "CryptEccSignature_fp.h"
63 #include "CryptEccKeyExchange_fp.h"
64 #endif
65 #if CC_MAC || CC_MAC_Start
66 #   include "CryptSmac_fp.h"
67 #   if ALG_CMAC
68 #       include "CryptCmac_fp.h"
69 #   endif
70 #endif
```

Support library

```
71 #include "SupportLibraryFunctionPrototypes_fp.h"
```

Linkage to platform functions

```
72 #include "Platform_fp.h"
73 #endif
```

5.12 LibSupport.h

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

```

1  #ifndef _LIB_SUPPORT_H_
2  #define _LIB_SUPPORT_H_
3  #ifndef RADIX_BITS
4  #   if defined(_x86_64_) || defined(_WIN64) || defined(_M_X64) || defined(_M_ARM64)
5  #       define RADIX_BITS          64
6  #   else
7  #       define RADIX_BITS          32
8  #   endif
9  #endif // RADIX_BITS

```

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

```

10 #define LIB_JOIN(x,y) x##y
11 #define LIB_CONCAT(x,y) LIB_JOIN(x, y)
12 #define LIB_QUOTE(_STRING_) #_STRING_
13 #define LIB_INCLUDE2(_LIB_, _TYPE_) LIB_QUOTE(##_LIB_##/TpmTo##_LIB_##_TYPE_.h)
14 #define LIB_INCLUDE(_LIB_, _TYPE_) LIB_INCLUDE2(_LIB_, _TYPE_)
15 #define SYM_LIBRARY LIB_CONCAT(SYM_LIB_, SYM_LIB)
16 #define HASH_LIBRARY(_LIB_) LIB_CONCAT(HASH_LIB_, HASH_LIB)
17 #define MATH_LIBRARY(_LIB_) LIB_CONCAT(MATH_LIB_, MATH_LIB)

```

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

```

18 #include LIB_INCLUDE(SYM_LIB, Sym)
19 #include LIB_INCLUDE(HASH_LIB, Hash)
20 #undef MIN
21 #undef MAX
22 #endif // _LIB_SUPPORT_H_

```

5.13 NV.h

5.13.1 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  #ifdef    TPM_NT_ORDINARY
```

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

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

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

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

5.13.2 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

```
16 #    define IsNvOrdinaryIndex(attributes) \
17     (GET_TPM_NT(attributes) == TPM_NT_ORDINARY) \
18 #    define IsNvCounterIndex(attributes) \
19     (GET_TPM_NT(attributes) == TPM_NT_COUNTER) \
20 #    define IsNvBitsIndex(attributes) \
21     (GET_TPM_NT(attributes) == TPM_NT_BITS) \
22 #    define IsNvExtendIndex(attributes) \
23     (GET_TPM_NT(attributes) == TPM_NT_EXTEND) \
24 #ifndef TPM_NT_PIN_PASS
25 #    define IsNvPinPassIndex(attributes) \
26     (GET_TPM_NT(attributes) == TPM_NT_PIN_PASS) \
27 #endif
28 #ifndef TPM_NT_PIN_FAIL
29 #    define IsNvPinFailIndex(attributes) \
30     (GET_TPM_NT(attributes) == TPM_NT_PIN_FAIL) \
31 #endif
32 typedef struct {
33     UINT32    size;
34     TPM_HANDLE handle;
35 } NV_ENTRY_HEADER;
36 #define NV_EVICT_OBJECT_SIZE \
37     (sizeof(UINT32) + sizeof(TPM_HANDLE) + sizeof(OBJECT))
38 #define NV_INDEX_COUNTER_SIZE \
39     (sizeof(UINT32) + sizeof(NV_INDEX) + sizeof(UINT64))
40 #define NV_RAM_INDEX_COUNTER_SIZE \
41     (sizeof(NV_RAM_HEADER) + sizeof(UINT64))
```

```

42 typedef struct {
43     UINT32      size;
44     TPM_HANDLE  handle;
45     TPMA_NV     attributes;
46 } 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 UIt64m *maxCounter* value.

```

47 typedef UINT32 NV_LIST_TERMINATOR[3];

```

5.13.3 Orderly RAM Values

The following defines are for accessing orderly RAM values. This is the initialize for the RAM reference iterator.

```

48 #define NV_RAM_REF_INIT 0

```

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

```

49 #define RAM_ORDERLY_START \
50     (&s_indexOrderlyRam[0])

```

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

```

51 #define NV_ORDERLY_START \
52     (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

```

53 #define RAM_ORDERLY_END \
54     (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.

```

55 #define NV_ORDERLY_END \
56     (NV_ORDERLY_START + sizeof(s_indexOrderlyRam))

```

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

```

57 #define ORDERLY_RAM_ADDRESS_OK(start, offset) \
58     ((start >= RAM_ORDERLY_START) && ((start + offset - 1) < RAM_ORDERLY_END))
59 #define RETURN_IF_NV_IS_NOT_AVAILABLE \
60 { \
61     if(g_NvStatus != TPM_RC_SUCCESS) \
62         return g_NvStatus; \
63 }

```

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

```

64 #define RETURN_IF_ORDERLY \
65 { \
66     if(NvClearOrderly() != TPM_RC_SUCCESS) \
67         return g_NvStatus; \
68 }
69 #define NV_IS_AVAILABLE (g_NvStatus == TPM_RC_SUCCESS)
70 #define IS_ORDERLY(value) (value < SU_DA_USED_VALUE)
71 #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.

```
72 #define SET_NV_UPDATE(type)      g_updateNV |= (type)
73 #endif // _NV_H_
```

DRAFT

5.14 PRNG_TestVectors.h

```

1  #ifndef      _MSBN_DRBG_TEST_VECTORS_H
2  #define      _MSBN_DRBG_TEST_VECTORS_H
3  // #if DRBG_ALGORITHM == TPM_ALG_AES && DRBG_KEY_BITS == 256
4  #if DRBG_KEY_SIZE_BITS == 256

```

Entropy is the size of a 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

```

5  #   define DRBG_TEST_INITIATE_ENTROPY          \
6      0x0d, 0x15, 0xaa, 0x80, 0xb1, 0x6c, 0x3a, 0x10, \
7      0x90, 0x6c, 0xfe, 0xdb, 0x79, 0x5d, 0xae, 0x0b, \
8      0x5b, 0x81, 0x04, 0x1c, 0x5c, 0x5b, 0xfa, 0xcb, \
9      0x37, 0x3d, 0x44, 0x40, 0xd9, 0x12, 0x0f, 0x7e, \
10     0x3d, 0x6c, 0xf9, 0x09, 0x86, 0xcf, 0x52, 0xd8, \
11     0x5d, 0x3e, 0x94, 0x7d, 0x8c, 0x06, 0x1f, 0x91
12 #   define DRBG_TEST_RESEED_ENTROPY            \
13     0x6e, 0xe7, 0x93, 0xa3, 0x39, 0x55, 0xd7, 0x2a, \
14     0xd1, 0x2f, 0xd8, 0x0a, 0x8a, 0x3f, 0xcf, 0x95, \
15     0xed, 0x3b, 0x4d, 0xac, 0x57, 0x95, 0xfe, 0x25, \
16     0xcf, 0x86, 0x9f, 0x7c, 0x27, 0x57, 0x3b, 0xbc, \
17     0x56, 0xf1, 0xac, 0xae, 0x13, 0xa6, 0x50, 0x42, \
18     0xb3, 0x40, 0x09, 0x3c, 0x46, 0x4a, 0x7a, 0x22
19 #   define DRBG_TEST_GENERATED_INTERM          \
20     0x28, 0xe0, 0xeb, 0xb8, 0x21, 0x01, 0x66, 0x50, \
21     0x8c, 0x8f, 0x65, 0xf2, 0x20, 0x7b, 0xd0, 0xa3
22 #   define DRBG_TEST_GENERATED                  \
23     0x94, 0x6f, 0x51, 0x82, 0xd5, 0x45, 0x10, 0xb9, \
24     0x46, 0x12, 0x48, 0xf5, 0x71, 0xca, 0x06, 0xc9
25 #elif DRBG_KEY_SIZE_BITS == 128
26 #   define DRBG_TEST_INITIATE_ENTROPY          \
27     0x8f, 0xc1, 0x1b, 0xdb, 0x5a, 0xab, 0xb7, 0xe0, \
28     0x93, 0xb6, 0x14, 0x28, 0xe0, 0x90, 0x73, 0x03, \
29     0xcb, 0x45, 0x9f, 0x3b, 0x60, 0x0d, 0xad, 0x87, \
30     0x09, 0x55, 0xf2, 0x2d, 0xa8, 0x0a, 0x44, 0xf8
31 #   define DRBG_TEST_RESEED_ENTROPY            \
32     0x0c, 0xd5, 0x3c, 0xd5, 0xec, 0xcd, 0x5a, 0x10, \
33     0xd7, 0xea, 0x26, 0x61, 0x11, 0x25, 0x9b, 0x05, \
34     0x57, 0x4f, 0xc6, 0xdd, 0xd8, 0xbe, 0xd8, 0xbd, \
35     0x72, 0x37, 0x8c, 0xf8, 0x2f, 0x1d, 0xba, 0x2a
36 #define DRBG_TEST_GENERATED_INTERM            \
37     0xdc, 0x3c, 0xf6, 0xbf, 0x5b, 0xd3, 0x41, 0x13, \
38     0x5f, 0x2c, 0x68, 0x11, 0xa1, 0x07, 0x1c, 0x87
39 #   define DRBG_TEST_GENERATED                  \
40     0xb6, 0x18, 0x50, 0xde, 0xcf, 0xd7, 0x10, 0x6d, \
41     0x44, 0x76, 0x9a, 0x8e, 0x6e, 0x8c, 0x1a, 0xd4
42 #endif
43 #endif //      _MSBN_DRBG_TEST_VECTORS_H

```

5.15 SelfTest.h

5.15.1 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_
```

5.15.2 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  #define NUM_SYMS      (AES_ENTRIES + SM4_ENTRIES + CAMELLIA_ENTRIES + TDES_ENTRIES)
9  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 TPM_SYM_MODE_FIRST and TPM_SYM_MODE_LAST

```
10 #define TPM_SYM_MODE_FIRST    ALG_CTR_VALUE
11 #define TPM_SYM_MODE_LAST    ALG_ECB_VALUE
12 #define NUM_SYM_MODES        (TPM_SYM_MODE_LAST - TPM_SYM_MODE_FIRST + 1)
```

Define a type to hold a bit vector for the modes.

```
13 #if NUM_SYM_MODES <= 0
14 #error "No symmetric modes implemented"
15 #elif NUM_SYM_MODES <= 8
16 typedef BYTE        SYM_MODES;
17 #elif NUM_SYM_MODES <= 16
18 typedef UINT16       SYM_MODES;
19 #elif NUM_SYM_MODES <= 32
20 typedef UINT32       SYM_MODES;
21 #else
22 #error "Too many symmetric modes"
23 #endif
24 typedef struct SYMMETRIC_TEST_VECTOR {
25     const TPM_ALG_ID    alg;                // the algorithm
26     const UINT16        keyBits;            // bits in the key
27     const BYTE          *key;               // The test key
28     const UINT32        ivSize;             // block size of the algorithm
29     const UINT32        dataInOutSize;      // size to encrypt/decrypt
30     const BYTE          *dataIn;            // data to encrypt
31     const BYTE          *dataOut[NUM_SYM_MODES]; // data to decrypt
32 } SYMMETRIC_TEST_VECTOR;
33 #if ALG_SHA512
34 #   define DEFAULT_TEST_HASH            ALG_SHA512_VALUE
35 #   define DEFAULT_TEST_DIGEST_SIZE     SHA512_DIGEST_SIZE
36 #   define DEFAULT_TEST_HASH_BLOCK_SIZE SHA512_BLOCK_SIZE
37 #elif ALG_SHA384
38 #   define DEFAULT_TEST_HASH            ALG_SHA384_VALUE
```

```
39 #       define DEFAULT_TEST_DIGEST_SIZE      SHA384_DIGEST_SIZE
40 #       define DEFAULT_TEST_HASH_BLOCK_SIZE   SHA384_BLOCK_SIZE
41 #elif ALG_SHA256
42 #       define DEFAULT_TEST_HASH              ALG_SHA256_VALUE
43 #       define DEFAULT_TEST_DIGEST_SIZE       SHA256_DIGEST_SIZE
44 #       define DEFAULT_TEST_HASH_BLOCK_SIZE   SHA256_BLOCK_SIZE
45 #elif ALG_SHA1
46 #       define DEFAULT_TEST_HASH              ALG_SHA1_VALUE
47 #       define DEFAULT_TEST_DIGEST_SIZE       SHA1_DIGEST_SIZE
48 #       define DEFAULT_TEST_HASH_BLOCK_SIZE   SHA1_BLOCK_SIZE
49 #endif
50 #endif      // _SELF_TEST_H_
```

5.16 SupportLibraryFunctionPrototypes_fp.h

5.16.1 Introduction

This file contains the function prototypes for the functions that need to be present in the selected match 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
```

5.16.2 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);
```

5.16.3 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  void
6  MathLibraryCompatibilityCheck(
7      void
8  );
```

5.16.4 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);
```

5.16.5 BnMult()

Multiplies two numbers and returns the result

```
11 LIB_EXPORT BOOL
12 BnMult(bigNum result, bigConst multiplicand, bigConst multiplier);
```

5.16.6 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);
```

5.16.7 BnMod()

```
16 #define BnMod(a, b)      BnDiv(NULL, (a), (a), (b))
```

5.16.8 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);
```

5.16.9 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);
```

5.16.10 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);
```

5.16.11 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);
```

5.16.12 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);
```

5.16.13 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);
```

5.16.14 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);
```

5.16.14.1 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  #endif
```

5.17 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 #endif
```


5.18 Tpm.h

Root header file for building any TPM.lib code

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

5.19 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  #undef YES
4  #define YES 1
5  #undef NO
6  #define NO 0

```

Allow the command line to specify a **profile** file. E.g., PROFILE=/the/profile.h

```

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

```

Need an unambiguous definition for DEBUG. Don't change this

```

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

```

This definition is required for the re-factored code

```

23 #if (!defined USE_BN_ECC_DATA) \
24     || ((USE_BN_ECC_DATA != NO) && (USE_BN_ECC_DATA != YES))
25 #   undef USE_BN_ECC_DATA
26 #   define USE_BN_ECC_DATA YES // Default: Either YES or NO
27 #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

```

28 #if !(defined SIMULATION) || ((SIMULATION != NO) && (SIMULATION != YES))
29 #   undef SIMULATION
30 #   define SIMULATION YES // Default: Either YES or NO
31 #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.

```

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

```

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

```

42 #if !(defined USE_DA_USED) || ((USE_DA_USED != NO) && (USE_DA_USED != YES))
43 #   undef USE_DA_USED
44 #   define USE_DA_USED YES // Default: Either YES or NO
45 #endif

```

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

```

46 #if !(defined TABLE_DRIVEN_DISPATCH) \
47     || ((TABLE_DRIVEN_DISPATCH != NO) && (TABLE_DRIVEN_DISPATCH != YES))
48 #   undef TABLE_DRIVEN_DISPATCH
49 #   define TABLE_DRIVEN_DISPATCH YES // Default: Either YES or NO
50 #endif

```

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

```

51 #if !(defined SELF_TEST) || ((SELF_TEST != NO) && (SELF_TEST != YES))
52 #   undef SELF_TEST
53 #   define SELF_TEST YES // Default: Either YES or NO
54 #endif

```

Enable the generation of RSA primes using a sieve.

```

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

```

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

```

59 #if RSA_KEY_SIEVE && SIMULATION
60 #   if !(defined RSA_INSTRUMENT) \
61         || ((RSA_INSTRUMENT != NO) && (RSA_INSTRUMENT != YES))
62 #       undef RSA_INSTRUMENT
63 #       define RSA_INSTRUMENT NO // Default: Either YES or NO
64 #   endif
65 #endif

```

This switch enables the RNG state save and restore

```

66 #if !(defined _DRBG_STATE_SAVE) \
67 || ((_DRBG_STATE_SAVE != NO) && (_DRBG_STATE_SAVE != YES))
68 # undef _DRBG_STATE_SAVE
69 # define _DRBG_STATE_SAVE YES // Default: Either YES or NO
70 #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.

```

71 #if !(defined COMPRESSED_LISTS) \
72 || ((COMPRESSED_LISTS != NO) && (COMPRESSED_LISTS != YES))
73 # undef COMPRESSED_LISTS
74 # define COMPRESSED_LISTS YES // Default: Either YES or NO
75 #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.

```

76 #if !(defined CLOCK_STOPS) || ((CLOCK_STOPS != NO) && (CLOCK_STOPS != YES))
77 # undef CLOCK_STOPS
78 # define CLOCK_STOPS NO // Default: Either YES or NO
79 #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.

```

80 #if (defined USE_MARSHALING_DEFINES) && (USE_MARSHALING_DEFINES != NO)
81 # undef USE_MARSHALING_DEFINES
82 # define USE_MARSHALING_DEFINES YES
83 #else
84 # define USE_MARSHALING_DEFINES YES // Default: Either YES or NO
85 #endif

```

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

```

86 #if SIMULATION && DEBUG

```

Enables use of the key cache. Default is YES

```

87 # if !(defined USE_RSA_KEY_CACHE) \
88 || ((USE_RSA_KEY_CACHE != NO) && (USE_RSA_KEY_CACHE != YES))
89 # undef USE_RSA_KEY_CACHE
90 # define USE_RSA_KEY_CACHE YES // Default: Either YES or NO
91 # 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

```

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

```

```

98 # else
99 #     undef    USE_KEY_CACHE_FILE
100 #     define   USE_KEY_CACHE_FILE    NO
101 # 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.

```

102 # if !(defined USE_DEBUG_RNG) || ((USE_DEBUG_RNG != NO) && (USE_DEBUG_RNG != YES))
103 #     undef    USE_DEBUG_RNG
104 #     define   USE_DEBUG_RNG    YES    // Default: Either YES or NO
105 # endif

```

Don't 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

```

106 #else
107 # define USE_RSA_KEY_CACHE    NO
108 # define USE_RSA_KEY_CACHE_FILE    NO
109 # define USE_DEBUG_RNG    NO
110 #endif // DEBUG && SIMULATION
111 #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 isn't 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.

```

112 # if !(defined COMPILER_CHECKS) \
113 || ((COMPILER_CHECKS != NO) && (COMPILER_CHECKS != YES))
114 #     undef    COMPILER_CHECKS
115 #     define   COMPILER_CHECKS    NO    // Default: Either YES or NO
116 # endif

```

Some of the values (such as sizes) are the result of different options set in Implementation.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.

```

117 # if !(defined RUNTIME_SIZE_CHECKS) \
118 || ((RUNTIME_SIZE_CHECKS != NO) && (RUNTIME_SIZE_CHECKS != YES))
119 #     undef    RUNTIME_SIZE_CHECKS
120 #     define   RUNTIME_SIZE_CHECKS    NO    // Default: Either YES or NO
121 # 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)

```

122 # if !(defined DRBG_DEBUG_PRINT) \
123 || ((DRBG_DEBUG_PRINT != NO) && (DRBG_DEBUG_PRINT != YES))
124 #     undef    DRBG_DEBUG_PRINT
125 #     define   DRBG_DEBUG_PRINT    NO    // Default: Either YES or NO
126 # endif

```

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

```

127 # if !(defined FAIL_TRACE) || ((FAIL_TRACE != NO) && (FAIL_TRACE != YES))
128 #     undef    FAIL_TRACE
129 #     define    FAIL_TRACE        YES        // Default: Either YES or NO
130 # endif
131 #endif // DEBUG

```

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

```

132 #if !(defined ACCUMULATE_SELF_HEAL_TIMER) \
133     || ((ACCUMULATE_SELF_HEAL_TIMER != NO) && (ACCUMULATE_SELF_HEAL_TIMER != YES))
134 #     undef    ACCUMULATE_SELF_HEAL_TIMER
135 #     define    ACCUMULATE_SELF_HEAL_TIMER        YES        // Default: Either YES or NO
136 #endif

```

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

```

137 #if !(defined USE_SPEC_COMPLIANT_PROOFS) \
138     || ((USE_SPEC_COMPLIANT_PROOFS != NO) && (USE_SPEC_COMPLIANT_PROOFS != YES))
139 #     undef    USE_SPEC_COMPLIANT_PROOFS
140 #     define    USE_SPEC_COMPLIANT_PROOFS        YES        // Default: Either YES or NO
141 #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.

```

142 #if !(defined SKIP_PROOF_ERRORS) \
143     || ((SKIP_PROOF_ERRORS != NO) && (SKIP_PROOF_ERRORS != YES))
144 #     undef    SKIP_PROOF_ERRORS
145 #     define    SKIP_PROOF_ERRORS        NO        // Default: Either YES or NO
146 #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

```

147 #if !(defined USE_BIT_FIELD_STRUCTURES) \
148     || ((USE_BIT_FIELD_STRUCTURES != NO) && (USE_BIT_FIELD_STRUCTURES != YES))
149 #     undef    USE_BIT_FIELD_STRUCTURES
150 #     define    USE_BIT_FIELD_STRUCTURES        DEBUG        // Default: Either YES or NO
151 #endif

```

This define is used to enable any runtime checks of the interface between the cryptographic library (e.g., OpenSSL) and the thinking layer.

```

152 #if !(defined LIBRARY_COMPATIBILITY_CHECK) \
153     || ((LIBRARY_COMPATIBILITY_CHECK != NO) && (LIBRARY_COMPATIBILITY_CHECK != YES))
154 #     undef    LIBRARY_COMPATIBILITY_CHECK
155 #     define    LIBRARY_COMPATIBILITY_CHECK        NO        // Default: Either YES or NO
156 #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.

```

157 #define    ALG_YES        YES
158 #define    ALG_NO        NO
159 #define    CC_YES        YES
160 #define    CC_NO        NO

```

161 ~~#endif~~ // _TPM_BUILD_SWITCHES_H_

DRAFT

5.20 TpmError.h

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

5.21 TpmTypes.h

```

1  #ifndef _TPM_TYPES_H_
2  #define _TPM_TYPES_H_

```

Table 1:2 - 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)

```

Table 1:3 - 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)

```

Table 2:12 - 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 CC_VEND 0x20000000
223 #define TPM_CC_Vendor_TCG_Test (TPM_CC) (0x20000000)

```

Table 2:5 - Definition of Types for Documentation Clarity

```

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

```

Table 2:6 - Definition of TPM_SPEC Constants

```

236 typedef UINT32 TPM_SPEC;
237 #define TYPE_OF_TPM_SPEC UINT32
238 #define SPEC_FAMILY 0x322E3000
239 #define TPM_SPEC_FAMILY (TPM_SPEC) (SPEC_FAMILY)

```



```

240 #define SPEC_LEVEL 00
241 #define TPM_SPEC_LEVEL (TPM_SPEC) (SPEC_LEVEL)
242 #define SPEC_VERSION 155
243 #define TPM_SPEC_VERSION (TPM_SPEC) (SPEC_VERSION)
244 #define SPEC_YEAR 2019
245 #define TPM_SPEC_YEAR (TPM_SPEC) (SPEC_YEAR)
246 #define SPEC_DAY_OF_YEAR 107
247 #define TPM_SPEC_DAY_OF_YEAR (TPM_SPEC) (SPEC_DAY_OF_YEAR)

```

Table 2:7 - Definition of TPM_GENERATED Constants

```

248 typedef UINT32 TPM_GENERATED;
249 #define TYPE_OF_TPM_GENERATED UINT32
250 #define TPM_GENERATED_VALUE (TPM_GENERATED) (0xFF544347)

```

Table 2:16 - Definition of TPM_RC Constants

```

251 typedef UINT32 TPM_RC;
252 #define TYPE_OF_TPM_RC UINT32
253 #define TPM_RC_SUCCESS (TPM_RC) (0x000)
254 #define TPM_RC_BAD_TAG (TPM_RC) (0x01E)
255 #define RC_VER1 (TPM_RC) (0x100)
256 #define TPM_RC_INITIALIZE (TPM_RC) (RC_VER1+0x000)
257 #define TPM_RC_FAILURE (TPM_RC) (RC_VER1+0x001)
258 #define TPM_RC_SEQUENCE (TPM_RC) (RC_VER1+0x003)
259 #define TPM_RC_PRIVATE (TPM_RC) (RC_VER1+0x00B)
260 #define TPM_RC_HMAC (TPM_RC) (RC_VER1+0x019)
261 #define TPM_RC_DISABLED (TPM_RC) (RC_VER1+0x020)
262 #define TPM_RC_EXCLUSIVE (TPM_RC) (RC_VER1+0x021)
263 #define TPM_RC_AUTH_TYPE (TPM_RC) (RC_VER1+0x024)
264 #define TPM_RC_AUTH_MISSING (TPM_RC) (RC_VER1+0x025)
265 #define TPM_RC_POLICY (TPM_RC) (RC_VER1+0x026)
266 #define TPM_RC_PCR (TPM_RC) (RC_VER1+0x027)
267 #define TPM_RC_PCR_CHANGED (TPM_RC) (RC_VER1+0x028)
268 #define TPM_RC_UPGRADE (TPM_RC) (RC_VER1+0x02D)
269 #define TPM_RC_TOO_MANY_CONTEXTS (TPM_RC) (RC_VER1+0x02E)
270 #define TPM_RC_AUTH_UNAVAILABLE (TPM_RC) (RC_VER1+0x02F)
271 #define TPM_RC_REBOOT (TPM_RC) (RC_VER1+0x030)
272 #define TPM_RC_UNBALANCED (TPM_RC) (RC_VER1+0x031)
273 #define TPM_RC_COMMAND_SIZE (TPM_RC) (RC_VER1+0x042)
274 #define TPM_RC_COMMAND_CODE (TPM_RC) (RC_VER1+0x043)
275 #define TPM_RC_AUTHSIZE (TPM_RC) (RC_VER1+0x044)
276 #define TPM_RC_AUTH_CONTEXT (TPM_RC) (RC_VER1+0x045)
277 #define TPM_RC_NV_RANGE (TPM_RC) (RC_VER1+0x046)
278 #define TPM_RC_NV_SIZE (TPM_RC) (RC_VER1+0x047)
279 #define TPM_RC_NV_LOCKED (TPM_RC) (RC_VER1+0x048)
280 #define TPM_RC_NV_AUTHORIZATION (TPM_RC) (RC_VER1+0x049)
281 #define TPM_RC_NV_UNINITIALIZED (TPM_RC) (RC_VER1+0x04A)
282 #define TPM_RC_NV_SPACE (TPM_RC) (RC_VER1+0x04B)
283 #define TPM_RC_NV_DEFINED (TPM_RC) (RC_VER1+0x04C)
284 #define TPM_RC_BAD_CONTEXT (TPM_RC) (RC_VER1+0x050)
285 #define TPM_RC_CPHASH (TPM_RC) (RC_VER1+0x051)
286 #define TPM_RC_PARENT (TPM_RC) (RC_VER1+0x052)
287 #define TPM_RC_NEEDS_TEST (TPM_RC) (RC_VER1+0x053)
288 #define TPM_RC_NO_RESULT (TPM_RC) (RC_VER1+0x054)
289 #define TPM_RC_SENSITIVE (TPM_RC) (RC_VER1+0x055)
290 #define RC_MAX_FM0 (TPM_RC) (RC_VER1+0x07F)
291 #define RC_FMT1 (TPM_RC) (0x080)
292 #define TPM_RC_ASYMMETRIC (TPM_RC) (RC_FMT1+0x001)
293 #define TPM_RCS_ASYMMETRIC (TPM_RC) (RC_FMT1+0x001)
294 #define TPM_RC_ATTRIBUTES (TPM_RC) (RC_FMT1+0x002)
295 #define TPM_RCS_ATTRIBUTES (TPM_RC) (RC_FMT1+0x002)
296 #define TPM_RC_HASH (TPM_RC) (RC_FMT1+0x003)
297 #define TPM_RCS_HASH (TPM_RC) (RC_FMT1+0x003)
298 #define TPM_RC_VALUE (TPM_RC) (RC_FMT1+0x004)

```

```

299 #define TPM_RCS_VALUE (TPM_RC) (RC_FMT1+0x004)
300 #define TPM_RC_HIERARCHY (TPM_RC) (RC_FMT1+0x005)
301 #define TPM_RCS_HIERARCHY (TPM_RC) (RC_FMT1+0x005)
302 #define TPM_RC_KEY_SIZE (TPM_RC) (RC_FMT1+0x007)
303 #define TPM_RCS_KEY_SIZE (TPM_RC) (RC_FMT1+0x007)
304 #define TPM_RC_MGF (TPM_RC) (RC_FMT1+0x008)
305 #define TPM_RCS_MGF (TPM_RC) (RC_FMT1+0x008)
306 #define TPM_RC_MODE (TPM_RC) (RC_FMT1+0x009)
307 #define TPM_RCS_MODE (TPM_RC) (RC_FMT1+0x009)
308 #define TPM_RC_TYPE (TPM_RC) (RC_FMT1+0x00A)
309 #define TPM_RCS_TYPE (TPM_RC) (RC_FMT1+0x00A)
310 #define TPM_RC_HANDLE (TPM_RC) (RC_FMT1+0x00B)
311 #define TPM_RCS_HANDLE (TPM_RC) (RC_FMT1+0x00B)
312 #define TPM_RC_KDF (TPM_RC) (RC_FMT1+0x00C)
313 #define TPM_RCS_KDF (TPM_RC) (RC_FMT1+0x00C)
314 #define TPM_RC_RANGE (TPM_RC) (RC_FMT1+0x00D)
315 #define TPM_RCS_RANGE (TPM_RC) (RC_FMT1+0x00D)
316 #define TPM_RC_AUTH_FAIL (TPM_RC) (RC_FMT1+0x00E)
317 #define TPM_RCS_AUTH_FAIL (TPM_RC) (RC_FMT1+0x00E)
318 #define TPM_RC_NONCE (TPM_RC) (RC_FMT1+0x00F)
319 #define TPM_RCS_NONCE (TPM_RC) (RC_FMT1+0x00F)
320 #define TPM_RC_PP (TPM_RC) (RC_FMT1+0x010)
321 #define TPM_RCS_PP (TPM_RC) (RC_FMT1+0x010)
322 #define TPM_RC_SCHEME (TPM_RC) (RC_FMT1+0x012)
323 #define TPM_RCS_SCHEME (TPM_RC) (RC_FMT1+0x012)
324 #define TPM_RC_SIZE (TPM_RC) (RC_FMT1+0x015)
325 #define TPM_RCS_SIZE (TPM_RC) (RC_FMT1+0x015)
326 #define TPM_RC_SYMMETRIC (TPM_RC) (RC_FMT1+0x016)
327 #define TPM_RCS_SYMMETRIC (TPM_RC) (RC_FMT1+0x016)
328 #define TPM_RC_TAG (TPM_RC) (RC_FMT1+0x017)
329 #define TPM_RCS_TAG (TPM_RC) (RC_FMT1+0x017)
330 #define TPM_RC_SELECTOR (TPM_RC) (RC_FMT1+0x018)
331 #define TPM_RCS_SELECTOR (TPM_RC) (RC_FMT1+0x018)
332 #define TPM_RC_INSUFFICIENT (TPM_RC) (RC_FMT1+0x01A)
333 #define TPM_RCS_INSUFFICIENT (TPM_RC) (RC_FMT1+0x01A)
334 #define TPM_RC_SIGNATURE (TPM_RC) (RC_FMT1+0x01B)
335 #define TPM_RCS_SIGNATURE (TPM_RC) (RC_FMT1+0x01B)
336 #define TPM_RC_KEY (TPM_RC) (RC_FMT1+0x01C)
337 #define TPM_RCS_KEY (TPM_RC) (RC_FMT1+0x01C)
338 #define TPM_RC_POLICY_FAIL (TPM_RC) (RC_FMT1+0x01D)
339 #define TPM_RCS_POLICY_FAIL (TPM_RC) (RC_FMT1+0x01D)
340 #define TPM_RC_INTEGRITY (TPM_RC) (RC_FMT1+0x01F)
341 #define TPM_RCS_INTEGRITY (TPM_RC) (RC_FMT1+0x01F)
342 #define TPM_RC_TICKET (TPM_RC) (RC_FMT1+0x020)
343 #define TPM_RCS_TICKET (TPM_RC) (RC_FMT1+0x020)
344 #define TPM_RC_RESERVED_BITS (TPM_RC) (RC_FMT1+0x021)
345 #define TPM_RCS_RESERVED_BITS (TPM_RC) (RC_FMT1+0x021)
346 #define TPM_RC_BAD_AUTH (TPM_RC) (RC_FMT1+0x022)
347 #define TPM_RCS_BAD_AUTH (TPM_RC) (RC_FMT1+0x022)
348 #define TPM_RC_EXPIRED (TPM_RC) (RC_FMT1+0x023)
349 #define TPM_RCS_EXPIRED (TPM_RC) (RC_FMT1+0x023)
350 #define TPM_RC_POLICY_CC (TPM_RC) (RC_FMT1+0x024)
351 #define TPM_RCS_POLICY_CC (TPM_RC) (RC_FMT1+0x024)
352 #define TPM_RC_BINDING (TPM_RC) (RC_FMT1+0x025)
353 #define TPM_RCS_BINDING (TPM_RC) (RC_FMT1+0x025)
354 #define TPM_RC_CURVE (TPM_RC) (RC_FMT1+0x026)
355 #define TPM_RCS_CURVE (TPM_RC) (RC_FMT1+0x026)
356 #define TPM_RC_ECC_POINT (TPM_RC) (RC_FMT1+0x027)
357 #define TPM_RCS_ECC_POINT (TPM_RC) (RC_FMT1+0x027)
358 #define RC_WARN (TPM_RC) (0x900)
359 #define TPM_RC_CONTEXT_GAP (TPM_RC) (RC_WARN+0x001)
360 #define TPM_RC_OBJECT_MEMORY (TPM_RC) (RC_WARN+0x002)
361 #define TPM_RC_SESSION_MEMORY (TPM_RC) (RC_WARN+0x003)
362 #define TPM_RC_MEMORY (TPM_RC) (RC_WARN+0x004)
363 #define TPM_RC_SESSION_HANDLES (TPM_RC) (RC_WARN+0x005)
364 #define TPM_RC_OBJECT_HANDLES (TPM_RC) (RC_WARN+0x006)

```



```

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

```

Table 2:17 - Definition of TPM_CLOCK_ADJUST Constants

```

407 typedef INT8 TPM_CLOCK_ADJUST;
408 #define TYPE_OF_TPM_CLOCK_ADJUST UINT8
409 #define TPM_CLOCK_COARSE_SLOWER (TPM_CLOCK_ADJUST) (-3)
410 #define TPM_CLOCK_MEDIUM_SLOWER (TPM_CLOCK_ADJUST) (-2)
411 #define TPM_CLOCK_FINE_SLOWER (TPM_CLOCK_ADJUST) (-1)
412 #define TPM_CLOCK_NO_CHANGE (TPM_CLOCK_ADJUST) (0)
413 #define TPM_CLOCK_FINE_FASTER (TPM_CLOCK_ADJUST) (1)
414 #define TPM_CLOCK_MEDIUM_FASTER (TPM_CLOCK_ADJUST) (2)
415 #define TPM_CLOCK_COARSE_FASTER (TPM_CLOCK_ADJUST) (3)

```

Table 2:18 - Definition of TPM_EO Constants

```

416 typedef UINT16 TPM_EO;
417 #define TYPE_OF_TPM_EO UINT16
418 #define TPM_EO_EQ (TPM_EO) (0x0000)
419 #define TPM_EO_NEQ (TPM_EO) (0x0001)
420 #define TPM_EO_SIGNED_GT (TPM_EO) (0x0002)
421 #define TPM_EO_UNSIGNED_GT (TPM_EO) (0x0003)
422 #define TPM_EO_SIGNED_LT (TPM_EO) (0x0004)
423 #define TPM_EO_UNSIGNED_LT (TPM_EO) (0x0005)

```

```

424 #define TPM_EO_SIGNED_GE      (TPM_EO) (0x0006)
425 #define TPM_EO_UNSIGNED_GE    (TPM_EO) (0x0007)
426 #define TPM_EO_SIGNED_LE      (TPM_EO) (0x0008)
427 #define TPM_EO_UNSIGNED_LE    (TPM_EO) (0x0009)
428 #define TPM_EO_BITSET         (TPM_EO) (0x000A)
429 #define TPM_EO_BITCLEAR       (TPM_EO) (0x000B)

```

Table 2:19 - Definition of TPM_ST Constants

```

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

```

Table 2:20 - Definition of TPM_SU Constants

```

450 typedef UINT16          TPM_SU;
451 #define TYPE_OF TPM_SU  UINT16
452 #define TPM_SU_CLEAR     (TPM_SU) (0x0000)
453 #define TPM_SU_STATE     (TPM_SU) (0x0001)

```

Table 2:21 - Definition of TPM_SE Constants

```

454 typedef UINT8          TPM_SE;
455 #define TYPE_OF TPM_SE  UINT8
456 #define TPM_SE_HMAC      (TPM_SE) (0x00)
457 #define TPM_SE_POLICY     (TPM_SE) (0x01)
458 #define TPM_SE_TRIAL     (TPM_SE) (0x03)

```

Table 2:22 - Definition of TPM_CAP Constants

```

459 typedef UINT32          TPM_CAP;
460 #define TYPE_OF TPM_CAP  UINT32
461 #define TPM_CAP_FIRST      (TPM_CAP) (0x00000000)
462 #define TPM_CAP_ALGS       (TPM_CAP) (0x00000000)
463 #define TPM_CAP_HANDLES    (TPM_CAP) (0x00000001)
464 #define TPM_CAP_COMMANDS   (TPM_CAP) (0x00000002)
465 #define TPM_CAP_PP_COMMANDS (TPM_CAP) (0x00000003)
466 #define TPM_CAP_AUDIT_COMMANDS (TPM_CAP) (0x00000004)
467 #define TPM_CAP_PCRS       (TPM_CAP) (0x00000005)
468 #define TPM_CAP_TPM_PROPERTIES (TPM_CAP) (0x00000006)
469 #define TPM_CAP_PCR_PROPERTIES (TPM_CAP) (0x00000007)
470 #define TPM_CAP_ECC_CURVES (TPM_CAP) (0x00000008)
471 #define TPM_CAP_AUTH_POLICIES (TPM_CAP) (0x00000009)
472 #define TPM_CAP_LAST       (TPM_CAP) (0x00000009)
473 #define TPM_CAP_VENDOR_PROPERTY (TPM_CAP) (0x00000100)

```

Table 2:23 - Definition of TPM_PT Constants

```

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

```

```

538 #define TPM_PT_ALGORITHM_SET (TPM_PT) (PT_VAR+12)
539 #define TPM_PT_LOADED_CURVES (TPM_PT) (PT_VAR+13)
540 #define TPM_PT_LOCKOUT_COUNTER (TPM_PT) (PT_VAR+14)
541 #define TPM_PT_MAX_AUTH_FAIL (TPM_PT) (PT_VAR+15)
542 #define TPM_PT_LOCKOUT_INTERVAL (TPM_PT) (PT_VAR+16)
543 #define TPM_PT_LOCKOUT_RECOVERY (TPM_PT) (PT_VAR+17)
544 #define TPM_PT_NV_WRITE_RECOVERY (TPM_PT) (PT_VAR+18)
545 #define TPM_PT_AUDIT_COUNTER_0 (TPM_PT) (PT_VAR+19)
546 #define TPM_PT_AUDIT_COUNTER_1 (TPM_PT) (PT_VAR+20)

```

Table 2:24 - Definition of TPM_PT_PCR Constants

```

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

```

Table 2:25 - Definition of TPM_PS Constants

```

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

```

Table 2:26 - Definition of Types for Handles

```

584 typedef UINT32 TPM_HANDLE;
585 #define TYPE_OF_TPM_HANDLE UINT32

```

Table 2:27 - Definition of TPM_HT Constants

```

586 typedef UINT8 TPM_HT;
587 #define TYPE_OF_TPM_HT UINT8
588 #define TPM_HT_PCR (TPM_HT) (0x00)
589 #define TPM_HT_NV_INDEX (TPM_HT) (0x01)

```

```

590 #define TPM_HT_HMAC_SESSION      (TPM_HT) (0x02)
591 #define TPM_HT_LOADED_SESSION    (TPM_HT) (0x02)
592 #define TPM_HT_POLICY_SESSION    (TPM_HT) (0x03)
593 #define TPM_HT_SAVED_SESSION     (TPM_HT) (0x03)
594 #define TPM_HT_PERMANENT         (TPM_HT) (0x40)
595 #define TPM_HT_TRANSIENT         (TPM_HT) (0x80)
596 #define TPM_HT_PERSISTENT        (TPM_HT) (0x81)
597 #define TPM_HT_AC                (TPM_HT) (0x90)

```

Table 2:28 - Definition of TPM_RH Constants

```

598 typedef TPM_HANDLE      TPM_RH;
599 #define TPM_RH_FIRST     (TPM_RH) (0x40000000)
600 #define TPM_RH_SRK       (TPM_RH) (0x40000000)
601 #define TPM_RH_OWNER     (TPM_RH) (0x40000001)
602 #define TPM_RH_REVOKE    (TPM_RH) (0x40000002)
603 #define TPM_RH_TRANSPORT (TPM_RH) (0x40000003)
604 #define TPM_RH_OPERATOR  (TPM_RH) (0x40000004)
605 #define TPM_RH_ADMIN     (TPM_RH) (0x40000005)
606 #define TPM_RH_EK        (TPM_RH) (0x40000006)
607 #define TPM_RH_NULL      (TPM_RH) (0x40000007)
608 #define TPM_RH_UNASSIGNED (TPM_RH) (0x40000008)
609 #define TPM_RS_PW        (TPM_RH) (0x40000009)
610 #define TPM_RH_LOCKOUT   (TPM_RH) (0x4000000A)
611 #define TPM_RH_ENDORSEMENT (TPM_RH) (0x4000000B)
612 #define TPM_RH_PLATFORM  (TPM_RH) (0x4000000C)
613 #define TPM_RH_PLATFORM_NV (TPM_RH) (0x4000000D)
614 #define TPM_RH_AUTH_00   (TPM_RH) (0x40000010)
615 #define TPM_RH_AUTH_FF   (TPM_RH) (0x4000010F)
616 #define TPM_RH_LAST      (TPM_RH) (0x4000010F)

```

Table 2:29 - Definition of TPM_HC Constants

```

617 typedef TPM_HANDLE      TPM_HC;
618 #define HR_HANDLE_MASK   (TPM_HC) (0x00FFFFFF)
619 #define HR_RANGE_MASK    (TPM_HC) (0xFF000000)
620 #define HR_SHIFT         (TPM_HC) (24)
621 #define HR_PCR           (TPM_HC) ((TPM_HT_PCR<<HR_SHIFT))
622 #define HR_HMAC_SESSION  (TPM_HC) ((TPM_HT_HMAC_SESSION<<HR_SHIFT))
623 #define HR_POLICY_SESSION (TPM_HC) ((TPM_HT_POLICY_SESSION<<HR_SHIFT))
624 #define HR_TRANSIENT     (TPM_HC) ((TPM_HT_TRANSIENT<<HR_SHIFT))
625 #define HR_PERSISTENT    (TPM_HC) ((TPM_HT_PERSISTENT<<HR_SHIFT))
626 #define HR_NV_INDEX      (TPM_HC) ((TPM_HT_NV_INDEX<<HR_SHIFT))
627 #define HR_PERMANENT     (TPM_HC) ((TPM_HT_PERMANENT<<HR_SHIFT))
628 #define PCR_FIRST        (TPM_HC) ((HR_PCR+0))
629 #define PCR_LAST         (TPM_HC) ((PCR_FIRST+IMPLEMENTATION_PCR-1))
630 #define HMAC_SESSION_FIRST (TPM_HC) ((HR_HMAC_SESSION+0))
631 #define HMAC_SESSION_LAST (TPM_HC) ((HMAC_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
632 #define LOADED_SESSION_FIRST (TPM_HC) (HMAC_SESSION_FIRST)
633 #define LOADED_SESSION_LAST (TPM_HC) (HMAC_SESSION_LAST)
634 #define POLICY_SESSION_FIRST (TPM_HC) ((HR_POLICY_SESSION+0))
635 #define POLICY_SESSION_LAST \
636     (TPM_HC) ((POLICY_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
637 #define TRANSIENT_FIRST  (TPM_HC) ((HR_TRANSIENT+0))
638 #define ACTIVE_SESSION_FIRST (TPM_HC) (POLICY_SESSION_FIRST)
639 #define ACTIVE_SESSION_LAST (TPM_HC) (POLICY_SESSION_LAST)
640 #define TRANSIENT_LAST   (TPM_HC) ((TRANSIENT_FIRST+MAX_LOADED_OBJECTS-1))
641 #define PERSISTENT_FIRST (TPM_HC) ((HR_PERSISTENT+0))
642 #define PERSISTENT_LAST  (TPM_HC) ((PERSISTENT_FIRST+0x00FFFFFF))
643 #define PLATFORM_PERSISTENT (TPM_HC) ((PERSISTENT_FIRST+0x00800000))
644 #define NV_INDEX_FIRST    (TPM_HC) ((HR_NV_INDEX+0))
645 #define NV_INDEX_LAST     (TPM_HC) ((NV_INDEX_FIRST+0x00FFFFFF))
646 #define PERMANENT_FIRST   (TPM_HC) (TPM_RH_FIRST)
647 #define PERMANENT_LAST    (TPM_HC) (TPM_RH_LAST)
648 #define HR_NV_AC          (TPM_HC) (((TPM_HT_NV_INDEX<<HR_SHIFT)+0xD00000))

```



```

649 #define NV_AC_FIRST          (TPM_HC) ((HR_NV_AC+0))
650 #define NV_AC_LAST          (TPM_HC) ((HR_NV_AC+0x0000FFFF))
651 #define HR_AC                (TPM_HC) ((TPM_HT_AC<<HR_SHIFT))
652 #define AC_FIRST            (TPM_HC) ((HR_AC+0))
653 #define AC_LAST             (TPM_HC) ((HR_AC+0x0000FFFF))
654 #define TYPE_OF_TPMA_ALGORITHM  UINT32
655 #define TPMA_ALGORITHM_TO_UINT32(a)  (*(UINT32 *)&(a))
656 #define UINT32_TO_TPMA_ALGORITHM(a)  (*(TPMA_ALGORITHM *)&(a))
657 #define TPMA_ALGORITHM_TO_BYTE_ARRAY(i, a) \
658     UINT32_TO_BYTE_ARRAY((TPMA_ALGORITHM_TO_UINT32(i)), (a))
659 #define BYTE_ARRAY_TO_TPMA_ALGORITHM(i, a) \
660     {UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
661     i = UINT32_TO_TPMA_ALGORITHM(x); \
662     }
663 #if USE_BIT_FIELD_STRUCTURES
664 typedef struct TPMA_ALGORITHM { // Table 2:30
665     unsigned asymmetric : 1;
666     unsigned symmetric : 1;
667     unsigned hash : 1;
668     unsigned object : 1;
669     unsigned Reserved_bits_at_4 : 4;
670     unsigned signing : 1;
671     unsigned encrypting : 1;
672     unsigned method : 1;
673     unsigned Reserved_bits_at_11 : 21;
674 } TPMA_ALGORITHM; /* Bits */

```

This is the initializer for a TPMA_ALGORITHM structure

```

675 #define TPMA_ALGORITHM_INITIALIZER( \
676     asymmetric, symmetric, hash, object, bits_at_4, \
677     signing, encrypting, method, bits_at_11) \
678     {asymmetric, symmetric, hash, object, bits_at_4, \
679     signing, encrypting, method, bits_at_11}
680 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:30 TPMA_ALGORITHM using bit masking

```

681 typedef UINT32 TPMA_ALGORITHM;
682 #define TYPE_OF_TPMA_ALGORITHM  UINT32
683 #define TPMA_ALGORITHM_asymmetric ((TPMA_ALGORITHM)1 << 0)
684 #define TPMA_ALGORITHM_symmetric ((TPMA_ALGORITHM)1 << 1)
685 #define TPMA_ALGORITHM_hash ((TPMA_ALGORITHM)1 << 2)
686 #define TPMA_ALGORITHM_object ((TPMA_ALGORITHM)1 << 3)
687 #define TPMA_ALGORITHM_signing ((TPMA_ALGORITHM)1 << 8)
688 #define TPMA_ALGORITHM_encrypting ((TPMA_ALGORITHM)1 << 9)
689 #define TPMA_ALGORITHM_method ((TPMA_ALGORITHM)1 << 10)

```

This is the initializer for a TPMA_ALGORITHM bit array.

```

690 #define TPMA_ALGORITHM_INITIALIZER( \
691     asymmetric, symmetric, hash, object, bits_at_4, \
692     signing, encrypting, method, bits_at_11) \
693     {(asymmetric << 0) + (symmetric << 1) + (hash << 2) + \
694     (object << 3) + (signing << 8) + (encrypting << 9) + \
695     (method << 10)}
696 #endif // USE_BIT_FIELD_STRUCTURES
697 #define TYPE_OF_TPMA_OBJECT  UINT32
698 #define TPMA_OBJECT_TO_UINT32(a)  (*(UINT32 *)&(a))
699 #define UINT32_TO_TPMA_OBJECT(a)  (*(TPMA_OBJECT *)&(a))
700 #define TPMA_OBJECT_TO_BYTE_ARRAY(i, a) \
701     UINT32_TO_BYTE_ARRAY((TPMA_OBJECT_TO_UINT32(i)), (a))
702 #define BYTE_ARRAY_TO_TPMA_OBJECT(i, a) \
703     {UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_OBJECT(x); }

```

```

704 #if USE_BIT_FIELD_STRUCTURES
705 typedef struct TPMA_OBJECT { // Table 2:31
706     unsigned Reserved_bit_at_0 : 1;
707     unsigned fixedTPM : 1;
708     unsigned stClear : 1;
709     unsigned Reserved_bit_at_3 : 1;
710     unsigned fixedParent : 1;
711     unsigned sensitiveDataOrigin : 1;
712     unsigned userWithAuth : 1;
713     unsigned adminWithPolicy : 1;
714     unsigned Reserved_bits_at_8 : 2;
715     unsigned noDA : 1;
716     unsigned encryptedDuplication : 1;
717     unsigned Reserved_bits_at_12 : 4;
718     unsigned restricted : 1;
719     unsigned decrypt : 1;
720     unsigned sign : 1;
721     unsigned x509sign : 1;
722     unsigned Reserved_bits_at_20 : 12;
723 } TPMA_OBJECT; /* Bits */

```

This is the initializer for a TPMA_OBJECT structure

```

724 #define TPMA_OBJECT_INITIALIZER( \
725     bit_at_0, fixedtpm, stclear, \
726     bit_at_3, fixedparent, sensitivedataorigin, \
727     userwithauth, adminwithpolicy, bits_at_8, \
728     noda, encryptedduplication, bits_at_12, \
729     restricted, decrypt, sign, \
730     x509sign, bits_at_20) \
731 {bit_at_0, fixedtpm, stclear, \
732     bit_at_3, fixedparent, sensitivedataorigin, \
733     userwithauth, adminwithpolicy, bits_at_8, \
734     noda, encryptedduplication, bits_at_12, \
735     restricted, decrypt, sign, \
736     x509sign, bits_at_20}
737 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:31 TPMA_OBJECT using bit masking

```

738 typedef UINT32 TPMA_OBJECT;
739 #define TYPE_OF_TPMA_OBJECT UINT32
740 #define TPMA_OBJECT_fixedTPM ((TPMA_OBJECT)1 << 1)
741 #define TPMA_OBJECT_stClear ((TPMA_OBJECT)1 << 2)
742 #define TPMA_OBJECT_fixedParent ((TPMA_OBJECT)1 << 4)
743 #define TPMA_OBJECT_sensitiveDataOrigin ((TPMA_OBJECT)1 << 5)
744 #define TPMA_OBJECT_userWithAuth ((TPMA_OBJECT)1 << 6)
745 #define TPMA_OBJECT_adminWithPolicy ((TPMA_OBJECT)1 << 7)
746 #define TPMA_OBJECT_noDA ((TPMA_OBJECT)1 << 10)
747 #define TPMA_OBJECT_encryptedDuplication ((TPMA_OBJECT)1 << 11)
748 #define TPMA_OBJECT_restricted ((TPMA_OBJECT)1 << 16)
749 #define TPMA_OBJECT_decrypt ((TPMA_OBJECT)1 << 17)
750 #define TPMA_OBJECT_sign ((TPMA_OBJECT)1 << 18)
751 #define TPMA_OBJECT_x509sign ((TPMA_OBJECT)1 << 19)

```

This is the initializer for a TPMA_OBJECT bit array.

```

752 #define TPMA_OBJECT_INITIALIZER( \
753     bit_at_0, fixedtpm, stclear, \
754     bit_at_3, fixedparent, sensitivedataorigin, \
755     userwithauth, adminwithpolicy, bits_at_8, \
756     noda, encryptedduplication, bits_at_12, \
757     restricted, decrypt, sign, \
758     x509sign, bits_at_20) \

```



```

759         {(fixedtpm << 1)                + (stclear << 2)                + \
760         (fixedparent << 4)              + (sensitiveorigin << 5)        + \
761         (userwithauth << 6)             + (adminwithpolicy << 7)      + \
762         (noda << 10)                    + (encryptedduplication << 11) + \
763         (restricted << 16)              + (decrypt << 17)           + \
764         (sign << 18)                    + (x509sign << 19)}
765 #endif // USE_BIT_FIELD_STRUCTURES
766 #define TYPE_OF_TPMA_SESSION            UINT8
767 #define TPMA_SESSION_TO_UINT8(a)        (*(UINT8 *)&(a))
768 #define UINT8_TO_TPMA_SESSION(a)        (*(TPMA_SESSION *)&(a))
769 #define TPMA_SESSION_TO_BYTE_ARRAY(i, a) \
770     UINT8_TO_BYTE_ARRAY((TPMA_SESSION_TO_UINT8(i)), (a))
771 #define BYTE_ARRAY_TO_TPMA_SESSION(i, a) \
772     { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_SESSION(x); }
773 #if USE_BIT_FIELD_STRUCTURES
774 typedef struct TPMA_SESSION {           // Table 2:32
775     unsigned    continueSession        : 1;
776     unsigned    auditExclusive         : 1;
777     unsigned    auditReset             : 1;
778     unsigned    Reserved_bits_at_3     : 2;
779     unsigned    decrypt                : 1;
780     unsigned    encrypt                : 1;
781     unsigned    audit                  : 1;
782 } TPMA_SESSION;                       /* Bits */

```

This is the initializer for a TPMA_SESSION structure

```

783 #define TPMA_SESSION_INITIALIZER(      \
784     continuesession, auditexclusive,    auditreset,    bits_at_3, \
785     decrypt, encrypt, audit)           \
786     {continuesession, auditexclusive,    auditreset,    bits_at_3, \
787     decrypt, encrypt, audit}
788 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:32 TPMA_SESSION using bit masking

```

789 typedef UINT8 TPMA_SESSION;
790 #define TYPE_OF_TPMA_SESSION            UINT8
791 #define TPMA_SESSION_continueSession    ((TPMA_SESSION)1 << 0)
792 #define TPMA_SESSION_auditExclusive     ((TPMA_SESSION)1 << 1)
793 #define TPMA_SESSION_auditReset         ((TPMA_SESSION)1 << 2)
794 #define TPMA_SESSION_decrypt            ((TPMA_SESSION)1 << 5)
795 #define TPMA_SESSION_encrypt            ((TPMA_SESSION)1 << 6)
796 #define TPMA_SESSION_audit              ((TPMA_SESSION)1 << 7)

```

This is the initializer for a TPMA_SESSION bit array.

```

797 #define TPMA_SESSION_INITIALIZER(      \
798     continuesession, auditexclusive,    auditreset,    bits_at_3, \
799     decrypt, encrypt, audit)           \
800     {(continuesession << 0) + (auditexclusive << 1) + \
801     (auditreset << 2) + (decrypt << 5) + \
802     (encrypt << 6) + (audit << 7)}
803 #endif // USE_BIT_FIELD_STRUCTURES
804 #define TYPE_OF_TPMA_LOCALITY            UINT8
805 #define TPMA_LOCALITY_TO_UINT8(a)        (*(UINT8 *)&(a))
806 #define UINT8_TO_TPMA_LOCALITY(a)        (*(TPMA_LOCALITY *)&(a))
807 #define TPMA_LOCALITY_TO_BYTE_ARRAY(i, a) \
808     UINT8_TO_BYTE_ARRAY((TPMA_LOCALITY_TO_UINT8(i)), (a))
809 #define BYTE_ARRAY_TO_TPMA_LOCALITY(i, a) \
810     { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_LOCALITY(x); }
811 #if USE_BIT_FIELD_STRUCTURES
812 typedef struct TPMA_LOCALITY {           // Table 2:33
813     unsigned    TPM_LOC_ZERO            : 1;

```

```

814     unsigned    TPM_LOC_ONE           : 1;
815     unsigned    TPM_LOC_TWO           : 1;
816     unsigned    TPM_LOC_THREE        : 1;
817     unsigned    TPM_LOC_FOUR         : 1;
818     unsigned    Extended              : 3;
819 } TPMA_LOCALITY;                                /* Bits */

```

This is the initializer for a TPMA_LOCALITY structure

```

820 #define TPMA_LOCALITY_INITIALIZER(          \
821     tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three, \
822     tpm_loc_four, extended)                \
823     {tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three, \
824     tpm_loc_four, extended}                \
825 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:33 TPMA_LOCALITY using bit masking

```

826 typedef UINT8                                TPMA_LOCALITY;
827 #define TYPE_OF_TPMA_LOCALITY                UINT8
828 #define TPMA_LOCALITY_TPM_LOC_ZERO           ((TPMA_LOCALITY)1 << 0)
829 #define TPMA_LOCALITY_TPM_LOC_ONE            ((TPMA_LOCALITY)1 << 1)
830 #define TPMA_LOCALITY_TPM_LOC_TWO            ((TPMA_LOCALITY)1 << 2)
831 #define TPMA_LOCALITY_TPM_LOC_THREE          ((TPMA_LOCALITY)1 << 3)
832 #define TPMA_LOCALITY_TPM_LOC_FOUR           ((TPMA_LOCALITY)1 << 4)
833 #define TPMA_LOCALITY_Extended_SHIFT        5
834 #define TPMA_LOCALITY_Extended               ((TPMA_LOCALITY)0x7 << 5)

```

This is the initializer for a TPMA_LOCALITY bit array.

```

835 #define TPMA_LOCALITY_INITIALIZER(          \
836     tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three, \
837     tpm_loc_four, extended)                \
838     {(tpm_loc_zero << 0) + (tpm_loc_one << 1) + (tpm_loc_two << 2) + \
839     (tpm_loc_three << 3) + (tpm_loc_four << 4) + (extended << 5)} \
840 #endif // USE_BIT_FIELD_STRUCTURES
841 #define TYPE_OF_TPMA_PERMANENT             UINT32
842 #define TPMA_PERMANENT_TO_UINT32(a)        (*(UINT32 *)&(a))
843 #define UINT32_TO_TPMA_PERMANENT(a)        (*(TPMA_PERMANENT *)&(a))
844 #define TPMA_PERMANENT_TO_BYTE_ARRAY(i, a) \
845     UINT32_TO_BYTE_ARRAY((TPMA_PERMANENT_TO_UINT32(i)), (a)) \
846 #define BYTE_ARRAY_TO_TPMA_PERMANENT(i, a) \
847     {UINT32 x = BYTE_ARRAY_TO_UINT32(a); \
848     i = UINT32_TO_TPMA_PERMANENT(x); \
849     } \
850 #if USE_BIT_FIELD_STRUCTURES
851 typedef struct TPMA_PERMANENT {             // Table 2:34
852     unsigned    ownerAuthSet              : 1;
853     unsigned    endorsementAuthSet        : 1;
854     unsigned    lockoutAuthSet            : 1;
855     unsigned    Reserved_bits_at_3        : 5;
856     unsigned    disableClear              : 1;
857     unsigned    inLockout                 : 1;
858     unsigned    tpmGeneratedEPS           : 1;
859     unsigned    Reserved_bits_at_11       : 21;
860 } TPMA_PERMANENT;                          /* Bits */

```

This is the initializer for a TPMA_PERMANENT structure

```

861 #define TPMA_PERMANENT_INITIALIZER(          \
862     ownerauthset, endorsementauthset, lockoutauthset, \
863     bits_at_3, disableclear, inlockout, \
864     tpmgeneratedeps, bits_at_11)            \
865     {ownerauthset, endorsementauthset, lockoutauthset, \

```

```

866         bits_at_3,          disableclear,          inlockout,          \
867         tpmgeneratedeps,    bits_at_11}
868 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:34 TPMA_PERMANENT using bit masking

```

869 typedef UINT32          TPMA_PERMANENT;
870 #define TYPE_OF_TPMA_PERMANENT  UINT32
871 #define TPMA_PERMANENT_ownerAuthSet  ((TPMA_PERMANENT)1 << 0)
872 #define TPMA_PERMANENT_endorsementAuthSet  ((TPMA_PERMANENT)1 << 1)
873 #define TPMA_PERMANENT_lockoutAuthSet  ((TPMA_PERMANENT)1 << 2)
874 #define TPMA_PERMANENT_disableClear  ((TPMA_PERMANENT)1 << 8)
875 #define TPMA_PERMANENT_inLockout  ((TPMA_PERMANENT)1 << 9)
876 #define TPMA_PERMANENT_tpmGeneratedEPS  ((TPMA_PERMANENT)1 << 10)

```

This is the initializer for a TPMA_PERMANENT bit array.

```

877 #define TPMA_PERMANENT_INITIALIZER(          \
878     ownerauthset,      endorsementauthset, lockoutauthset,          \
879     bits_at_3,          disableclear,          inlockout,          \
880     tpmgeneratedeps,    bits_at_11)          \
881     { (ownerauthset << 0)      + (endorsementauthset << 1) +          \
882       (lockoutauthset << 2)    + (disableclear << 8)      +          \
883       (inlockout << 9)        + (tpmgeneratedeps << 10)}          \
884 #endif // USE_BIT_FIELD_STRUCTURES
885 #define TYPE_OF_TPMA_STARTUP_CLEAR  UINT32
886 #define TPMA_STARTUP_CLEAR_TO_UINT32(a)  ((*((UINT32 *)&(a)))
887 #define UINT32_TO_TPMA_STARTUP_CLEAR(a)  ((*((TPMA_STARTUP_CLEAR *)&(a)))
888 #define TPMA_STARTUP_CLEAR_TO_BYTE_ARRAY(i, a)          \
889     UINT32_TO_BYTE_ARRAY((TPMA_STARTUP_CLEAR_TO_UINT32(i)), (a))
890 #define BYTE_ARRAY_TO_TPMA_STARTUP_CLEAR(i, a)          \
891     {UINT32 x = BYTE_ARRAY_TO_UINT32(a);          \
892       i = UINT32_TO_TPMA_STARTUP_CLEAR(x);          \
893     }
894 #if USE_BIT_FIELD_STRUCTURES
895 typedef struct TPMA_STARTUP_CLEAR {          // Table 2:35
896     unsigned    phEnable      : 1;
897     unsigned    shEnable      : 1;
898     unsigned    ehEnable      : 1;
899     unsigned    phEnableNV     : 1;
900     unsigned    Reserved_bits_at_4 : 27;
901     unsigned    orderly       : 1;
902 } TPMA_STARTUP_CLEAR;          /* Bits */

```

This is the initializer for a TPMA_STARTUP_CLEAR structure

```

903 #define TPMA_STARTUP_CLEAR_INITIALIZER(          \
904     phenable, shenable, ehenable, phenablenv, bits_at_4, orderly)          \
905     {phenable, shenable, ehenable, phenablenv, bits_at_4, orderly}
906 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:35 TPMA_STARTUP_CLEAR using bit masking

```

907 typedef UINT32          TPMA_STARTUP_CLEAR;
908 #define TYPE_OF_TPMA_STARTUP_CLEAR  UINT32
909 #define TPMA_STARTUP_CLEAR_phEnable  ((TPMA_STARTUP_CLEAR)1 << 0)
910 #define TPMA_STARTUP_CLEAR_shEnable  ((TPMA_STARTUP_CLEAR)1 << 1)
911 #define TPMA_STARTUP_CLEAR_ehEnable  ((TPMA_STARTUP_CLEAR)1 << 2)
912 #define TPMA_STARTUP_CLEAR_phEnableNV  ((TPMA_STARTUP_CLEAR)1 << 3)
913 #define TPMA_STARTUP_CLEAR_orderly  ((TPMA_STARTUP_CLEAR)1 << 31)

```

This is the initializer for a TPMA_STARTUP_CLEAR bit array.

```

914 #define TPMA_STARTUP_CLEAR_INITIALIZER(
915     phenable, shenable, ehenable, phenablenv, bits_at_4, orderly)
916     {(phenable << 0) + (shenable << 1) + (ehenable << 2) +
917     (phenablenv << 3) + (orderly << 31)}
918 #endif // USE_BIT_FIELD_STRUCTURES
919 #define TYPE_OF_TPMA_MEMORY UINT32
920 #define TPMA_MEMORY_TO_UINT32(a)      (*(UINT32 *)&(a))
921 #define UINT32_TO_TPMA_MEMORY(a)      (*(TPMA_MEMORY *)&(a))
922 #define TPMA_MEMORY_TO_BYTE_ARRAY(i, a)
923     UINT32_TO_BYTE_ARRAY((TPMA_MEMORY_TO_UINT32(i)), (a))
924 #define BYTE_ARRAY_TO_TPMA_MEMORY(i, a)
925     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_MEMORY(x); }
926 #if USE_BIT_FIELD_STRUCTURES
927 typedef struct TPMA_MEMORY {
928     unsigned sharedRAM      : 1;
929     unsigned sharedNV       : 1;
930     unsigned objectCopiedToRam : 1;
931     unsigned Reserved_bits_at_3 : 29;
932 } TPMA_MEMORY;

```

// Table 2:36

/* Bits */

This is the initializer for a TPMA_MEMORY structure

```

933 #define TPMA_MEMORY_INITIALIZER(
934     sharedram, sharednv, objectcopiedtoram, bits_at_3)
935     {sharedram, sharednv, objectcopiedtoram, bits_at_3}
936 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:36 TPMA_MEMORY using bit masking

```

937 typedef UINT32      TPMA_MEMORY;
938 #define TYPE_OF_TPMA_MEMORY      UINT32
939 #define TPMA_MEMORY_sharedRAM    ((TPMA_MEMORY)1 << 0)
940 #define TPMA_MEMORY_sharedNV    ((TPMA_MEMORY)1 << 1)
941 #define TPMA_MEMORY_objectCopiedToRam ((TPMA_MEMORY)1 << 2)

```

This is the initializer for a TPMA_MEMORY bit array.

```

942 #define TPMA_MEMORY_INITIALIZER(
943     sharedram, sharednv, objectcopiedtoram, bits_at_3)
944     {(sharedram << 0) + (sharednv << 1) + (objectcopiedtoram << 2)}
945 #endif // USE_BIT_FIELD_STRUCTURES
946 #define TYPE_OF_TPMA_CC      UINT32
947 #define TPMA_CC_TO_UINT32(a)  (*(UINT32 *)&(a))
948 #define UINT32_TO_TPMA_CC(a)  (*(TPMA_CC *)&(a))
949 #define TPMA_CC_TO_BYTE_ARRAY(i, a)
950     UINT32_TO_BYTE_ARRAY((TPMA_CC_TO_UINT32(i)), (a))
951 #define BYTE_ARRAY_TO_TPMA_CC(i, a)
952     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_CC(x); }
953 #if USE_BIT_FIELD_STRUCTURES
954 typedef struct TPMA_CC {
955     unsigned commandIndex      : 16;
956     unsigned Reserved_bits_at_16 : 6;
957     unsigned nv                : 1;
958     unsigned extensive         : 1;
959     unsigned flushed           : 1;
960     unsigned cHandles          : 3;
961     unsigned rHandle           : 1;
962     unsigned v                 : 1;
963     unsigned Reserved_bits_at_30 : 2;
964 } TPMA_CC;

```

// Table 2:37

/* Bits */

This is the initializer for a TPMA_CC structure

```

965 #define TPMA_CC_INITIALIZER(

```

```

966         commandindex, bits_at_16, nv, extensive, flushed, \
967         chandles, rhandle, v, bits_at_30) \
968         {commandindex, bits_at_16, nv, extensive, flushed, \
969         chandles, rhandle, v, bits_at_30}
970 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:37 TPMA_CC using bit masking

```

971 typedef UINT32 TPMA_CC;
972 #define TYPE_OF_TPMA_CC UINT32
973 #define TPMA_CC_commandIndex_SHIFT 0
974 #define TPMA_CC_commandIndex ((TPMA_CC)0xffff << 0)
975 #define TPMA_CC_nv ((TPMA_CC)1 << 22)
976 #define TPMA_CC_extensive ((TPMA_CC)1 << 23)
977 #define TPMA_CC_flushed ((TPMA_CC)1 << 24)
978 #define TPMA_CC_chandles_SHIFT 25
979 #define TPMA_CC_chandles ((TPMA_CC)0x7 << 25)
980 #define TPMA_CC_rHandle ((TPMA_CC)1 << 28)
981 #define TPMA_CC_v ((TPMA_CC)1 << 29)

```

This is the initializer for a TPMA_CC bit array.

```

982 #define TPMA_CC_INITIALIZER( \
983     commandindex, bits_at_16, nv, extensive, flushed, \
984     chandles, rhandle, v, bits_at_30) \
985     { (commandindex << 0) + (nv << 22) + (extensive << 23) + \
986     (flushed << 24) + (chandles << 25) + (rhandle << 28) + \
987     (v << 29) }
988 #endif // USE_BIT_FIELD_STRUCTURES
989 #define TYPE_OF_TPMA_MODES UINT32
990 #define TPMA_MODES_TO_UINT32(a) (*(UINT32 *)&(a))
991 #define UINT32_TO_TPMA_MODES(a) (*(TPMA_MODES *)&(a))
992 #define TPMA_MODES_TO_BYTE_ARRAY(i, a) \
993     UINT32_TO_BYTE_ARRAY(TPMA_MODES_TO_UINT32(i), (a))
994 #define BYTE_ARRAY_TO_TPMA_MODES(i, a) \
995     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_MODES(x); }
996 #if USE_BIT_FIELD_STRUCTURES
997 typedef struct TPMA_MODES { // Table 2:38
998     unsigned FIPS_140_2 : 1;
999     unsigned Reserved_bits_at_1 : 31;
1000 } TPMA_MODES; /* Bits */

```

This is the initializer for a TPMA_MODES structure

```

1001 #define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {fips_140_2, bits_at_1}
1002 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:38 TPMA_MODES using bit masking

```

1003 typedef UINT32 TPMA_MODES;
1004 #define TYPE_OF_TPMA_MODES UINT32
1005 #define TPMA_MODES_FIPS_140_2 ((TPMA_MODES)1 << 0)

```

This is the initializer for a TPMA_MODES bit array.

```

1006 #define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {(fips_140_2 << 0)}
1007 #endif // USE_BIT_FIELD_STRUCTURES
1008 #define TYPE_OF_TPMA_X509_KEY_USAGE UINT32
1009 #define TPMA_X509_KEY_USAGE_TO_UINT32(a) (*(UINT32 *)&(a))
1010 #define UINT32_TO_TPMA_X509_KEY_USAGE(a) (*(TPMA_X509_KEY_USAGE *)&(a))
1011 #define TPMA_X509_KEY_USAGE_TO_BYTE_ARRAY(i, a) \
1012     UINT32_TO_BYTE_ARRAY(TPMA_X509_KEY_USAGE_TO_UINT32(i), (a))
1013 #define BYTE_ARRAY_TO_TPMA_X509_KEY_USAGE(i, a) \

```



```

1014         {UINT32 x = BYTE_ARRAY_TO_UINT32(a);
1015         i = UINT32_TO_TPMA_X509_KEY_USAGE(x);
1016         }
1017 #if USE_BIT_FIELD_STRUCTURES
1018 typedef struct TPMA_X509_KEY_USAGE {           // Table 2:39
1019     unsigned    digitalSignature      : 1;
1020     unsigned    nonrepudiation        : 1;
1021     unsigned    keyEncipherment       : 1;
1022     unsigned    dataEncipherment      : 1;
1023     unsigned    keyAgreement          : 1;
1024     unsigned    keyCertSign           : 1;
1025     unsigned    crlSign               : 1;
1026     unsigned    encipherOnly          : 1;
1027     unsigned    decipherOnly          : 1;
1028     unsigned    Reserved_bits_at_9    : 23;
1029 } TPMA_X509_KEY_USAGE;                       /* Bits */

```

This is the initializer for a TPMA_X509_KEY_USAGE structure

```

1030 #define TPMA_X509_KEY_USAGE_INITIALIZER(
1031     digitalsignature, nonrepudiation, keyencipherment,
1032     dataencipherment, keyagreement, keycertsign,
1033     crlsign, encipheronly, decipheronly,
1034     bits_at_9)
1035 {digitalsignature, nonrepudiation, keyencipherment,
1036     dataencipherment, keyagreement, keycertsign,
1037     crlsign, encipheronly, decipheronly,
1038     bits_at_9}
1039 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:39 TPMA_X509_KEY_USAGE using bit masking

```

1040 typedef UINT32 TPMA_X509_KEY_USAGE;
1041 #define TYPE_OF_TPMA_X509_KEY_USAGE UINT32
1042 #define TPMA_X509_KEY_USAGE_digitalSignature ((TPMA_X509_KEY_USAGE)1 << 0)
1043 #define TPMA_X509_KEY_USAGE_nonrepudiation ((TPMA_X509_KEY_USAGE)1 << 1)
1044 #define TPMA_X509_KEY_USAGE_keyEncipherment ((TPMA_X509_KEY_USAGE)1 << 2)
1045 #define TPMA_X509_KEY_USAGE_dataEncipherment ((TPMA_X509_KEY_USAGE)1 << 3)
1046 #define TPMA_X509_KEY_USAGE_keyAgreement ((TPMA_X509_KEY_USAGE)1 << 4)
1047 #define TPMA_X509_KEY_USAGE_keyCertSign ((TPMA_X509_KEY_USAGE)1 << 5)
1048 #define TPMA_X509_KEY_USAGE_crlSign ((TPMA_X509_KEY_USAGE)1 << 6)
1049 #define TPMA_X509_KEY_USAGE_encipherOnly ((TPMA_X509_KEY_USAGE)1 << 7)
1050 #define TPMA_X509_KEY_USAGE_decipherOnly ((TPMA_X509_KEY_USAGE)1 << 8)

```

This is the initializer for a TPMA_X509_KEY_USAGE bit array.

```

1051 #define TPMA_X509_KEY_USAGE_INITIALIZER(
1052     digitalsignature, nonrepudiation, keyencipherment,
1053     dataencipherment, keyagreement, keycertsign,
1054     crlsign, encipheronly, decipheronly,
1055     bits_at_9)
1056 { (digitalsignature << 0) + (nonrepudiation << 1) +
1057   (keyencipherment << 2) + (dataencipherment << 3) +
1058   (keyagreement << 4) + (keycertsign << 5) +
1059   (crlsign << 6) + (encipheronly << 7) +
1060   (decipheronly << 8) }
1061 #endif // USE_BIT_FIELD_STRUCTURES
1062 typedef BYTE TPMA_YES_NO; // Table 2:40 /* Interface */
1063 typedef TPM_HANDLE TPMA_DH_OBJECT; // Table 2:41 /* Interface */
1064 typedef TPM_HANDLE TPMA_DH_PARENT; // Table 2:42 /* Interface */
1065 typedef TPM_HANDLE TPMA_DH_PERSISTENT; // Table 2:43 /* Interface */
1066 typedef TPM_HANDLE TPMA_DH_ENTITY; // Table 2:44 /* Interface */
1067 typedef TPM_HANDLE TPMA_DH_PCR; // Table 2:45 /* Interface */
1068 typedef TPM_HANDLE TPMA_SH_AUTH_SESSION; // Table 2:46 /* Interface */

```

```

1069 typedef TPM_HANDLE TPMI_SH HMAC; // Table 2:47 /* Interface */
1070 typedef TPM_HANDLE TPMI_SH POLICY; // Table 2:48 /* Interface */
1071 typedef TPM_HANDLE TPMI_DH CONTEXT; // Table 2:49 /* Interface */
1072 typedef TPM_HANDLE TPMI_DH SAVED; // Table 2:50 /* Interface */
1073 typedef TPM_HANDLE TPMI_RH HIERARCHY; // Table 2:51 /* Interface */
1074 typedef TPM_HANDLE TPMI_RH ENABLES; // Table 2:52 /* Interface */
1075 typedef TPM_HANDLE TPMI_RH HIERARCHY_AUTH; // Table 2:53 /* Interface */
1076 typedef TPM_HANDLE TPMI_RH PLATFORM; // Table 2:54 /* Interface */
1077 typedef TPM_HANDLE TPMI_RH OWNER; // Table 2:55 /* Interface */
1078 typedef TPM_HANDLE TPMI_RH ENDORSEMENT; // Table 2:56 /* Interface */
1079 typedef TPM_HANDLE TPMI_RH PROVISION; // Table 2:57 /* Interface */
1080 typedef TPM_HANDLE TPMI_RH CLEAR; // Table 2:58 /* Interface */
1081 typedef TPM_HANDLE TPMI_RH NV_AUTH; // Table 2:59 /* Interface */
1082 typedef TPM_HANDLE TPMI_RH LOCKOUT; // Table 2:60 /* Interface */
1083 typedef TPM_HANDLE TPMI_RH NV_INDEX; // Table 2:61 /* Interface */
1084 typedef TPM_HANDLE TPMI_RH AC; // Table 2:62 /* Interface */
1085 typedef TPM_ALG_ID TPMI_ALG_HASH; // Table 2:63 /* Interface */
1086 typedef TPM_ALG_ID TPMI_ALG ASYM; // Table 2:64 /* Interface */
1087 typedef TPM_ALG_ID TPMI_ALG SYM; // Table 2:65 /* Interface */
1088 typedef TPM_ALG_ID TPMI_ALG SYM_OBJECT; // Table 2:66 /* Interface */
1089 typedef TPM_ALG_ID TPMI_ALG SYM_MODE; // Table 2:67 /* Interface */
1090 typedef TPM_ALG_ID TPMI_ALG KDF; // Table 2:68 /* Interface */
1091 typedef TPM_ALG_ID TPMI_ALG SIG_SCHEME; // Table 2:69 /* Interface */
1092 typedef TPM_ALG_ID TPMI_ECC_KEY_EXCHANGE; // Table 2:70 /* Interface */
1093 typedef TPM_ST TPMI_ST COMMAND_TAG; // Table 2:71 /* Interface */
1094 typedef TPM_ALG_ID TPMI_ALG MAC_SCHEME; // Table 2:72 /* Interface */
1095 typedef TPM_ALG_ID TPMI_ALG CIPHER_MODE; // Table 2:73 /* Interface */
1096 typedef BYTE TPMS_EMPTY; // Table 2:74
1097 typedef struct { // Table 2:75
1098     TPM_ALG_ID alg;
1099     TPMA_ALGORITHM attributes;
1100 } TPMS_ALGORITHM_DESCRIPTION; /* Structure */
1101 typedef union { // Table 2:76
1102     #if ALG_SHA1
1103     BYTE sha1[SHA1_DIGEST_SIZE];
1104     #endif // ALG_SHA1
1105     #if ALG_SHA256
1106     BYTE sha256[SHA256_DIGEST_SIZE];
1107     #endif // ALG_SHA256
1108     #if ALG_SHA384
1109     BYTE sha384[SHA384_DIGEST_SIZE];
1110     #endif // ALG_SHA384
1111     #if ALG_SHA512
1112     BYTE sha512[SHA512_DIGEST_SIZE];
1113     #endif // ALG_SHA512
1114     #if ALG_SM3_256
1115     BYTE sm3_256[SM3_256_DIGEST_SIZE];
1116     #endif // ALG_SM3_256
1117     #if ALG_SHA3_256
1118     BYTE sha3_256[SHA3_256_DIGEST_SIZE];
1119     #endif // ALG_SHA3_256
1120     #if ALG_SHA3_384
1121     BYTE sha3_384[SHA3_384_DIGEST_SIZE];
1122     #endif // ALG_SHA3_384
1123     #if ALG_SHA3_512
1124     BYTE sha3_512[SHA3_512_DIGEST_SIZE];
1125     #endif // ALG_SHA3_512
1126 } TPMU_HASH; /* Structure */
1127 typedef struct { // Table 2:77
1128     TPMI_ALG_HASH hashAlg;
1129     TPMU_HASH digest;
1130 } TPMT_HASH; /* Structure */
1131 typedef union { // Table 2:78
1132     struct {
1133         UINT16 size;
1134         BYTE buffer[sizeof(TPMU_HASH)];

```



```

1135     }                t;
1136     TPM2B            b;
1137 } TPM2B_DIGEST;
1138 typedef union {
1139     struct {
1140         UINT16        size;
1141         BYTE          buffer[sizeof(TPMT_HA)];
1142     }                t;
1143     TPM2B            b;
1144 } TPM2B_DATA;

```

/* Structure */
// Table 2:79

/* Structure */

Table 2:80 - Definition of Types for TPM2B_NONCE

```

1145 typedef TPM2B_DIGEST    TPM2B_NONCE;

```

Table 2:81 - Definition of Types for TPM2B_AUTH

```

1146 typedef TPM2B_DIGEST    TPM2B_AUTH;

```

Table 2:82 - Definition of Types for TPM2B_OPERAND

```

1147 typedef TPM2B_DIGEST    TPM2B_OPERAND;
1148 typedef union {
1149     struct {
1150         UINT16        size;
1151         BYTE          buffer[1024];
1152     }                t;
1153     TPM2B            b;
1154 } TPM2B_EVENT;
1155 typedef union {
1156     struct {
1157         UINT16        size;
1158         BYTE          buffer[MAX_DIGEST_BUFFER];
1159     }                t;
1160     TPM2B            b;
1161 } TPM2B_MAX_BUFFER;
1162 typedef union {
1163     struct {
1164         UINT16        size;
1165         BYTE          buffer[MAX_NV_BUFFER_SIZE];
1166     }                t;
1167     TPM2B            b;
1168 } TPM2B_MAX_NV_BUFFER;
1169 typedef union {
1170     struct {
1171         UINT16        size;
1172         BYTE          buffer[sizeof(UINT64)];
1173     }                t;
1174     TPM2B            b;
1175 } TPM2B_TIMEOUT;
1176 typedef union {
1177     struct {
1178         UINT16        size;
1179         BYTE          buffer[MAX_SYM_BLOCK_SIZE];
1180     }                t;
1181     TPM2B            b;
1182 } TPM2B_IV;
1183 typedef union {
1184     TPMT_HA          digest;
1185     TPM_HANDLE       handle;
1186 } TPMU_NAME;
1187 typedef union {
1188     struct {
1189         UINT16        size;

```

// Table 2:83

/* Structure */
// Table 2:84

/* Structure */
// Table 2:85

/* Structure */
// Table 2:86

/* Structure */
// Table 2:87

/* Structure */
// Table 2:88

/* Structure */
// Table 2:89

```

1190         BYTE                name[sizeof(TPMU_NAME)];
1191     }                        t;
1192     TPM2B                    b;
1193 } TPM2B_NAME;                /* Structure */
1194 typedef struct {              /* Table 2:90
1195     UINT8                    sizeofSelect;
1196     BYTE                     pcrSelect[PCR_SELECT_MAX];
1197 } TPMS_PCR_SELECT;           /* Structure */
1198 typedef struct {              /* Table 2:91
1199     TPMI_ALG_HASH            hash;
1200     UINT8                    sizeofSelect;
1201     BYTE                     pcrSelect[PCR_SELECT_MAX];
1202 } TPMS_PCR_SELECTION;        /* Structure */
1203 typedef struct {              /* Table 2:94
1204     TPM_ST                    tag;
1205     TPMI_RH_HIERARCHY         hierarchy;
1206     TPM2B_DIGEST              digest;
1207 } TPMT_TK_CREATION;          /* Structure */
1208 typedef struct {              /* Table 2:95
1209     TPM_ST                    tag;
1210     TPMI_RH_HIERARCHY         hierarchy;
1211     TPM2B_DIGEST              digest;
1212 } TPMT_TK_VERIFIED;          /* Structure */
1213 typedef struct {              /* Table 2:96
1214     TPM_ST                    tag;
1215     TPMI_RH_HIERARCHY         hierarchy;
1216     TPM2B_DIGEST              digest;
1217 } TPMT_TK_AUTH;              /* Structure */
1218 typedef struct {              /* Table 2:97
1219     TPM_ST                    tag;
1220     TPMI_RH_HIERARCHY         hierarchy;
1221     TPM2B_DIGEST              digest;
1222 } TPMT_TK_HASHCHECK;         /* Structure */
1223 typedef struct {              /* Table 2:98
1224     TPM_ALG_ID                alg;
1225     TPMA_ALGORITHM            algProperties;
1226 } TPMS_ALG_PROPERTY;         /* Structure */
1227 typedef struct {              /* Table 2:99
1228     TPM_PT                    property;
1229     UINT32                    value;
1230 } TPMS_TAGGED_PROPERTY;      /* Structure */
1231 typedef struct {              /* Table 2:100
1232     TPM_PT_PCR                tag;
1233     UINT8                    sizeofSelect;
1234     BYTE                     pcrSelect[PCR_SELECT_MAX];
1235 } TPMS_TAGGED_PCR_SELECT;    /* Structure */
1236 typedef struct {              /* Table 2:101
1237     TPM_HANDLE                handle;
1238     TPMT_HA                    policyHash;
1239 } TPMS_TAGGED_POLICY;        /* Structure */
1240 typedef struct {              /* Table 2:102
1241     UINT32                    count;
1242     TPM_CC                    commandCodes[MAX_CAP_CC];
1243 } TPML_CC;                    /* Structure */
1244 typedef struct {              /* Table 2:103
1245     UINT32                    count;
1246     TPMA_CC                    commandAttributes[MAX_CAP_CC];
1247 } TPML_CCA;                    /* Structure */
1248 typedef struct {              /* Table 2:104
1249     UINT32                    count;
1250     TPM_ALG_ID                algorithms[MAX_ALG_LIST_SIZE];
1251 } TPML_ALG;                    /* Structure */
1252 typedef struct {              /* Table 2:105
1253     UINT32                    count;
1254     TPM_HANDLE                handle[MAX_CAP_HANDLES];
1255 } TPML_HANDLE;                /* Structure */

```

```

1256 typedef struct {                                     // Table 2:106
1257     UINT32 count;
1258     TPM2B_DIGEST digests[8];
1259 } TPML_DIGEST;                                       /* Structure */
1260 typedef struct {                                     // Table 2:107
1261     UINT32 count;
1262     TPMT_HA digests[HASH_COUNT];
1263 } TPML_DIGEST_VALUES;                               /* Structure */
1264 typedef struct {                                     // Table 2:108
1265     UINT32 count;
1266     TPMS_PCR_SELECTION pcrSelections[HASH_COUNT];
1267 } TPML_PCR_SELECTION;                               /* Structure */
1268 typedef struct {                                     // Table 2:109
1269     UINT32 count;
1270     TPMS_ALG_PROPERTY algProperties[MAX_CAP_ALGS];
1271 } TPML_ALG_PROPERTY;                               /* Structure */
1272 typedef struct {                                     // Table 2:110
1273     UINT32 count;
1274     TPMS_TAGGED_PROPERTY tpmProperty[MAX_TPM_PROPERTIES];
1275 } TPML_TAGGED_TPM_PROPERTY;                         /* Structure */
1276 typedef struct {                                     // Table 2:111
1277     UINT32 count;
1278     TPMS_TAGGED_PCR_SELECT pcrProperty[MAX_PCR_PROPERTIES];
1279 } TPML_TAGGED_PCR_PROPERTY;                         /* Structure */
1280 typedef struct {                                     // Table 2:112
1281     UINT32 count;
1282     TPM_ECC_CURVE eccCurves[MAX_ECC_CURVES];
1283 } TPML_ECC_CURVE;                                  /* Structure */
1284 typedef struct {                                     // Table 2:113
1285     UINT32 count;
1286     TPMS_TAGGED_POLICY policies[MAX_TAGGED_POLICIES];
1287 } TPML_TAGGED_POLICY;                               /* Structure */
1288 typedef union {                                     // Table 2:114
1289     TPML_ALG_PROPERTY algorithms;
1290     TPML_HANDLE handles;
1291     TPML_CCA command;
1292     TPML_CC ppCommands;
1293     TPML_CC auditCommands;
1294     TPML_PCR_SELECTION assignedPCR;
1295     TPML_TAGGED_TPM_PROPERTY tpmProperties;
1296     TPML_TAGGED_PCR_PROPERTY pcrProperties;
1297 #if ALG_ECC
1298     TPML_ECC_CURVE eccCurves;
1299 #endif // ALG_ECC
1300     TPML_TAGGED_POLICY authPolicies;
1301 } TPMU_CAPABILITIES;                               /* Structure */
1302 typedef struct {                                     // Table 2:115
1303     TPM_CAP capability;
1304     TPMU_CAPABILITIES data;
1305 } TPMS_CAPABILITY_DATA;                             /* Structure */
1306 typedef struct {                                     // Table 2:116
1307     UINT64 clock;
1308     UINT32 resetCount;
1309     UINT32 restartCount;
1310     TPMI_YES_NO safe;
1311 } TPMS_CLOCK_INFO;                                 /* Structure */
1312 typedef struct {                                     // Table 2:117
1313     UINT64 time;
1314     TPMS_CLOCK_INFO clockInfo;
1315 } TPMS_TIME_INFO;                                  /* Structure */
1316 typedef struct {                                     // Table 2:118
1317     TPMS_TIME_INFO time;
1318     UINT64 firmwareVersion;
1319 } TPMS_TIME_ATTEST_INFO;                           /* Structure */
1320 typedef struct {                                     // Table 2:119
1321     TPM2B_NAME name;

```

```

1322     TPM2B_NAME                qualifiedName;
1323 } TPMS_CERTIFY_INFO;          /* Structure */
1324 typedef struct {               /* Table 2:120
1325     TPML_PCR_SELECTION        pcrSelect;
1326     TPM2B_DIGEST              pcrDigest;
1327 } TPMS_QUOTE_INFO;            /* Structure */
1328 typedef struct {               /* Table 2:121
1329     UINT64                    auditCounter;
1330     TPM_ALG_ID                digestAlg;
1331     TPM2B_DIGEST              auditDigest;
1332     TPM2B_DIGEST              commandDigest;
1333 } TPMS_COMMAND_AUDIT_INFO;    /* Structure */
1334 typedef struct {               /* Table 2:122
1335     TPMI_YES_NO               exclusiveSession;
1336     TPM2B_DIGEST              sessionDigest;
1337 } TPMS_SESSION_AUDIT_INFO;    /* Structure */
1338 typedef struct {               /* Table 2:123
1339     TPM2B_NAME                objectName;
1340     TPM2B_DIGEST              creationHash;
1341 } TPMS_CREATION_INFO;         /* Structure */
1342 typedef struct {               /* Table 2:124
1343     TPM2B_NAME                indexName;
1344     UINT16                    offset;
1345     TPM2B_MAX_NV_BUFFER       nvContents;
1346 } TPMS_NV_CERTIFY_INFO;       /* Structure */
1347 typedef struct {               /* Table 2:125
1348     TPM2B_NAME                indexName;
1349     TPM2B_DIGEST              nvDigest;
1350 } TPMS_NV_DIGEST_CERTIFY_INFO; /* Structure */
1351 typedef TPM_ST                TPMI_ST_ATTEST; /* Table 2:126 /* Interface */
1352 typedef union {                /* Table 2:127
1353     TPMS_CERTIFY_INFO          certify;
1354     TPMS_CREATION_INFO         creation;
1355     TPMS_QUOTE_INFO            quote;
1356     TPMS_COMMAND_AUDIT_INFO    commandAudit;
1357     TPMS_SESSION_AUDIT_INFO    sessionAudit;
1358     TPMS_TIME_ATTEST_INFO      time;
1359     TPMS_NV_CERTIFY_INFO       nv;
1360     TPMS_NV_DIGEST_CERTIFY_INFO nvDigest;
1361 } TPMU_ATTEST;                 /* Structure */
1362 typedef struct {               /* Table 2:128
1363     TPM_GENERATED              magic;
1364     TPMI_ST_ATTEST             type;
1365     TPM2B_NAME                 qualifiedSigner;
1366     TPM2B_DATA                 extraData;
1367     TPMS_CLOCK_INFO            clockInfo;
1368     UINT64                    firmwareVersion;
1369     TPMU_ATTEST                attested;
1370 } TPMS_ATTEST;                 /* Structure */
1371 typedef union {                /* Table 2:129
1372     struct {
1373         UINT16                size;
1374         BYTE                   attestationData[sizeof(TPMS_ATTEST)];
1375     } t;
1376     TPM2B                     b;
1377 } TPM2B_ATTEST;                 /* Structure */
1378 typedef struct {               /* Table 2:130
1379     TPMI_SH_AUTH_SESSION       sessionHandle;
1380     TPM2B_NONCE                nonce;
1381     TPMA_SESSION               sessionAttributes;
1382     TPM2B_AUTH                 hmac;
1383 } TPMS_AUTH_COMMAND;           /* Structure */
1384 typedef struct {               /* Table 2:131
1385     TPM2B_NONCE                nonce;
1386     TPMA_SESSION               sessionAttributes;
1387     TPM2B_AUTH                 hmac;

```

```

1388 } TPMS_AUTH_RESPONSE;
1389 typedef TPM_KEY_BITS TPMI_TDES_KEY_BITS; /* Structure */
1390 typedef TPM_KEY_BITS TPMI_AES_KEY_BITS; /* Table 2:132 */ /* Interface */
1391 typedef TPM_KEY_BITS TPMI_SM4_KEY_BITS; /* Table 2:132 */ /* Interface */
1392 typedef TPM_KEY_BITS TPMI_CAMELLIA_KEY_BITS; /* Table 2:132 */ /* Interface */
1393 typedef union {
1394     #if ALG_TDES
1395         TPMI_TDES_KEY_BITS tdes;
1396     #endif // ALG_TDES
1397     #if ALG_AES
1398         TPMI_AES_KEY_BITS aes;
1399     #endif // ALG_AES
1400     #if ALG_SM4
1401         TPMI_SM4_KEY_BITS sm4;
1402     #endif // ALG_SM4
1403     #if ALG_CAMELLIA
1404         TPMI_CAMELLIA_KEY_BITS camellia;
1405     #endif // ALG_CAMELLIA
1406     TPM_KEY_BITS sym;
1407     #if ALG_XOR
1408         TPMI_ALG_HASH xor;
1409     #endif // ALG_XOR
1410 } TPMU_SYM_KEY_BITS; /* Structure */
1411 typedef union { /* Table 2:134 */
1412     #if ALG_TDES
1413         TPMI_ALG_SYM_MODE tdes;
1414     #endif // ALG_TDES
1415     #if ALG_AES
1416         TPMI_ALG_SYM_MODE aes;
1417     #endif // ALG_AES
1418     #if ALG_SM4
1419         TPMI_ALG_SYM_MODE sm4;
1420     #endif // ALG_SM4
1421     #if ALG_CAMELLIA
1422         TPMI_ALG_SYM_MODE camellia;
1423     #endif // ALG_CAMELLIA
1424     TPMI_ALG_SYM_MODE sym;
1425 } TPMU_SYM_MODE; /* Structure */
1426 typedef struct { /* Table 2:136 */
1427     TPMI_ALG_SYM algorithm;
1428     TPMU_SYM_KEY_BITS keyBits;
1429     TPMU_SYM_MODE mode;
1430 } TPMT_SYM_DEF; /* Structure */
1431 typedef struct { /* Table 2:137 */
1432     TPMI_ALG_SYM_OBJECT algorithm;
1433     TPMU_SYM_KEY_BITS keyBits;
1434     TPMU_SYM_MODE mode;
1435 } TPMT_SYM_DEF_OBJECT; /* Structure */
1436 typedef union { /* Table 2:138 */
1437     struct {
1438         UINT16 size;
1439         BYTE buffer[MAX_SYM_KEY_BYTES];
1440     } t;
1441     TPM2B b;
1442 } TPM2B_SYM_KEY; /* Structure */
1443 typedef struct { /* Table 2:139 */
1444     TPMT_SYM_DEF_OBJECT sym;
1445 } TPMS_SYMCIPHER_PARMS; /* Structure */
1446 typedef union { /* Table 2:140 */
1447     struct {
1448         UINT16 size;
1449         BYTE buffer[LABEL_MAX_BUFFER];
1450     } t;
1451     TPM2B b;
1452 } TPM2B_LABEL; /* Structure */
1453 typedef struct { /* Table 2:141 */

```

```

1454     TPM2B_LABEL                label;
1455     TPM2B_LABEL                context;
1456 } TPMS_DERIVE;                /* Structure */
1457 typedef union {                /* Table 2:142
1458     struct {
1459         UINT16                size;
1460         BYTE                  buffer[sizeof(TPMS_DERIVE)];
1461     } t;
1462     TPM2B                    b;
1463 } TPM2B_DERIVE;                /* Structure */
1464 typedef union {                /* Table 2:143
1465     BYTE                  create[MAX_SYM_DATA];
1466     TPMS_DERIVE            derive;
1467 } TPMU_SENSITIVE_CREATE;       /* Structure */
1468 typedef union {                /* Table 2:144
1469     struct {
1470         UINT16                size;
1471         BYTE                  buffer[sizeof(TPMU_SENSITIVE_CREATE)];
1472     } t;
1473     TPM2B                    b;
1474 } TPM2B_SENSITIVE_DATA;        /* Structure */
1475 typedef struct {                /* Table 2:145
1476     TPM2B_AUTH              userAuth;
1477     TPM2B_SENSITIVE_DATA    data;
1478 } TPMS_SENSITIVE_CREATE;       /* Structure */
1479 typedef struct {                /* Table 2:146
1480     UINT16                size;
1481     TPMS_SENSITIVE_CREATE  sensitive;
1482 } TPM2B_SENSITIVE_CREATE;       /* Structure */
1483 typedef struct {                /* Table 2:147
1484     TPMI_ALG_HASH            hashAlg;
1485 } TPMS_SCHEME_HASH;            /* Structure */
1486 typedef struct {                /* Table 2:148
1487     TPMI_ALG_HASH            hashAlg;
1488     UINT16                count;
1489 } TPMS_SCHEME_ECDSA;           /* Structure */
1490 typedef TPM_ALG_ID          TPMI_ALG_KEYEDHASH_SCHEME;

```

Table 2:150 - Definition of Types for HMAC_SIG_SCHEME

```

1491 typedef TPMS_SCHEME_HASH      TPMS_SCHEME_HMAC;
1492 typedef struct {                /* Table 2:151
1493     TPMI_ALG_HASH            hashAlg;
1494     TPMI_ALG_KDF              kdf;
1495 } TPMS_SCHEME_XOR;             /* Structure */
1496 typedef union {                /* Table 2:152
1497     #if ALG_HMAC
1498         TPMS_SCHEME_HMAC      hmac;
1499     #endif // ALG_HMAC
1500     #if ALG_XOR
1501         TPMS_SCHEME_XOR        xor;
1502     #endif // ALG_XOR
1503 } TPMU_SCHEME_KEYEDHASH;       /* Structure */
1504 typedef struct {                /* Table 2:153
1505     TPMI_ALG_KEYEDHASH_SCHEME scheme;
1506     TPMU_SCHEME_KEYEDHASH      details;
1507 } TPMT_KEYEDHASH_SCHEME;       /* Structure */

```

Table 2:154 - Definition of Types for RSA Signature Schemes

```

1508 typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_RSASSA;
1509 typedef TPMS_SCHEME_HASH      TPMS_SIG_SCHEME_RSAPSS;

```

Table 2:155 - Definition of Types for ECC Signature Schemes


```

1510 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_ECDSA;
1511 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_SM2;
1512 typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_ECSCNORR;
1513 typedef TPMS_SCHEME_ECDA A TPMS_SIG_SCHEME_ECDA A;
1514 typedef union { // Table 2:156
1515     #if ALG_ECC
1516         TPMS_SIG_SCHEME_ECDA A ecda a;
1517     #endif // ALG_ECC
1518     #if ALG_RSASSA
1519         TPMS_SIG_SCHEME_RSASSA rsassa;
1520     #endif // ALG_RSASSA
1521     #if ALG_RSAPSS
1522         TPMS_SIG_SCHEME_RSAPSS rsapss;
1523     #endif // ALG_RSAPSS
1524     #if ALG_ECDSA
1525         TPMS_SIG_SCHEME_ECDSA ecdsa;
1526     #endif // ALG_ECDSA
1527     #if ALG_SM2
1528         TPMS_SIG_SCHEME_SM2 sm2;
1529     #endif // ALG_SM2
1530     #if ALG_ECSCNORR
1531         TPMS_SIG_SCHEME_ECSCNORR ecschnorr;
1532     #endif // ALG_ECSCNORR
1533     #if ALG_HMAC
1534         TPMS_SCHEME_HMAC hmac;
1535     #endif // ALG_HMAC
1536     TPMS_SCHEME_HASH any;
1537 } TPMU_SIG_SCHEME; /* Structure */
1538 typedef struct { // Table 2:157
1539     TPMI_ALG_SIG_SCHEME scheme;
1540     TPMU_SIG_SCHEME details;
1541 } TPMT_SIG_SCHEME; /* Structure */

```

Table 2:158 - Definition of Types for Encryption Schemes

```

1542 typedef TPMS_SCHEME_HASH TPMS_ENC_SCHEME_OAEP;
1543 typedef TPMS_EMPTY TPMS_ENC_SCHEME_RSAES;

```

Table 2:159 - Definition of Types for ECC Key Exchange

```

1544 typedef TPMS_SCHEME_HASH TPMS_KEY_SCHEME_ECDH;
1545 typedef TPMS_SCHEME_HASH TPMS_KEY_SCHEME_ECMQV;

```

Table 2:160 - Definition of Types for KDF Schemes

```

1546 typedef TPMS_SCHEME_HASH TPMS_SCHEME_MGF1;
1547 typedef TPMS_SCHEME_HASH TPMS_SCHEME_KDF1_SP800_56A;
1548 typedef TPMS_SCHEME_HASH TPMS_SCHEME_KDF2;
1549 typedef TPMS_SCHEME_HASH TPMS_SCHEME_KDF1_SP800_108;
1550 typedef union { // Table 2:161
1551     #if ALG_MGF1
1552         TPMS_SCHEME_MGF1 mgf1;
1553     #endif // ALG_MGF1
1554     #if ALG_KDF1_SP800_56A
1555         TPMS_SCHEME_KDF1_SP800_56A kdf1_sp800_56a;
1556     #endif // ALG_KDF1_SP800_56A
1557     #if ALG_KDF2
1558         TPMS_SCHEME_KDF2 kdf2;
1559     #endif // ALG_KDF2
1560     #if ALG_KDF1_SP800_108
1561         TPMS_SCHEME_KDF1_SP800_108 kdf1_sp800_108;
1562     #endif // ALG_KDF1_SP800_108
1563 } TPMU_KDF_SCHEME; /* Structure */
1564 typedef struct { // Table 2:162

```



```

1565     TPMI_ALG_KDF                scheme;
1566     TPMU_KDF_SCHEME             details;
1567 } TPMT_KDF_SCHEME;             /* Structure */
1568 typedef TPM_ALG_ID              TPMI_ALG_ASYNC_SCHEME; /* Table 2:163 */ /* Interface */
1569 typedef union {                 /* Table 2:164 */
1570     #if ALG_ECDH
1571         TPMS_KEY_SCHEME_ECDH     ecdh;
1572     #endif // ALG_ECDH
1573     #if ALG_ECMQV
1574         TPMS_KEY_SCHEME_ECMQV    ecmqv;
1575     #endif // ALG_ECMQV
1576     #if ALG_ECC
1577         TPMS_SIG_SCHEME_ECDA     ecdaa;
1578     #endif // ALG_ECC
1579     #if ALG_RSASSA
1580         TPMS_SIG_SCHEME_RSASSA   rsassa;
1581     #endif // ALG_RSASSA
1582     #if ALG_RSAPSS
1583         TPMS_SIG_SCHEME_RSAPSS   rsapss;
1584     #endif // ALG_RSAPSS
1585     #if ALG_ECDSA
1586         TPMS_SIG_SCHEME_ECDSA    ecdsa;
1587     #endif // ALG_ECDSA
1588     #if ALG_SM2
1589         TPMS_SIG_SCHEME_SM2      sm2;
1590     #endif // ALG_SM2
1591     #if ALG_ECSCHNORR
1592         TPMS_SIG_SCHEME_ECSCHNORR ecschnorr;
1593     #endif // ALG_ECSCHNORR
1594     #if ALG_RSAES
1595         TPMS_ENC_SCHEME_RSAES    rsaes;
1596     #endif // ALG_RSAES
1597     #if ALG_OAEP
1598         TPMS_ENC_SCHEME_OAEP     oaep;
1599     #endif // ALG_OAEP
1600     TPMS_SCHEME_HASH             anySig;
1601 } TPMU_ASYNC_SCHEME;           /* Structure */
1602 typedef struct {                /* Table 2:165 */
1603     TPMI_ALG_ASYNC_SCHEME        scheme;
1604     TPMU_ASYNC_SCHEME            details;
1605 } TPMT_ASYNC_SCHEME;           /* Structure */
1606 typedef TPM_ALG_ID              TPMI_ALG_RSA_SCHEME; /* Table 2:166 */ /* Interface */
1607 typedef struct {                /* Table 2:167 */
1608     TPMI_ALG_RSA_SCHEME          scheme;
1609     TPMU_ASYNC_SCHEME            details;
1610 } TPMT_RSA_SCHEME;             /* Structure */
1611 typedef TPM_ALG_ID              TPMI_ALG_RSA_DECRYPT; /* Table 2:168 */ /* Interface */
1612 typedef struct {                /* Table 2:169 */
1613     TPMI_ALG_RSA_DECRYPT          scheme;
1614     TPMU_ASYNC_SCHEME            details;
1615 } TPMT_RSA_DECRYPT;             /* Structure */
1616 typedef union {                 /* Table 2:170 */
1617     struct {
1618         UINT16                    size;
1619         BYTE                      buffer[MAX_RSA_KEY_BYTES];
1620     } t;
1621     TPM2B                        b;
1622 } TPM2B_PUBLIC_KEY_RSA;         /* Structure */
1623 typedef TPM_KEY_BITS            TPMI_RSA_KEY_BITS; /* Table 2:171 */ /* Interface */
1624 typedef union {                 /* Table 2:172 */
1625     struct {
1626         UINT16                    size;
1627         BYTE                      buffer[RSA_PRIVATE_SIZE];
1628     } t;
1629     TPM2B                        b;
1630 } TPM2B_PRIVATE_KEY_RSA;        /* Structure */

```

```

1631 typedef union { // Table 2:173
1632     struct {
1633         UINT16 size;
1634         BYTE buffer[MAX_ECC_KEY_BYTES];
1635     } t;
1636     TPM2B b;
1637 } TPM2B_ECC_PARAMETER; /* Structure */
1638 typedef struct { // Table 2:174
1639     TPM2B_ECC_PARAMETER x;
1640     TPM2B_ECC_PARAMETER y;
1641 } TPMS_ECC_POINT; /* Structure */
1642 typedef struct { // Table 2:175
1643     UINT16 size;
1644     TPMS_ECC_POINT point;
1645 } TPM2B_ECC_POINT; /* Structure */
1646 typedef TPM_ALG_ID TPMI_ALG_ECC_SCHEME; // Table 2:176 /* Interface */
1647 typedef TPM_ECC_CURVE TPMI_ECC_CURVE; // Table 2:177 /* Interface */
1648 typedef struct { // Table 2:178
1649     TPMI_ALG_ECC_SCHEME scheme;
1650     TPMU_ASYM_SCHEME details;
1651 } TPMT_ECC_SCHEME; /* Structure */
1652 typedef struct { // Table 2:179
1653     TPM_ECC_CURVE curveID;
1654     UINT16 keySize;
1655     TPMT_KDF_SCHEME kdf;
1656     TPMT_ECC_SCHEME sign;
1657     TPM2B_ECC_PARAMETER p;
1658     TPM2B_ECC_PARAMETER a;
1659     TPM2B_ECC_PARAMETER b;
1660     TPM2B_ECC_PARAMETER gX;
1661     TPM2B_ECC_PARAMETER gY;
1662     TPM2B_ECC_PARAMETER n;
1663     TPM2B_ECC_PARAMETER h;
1664 } TPMS_ALGORITHM_DETAIL_ECC; /* Structure */
1665 typedef struct { // Table 2:180
1666     TPMI_ALG_HASH hash;
1667     TPM2B_PUBLIC_KEY_RSA sig;
1668 } TPMS_SIGNATURE_RSA; /* Structure */

```

Table 2:181 - Definition of Types for Signature

```

1669 typedef TPMS_SIGNATURE_RSA TPMS_SIGNATURE_RSASSA;
1670 typedef TPMS_SIGNATURE_RSA TPMS_SIGNATURE_RSAPSS;
1671 typedef struct { // Table 2:182
1672     TPMI_ALG_HASH hash;
1673     TPM2B_ECC_PARAMETER signatureR;
1674     TPM2B_ECC_PARAMETER signatureS;
1675 } TPMS_SIGNATURE_ECC; /* Structure */

```

Table 2:183 - Definition of Types for TPMS_SIGNATURE_ECC

```

1676 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECDA;
1677 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECDSA;
1678 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_SM2;
1679 typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECSCHNORR;
1680 typedef union { // Table 2:184
1681     #if ALG_ECC
1682         TPMS_SIGNATURE_ECDA ecda;
1683     #endif // ALG_ECC
1684     #if ALG_RSA
1685         TPMS_SIGNATURE_RSASSA rsassa;
1686     #endif // ALG_RSA
1687     #if ALG_RSA
1688         TPMS_SIGNATURE_RSAPSS rsapss;
1689     #endif // ALG_RSA

```

```

1690 #if ALG_ECC
1691     TPMS_SIGNATURE_ECDSA                ecdsa;
1692 #endif // ALG_ECC
1693 #if ALG_ECC
1694     TPMS_SIGNATURE_SM2                  sm2;
1695 #endif // ALG_ECC
1696 #if ALG_ECC
1697     TPMS_SIGNATURE_ECSCNORR            ecschnorr;
1698 #endif // ALG_ECC
1699 #if ALG_HMAC
1700     TPMT_HA                             hmac;
1701 #endif // ALG_HMAC
1702     TPMS_SCHEME_HASH                    any;
1703 } TPMU_SIGNATURE;                      /* Structure */
1704 typedef struct {                        /* Table 2:185 */
1705     TPMI_ALG_SIG_SCHEME                sigAlg;
1706     TPMU_SIGNATURE                     signature;
1707 } TPMT_SIGNATURE;                      /* Structure */
1708 typedef union {                         /* Table 2:186 */
1709 #if ALG_ECC
1710     BYTE                                ecc[sizeof(TPMS_ECC_POINT)];
1711 #endif // ALG_ECC
1712 #if ALG_RSA
1713     BYTE                                rsa[MAX_RSA_KEY_BYTES];
1714 #endif // ALG_RSA
1715 #if ALG_SYMCIPHER
1716     BYTE                                symmetric[sizeof(TPM2B_DIGEST)];
1717 #endif // ALG_SYMCIPHER
1718 #if ALG_KEYEDHASH
1719     BYTE                                keyedHash[sizeof(TPM2B_DIGEST)];
1720 #endif // ALG_KEYEDHASH
1721 } TPMU_ENCRYPTED_SECRET;                /* Structure */
1722 typedef union {                         /* Table 2:187 */
1723     struct {
1724         UINT16                          size;
1725         BYTE                             secret[sizeof(TPMU_ENCRYPTED_SECRET)];
1726     } t;
1727     TPM2B                                b;
1728 } TPM2B_ENCRYPTED_SECRET;                /* Structure */
1729 typedef TPM_ALG_ID                      TPMI_ALG_PUBLIC; /* Table 2:188 */ /* Interface */
1730 typedef union {                         /* Table 2:189 */
1731 #if ALG_KEYEDHASH
1732     TPM2B_DIGEST                        keyedHash;
1733 #endif // ALG_KEYEDHASH
1734 #if ALG_SYMCIPHER
1735     TPM2B_DIGEST                        sym;
1736 #endif // ALG_SYMCIPHER
1737 #if ALG_RSA
1738     TPM2B_PUBLIC_KEY_RSA                rsa;
1739 #endif // ALG_RSA
1740 #if ALG_ECC
1741     TPMS_ECC_POINT                      ecc;
1742 #endif // ALG_ECC
1743     TPMS_DERIVE                         derive;
1744 } TPMU_PUBLIC_ID;                      /* Structure */
1745 typedef struct {                        /* Table 2:190 */
1746     TPMT_KEYEDHASH_SCHEME               scheme;
1747 } TPMS_KEYEDHASH_PARMS;                /* Structure */
1748 typedef struct {                        /* Table 2:191 */
1749     TPMT_SYM_DEF_OBJECT                 symmetric;
1750     TPMT_ASYM_SCHEME                   scheme;
1751 } TPMS_ASYM_PARMS;                     /* Structure */
1752 typedef struct {                        /* Table 2:192 */
1753     TPMT_SYM_DEF_OBJECT                 symmetric;
1754     TPMT_RSA_SCHEME                     scheme;
1755     TPMI_RSA_KEY_BITS                   keyBits;

```

```

1756     UINT32                                exponent;
1757 } TPMS_RSA_PARMS;                                /* Structure */
1758 typedef struct {                                // Table 2:193
1759     TPMT_SYM_DEF_OBJECT                    symmetric;
1760     TPMT_ECC_SCHEME                        scheme;
1761     TPMI_ECC_CURVE                        curveID;
1762     TPMT_KDF_SCHEME                        kdf;
1763 } TPMS_ECC_PARMS;                                /* Structure */
1764 typedef union {                                // Table 2:194
1765     #if ALG_KEYEDHASH
1766         TPMS_KEYEDHASH_PARMS                keyedHashDetail;
1767     #endif // ALG_KEYEDHASH
1768     #if ALG_SYMCIPHER
1769         TPMS_SYMCIPHER_PARMS                symDetail;
1770     #endif // ALG_SYMCIPHER
1771     #if ALG_RSA
1772         TPMS_RSA_PARMS                        rsaDetail;
1773     #endif // ALG_RSA
1774     #if ALG_ECC
1775         TPMS_ECC_PARMS                        eccDetail;
1776     #endif // ALG_ECC
1777         TPMS_ASYM_PARMS                        asymDetail;
1778 } TPMU_PUBLIC_PARMS;                                /* Structure */
1779 typedef struct {                                // Table 2:195
1780     TPMI_ALG_PUBLIC                        type;
1781     TPMU_PUBLIC_PARMS                    parameters;
1782 } TPMT_PUBLIC_PARMS;                                /* Structure */
1783 typedef struct {                                // Table 2:196
1784     TPMI_ALG_PUBLIC                        type;
1785     TPMI_ALG_HASH                        nameAlg;
1786     TPMA_OBJECT                        objectAttributes;
1787     TPM2B_DIGEST                        authPolicy;
1788     TPMU_PUBLIC_PARMS                    parameters;
1789     TPMU_PUBLIC_ID                        unique;
1790 } TPMT_PUBLIC;                                /* Structure */
1791 typedef struct {                                // Table 2:197
1792     UINT16                                size;
1793     TPMT_PUBLIC                        publicArea;
1794 } TPM2B_PUBLIC;                                /* Structure */
1795 typedef union {                                // Table 2:198
1796     struct {
1797         UINT16                                size;
1798         BYTE                                buffer[sizeof(TPMT_PUBLIC)];
1799     } t;
1800     TPM2B                                b;
1801 } TPM2B_TEMPLATE;                                /* Structure */
1802 typedef union {                                // Table 2:199
1803     struct {
1804         UINT16                                size;
1805         BYTE                                buffer[PRIVATE_VENDOR_SPECIFIC_BYTES];
1806     } t;
1807     TPM2B                                b;
1808 } TPM2B_PRIVATE_VENDOR_SPECIFIC;                /* Structure */
1809 typedef union {                                // Table 2:200
1810     #if ALG_RSA
1811         TPM2B_PRIVATE_KEY_RSA                rsa;
1812     #endif // ALG_RSA
1813     #if ALG_ECC
1814         TPM2B_ECC_PARAMETER                ecc;
1815     #endif // ALG_ECC
1816     #if ALG_KEYEDHASH
1817         TPM2B_SENSITIVE_DATA                bits;
1818     #endif // ALG_KEYEDHASH
1819     #if ALG_SYMCIPHER
1820         TPM2B_SYM_KEY                        sym;
1821     #endif // ALG_SYMCIPHER

```

```

1822     TPM2B_PRIVATE_VENDOR_SPECIFIC    any;
1823 } TPMU_SENSITIVE_COMPOSITE;          /* Structure */
1824 typedef struct {                     /* Table 2:201
1825     TPMI_ALG_PUBLIC                  sensitiveType;
1826     TPM2B_AUTH                      authValue;
1827     TPM2B_DIGEST                    seedValue;
1828     TPMU_SENSITIVE_COMPOSITE        sensitive;
1829 } TPMT_SENSITIVE;                   /* Structure */
1830 typedef struct {                     /* Table 2:202
1831     UINT16                          size;
1832     TPMT_SENSITIVE                  sensitiveArea;
1833 } TPM2B_SENSITIVE;                  /* Structure */
1834 typedef struct {                     /* Table 2:203
1835     TPM2B_DIGEST                    integrityOuter;
1836     TPM2B_DIGEST                    integrityInner;
1837     TPM2B_SENSITIVE                 sensitive;
1838 } _PRIVATE;                         /* Structure */
1839 typedef union {                     /* Table 2:204
1840     struct {
1841         UINT16                      size;
1842         BYTE                        buffer[sizeof(_PRIVATE)];
1843     }                               t;
1844     TPM2B                          b;
1845 } TPM2B_PRIVATE;                   /* Structure */
1846 typedef struct {                     /* Table 2:205
1847     TPM2B_DIGEST                    integrityHMAC;
1848     TPM2B_DIGEST                    encIdentity;
1849 } TPMS_ID_OBJECT;                  /* Structure */
1850 typedef union {                     /* Table 2:206
1851     struct {
1852         UINT16                      size;
1853         BYTE                        credential[sizeof(TPMS_ID_OBJECT)];
1854     }                               t;
1855     TPM2B                          b;
1856 } TPM2B_ID_OBJECT;                 /* Structure */
1857 #define TYPE_OF_TPM_NV_INDEX        UINT32
1858 #define TPM_NV_INDEX_TO_UINT32(a)    (*(UINT32 *)&(a))
1859 #define UINT32_TO_TPM_NV_INDEX(a)    (*(TPM_NV_INDEX *)&(a))
1860 #define TPM_NV_INDEX_TO_BYTE_ARRAY(i, a) \
1861     UINT32_TO_BYTE_ARRAY(TPM_NV_INDEX_TO_UINT32(i), (a)) \
1862 #define BYTE_ARRAY_TO_TPM_NV_INDEX(i, a) \
1863     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPM_NV_INDEX(x); }
1864 #if USE_BIT_FIELD_STRUCTURES
1865 typedef struct TPM_NV_INDEX {        /* Table 2:207
1866     unsigned    index                : 24;
1867     unsigned    RH_NV                : 8;
1868 } TPM_NV_INDEX;                     /* Bits */

```

This is the initializer for a TPM_NV_INDEX structure

```

1869 #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) {index, rh_nv}
1870 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:207 TPM_NV_INDEX using bit masking

```

1871 typedef UINT32    TPM_NV_INDEX;
1872 #define TYPE_OF_TPM_NV_INDEX    UINT32
1873 #define TPM_NV_INDEX_index_SHIFT    0
1874 #define TPM_NV_INDEX_index        ((TPM_NV_INDEX)0xffffffff << 0)
1875 #define TPM_NV_INDEX_RH_NV_SHIFT    24
1876 #define TPM_NV_INDEX_RH_NV        ((TPM_NV_INDEX)0xff << 24)

```

This is the initializer for a TPM_NV_INDEX bit array.

```

1877 #define TPM_NV_INDEX_INITIALIZER(index, rh_nv) {(index << 0) + (rh_nv << 24)}
1878 #endif // USE_BIT_FIELD_STRUCTURES

```

Table 2:208 - Definition of TPM_NT Constants

```

1879 typedef UINT32          TPM_NT;
1880 #define TYPE_OF_TPM_NT  UINT32
1881 #define TPM_NT_ORDINARY  (TPM_NT) (0x0)
1882 #define TPM_NT_COUNTER   (TPM_NT) (0x1)
1883 #define TPM_NT_BITS      (TPM_NT) (0x2)
1884 #define TPM_NT_EXTEND     (TPM_NT) (0x4)
1885 #define TPM_NT_PIN_FAIL   (TPM_NT) (0x8)
1886 #define TPM_NT_PIN_PASS   (TPM_NT) (0x9)
1887 typedef struct {          // Table 2:209
1888     UINT32                pinCount;
1889     UINT32                pinLimit;
1890 } TPMS_NV_PIN_COUNTER_PARAMETERS; // Structure */
1891 #define TYPE_OF_TPMA_NV  UINT32
1892 #define TPMA_NV_TO_UINT32(a) ((UINT32 *)&(a))
1893 #define UINT32_TO_TPMA_NV(a) ((TPMA_NV *)&(a))
1894 #define TPMA_NV_TO_BYTE_ARRAY(i, a) \
1895     UINT32_TO_BYTE_ARRAY((TPMA_NV_TO_UINT32(i)), (a)) \
1896 #define BYTE_ARRAY_TO_TPMA_NV(i, a) \
1897     { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_NV(x); }
1898 #if USE_BIT_FIELD_STRUCTURES
1899 typedef struct TPMA_NV {    // Table 2:210
1900     unsigned    PPWRITE      : 1;
1901     unsigned    OWNERWRITE   : 1;
1902     unsigned    AUTHWRITE    : 1;
1903     unsigned    POLICYWRITE   : 1;
1904     unsigned    TPM_NT       : 4;
1905     unsigned    Reserved_bits_at_8 : 2;
1906     unsigned    POLICY_DELETE : 1;
1907     unsigned    WRITELOCKED   : 1;
1908     unsigned    WRITEALL      : 1;
1909     unsigned    WRITEDEFINE   : 1;
1910     unsigned    WRITE_STCLEAR : 1;
1911     unsigned    GLOBALLOCK    : 1;
1912     unsigned    PPREAD        : 1;
1913     unsigned    OWNERREAD     : 1;
1914     unsigned    AUTHREAD      : 1;
1915     unsigned    POLICYREAD     : 1;
1916     unsigned    Reserved_bits_at_20 : 5;
1917     unsigned    NO_DA         : 1;
1918     unsigned    ORDERLY       : 1;
1919     unsigned    CLEAR_STCLEAR  : 1;
1920     unsigned    READLOCKED    : 1;
1921     unsigned    WRITTEN       : 1;
1922     unsigned    PLATFORMCREATE : 1;
1923     unsigned    READ_STCLEAR   : 1;
1924 } TPMA_NV; // Bits */

```

This is the initializer for a TPMA_NV structure

```

1925 #define TPMA_NV_INITIALIZER( \
1926     ppwrite,      ownerwrite,  authwrite,  policywrite, \
1927     tpm_nt,       bits_at_8,   policy_delete, writelocked, \
1928     writeall,     writedefine, write_stclear, globallock, \
1929     ppread,       ownerread,   authread,   policyread, \
1930     bits_at_20,   no_da,       orderly,     clear_stclear, \
1931     readlocked,   written,     platformcreate, read_stclear) \
1932     {ppwrite,      ownerwrite,  authwrite,  policywrite, \
1933     tpm_nt,       bits_at_8,   policy_delete, writelocked, \
1934     writeall,     writedefine, write_stclear, globallock, \
1935     ppread,       ownerread,   authread,   policyread, \

```



```

1936         bits_at_20,      no_da,      orderly,      clear_stclear,      \
1937         readlocked,      written,      platformcreate, read_stclear}
1938 #else // USE_BIT_FIELD_STRUCTURES

```

This implements Table 2:210 TPMA_NV using bit masking

```

1939 typedef UINT32          TPMA_NV;
1940 #define TYPE_OF_TPMA_NV  UINT32
1941 #define TPMA_NV_PPWRITE   ((TPMA_NV)1 << 0)
1942 #define TPMA_NV_OWNERWRITE ((TPMA_NV)1 << 1)
1943 #define TPMA_NV_AUTHWRITE ((TPMA_NV)1 << 2)
1944 #define TPMA_NV_POLICYWRITE ((TPMA_NV)1 << 3)
1945 #define TPMA_NV_TPM_NT_SHIFT 4
1946 #define TPMA_NV_TPM_NT      ((TPMA_NV)0xf << 4)
1947 #define TPMA_NV_POLICY_DELETE ((TPMA_NV)1 << 10)
1948 #define TPMA_NV_WRITELOCKED ((TPMA_NV)1 << 11)
1949 #define TPMA_NV_WRITEALL ((TPMA_NV)1 << 12)
1950 #define TPMA_NV_WRIEDEFINE ((TPMA_NV)1 << 13)
1951 #define TPMA_NV_WRITE_STCLEAR ((TPMA_NV)1 << 14)
1952 #define TPMA_NV_GLOBALLOCK ((TPMA_NV)1 << 15)
1953 #define TPMA_NV_PPREAD ((TPMA_NV)1 << 16)
1954 #define TPMA_NV_OWNERREAD ((TPMA_NV)1 << 17)
1955 #define TPMA_NV_AUTHREAD ((TPMA_NV)1 << 18)
1956 #define TPMA_NV_POLICYREAD ((TPMA_NV)1 << 19)
1957 #define TPMA_NV_NO_DA ((TPMA_NV)1 << 25)
1958 #define TPMA_NV_ORDERLY ((TPMA_NV)1 << 26)
1959 #define TPMA_NV_CLEAR_STCLEAR ((TPMA_NV)1 << 27)
1960 #define TPMA_NV_READLOCKED ((TPMA_NV)1 << 28)
1961 #define TPMA_NV_WRITTEN ((TPMA_NV)1 << 29)
1962 #define TPMA_NV_PLATFORMCREATE ((TPMA_NV)1 << 30)
1963 #define TPMA_NV_READ_STCLEAR ((TPMA_NV)1 << 31)

```

This is the initializer for a TPMA_NV bit array.

```

1964 #define TPMA_NV_INITIALIZER(
1965     ppwrite,      ownerwrite,      authwrite,      policywrite,      \
1966     tpm_nt,      bits_at_8,      policy_delete,      writelocked,      \
1967     writeall,      writedefine,      write_stclear,      globallock,      \
1968     ppread,      ownerread,      authread,      policyread,      \
1969     bits_at_20,      no_da,      orderly,      clear_stclear,      \
1970     readlocked,      written,      platformcreate,      read_stclear)      \
1971     { (ppwrite << 0) + (ownerwrite << 1) +
1972       (authwrite << 2) + (policywrite << 3) +
1973       (tpm_nt << 4) + (policy_delete << 10) +
1974       (writelocked << 11) + (writeall << 12) +
1975       (writedefine << 13) + (write_stclear << 14) +
1976       (globallock << 15) + (ppread << 16) +
1977       (ownerread << 17) + (authread << 18) +
1978       (policyread << 19) + (no_da << 25) +
1979       (orderly << 26) + (clear_stclear << 27) +
1980       (readlocked << 28) + (written << 29) +
1981       (platformcreate << 30) + (read_stclear << 31)}
1982 #endif // USE_BIT_FIELD_STRUCTURES
1983 typedef struct { // Table 2:211
1984     TPMI_RH_NV_INDEX      nvIndex;
1985     TPMI_ALG_HASH          nameAlg;
1986     TPMA_NV                attributes;
1987     TPM2B_DIGEST           authPolicy;
1988     UINT16                 dataSize;
1989 } TPMS_NV_PUBLIC; // /* Structure */
1990 typedef struct { // Table 2:212
1991     UINT16                 size;
1992     TPMS_NV_PUBLIC         nvPublic;
1993 } TPM2B_NV_PUBLIC; // /* Structure */
1994 typedef union { // Table 2:213

```



```

1995     struct {
1996         UINT16          size;
1997         BYTE            buffer[MAX_CONTEXT_SIZE];
1998     } t;
1999     TPM2B              b;
2000 } TPM2B_CONTEXT_SENSITIVE;          /* Structure */
2001 typedef struct {                  /* Table 2:214
2002     TPM2B_DIGEST            integrity;
2003     TPM2B_CONTEXT_SENSITIVE encrypted;
2004 } TPMS_CONTEXT_DATA;              /* Structure */
2005 typedef union {                  /* Table 2:215
2006     struct {
2007         UINT16          size;
2008         BYTE            buffer[sizeof(TPMS_CONTEXT_DATA)];
2009     } t;
2010     TPM2B              b;
2011 } TPM2B_CONTEXT_DATA;              /* Structure */
2012 typedef struct {                  /* Table 2:216
2013     UINT64              sequence;
2014     TPMI_DH_SAVED        savedHandle;
2015     TPMI_RH_HIERARCHY    hierarchy;
2016     TPM2B_CONTEXT_DATA   contextBlob;
2017 } TPMS_CONTEXT;                  /* Structure */
2018 typedef struct {                  /* Table 2:218
2019     TPML_PCR_SELECTION   pcrSelect;
2020     TPM2B_DIGEST          pcrDigest;
2021     TPMA_LOCALITY         locality;
2022     TPM_ALG_ID            parentNameAlg;
2023     TPM2B_NAME             parentName;
2024     TPM2B_NAME             parentQualifiedName;
2025     TPM2B_DATA             outsideInfo;
2026 } TPMS_CREATION_DATA;            /* Structure */
2027 typedef struct {                  /* Table 2:219
2028     UINT16          size;
2029     TPMS_CREATION_DATA creationData;
2030 } TPM2B_CREATION_DATA;            /* Structure */

```

Table 2:220 - Definition of TPM_AT Constants

```

2031 typedef UINT32      TPM_AT;
2032 #define TYPE_OF_TPM_AT  UINT32
2033 #define TPM_AT_ANY      (TPM_AT) (0x00000000)
2034 #define TPM_AT_ERROR    (TPM_AT) (0x00000001)
2035 #define TPM_AT_PV1      (TPM_AT) (0x00000002)
2036 #define TPM_AT_VEND     (TPM_AT) (0x80000000)

```

Table 2:221 - Definition of TPM_AE Constants

```

2037 typedef UINT32      TPM_AE;
2038 #define TYPE_OF_TPM_AE  UINT32
2039 #define TPM_AE_NONE     (TPM_AE) (0x00000000)
2040 typedef struct {                  /* Table 2:222
2041     TPM_AT              tag;
2042     UINT32              data;
2043 } TPMS_AC_OUTPUT;              /* Structure */
2044 typedef struct {                  /* Table 2:223
2045     UINT32              count;
2046     TPMS_AC_OUTPUT       acCapabilities[MAX_AC_CAPABILITIES];
2047 } TPML_AC_CAPABILITIES;        /* Structure */
2048 #endif // _TPM_TYPES_H

```

5.22 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
```

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

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

5.23 swap.h

```

1  #ifndef _SWAP_H
2  #define _SWAP_H
3  #if LITTLE_ENDIAN_TPM
4  #define TO_BIG_ENDIAN_UINT16(i)    REVERSE_ENDIAN_16(i)
5  #define FROM_BIG_ENDIAN_UINT16(i)  REVERSE_ENDIAN_16(i)
6  #define TO_BIG_ENDIAN_UINT32(i)    REVERSE_ENDIAN_32(i)
7  #define FROM_BIG_ENDIAN_UINT32(i)  REVERSE_ENDIAN_32(i)
8  #define TO_BIG_ENDIAN_UINT64(i)    REVERSE_ENDIAN_64(i)
9  #define FROM_BIG_ENDIAN_UINT64(i)  REVERSE_ENDIAN_64(i)
10 #else
11 #define TO_BIG_ENDIAN_UINT16(i)    (i)
12 #define FROM_BIG_ENDIAN_UINT16(i)  (i)
13 #define TO_BIG_ENDIAN_UINT32(i)    (i)
14 #define FROM_BIG_ENDIAN_UINT32(i)  (i)
15 #define TO_BIG_ENDIAN_UINT64(i)    (i)
16 #define FROM_BIG_ENDIAN_UINT64(i)  (i)
17 #endif
18 #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

```

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

```

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

```

29 #define BYTE_ARRAY_TO_UINT8(b)    *((uint8_t *) (b))
30 #define BYTE_ARRAY_TO_UINT16(b)   *((uint16_t *) (b))
31 #define BYTE_ARRAY_TO_UINT32(b)   *((uint32_t *) (b))
32 #define BYTE_ARRAY_TO_UINT64(b)   *((uint64_t *) (b))

```

Disaggregate a UINT into a byte array

```

33 #define UINT8_TO_BYTE_ARRAY(i, b)  {*((uint8_t *) (b)) = (i);}
34 #define UINT16_TO_BYTE_ARRAY(i, b) {*((uint16_t *) (b)) = (i);}
35 #define UINT32_TO_BYTE_ARRAY(i, b) {*((uint32_t *) (b)) = (i);}
36 #define UINT64_TO_BYTE_ARRAY(i, b) {*((uint64_t *) (b)) = (i);}
37 #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

```

38 #define BYTE_ARRAY_TO_UINT8(b)    *((uint8_t *) (b))
39 #define BYTE_ARRAY_TO_UINT16(b)   REVERSE_ENDIAN_16(*((uint16_t *) (b)))
40 #define BYTE_ARRAY_TO_UINT32(b)   REVERSE_ENDIAN_32(*((uint32_t *) (b)))
41 #define BYTE_ARRAY_TO_UINT64(b)   REVERSE_ENDIAN_64(*((uint64_t *) (b)))

```

Disaggregate a UINT into a byte array

```

42 #define UINT8_TO_BYTE_ARRAY(i, b)  {*((uint8_t *) (b)) = (i);}

```

```
43 #define UINT16_TO_BYTE_ARRAY(i, b) {*((uint16_t *) (b)) = REVERSE_ENDIAN_16(i);}
44 #define UINT32_TO_BYTE_ARRAY(i, b) {*((uint32_t *) (b)) = REVERSE_ENDIAN_32(i);}
45 #define UINT64_TO_BYTE_ARRAY(i, b) {*((uint64_t *) (b)) = REVERSE_ENDIAN_64(i);}
46 #endif // BIG_ENDIAN_TPM
47 #endif // AUTO_ALIGN == NO
48 #endif // _SWAP_H
```

DRAFT

6 Main

6.1 Introduction

The files in this section 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"
```

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 BuildResponseSessions() 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().

```

4  LIB_EXPORT void
5  ExecuteCommand(
6      uint32_t      requestSize,    // IN: command buffer size
7      unsigned char *request,      // IN: command buffer
8      uint32_t      *responseSize, // IN/OUT: response buffer size
9      unsigned char **response     // IN/OUT: response buffer
10 )
11 {
12     // Command local variables
13     UINT32      commandSize;
14     COMMAND     command;
15
16     // Response local variables
17     UINT32      maxResponse = *responseSize;
18     TPM_RC      result;      // return code for the command
19
20     // This next function call is used in development to size the command and response
21     // buffers. The values printed are the sizes of the internal structures and
22     // not the sizes of the canonical forms of the command response structures. Also,
23     // the sizes do not include the tag, command.code, requestSize, or the authorization
24     // fields.
25     //CommandResponseSizes();
26     // Set flags for NV access state. This should happen before any other
27     // operation that may require a NV write. Note, that this needs to be done

```

```

28 // even when in failure mode. Otherwise, g_updateNV would stay SET while in
29 // Failure mode and the NV would be written on each call.
30 g_updateNV = UT_NONE;
31 g_clearOrderly = FALSE;
32 if(g_inFailureMode)
33 {
34     // Do failure mode processing
35     TpmFailureMode(requestSize, request, responseSize, response);
36     return;
37 }
38 // Query platform to get the NV state. The result state is saved internally
39 // and will be reported by NvIsAvailable(). The reference code requires that
40 // accessibility of NV does not change during the execution of a command.
41 // Specifically, if NV is available when the command execution starts and then
42 // is not available later when it is necessary to write to NV, then the TPM
43 // will go into failure mode.
44 NvCheckState();
45
46 // Due to the limitations of the simulation, TPM clock must be explicitly
47 // synchronized with the system clock whenever a command is received.
48 // This function call is not necessary in a hardware TPM. However, taking
49 // a snapshot of the hardware timer at the beginning of the command allows
50 // the time value to be consistent for the duration of the command execution.
51 TimeUpdateToCurrent();
52
53 // Any command through this function will unceremoniously end the
54 // _TPM_Hash_Data/_TPM_Hash_End sequence.
55 if(g_DRTMHandle != TPM_RH_UNASSIGNED)
56     ObjectTerminateEvent();
57
58 // Get command buffer size and command buffer.
59 command.parameterBuffer = request;
60 command.parameterSize = requestSize;
61
62 // Parse command header: tag, commandSize and command.code.
63 // First parse the tag. The unmarshaling routine will validate
64 // that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
65 result = TPMI_ST_COMMAND_TAG_Unmarshal(&command.tag,
66                                         &command.parameterBuffer,
67                                         &command.parameterSize);
68 if(result != TPM_RC_SUCCESS)
69     goto Cleanup;
70 // Unmarshal the commandSize indicator.
71 result = UINT32_Unmarshal(&commandSize,
72                           &command.parameterBuffer,
73                           &command.parameterSize);
74 if(result != TPM_RC_SUCCESS)
75     goto Cleanup;
76 // On a TPM that receives bytes on a port, the number of bytes that were
77 // received on that port is requestSize it must be identical to commandSize.
78 // In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
79 // by the implementation. The check against MAX_COMMAND_SIZE may be redundant
80 // as the input processing (the function that receives the command bytes and
81 // places them in the input buffer) would likely have the input truncated when
82 // it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
83 if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
84 {
85     result = TPM_RC_COMMAND_SIZE;
86     goto Cleanup;
87 }
88 // Unmarshal the command code.
89 result = TPM_CC_Unmarshal(&command.code, &command.parameterBuffer,
90                           &command.parameterSize);
91 if(result != TPM_RC_SUCCESS)
92     goto Cleanup;
93 // Check to see if the command is implemented.

```



```

94     command.index = CommandCodeToCommandIndex(command.code);
95     if(UNIMPLEMENTED_COMMAND_INDEX == command.index)
96     {
97         result = TPM_RC_COMMAND_CODE;
98         goto Cleanup;
99     }
100 #if FIELD_UPGRADE_IMPLEMENTED == YES
101 // If the TPM is in FUM, then the only allowed command is
102 // TPM_CC_FieldUpgradeData.
103 if(IsFieldUpgradeMode() && (command.code != TPM_CC_FieldUpgradeData))
104 {
105     result = TPM_RC_UPGRADE;
106     goto Cleanup;
107 }
108 else
109 #endif
110 // Excepting FUM, the TPM only accepts TPM2_Startup() after
111 // TPM_Init. After getting a TPM2_Startup(), TPM2_Startup()
112 // is no longer allowed.
113 if((!TPMIsStarted() && command.code != TPM_CC_Startup)
114    || (TPMIsStarted() && command.code == TPM_CC_Startup))
115 {
116     result = TPM_RC_INITIALIZE;
117     goto Cleanup;
118 }
119 // Start regular command process.
120 NvIndexCacheInit();
121 // Parse Handle buffer.
122 result = ParseHandleBuffer(&command);
123 if(result != TPM_RC_SUCCESS)
124     goto Cleanup;
125 // All handles in the handle area are required to reference TPM-resident
126 // entities.
127 result = EntityGetLoadStatus(&command);
128 if(result != TPM_RC_SUCCESS)
129     goto Cleanup;
130 // Authorization session handling for the command.
131 ClearCpRpHashes(&command);
132 if(command.tag == TPM_ST_SESSIONS)
133 {
134     // Find out session buffer size.
135     result = UINT32_Unmarshal((UINT32 *)&command.authSize,
136                             &command.parameterBuffer,
137                             &command.parameterSize);
138     if(result != TPM_RC_SUCCESS)
139         goto Cleanup;
140     // Perform sanity check on the unmarshaled value. If it is smaller than
141     // the smallest possible session or larger than the remaining size of
142     // the command, then it is an error. NOTE: This check could pass but the
143     // session size could still be wrong. That will be determined after the
144     // sessions are unmarshaled.
145     if(command.authSize < 9
146        || command.authSize > command.parameterSize)
147     {
148         result = TPM_RC_SIZE;
149         goto Cleanup;
150     }
151     command.parameterSize -= command.authSize;
152
153     // The actions of ParseSessionBuffer() are described in the introduction.
154     // As the sessions are parsed command.parameterBuffer is advanced so, on a
155     // successful return, command.parameterBuffer should be pointing at the
156     // first byte of the parameters.
157     result = ParseSessionBuffer(&command);
158     if(result != TPM_RC_SUCCESS)
159         goto Cleanup;

```

```

160     }
161     else
162     {
163         command.authSize = 0;
164         // The command has no authorization sessions.
165         // If the command requires authorizations, then CheckAuthNoSession() will
166         // return an error.
167         result = CheckAuthNoSession(&command);
168         if(result != TPM_RC_SUCCESS)
169             goto Cleanup;
170     }
171     // Set up the response buffer pointers. CommandDispatch will marshal the
172     // response parameters starting at the address in command.responseBuffer.
173     /*response = MemoryGetResponseBuffer(command.index);
174     // leave space for the command header
175     command.responseBuffer = *response + STD_RESPONSE_HEADER;
176
177     // leave space for the parameter size field if needed
178     if(command.tag == TPM_ST_SESSIONS)
179         command.responseBuffer += sizeof(UINT32);
180     if(IsHandleInResponse(command.index))
181         command.responseBuffer += sizeof(TPM_HANDLE);
182
183     // CommandDispatcher returns a response handle buffer and a response parameter
184     // buffer if it succeeds. It will also set the parameterSize field in the
185     // buffer if the tag is TPM_RC_SESSIONS.
186     result = CommandDispatcher(&command);
187     if(result != TPM_RC_SUCCESS)
188         goto Cleanup;
189
190     // Build the session area at the end of the parameter area.
191     BuildResponseSession(&command);
192
193 Cleanup:
194     if(g_clearOrderly == TRUE
195        && NV_IS_ORDERLY)
196     {
197 #if USE_DA_USED
198         gp.orderlyState = g_daUsed ? SU_DA_USED_VALUE : SU_NONE_VALUE;
199 #else
200         gp.orderlyState = SU_NONE_VALUE;
201 #endif
202         NV_SYNC_PERSISTENT(orderlyState);
203     }
204     // This implementation loads an "evict" object to a transient object slot in
205     // RAM whenever an "evict" object handle is used in a command so that the
206     // access to any object is the same. These temporary objects need to be
207     // cleared from RAM whether the command succeeds or fails.
208     ObjectCleanupEvict();
209
210     // The parameters and sessions have been marshaled. Now tack on the header and
211     // set the sizes
212     BuildResponseHeader(&command, *response, result);
213
214     // Try to commit all the writes to NV if any NV write happened during this
215     // command execution. This check should be made for both succeeded and failed
216     // commands, because a failed one may trigger a NV write in DA logic as well.
217     // This is the only place in the command execution path that may call the NV
218     // commit. If the NV commit fails, the TPM should be put in failure mode.
219     if((g_updateNV != UT_NONE) && !g_inFailureMode)
220     {
221         if(g_updateNV == UT_ORDERLY)
222             NvUpdateIndexOrderlyData();
223         if(!NvCommit())
224             FAIL(FATAL_ERROR_INTERNAL);
225         g_updateNV = UT_NONE;

```

```
226     }
227     pAssert((UINT32)command.parameterSize <= maxResponse);
228
229     // Clear unused bits in response buffer.
230     MemorySet(*response + *responseSize, 0, maxResponse - *responseSize);
231
232     // as a final act, and not before, update the response size.
233     *responseSize = (UINT32)command.parameterSize;
234
235     return;
236 }
```

DRAFT

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. *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. *respParmSize* 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.

6.3.1.1 Includes and Typedefs

```

1  #include "Tpm.h"
2  #if TABLE_DRIVEN_DISPATCH
3  typedef TPM_RC (NoFlagFunction) (void *target, BYTE **buffer, INT32 *size);
4  typedef TPM_RC (FlagFunction) (void *target, BYTE **buffer, INT32 *size, BOOL flag);
5  typedef FlagFunction *UNMARSHAL_t;
6  typedef INT16 (MarshalFunction) (void *source, BYTE **buffer, INT32 *size);
7  typedef MarshalFunction *MARSHAL_t;
8  typedef TPM_RC (COMMAND_NO_ARGS) (void);
9  typedef TPM_RC (COMMAND_IN_ARG) (void *in);
10 typedef TPM_RC (COMMAND_OUT_ARG) (void *out);
11 typedef TPM_RC (COMMAND_INOUT_ARG) (void *in, void *out);
12 typedef union COMMAND_t
13 {
14     COMMAND_NO_ARGS      *noArgs;
15     COMMAND_IN_ARG       *inArg;
16     COMMAND_OUT_ARG      *outArg;
17     COMMAND_INOUT_ARG    *inOutArg;
18 } 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.
<i>inSize</i>	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 <i>inSize</i> will have a value of 0.
<i>outSize</i>	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.
<i>typesOffsets</i>	this parameter points to the list of data types that are to be marshaled or unmarshaled. The list of types follows the <i>offsets</i> array. The offsets array is variable sized so the <i>typesOffset</i> field is necessary for the handle and command processing to be able to find the types that are being handled. The <i>offsets</i> array may be empty. The types structure is described below.
<i>offsets</i>	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 <code>CommandDispatcher()</code> hits this value, it will stop unmarshaling the parameters and call <i>command</i> . If a command has no response parameters and only one command parameter, then offsets can be an empty list.

```

19  typedef struct COMMAND_DESCRIPTOR_t
20  {
21      COMMAND_t      command;          // Address of the command
22      UINT16          inSize;           // Maximum size of the input structure
23      UINT16          outSize;          // Maximum size of the output structure
24      UINT16          typesOffset;      // address of the types field
25      UINT16          offsets[1];
26  } 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 types 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.

```

27  #if COMPRESSED_LISTS
28  #   define PAD_LIST 0
29  #else
30  #   define PAD_LIST 1
31  #endif
32  #define _COMMAND_TABLE_DISPATCH_
33  #include "CommandDispatchData.h"
34  #define TEST_COMMAND    TPM_CC_Startup
35  #define NEW_CC
36  #else
37  #include "Commands.h"
38  #endif

```

6.3.1.2 Marshal/Unmarshal Functions

6.3.1.2.1 ParseHandleBuffer()

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

```

39  TPM_RC
40  ParseHandleBuffer(
41      COMMAND                *command
42  )
43  {
44      TPM_RC                result;
45  #if TABLE_DRIVEN_DISPATCH
46      COMMAND_DESCRIPTOR_t  *desc;
47      BYTE                  *types;
48      BYTE                  type;
49      BYTE                  dtype;
50
51      // Make sure that nothing strange has happened
52      pAssert(command->index
53              < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
54      // Get the address of the descriptor for this command
55      desc = s_CommandDataArray[command->index];
56
57      pAssert(desc != NULL);
58      // Get the associated list of unmarshaling data types.
59      types = &((BYTE *)desc)[desc->typesOffset];
60
61      //   if(s_ccAttr[commandIndex].commandIndex == TEST_COMMAND)
62      //       commandIndex = commandIndex;
63      // No handles yet
64      command->handleNum = 0;
65

```

```

66     // Get the first type value
67     for(type = *types++;
68         // check each byte to make sure that we have not hit the start
69         // of the parameters
70         (dtype = (type & 0x7F)) < PARAMETER_FIRST_TYPE;
71         // get the next type
72         type = *types++)
73     {
74         // See if unmarshaling of this handle type requires a flag
75         if(dtype < HANDLE_FIRST_FLAG_TYPE)
76         {
77             // Look up the function to do the unmarshaling
78             NoFlagFunction *f = (NoFlagFunction *)UnmarshalArray[dtype];
79             // call it
80             result = f(&(command->handles[command->handleNum]),
81                     &command->parameterBuffer,
82                     &command->parameterSize);
83         }
84         else
85         {
86             // Look up the function
87             FlagFunction *f = UnmarshalArray[dtype];
88
89             // Call it setting the flag to the appropriate value
90             result = f(&(command->handles[command->handleNum]),
91                     &command->parameterBuffer,
92                     &command->parameterSize, (type & 0x80) != 0);
93         }
94         // Got a handle
95         // We do this first so that the match for the handle offset of the
96         // response code works correctly.
97         command->handleNum += 1;
98         if(result != TPM_RC_SUCCESS)
99             // if the unmarshaling failed, return the response code with the
100             // handle indication set
101             return result + TPM_RC_H + (command->handleNum * TPM_RC_1);
102     }
103 #else
104     BYTE                **handleBufferStart = &command->parameterBuffer;
105     INT32                *bufferRemainingSize = &command->parameterSize;
106     TPM_HANDLE           *handles = &command->handles[0];
107     UINT32               *handleCount = &command->handleNum;
108     *handleCount = 0;
109     switch(command->code)
110     {
111 #include "HandleProcess.h"
112 #undef handles
113         default:
114             FAIL(FATAL_ERROR_INTERNAL);
115             break;
116     }
117 #endif
118     return TPM_RC_SUCCESS;
119 }

```

6.3.1.2.2 CommandDispatcher()

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

```

120 TPM_RC
121 CommandDispatcher(
122     COMMAND                *command
123 )

```



```

124 {
125 #if !TABLE_DRIVEN_DISPATCH
126     TPM_RC      result;
127     BYTE        **paramBuffer = &command->parameterBuffer;
128     INT32       *paramBufferSize = &command->parameterSize;
129     BYTE        **responseBuffer = &command->responseBuffer;
130     INT32       *respParamSize = &command->parameterSize;
131     INT32       rSize;
132     TPM_HANDLE  *handles = &command->handles[0];
133 //
134     command->handleNum = 0;                                // The command-specific code knows how
135                                                            // many handles there are. This is for
136                                                            // cataloging the number of response
137                                                            // handles
138     MemoryIoBufferAllocationReset();                       // Initialize so that allocation will
139                                                            // work properly
140     switch (GetCommandCode (command->index))
141     {
142 #include "CommandDispatcher.h"
143
144         default:
145             FAIL(FATAL_ERROR_INTERNAL);
146             break;
147     }
148 Exit:
149     MemoryIoBufferZero();
150     return result;
151 #else
152     COMMAND_DESCRIPTOR_t *desc;
153     BYTE *types;
154     BYTE type;
155     UINT16 *offsets;
156     UINT16 offset = 0;
157     UINT32 maxInSize;
158     BYTE *commandIn;
159     INT32 maxOutSize;
160     BYTE *commandOut;
161     COMMAND_t cmd;
162     TPM_HANDLE *handles;
163     UINT32 hasInParameters = 0;
164     BOOL hasOutParameters = FALSE;
165     UINT32 pNum = 0;
166     BYTE dType; // dispatch type
167     TPM_RC result;
168 //
169 // Get the address of the descriptor for this command
170 pAssert(command->index
171         < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
172 desc = s_CommandDataArray[command->index];
173
174 // Get the list of parameter types for this command
175 pAssert(desc != NULL);
176 types = &((BYTE *)desc)[desc->typesOffset];
177
178 // Get a pointer to the list of parameter offsets
179 offsets = &desc->offsets[0];
180 // pointer to handles
181 handles = command->handles;
182
183 // Get the size required to hold all the unmarshaled parameters for this command
184 maxInSize = desc->inSize;
185 // and the size of the output parameter structure returned by this command
186 maxOutSize = desc->outSize;
187
188     MemoryIoBufferAllocationReset();
189     // Get a buffer for the input parameters

```

```

190     commandIn = MemoryGetInBuffer(maxInSize);
191     // And the output parameters
192     commandOut = (BYTE *)MemoryGetOutBuffer((UINT32)maxOutSize);
193
194     // Get the address of the action code dispatch
195     cmd = desc->command;
196
197     // Copy any handles into the input buffer
198     for(type = *types++; (type & 0x7F) < PARAMETER_FIRST_TYPE; type = *types++)
199     {
200         // 'offset' was initialized to zero so the first unmarshaling will always
201         // be to the start of the data structure
202         *(TPM_HANDLE *)&(commandIn[offset]) = *handles++;
203         // This check is used so that we don't have to add an additional offset
204         // value to the offsets list to correspond to the stop value in the
205         // command parameter list.
206         if(*types != 0xFF)
207             offset = *offsets++;
208     //     maxInSize -= sizeof(TPM_HANDLE);
209     hasInParameteres++;
210     }
211     // Exit loop with type containing the last value read from types
212     // maxInSize has the amount of space remaining in the command action input
213     // buffer. Make sure that we don't have more data to unmarshal than is going to
214     // fit.
215
216     // type contains the last value read from types so it is not necessary to
217     // reload it, which is good because *types now points to the next value
218     for(; (dType = (type & 0x7F)) <= PARAMETER_LAST_TYPE; type = *types++)
219     {
220         pNum++;
221         if(dType < PARAMETER_FIRST_FLAG_TYPE)
222         {
223             NoFlagFunction *f = (NoFlagFunction *)UnmarshalArray[dType];
224             result = f(&commandIn[offset], &command->parameterBuffer,
225                     &command->parameterSize);
226         }
227         else
228         {
229             FlagFunction *f = UnmarshalArray[dType];
230             result = f(&commandIn[offset], &command->parameterBuffer,
231                     &command->parameterSize,
232                     (type & 0x80) != 0);
233         }
234         if(result != TPM_RC_SUCCESS)
235         {
236             result += TPM_RC_P + (TPM_RC_1 * pNum);
237             goto Exit;
238         }
239
240         // This check is used so that we don't have to add an additional offset
241         // value to the offsets list to correspond to the stop value in the
242         // command parameter list.
243         if(*types != 0xFF)
244             offset = *offsets++;
245         hasInParameteres++;
246     }
247     // Should have used all the bytes in the input
248     if(command->parameterSize != 0)
249     {
250         result = TPM_RC_SIZE;
251         goto Exit;
252     }
253
254     // The command parameter unmarshaling stopped when it hit a value that was out
255     // of range for unmarshaling values and left *types pointing to the first

```

```

256 // marshaling type. If that type happens to be the STOP value, then there
257 // are no response parameters. So, set the flag to indicate if there are
258 // output parameters.
259 hasOutParameters = *types != 0xFF;
260
261 // There are four cases for calling, with and without input parameters and with
262 // and without output parameters.
263 if(hasInParameters > 0)
264 {
265     if(hasOutParameters)
266         result = cmd.inOutArg(commandIn, commandOut);
267     else
268         result = cmd.inArg(commandIn);
269 }
270 else
271 {
272     if(hasOutParameters)
273         result = cmd.outArg(commandOut);
274     else
275         result = cmd.noArgs();
276 }
277 if(result != TPM_RC_SUCCESS)
278     goto Exit;
279
280 // Offset in the marshaled output structure
281 offset = 0;
282
283 // Process the return handles, if any
284 command->handleNum = 0;
285
286 // Could make this a loop to process output handles but there is only ever
287 // one handle in the outputs (for now).
288 type = *types++;
289 if((dType = (type & 0x7F)) < RESPONSE_PARAMETER_FIRST_TYPE)
290 {
291     // The out->handle value was referenced as TPM_HANDLE in the
292     // action code so it has to be properly aligned.
293     command->handles[command->handleNum++] =
294         *((TPM_HANDLE *)&(commandOut[offset]));
295     maxOutSize -= sizeof(UINT32);
296     type = *types++;
297     offset = *offsets++;
298 }
299 // Use the size of the command action output buffer as the maximum for the
300 // number of bytes that can get marshaled. Since the marshaling code has
301 // no pointers to data, all of the data being returned has to be in the
302 // command action output buffer. If we try to marshal more bytes than
303 // could fit into the output buffer, we need to fail.
304 for(; (dType = (type & 0x7F)) <= RESPONSE_PARAMETER_LAST_TYPE
305     && !g_inFailureMode; type = *types++)
306 {
307     const MARSHAL_t f = MarshalArray[dType];
308
309     command->parameterSize += f(&commandOut[offset], &command->responseBuffer,
310                               &maxOutSize);
311     offset = *offsets++;
312 }
313 result = (maxOutSize < 0) ? TPM_RC_FAILURE : TPM_RC_SUCCESS;
314 Exit:
315     MemoryIoBufferZero();
316     return result;
317 #endif
318 }

```

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  #include "Tpm.h"
```

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

```
3  BOOL
4  IsDAExempted(
5      TPM_HANDLE    handle        // IN: entity handle
6  )
7  {
8      BOOL    result = FALSE;
9      //
10     switch(HandleGetType(handle))
11     {
12         case TPM_HT_PERMANENT:
13             // All permanent handles, other than TPM_RH_LOCKOUT, are exempt from
14             // DA protection.
15             result = (handle != TPM_RH_LOCKOUT);
16             break;
17             // When this function is called, a persistent object will have been loaded
18             // into an object slot and assigned a transient handle.
19         case TPM_HT_TRANSIENT:
20             {
21                 TPMA_OBJECT    attributes = ObjectGetPublicAttributes(handle);
22                 result = IS_ATTRIBUTE(attributes, TPMA_OBJECT, noDA);
23                 break;
24             }
25         case TPM_HT_NV_INDEX:
26             {
27                 NV_INDEX    *nvIndex = NvGetIndexInfo(handle, NULL);
28                 result = IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, NO_DA);
29                 break;
30             }
31     }
```

```

31     case TPM_HT_PCR:
32         // PCRs are always exempted from DA.
33         result = TRUE;
34         break;
35     default:
36         break;
37 }
38 return result;
39 }

```

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 Returns	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

```

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

```

```

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

```

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

```

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

```

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:

- the command requires the DUP role,
- the command requires the ADMIN role and the authorized entity is an object and its *adminWithPolicy* bit is SET, or
- the command requires the ADMIN role and the authorized entity is a permanent handle or an NV Index.
- 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

```

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

```


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

```

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

```

```

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

```

```

283         break;
284     }
285     return result;
286 }

```

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

```

287 static BOOL
288 IsAuthPolicyAvailable(
289     TPM_HANDLE      handle,          // IN: handle of entity
290     COMMAND_INDEX   commandIndex,   // IN: command index
291     UINT32          sessionIndex    // IN: session index
292 )
293 {
294     BOOL            result = FALSE;
295     //
296     switch(HandleGetType(handle))
297     {
298         case TPM_HT_PERMANENT:
299             switch(handle)
300             {
301                 // At this point hierarchy availability has already been checked.
302                 case TPM_RH_OWNER:
303                     if(gp.ownerPolicy.t.size != 0)
304                         result = TRUE;
305                     break;
306                 case TPM_RH_ENDORSEMENT:
307                     if(gp.endorsementPolicy.t.size != 0)
308                         result = TRUE;
309                     break;
310                 case TPM_RH_PLATFORM:
311                     if(gc.platformPolicy.t.size != 0)
312                         result = TRUE;
313                     break;
314                 case TPM_RH_LOCKOUT:
315                     if(gp.lockoutPolicy.t.size != 0)
316                         result = TRUE;
317                     break;
318                 default:
319                     break;
320             }
321             break;
322         case TPM_HT_TRANSIENT:
323             {
324                 // Object handle.
325                 // An evict object would already have been loaded and given a
326                 // transient object handle by this point.
327                 OBJECT *object = HandleToObject(handle);
328                 // Policy authorization is not available for an object with only
329                 // public portion loaded.
330                 if(object->attributes.publicOnly == CLEAR)
331                 {
332                     // Policy authorization is always available for an object but

```

```

333         // is never available for a sequence.
334         if (!ObjectIsSequence(object))
335             result = TRUE;
336     }
337     break;
338 }
339 case TPM_HT_NV_INDEX:
340     // An NV Index.
341     {
342         NV_INDEX          *nvIndex = NvGetIndexInfo(handle, NULL);
343         TPMA_NV           nvAttributes = nvIndex->publicArea.attributes;
344     //
345     // If the policy size is not zero, check if policy can be used.
346     if (nvIndex->publicArea.authPolicy.t.size != 0)
347     {
348         // If policy session is required for this handle, always
349         // uses policy regardless of the attributes bit setting
350         if (IsPolicySessionRequired(commandIndex, sessionIndex))
351             result = TRUE;
352         // Otherwise, the presence of the policy depends on the NV
353         // attributes.
354         else if (IsWriteOperation(commandIndex))
355         {
356             if (IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYWRITE))
357                 result = TRUE;
358         }
359         else
360         {
361             if (IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYREAD))
362                 result = TRUE;
363         }
364     }
365     }
366     break;
367 case TPM_HT_PCR:
368     // PCR handle.
369     if (PCRPolicyIsAvailable(handle))
370         result = TRUE;
371     break;
372 default:
373     break;
374 }
375 return result;
376 }

```

6.4.4 Session Parsing Functions

6.4.4.1 ClearCpRpHashes()

```

377 void
378 ClearCpRpHashes(
379     COMMAND      *command
380 )
381 {
382     #if ALG_SHA1
383     command->sha1CpHash.t.size = 0;
384     command->sha1RpHash.t.size = 0;
385     #endif
386     #if ALG_SHA256
387     command->sha256CpHash.t.size = 0;
388     command->sha256RpHash.t.size = 0;
389     #endif
390     #if ALG_SHA384
391     command->sha384CpHash.t.size = 0;

```

```

392     command->sha384RpHash.t.size = 0;
393 #endif
394 #if ALG_SHA512
395     command->sha512CpHash.t.size = 0;
396     command->sha512RpHash.t.size = 0;
397 #endif
398 #if ALG_SM3_256
399     command->sm3_256CpHash.t.size = 0;
400     command->sm3_256RpHash.t.size = 0;
401 #endif
402 }

```

6.4.4.2 GetCpHashPointer()

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

```

403 static TPM2B_DIGEST *
404 GetCpHashPointer(
405     COMMAND          *command,
406     TPMI_ALG_HASH    hashAlg
407 )
408 {
409     TPM2B_DIGEST      *retVal;
410     //
411     switch(hashAlg)
412     {
413 #if ALG_SHA1
414         case ALG_SHA1_VALUE:
415             retVal = (TPM2B_DIGEST *) &command->sha1CpHash;
416             break;
417 #endif
418 #if ALG_SHA256
419         case ALG_SHA256_VALUE:
420             retVal = (TPM2B_DIGEST *) &command->sha256CpHash;
421             break;
422 #endif
423 #if ALG_SHA384
424         case ALG_SHA384_VALUE:
425             retVal = (TPM2B_DIGEST *) &command->sha384CpHash;
426             break;
427 #endif
428 #if ALG_SHA512
429         case ALG_SHA512_VALUE:
430             retVal = (TPM2B_DIGEST *) &command->sha512CpHash;
431             break;
432 #endif
433 #if ALG_SM3_256
434         case ALG_SM3_256_VALUE:
435             retVal = (TPM2B_DIGEST *) &command->sm3_256CpHash;
436             break;
437 #endif
438         default:
439             retVal = NULL;
440             break;
441     }
442     return retVal;
443 }

```

6.4.4.3 GetRpHashPointer()

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

```

444 static TPM2B_DIGEST *

```

```

445 GetRpHashPointer(
446     COMMAND      *command,
447     TPMI_ALG_HASH hashAlg
448 )
449 {
450     TPM2B_DIGEST *retVal;
451     //
452     switch(hashAlg)
453     {
454     #if ALG_SHA1
455         case ALG_SHA1_VALUE:
456             retVal = (TPM2B_DIGEST *) &command->sha1RpHash;
457             break;
458     #endif
459     #if ALG_SHA256
460         case ALG_SHA256_VALUE:
461             retVal = (TPM2B_DIGEST *) &command->sha256RpHash;
462             break;
463     #endif
464     #if ALG_SHA384
465         case ALG_SHA384_VALUE:
466             retVal = (TPM2B_DIGEST *) &command->sha384RpHash;
467             break;
468     #endif
469     #if ALG_SHA512
470         case ALG_SHA512_VALUE:
471             retVal = (TPM2B_DIGEST *) &command->sha512RpHash;
472             break;
473     #endif
474     #if ALG_SM3_256
475         case ALG_SM3_256_VALUE:
476             retVal = (TPM2B_DIGEST *) &command->sm3_256RpHash;
477             break;
478     #endif
479         default:
480             retVal = NULL;
481             break;
482     }
483     return retVal;
484 }

```

6.4.4.4 ComputeCpHash()

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

```

485 static TPM2B_DIGEST *
486 ComputeCpHash(
487     COMMAND      *command,           // IN: command parsing structure
488     TPMI_ALG_HASH hashAlg           // IN: hash algorithm
489 )
490 {
491     UINT32      i;
492     HASH_STATE   hashState;
493     TPM2B_NAME   name;
494     TPM2B_DIGEST *cpHash;
495     //
496     // cpHash = hash(commandCode [ || authName1
497     //                      [ || authName2
498     //                      [ || authName 3 ]])
499     //                      [ || parameters])
500     // A cpHash can contain just a commandCode only if the lone session is
501     // an audit session.
502     // Get pointer to the hash value
503     cpHash = GetCpHashPointer(command, hashAlg);

```

```

504     if(cpHash->t.size == 0)
505     {
506         cpHash->t.size = CryptHashStart(&hashState, hashAlg);
507         // Add commandCode.
508         CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
509         // Add authNames for each of the handles.
510         for(i = 0; i < command->handleNum; i++)
511             CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
512                                                         &name)->b);
513         // Add the parameters.
514         CryptDigestUpdate(&hashState, command->parameterSize,
515                         command->parameterBuffer);
516         // Complete the hash.
517         CryptHashEnd2B(&hashState, &cpHash->b);
518     }
519     return cpHash;
520 }

```

6.4.4.5 GetCpHash()

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

```

521 static TPM2B_DIGEST *
522 GetCpHash(
523     COMMAND          *command,
524     TPMI_ALG_HASH    hashAlg
525 )
526 {
527     TPM2B_DIGEST      *cpHash = GetCpHashPointer(command, hashAlg);
528     //
529     pAssert(cpHash->t.size != 0);
530     return cpHash;
531 }

```

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 session-> <i>templateHash</i>
FALSE(0)	template hash not equal to session-> <i>templateHash</i>

```

532 static BOOL
533 CompareTemplateHash(
534     COMMAND          *command,          // IN: parsing structure
535     SESSION          *session          // IN: session data
536 )
537 {
538     BYTE              *pBuffer = command->parameterBuffer;
539     INT32              pSize = command->parameterSize;
540     TPM2B_DIGEST      tHash;
541     UINT16             size;
542     //
543     // Only try this for the three commands for which it is intended
544     if(command->code != TPM_CC_Create
545        && command->code != TPM_CC_CreatePrimary
546        #if CC_CreateLoaded
547        && command->code != TPM_CC_CreateLoaded
548        #endif

```



```

549     )
550     return FALSE;
551     // Assume that the first parameter is a TPM2B and unmarshal the size field
552     // Note: this will not affect the parameter buffer and size in the calling
553     // function.
554     if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
555         return FALSE;
556     // reduce the space in the buffer.
557     // NOTE: this could make pSize go negative if the parameters are not correct but
558     // the unmarshaling code does not try to unmarshal if the remaining size is
559     // negative.
560     pSize -= size;
561
562     // Advance the pointer
563     pBuffer += size;
564
565     // Get the size of what should be the template
566     if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
567         return FALSE;
568     // See if this is reasonable
569     if(size > pSize)
570         return FALSE;
571     // Hash the template data
572     tHash.t.size = CryptHashBlock(session->authHashAlg, size, pBuffer,
573                                   sizeof(tHash.t.buffer), tHash.t.buffer);
574     return(MemoryEqual2B(&session->u1.templateHash.b, &tHash.b));
575 }

```

6.4.4.7 CompareNameHash()

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

```

576 BOOL
577 CompareNameHash(
578     COMMAND      *command,      // IN: main parsing structure
579     SESSION      *session       // IN: session structure with nameHash
580 )
581 {
582     HASH_STATE      hashState;
583     TPM2B_DIGEST    nameHash;
584     UINT32          i;
585     TPM2B_NAME      name;
586     //
587     nameHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
588     // Add names.
589     for(i = 0; i < command->handleNum; i++)
590         CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
591                                                         &name) ->b);
592     // Complete hash.
593     CryptHashEnd2B(&hashState, &nameHash.b);
594     // and compare
595     return MemoryEqual(session->u1.nameHash.t.buffer, nameHash.t.buffer,
596                       nameHash.t.size);
597 }

```

6.4.4.8 CheckPWAuthSession()

This function validates the authorization provided in a PWP 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 Returns	Meaning
TPM_RC_AUTH_FAIL	authorization fails and increments DA failure count
TPM_RC_BAD_AUTH	authorization fails but DA does not apply

```

598 static TPM_RC
599 CheckPWAAuthSession(
600     UINT32      sessionIndex    // IN: index of session to be processed
601 )
602 {
603     TPM2B_AUTH    authValue;
604     TPM_HANDLE    associatedHandle = s_associatedHandles[sessionIndex];
605     //
606     // Strip trailing zeros from the password.
607     MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
608
609     // Get the authValue with trailing zeros removed
610     EntityGetAuthValue(associatedHandle, &authValue);
611
612     // Success if the values are identical.
613     if(MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &authValue.b))
614     {
615         return TPM_RC_SUCCESS;
616     }
617     else                                // if the digests are not identical
618     {
619         // Invoke DA protection if applicable.
620         return IncrementLockout(sessionIndex);
621     }
622 }

```

6.4.4.9 ComputeCommandHMAC()

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

```

623 static TPM2B_DIGEST *
624 ComputeCommandHMAC(
625     COMMAND      *command,        // IN: primary control structure
626     UINT32      sessionIndex,    // IN: index of session to be processed
627     TPM2B_DIGEST *hmac           // OUT: authorization HMAC
628 )
629 {
630     TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
631     TPM2B_KEY    key;
632     BYTE         marshalBuffer[sizeof(TPMA_SESSION)];
633     BYTE         *buffer;
634     UINT32       marshalSize;
635     HMAC_STATE    hmacState;
636     TPM2B_NONCE   *nonceDecrypt;
637     TPM2B_NONCE   *nonceEncrypt;
638     SESSION       *session;
639     //
640     nonceDecrypt = NULL;
641     nonceEncrypt = NULL;
642
643     // Determine if extra nonceTPM values are going to be required.
644     // If this is the first session (sessionIndex = 0) and it is an authorization
645     // session that uses an HMAC, then check if additional session nonces are to be
646     // included.
647     if(sessionIndex == 0
648        && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
649     {
650         // If there is a decrypt session and if this is not the decrypt session,

```

```

651     // then an extra nonce may be needed.
652     if(s_decryptSessionIndex != UNDEFINED_INDEX
653        && s_decryptSessionIndex != sessionIndex)
654     {
655         // Will add the nonce for the decrypt session.
656         SESSION *decryptSession
657             = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
658         nonceDecrypt = &decryptSession->nonceTPM;
659     }
660     // Now repeat for the encrypt session.
661     if(s_encryptSessionIndex != UNDEFINED_INDEX
662        && s_encryptSessionIndex != sessionIndex
663        && s_encryptSessionIndex != s_decryptSessionIndex)
664     {
665         // Have to have the nonce for the encrypt session.
666         SESSION *encryptSession
667             = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
668         nonceEncrypt = &encryptSession->nonceTPM;
669     }
670 }
671
672 // Continue with the HMAC processing.
673 session = SessionGet(s_sessionHandles[sessionIndex]);
674
675 // Generate HMAC key.
676 MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
677
678 // Check if the session has an associated handle and if the associated entity
679 // is the one to which the session is bound. If not, add the authValue of
680 // this entity to the HMAC key.
681 // If the session is bound to the object or the session is a policy session
682 // with no authValue required, do not include the authValue in the HMAC key.
683 // Note: For a policy session, its isBound attribute is CLEARED.
684 //
685 // Include the entity authValue if it is needed
686 if(session->attributes.includeAuth == SET)
687 {
688     TPM2B_AUTH authValue;
689     // Get the entity authValue with trailing zeros removed
690     EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
691     // add the authValue to the HMAC key
692     MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
693 }
694 // if the HMAC key size is 0, a NULL string HMAC is allowed
695 if(key.t.size == 0
696    && s_inputAuthValues[sessionIndex].t.size == 0)
697 {
698     hmac->t.size = 0;
699     return hmac;
700 }
701 // Start HMAC
702 hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
703
704 // Add cpHash
705 CryptDigestUpdate2B(&hmacState.hashState,
706                    &ComputeCpHash(command, session->authHashAlg)->b);
707 // Add nonces as required
708 CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
709 CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
710 if(nonceDecrypt != NULL)
711     CryptDigestUpdate2B(&hmacState.hashState, &nonceDecrypt->b);
712 if(nonceEncrypt != NULL)
713     CryptDigestUpdate2B(&hmacState.hashState, &nonceEncrypt->b);
714 // Add sessionAttributes
715 buffer = marshalBuffer;
716 marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),

```

```

717         &buffer, NULL);
718     CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
719     // Complete the HMAC computation
720     CryptHmacEnd2B(&hmacState, &hmac->b);
721
722     return hmac;
723 }

```

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 Returns	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

```

724 static TPM_RC
725 CheckSessionHMAC(
726     COMMAND          *command,      // IN: primary control structure
727     UINT32            sessionIndex  // IN: index of session to be processed
728 )
729 {
730     TPM2B_DIGEST      hmac;          // authHMAC for comparing
731     //
732     // Compute authHMAC
733     ComputeCommandHMAC(command, sessionIndex, &hmac);
734
735     // Compare the input HMAC with the authHMAC computed above.
736     if(!MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &hmac.b))
737     {
738         // If an HMAC session has a failure, invoke the anti-hammering
739         // if it applies to the authorized entity or the session.
740         // Otherwise, just indicate that the authorization is bad.
741         return IncrementLockout(sessionIndex);
742     }
743     return TPM_RC_SUCCESS;
744 }

```

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 Returns	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

```

745 static TPM_RC
746 CheckPolicyAuthSession(
747     COMMAND      *command,      // IN: primary parsing structure
748     UINT32        sessionIndex   // IN: index of session to be processed
749 )
750 {
751     SESSION      *session;
752     TPM2B_DIGEST authPolicy;
753     TPMT_ALG_HASH policyAlg;
754     UINT8        locality;
755 //
756 // Initialize pointer to the authorization session.
757 session = SessionGet(s_sessionHandles[sessionIndex]);
758
759 // If the command is TPM2_PolicySecret(), make sure that
760 // either password or authValue is required
761 if(command->code == TPM_CC_PolicySecret
762     && session->attributes.isPasswordNeeded == CLEAR
763     && session->attributes.isAuthValueNeeded == CLEAR)
764     return TPM_RC_MODE;
765 // See if the PCR counter for the session is still valid.
766 if(!SessionPCRValueIsCurrent(session))
767     return TPM_RC_PCR_CHANGED;
768 // Get authPolicy.
769 policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
770                                &authPolicy);
771 // Compare authPolicy.
772 if(!MemoryEqual2B(&session->u2.policyDigest.b, &authPolicy.b))
773     return TPM_RC_POLICY_FAIL;
774 // Policy is OK so check if the other factors are correct
775
776 // Compare policy hash algorithm.
777 if(policyAlg != session->authHashAlg)
778     return TPM_RC_POLICY_FAIL;
779
780 // Compare timeout.
781 if(session->timeout != 0)
782 {
783     // Cannot compare time if clock stop advancing. An TPM_RC_NV_UNAVAILABLE
784     // or TPM_RC_NV_RATE error may be returned here. This doesn't mean that
785     // a new nonce will be created just that, because TPM time can't advance
786     // we can't do time-based operations.
787     RETURN_IF_NV_IS_NOT_AVAILABLE;
788
789     if((session->timeout < g_time)
790        || (session->epoch != g_timeEpoch))
791         return TPM_RC_EXPIRED;
792 }
793 // If command code is provided it must match

```

```

794     if(session->commandCode != 0)
795     {
796         if(session->commandCode != command->code)
797             return TPM_RC_POLICY_CC;
798     }
799     else
800     {
801         // If command requires a DUP or ADMIN authorization, the session must have
802         // command code set.
803         AUTH_ROLE    role = CommandAuthRole(command->index, sessionIndex);
804         if(role == AUTH_ADMIN || role == AUTH_DUP)
805             return TPM_RC_POLICY_FAIL;
806     }
807     // Check command locality.
808     {
809         BYTE          sessionLocality[sizeof(TPMA_LOCALITY)];
810         BYTE          *buffer = sessionLocality;
811
812         // Get existing locality setting in canonical form
813         sessionLocality[0] = 0;
814         TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);
815
816         // See if the locality has been set
817         if(sessionLocality[0] != 0)
818         {
819             // If so, get the current locality
820             locality = _plat__LocalityGet();
821             if(locality < 5)
822             {
823                 if(((sessionLocality[0] & (1 << locality)) == 0)
824                     || sessionLocality[0] > 31)
825                     return TPM_RC_LOCALITY;
826             }
827             else if(locality > 31)
828             {
829                 if(sessionLocality[0] != locality)
830                     return TPM_RC_LOCALITY;
831             }
832             else
833             {
834                 // Could throw an assert here but a locality error is just
835                 // as good. It just means that, whatever the locality is, it isn't
836                 // the locality requested so...
837                 return TPM_RC_LOCALITY;
838             }
839         }
840     } // end of locality check
841     // Check physical presence.
842     if(session->attributes.isPPRequired == SET
843         && !_plat__PhysicalPresenceAsserted())
844         return TPM_RC_PP;
845     // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
846     // DUP role for this handle.
847     if(session->u1.cpHash.b.size != 0)
848     {
849         BOOL          OK;
850         if(session->attributes.isCpHashDefined)
851             // Compare cpHash.
852             OK = MemoryEqual2B(&session->u1.cpHash.b,
853                             &ComputeCpHash(command, session->authHashAlg->b));
854         else if(session->attributes.isTemplateSet)
855             OK = CompareTemplateHash(command, session);
856         else
857             OK = CompareNameHash(command, session);
858         if(!OK)
859             return TPM_RCS_POLICY_FAIL;

```



```

860     }
861     if(session->attributes.checkNvWritten)
862     {
863         NV_REF          locator;
864         NV_INDEX        *nvIndex;
865         //
866         // If this is not an NV index, the policy makes no sense so fail it.
867         if(HandleGetType(s_associatedHandles[sessionIndex]) != TPM_HT_NV_INDEX)
868             return TPM_RC_POLICY_FAIL;
869         // Get the index data
870         nvIndex = NvGetIndexInfo(s_associatedHandles[sessionIndex], &locator);
871
872         // Make sure that the TPMA_WRITTEN_ATTRIBUTE has the desired state
873         if((IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
874             != (session->attributes.nvWrittenState == SET))
875             return TPM_RC_POLICY_FAIL;
876     }
877     return TPM_RC_SUCCESS;
878 }

```

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 Returns	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

```

879 static TPM_RC
880 RetrieveSessionData(
881     COMMAND        *command           // IN: main parsing structure for command
882 )
883 {
884     int            i;
885     TPM_RC         result;
886     SESSION        *session;
887     TPMA_SESSION   sessionAttributes;
888     TPM_HT         sessionType;
889     INT32          sessionIndex;
890     TPM_RC         errorIndex;
891     //
892     s_decryptSessionIndex = UNDEFINED_INDEX;
893     s_encryptSessionIndex = UNDEFINED_INDEX;
894     s_auditSessionIndex = UNDEFINED_INDEX;
895
896     for(sessionIndex = 0; command->authSize > 0; sessionIndex++)
897     {
898         errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
899
900         // If maximum allowed number of sessions has been parsed, return a size
901         // error with a session number that is larger than the number of allowed
902         // sessions
903         if(sessionIndex == MAX_SESSION_NUM)
904             return TPM_RCS_SIZE + errorIndex;
905         // make sure that the associated handle for each session starts out
906         // unassigned
907         s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
908
909         // First parameter: Session handle.
910         result = TPMI_SH_AUTH_SESSION_Unmarshal(

```



```

911         &s_sessionHandles[sessionIndex],
912         &command->parameterBuffer,
913         &command->authSize, TRUE);
914 if(result != TPM_RC_SUCCESS)
915     return result + TPM_RC_S + g_rcIndex[sessionIndex];
916 // Second parameter: Nonce.
917 result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
918                               &command->parameterBuffer,
919                               &command->authSize);
920 if(result != TPM_RC_SUCCESS)
921     return result + TPM_RC_S + g_rcIndex[sessionIndex];
922 // Third parameter: sessionAttributes.
923 result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
924                                &command->parameterBuffer,
925                                &command->authSize);
926 if(result != TPM_RC_SUCCESS)
927     return result + TPM_RC_S + g_rcIndex[sessionIndex];
928 // Fourth parameter: authValue (PW or HMAC).
929 result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
930                              &command->parameterBuffer,
931                              &command->authSize);
932 if(result != TPM_RC_SUCCESS)
933     return result + errorIndex;
934
935 sessionAttributes = s_attributes[sessionIndex];
936 if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
937 {
938     // A PWAP session needs additional processing.
939     // Can't have any attributes set other than continueSession bit
940     if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt)
941        || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt)
942        || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit)
943        || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
944        || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset))
945         return TPM_RCS_ATTRIBUTES + errorIndex;
946     // The nonce size must be zero.
947     if(s_nonceCaller[sessionIndex].t.size != 0)
948         return TPM_RCS_NONCE + errorIndex;
949     continue;
950 }
951 // For not password sessions...
952 // Find out if the session is loaded.
953 if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
954     return TPM_RC_REFERENCE_S0 + sessionIndex;
955 sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
956 session = SessionGet(s_sessionHandles[sessionIndex]);
957
958 // Check if the session is an HMAC/policy session.
959 if((session->attributes.isPolicy == SET
960     && sessionType == TPM_HT_HMAC_SESSION)
961    || (session->attributes.isPolicy == CLEAR
962        && sessionType == TPM_HT_POLICY_SESSION))
963     return TPM_RCS_HANDLE + errorIndex;
964 // Check that this handle has not previously been used.
965 for(i = 0; i < sessionIndex; i++)
966 {
967     if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
968         return TPM_RCS_HANDLE + errorIndex;
969 }
970 // If the session is used for parameter encryption or audit as well, set
971 // the corresponding indexes.
972
973 // First process decrypt.
974 if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt))
975 {
976     // Check if the commandCode allows command parameter encryption.

```

```

977         if(DecryptSize(command->index) == 0)
978             return TPM_RCS_ATTRIBUTES + errorIndex;
979         // Encrypt attribute can only appear in one session
980         if(s_decryptSessionIndex != UNDEFINED_INDEX)
981             return TPM_RCS_ATTRIBUTES + errorIndex;
982         // Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
983         if(session->symmetric.algorithm == TPM_ALG_NULL)
984             return TPM_RCS_SYMMETRIC + errorIndex;
985         // All checks passed, so set the index for the session used to decrypt
986         // a command parameter.
987         s_decryptSessionIndex = sessionIndex;
988     }
989     // Now process encrypt.
990     if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt))
991     {
992         // Check if the commandCode allows response parameter encryption.
993         if(EncryptSize(command->index) == 0)
994             return TPM_RCS_ATTRIBUTES + errorIndex;
995         // Encrypt attribute can only appear in one session.
996         if(s_encryptSessionIndex != UNDEFINED_INDEX)
997             return TPM_RCS_ATTRIBUTES + errorIndex;
998         // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
999         if(session->symmetric.algorithm == TPM_ALG_NULL)
1000             return TPM_RCS_SYMMETRIC + errorIndex;
1001         // All checks passed, so set the index for the session used to encrypt
1002         // a response parameter.
1003         s_encryptSessionIndex = sessionIndex;
1004     }
1005     // At last process audit.
1006     if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit))
1007     {
1008         // Audit attribute can only appear in one session.
1009         if(s_auditSessionIndex != UNDEFINED_INDEX)
1010             return TPM_RCS_ATTRIBUTES + errorIndex;
1011         // An audit session can not be policy session.
1012         if(HandleGetType(s_sessionHandles[sessionIndex])
1013            == TPM_HT_POLICY_SESSION)
1014             return TPM_RCS_ATTRIBUTES + errorIndex;
1015         // If this is a reset of the audit session, or the first use
1016         // of the session as an audit session, it doesn't matter what
1017         // the exclusive state is. The session will become exclusive.
1018         if(!IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset)
1019            && session->attributes.isAudit == SET)
1020         {
1021             // Not first use or reset. If auditExclusive is SET, then this
1022             // session must be the current exclusive session.
1023             if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
1024                && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])
1025                 return TPM_RC_EXCLUSIVE;
1026         }
1027         s_auditSessionIndex = sessionIndex;
1028     }
1029     // Initialize associated handle as undefined. This will be changed when
1030     // the handles are processed.
1031     s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
1032 }
1033 command->sessionNum = sessionIndex;
1034 return TPM_RC_SUCCESS;
1035 }

```

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 Returns	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

```

1036 static TPM_RC
1037 CheckLockedOut(
1038     BOOL                lockoutAuthCheck    // IN: TRUE if checking is for lockoutAuth
1039 )
1040 {
1041     // If NV is unavailable, and current cycle state recorded in NV is not
1042     // SU_NONE_VALUE, refuse to check any authorization because we would
1043     // not be able to handle a DA failure.
1044     if(!NV_IS_AVAILABLE && NV_IS_ORDERLY)
1045         return g_NvStatus;
1046     // Check if DA info needs to be updated in NV.
1047     if(s_DAPendingOnNV)
1048     {
1049         // If NV is accessible,
1050         RETURN_IF_NV_IS_NOT_AVAILABLE;
1051
1052         // ... write the pending DA data and proceed.
1053         NV_SYNC_PERSISTENT(lockOutAuthEnabled);
1054         NV_SYNC_PERSISTENT(failedTries);
1055         s_DAPendingOnNV = FALSE;
1056     }
1057     // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
1058     // is disabled...
1059     if(lockoutAuthCheck)
1060     {
1061         if(gp.lockOutAuthEnabled == FALSE)
1062             return TPM_RC_LOCKOUT;
1063     }
1064     else
1065     {
1066         // ... or if the number of failed tries has been maxed out.
1067         if(gp.failedTries >= gp.maxTries)
1068             return TPM_RC_LOCKOUT;
1069     }
1070     #if USE_DA_USED
1071     // If the daUsed flag is not SET, then no DA validation until the
1072     // daUsed state is written to NV
1073     if(!g_daUsed)
1074     {
1075         RETURN_IF_NV_IS_NOT_AVAILABLE;
1076         g_daUsed = TRUE;
1077         gp.orderlyState = SU_DA_USED_VALUE;
1078         NV_SYNC_PERSISTENT(orderlyState);
1079         return TPM_RC_RETRY;
1080     }
1081     #endif
1082     return TPM_RC_SUCCESS;
1083 }

```

6.4.4.14 CheckAuthSession()

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

Error Returns	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	<i>authValue</i> or <i>authPolicy</i> unavailable

```

1084 static TPM_RC
1085 CheckAuthSession(
1086     COMMAND      *command,          // IN: primary parsing structure
1087     UINT32        sessionIndex      // IN: index of session to be processed
1088 )
1089 {
1090     TPM_RC        result = TPM_RC_SUCCESS;
1091     SESSION       *session = NULL;
1092     TPM_HANDLE    sessionHandle = s_sessionHandles[sessionIndex];
1093     TPM_HANDLE    associatedHandle = s_associatedHandles[sessionIndex];
1094     TPM_HT        sessionHandleType = HandleGetType(sessionHandle);
1095     //
1096     pAssert(sessionHandle != TPM_RH_UNASSIGNED);
1097
1098     // Take care of physical presence
1099     if(associatedHandle == TPM_RH_PLATFORM)
1100     {
1101         // If the physical presence is required for this command, check for PP
1102         // assertion. If it isn't asserted, no point going any further.
1103         if(PhysicalPresenceIsRequired(command->index)
1104            && !_plat_PhysicalPresenceAsserted())
1105             return TPM_RC_PP;
1106     }
1107     if(sessionHandle != TPM_RS_PW)
1108     {
1109         session = SessionGet(sessionHandle);
1110
1111         // Set includeAuth to indicate if DA checking will be required and if the
1112         // authValue will be included in any HMAC.
1113         if(sessionHandleType == TPM_HT_POLICY_SESSION)
1114         {
1115             // For a policy session, will check the DA status of the entity if either
1116             // isAuthValueNeeded or isPasswordNeeded is SET.
1117             session->attributes.includeAuth =
1118                 session->attributes.isAuthValueNeeded
1119                 || session->attributes.isPasswordNeeded;
1120         }
1121         else
1122         {
1123             // For an HMAC session, need to check unless the session
1124             // is bound.
1125             session->attributes.includeAuth =
1126                 !IsSessionBindEntity(s_associatedHandles[sessionIndex], session);
1127         }

```

```

1128     }
1129     // If the authorization session is going to use an authValue, then make sure
1130     // that access to that authValue isn't locked out.
1131     // Note: session == NULL for a PW session.
1132     if(session == NULL || session->attributes.includeAuth)
1133     {
1134         // See if entity is subject to logout.
1135         if(!IsDAExempted(associatedHandle))
1136         {
1137             // See if in logout
1138             result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
1139             if(result != TPM_RC_SUCCESS)
1140                 return result;
1141         }
1142     }
1143     // Policy or HMAC+PW?
1144     if(sessionHandleType != TPM_HT_POLICY_SESSION)
1145     {
1146         // for non-policy session make sure that a policy session is not required
1147         if(IsPolicySessionRequired(command->index, sessionIndex))
1148             return TPM_RC_AUTH_TYPE;
1149         // The authValue must be available.
1150         // Note: The authValue is going to be "used" even if it is an EmptyAuth.
1151         // and the session is bound.
1152         if(!IsAuthValueAvailable(associatedHandle, command->index, sessionIndex))
1153             return TPM_RC_AUTH_UNAVAILABLE;
1154     }
1155     else
1156     {
1157         // ... see if the entity has a policy, ...
1158         // Note: IsAuthPolicyAvailable will return FALSE if the sensitive area of the
1159         // object is not loaded
1160         if(!IsAuthPolicyAvailable(associatedHandle, command->index, sessionIndex))
1161             return TPM_RC_AUTH_UNAVAILABLE;
1162         // ... and check the policy session.
1163         result = CheckPolicyAuthSession(command, sessionIndex);
1164         if(result != TPM_RC_SUCCESS)
1165             return result;
1166     }
1167     // Check authorization according to the type
1168     if(session == NULL || session->attributes.isPasswordNeeded == SET)
1169         result = CheckPWAuthSession(sessionIndex);
1170     else
1171         result = CheckSessionHMAC(command, sessionIndex);
1172     // Do processing for PIN Indexes are only three possibilities for 'result' at
1173     // this point: TPM_RC_SUCCESS, TPM_RC_AUTH_FAIL, and TPM_RC_BAD_AUTH.
1174     // For all these cases, we would have to process a PIN index if the
1175     // authValue of the index was used for authorization.
1176     // See if we need to do anything to a PIN index
1177     if(TPM_HT_NV_INDEX == HandleGetType(associatedHandle))
1178     {
1179         NV_REF          locator;
1180         NV_INDEX        *nvIndex = NvGetIndexInfo(associatedHandle, &locator);
1181         NV_PIN          pinData;
1182         TPMA_NV          nvAttributes;
1183     //
1184     pAssert(nvIndex != NULL);
1185     nvAttributes = nvIndex->publicArea.attributes;
1186     // If this is a PIN FAIL index and the value has been written
1187     // then we can update the counter (increment or clear)
1188     if(IsNvPinFailIndex(nvAttributes)
1189         && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN))
1190     {
1191         pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1192         if(result != TPM_RC_SUCCESS)
1193             pinData.pin.pinCount++;

```

```

1194         else
1195             pinData.pin.pinCount = 0;
1196             NvWriteUINT64Data(nvIndex, pinData.intVal);
1197         }
1198         // If this is a PIN PASS Index, increment if we have used the
1199         // authorization value for anything other than NV_Read.
1200         // NOTE: If the counter has already hit the limit, then we
1201         // would not get here because the authorization value would not
1202         // be available and the TPM would have returned before it gets here
1203         else if(IsNvPinPassIndex(nvAttributes)
1204             && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN)
1205             && result == TPM_RC_SUCCESS)
1206         {
1207             // If the access is valid, then increment the use counter
1208             pinData.intVal = NvGetUINT64Data(nvIndex, locator);
1209             pinData.pin.pinCount++;
1210             NvWriteUINT64Data(nvIndex, pinData.intVal);
1211         }
1212     }
1213     return result;
1214 }
1215 #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 Returns	Meaning
TPM_RC_NV_UNAVAILABLE	NV is not available for write
TPM_RC_NV_RATE	NV is rate limiting

```

1216 static TPM_RC
1217 CheckCommandAudit(
1218     COMMAND *command
1219 )
1220 {
1221     // If the audit digest is clear and command audit is required, NV must be
1222     // available so that TPM2_GetCommandAuditDigest() is able to increment
1223     // audit counter. If NV is not available, the function bails out to prevent
1224     // the TPM from attempting an operation that would fail anyway.
1225     if(gp.commandAuditDigest.t.size == 0
1226         || GetCommandCode(command->index) == TPM_CC_GetCommandAuditDigest)
1227     {
1228         RETURN_IF_NV_IS_NOT_AVAILABLE;
1229     }
1230     // Make sure that the cpHash is computed for the algorithm
1231     ComputeCpHash(command, gp.auditHashAlg);
1232     return TPM_RC_SUCCESS;
1233 }
1234 #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 Returns	Meaning
various	parsing failure or authorization failure

```

1235 TPM_RC
1236 ParseSessionBuffer(
1237     COMMAND      *command          // IN: the structure that contains
1238 )
1239 {
1240     TPM_RC      result;
1241     UINT32      i;
1242     INT32       size = 0;
1243     TPM2B_AUTH  extraKey;
1244     UINT32      sessionIndex;
1245     TPM_RC      errorIndex;
1246     SESSION     *session = NULL;
1247     //
1248     // Check if a command allows any session in its session area.
1249     if(!IsSessionAllowed(command->index))
1250         return TPM_RC_AUTH_CONTEXT;
1251     // Default-initialization.
1252     command->sessionNum = 0;
1253
1254     result = RetrieveSessionData(command);
1255     if(result != TPM_RC_SUCCESS)
1256         return result;
1257     // There is no command in the TPM spec that has more handles than
1258     // MAX_SESSION_NUM.
1259     pAssert(command->handleNum <= MAX_SESSION_NUM);
1260
1261     // Associate the session with an authorization handle.
1262     for(i = 0; i < command->handleNum; i++)
1263     {
1264         if(CommandAuthRole(command->index, i) != AUTH_NONE)
1265         {
1266             // If the received session number is less than the number of handles
1267             // that requires authorization, an error should be returned.
1268             // Note: for all the TPM 2.0 commands, handles requiring
1269             // authorization come first in a command input and there are only ever
1270             // two values requiring authorization
1271             if(i > (command->sessionNum - 1))
1272                 return TPM_RC_AUTH_MISSING;
1273             // Record the handle associated with the authorization session
1274             s_associatedHandles[i] = command->handles[i];
1275         }
1276     }
1277     // Consistency checks are done first to avoid authorization failure when the
1278     // command will not be executed anyway.
1279     for(sessionIndex = 0; sessionIndex < command->sessionNum; sessionIndex++)
1280     {
1281         errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
1282         // PW session must be an authorization session
1283         if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
1284         {
1285             if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
1286                 return TPM_RCS_HANDLE + errorIndex;
1287             // a password session can't be audit, encrypt or decrypt
1288             if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1289                 || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1290                 || IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1291                 return TPM_RCS_ATTRIBUTES + errorIndex;
1292             session = NULL;
1293         }
1294         else
1295     {

```



```

1296     session = SessionGet(s_sessionHandles[sessionIndex]);
1297
1298     // A trial session can not appear in session area, because it cannot
1299     // be used for authorization, audit or encrypt/decrypt.
1300     if(session->attributes.isTrialPolicy == SET)
1301         return TPM_RCS_ATTRIBUTES + errorIndex;
1302
1303     // See if the session is bound to a DA protected entity
1304     // NOTE: Since a policy session is never bound, a policy is still
1305     // usable even if the object is DA protected and the TPM is in
1306     // lockout.
1307     if(session->attributes.isDaBound == SET)
1308     {
1309         result = CheckLockedOut(session->attributes.isLockoutBound == SET);
1310         if(result != TPM_RC_SUCCESS)
1311             return result;
1312     }
1313     // If this session is for auditing, make sure the cpHash is computed.
1314     if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit))
1315         ComputeCpHash(command, session->authHashAlg);
1316 }
1317
1318 // if the session has an associated handle, check the authorization
1319 if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
1320 {
1321     result = CheckAuthSession(command, sessionIndex);
1322     if(result != TPM_RC_SUCCESS)
1323         return RcSafeAddToResult(result, errorIndex);
1324 }
1325 else
1326 {
1327     // a session that is not for authorization must either be encrypt,
1328     // decrypt, or audit
1329     if(!IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)
1330        && !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt)
1331        && !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
1332         return TPM_RCS_ATTRIBUTES + errorIndex;
1333
1334     // no authValue included in any of the HMAC computations
1335     pAssert(session != NULL);
1336     session->attributes.includeAuth = CLEAR;
1337
1338     // check HMAC for encrypt/decrypt/audit only sessions
1339     result = CheckSessionHMAC(command, sessionIndex);
1340     if(result != TPM_RC_SUCCESS)
1341         return RcSafeAddToResult(result, errorIndex);
1342 }
1343 }
1344 #ifdef TPM_CC_GetCommandAuditDigest
1345     // Check if the command should be audited. Need to do this before any parameter
1346     // encryption so that the cpHash for the audit is correct
1347     if(CommandAuditIsRequired(command->index))
1348     {
1349         result = CheckCommandAudit(command);
1350         if(result != TPM_RC_SUCCESS)
1351             return result; // No session number to reference
1352     }
1353 #endif
1354 // Decrypt the first parameter if applicable. This should be the last operation
1355 // in session processing.
1356 // If the encrypt session is associated with a handle and the handle's
1357 // authValue is available, then authValue is concatenated with sessionKey to
1358 // generate encryption key, no matter if the handle is the session bound entity
1359 // or not.
1360 if(s_decryptSessionIndex != UNDEFINED_INDEX)
1361 {

```

```

1362     // If this is an authorization session, include the authValue in the
1363     // generation of the decryption key
1364     if(s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED)
1365     {
1366         EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
1367                             &extraKey);
1368     }
1369     else
1370     {
1371         extraKey.b.size = 0;
1372     }
1373     size = DecryptSize(command->index);
1374     result = CryptParameterDecryption(s_sessionHandles[s_decryptSessionIndex],
1375                                     &s_nonceCaller[s_decryptSessionIndex].b,
1376                                     command->parameterSize, (UINT16)size,
1377                                     &extraKey,
1378                                     command->parameterBuffer);
1379     if(result != TPM_RC_SUCCESS)
1380         return RcSafeAddToResult(result,
1381                                 TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
1382 }
1383
1384 return TPM_RC_SUCCESS;
1385 }

```

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 Returns	Meaning
TPM_RC_AUTH_MISSING	failure - one or more handles require authorization

```

1386 TPM_RC
1387 CheckAuthNoSession(
1388     COMMAND *command // IN: command parsing structure
1389 )
1390 {
1391     UINT32 i;
1392     TPM_RC result = TPM_RC_SUCCESS;
1393     //
1394     // Check if the command requires authorization
1395     for(i = 0; i < command->handleNum; i++)
1396     {
1397         if(CommandAuthRole(command->index, i) != AUTH_NONE)
1398             return TPM_RC_AUTH_MISSING;
1399     }
1400 #ifdef TPM_CC_GetCommandAuditDigest
1401     // Check if the command should be audited.
1402     if(CommandAuditIsRequired(command->index))
1403     {
1404         result = CheckCommandAudit(command);
1405         if(result != TPM_RC_SUCCESS)
1406             return result;
1407     }
1408 #endif
1409     // Initialize number of sessions to be 0
1410     command->sessionNum = 0;
1411
1412     return TPM_RC_SUCCESS;
1413 }

```

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.

```

1414 static TPM2B_DIGEST *
1415 ComputeRpHash(
1416     COMMAND      *command,          // IN: command structure
1417     TPM_ALG_ID    hashAlg,          // IN: hash algorithm to compute rpHash
1418 )
1419 {
1420     TPM2B_DIGEST *rpHash = GetRpHashPointer(command, hashAlg);
1421     HASH_STATE    hashState;
1422     //
1423     if(rpHash->t.size == 0)
1424     {
1425         // rpHash := hash(responseCode || commandCode || parameters)
1426
1427         // Initiate hash creation.
1428         rpHash->t.size = CryptHashStart(&hashState, hashAlg);
1429
1430         // Add hash constituents.
1431         CryptDigestUpdateInt(&hashState, sizeof(TPM_RC), TPM_RC_SUCCESS);
1432         CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
1433         CryptDigestUpdate(&hashState, command->parameterSize,
1434                         command->parameterBuffer);
1435         // Complete hash computation.
1436         CryptHashEnd2B(&hashState, &rpHash->b);
1437     }
1438     return rpHash;
1439 }

```

6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```

1440 static void
1441 InitAuditSession(
1442     SESSION      *session          // session to be initialized
1443 )
1444 {
1445     // Mark session as an audit session.
1446     session->attributes.isAudit = SET;
1447
1448     // Audit session can not be bound.
1449     session->attributes.isBound = CLEAR;
1450
1451     // Size of the audit log is the size of session hash algorithm digest.
1452     session->u2.auditDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
1453
1454     // Set the original digest value to be 0.
1455     MemorySet(&session->u2.auditDigest.t.buffer,
1456             0,
1457             session->u2.auditDigest.t.size);
1458     return;

```

1459 }

6.4.5.4 UpdateAuditDigest

Function to update an audit digest

```

1460 static void
1461 UpdateAuditDigest(
1462     COMMAND      *command,
1463     TPMI_ALG_HASH hashAlg,
1464     TPM2B_DIGEST *digest
1465 )
1466 {
1467     HASH_STATE hashState;
1468     TPM2B_DIGEST *cpHash = GetCpHash(command, hashAlg);
1469     TPM2B_DIGEST *rpHash = ComputeRpHash(command, hashAlg);
1470     //
1471     pAssert(cpHash != NULL);
1472
1473     // digestNew := hash (digestOld || cpHash || rpHash)
1474     // Start hash computation.
1475     digest->t.size = CryptHashStart(&hashState, hashAlg);
1476     // Add old digest.
1477     CryptDigestUpdate2B(&hashState, &digest->b);
1478     // Add cpHash
1479     CryptDigestUpdate2B(&hashState, &cpHash->b);
1480     // Add rpHash
1481     CryptDigestUpdate2B(&hashState, &rpHash->b);
1482     // Finalize the hash.
1483     CryptHashEnd2B(&hashState, &digest->b);
1484 }

```

6.4.5.5 Audit()

This function updates the audit digest in an audit session.

```

1485 static void
1486 Audit(
1487     COMMAND      *command,           // IN: primary control structure
1488     SESSION      *auditSession      // IN: loaded audit session
1489 )
1490 {
1491     UpdateAuditDigest(command, auditSession->authHashAlg,
1492                       &auditSession->u2.auditDigest);
1493     return;
1494 }
1495 #ifdef TPM_CC_GetCommandAuditDigest

```

6.4.5.6 CommandAudit()

This function updates the command audit digest.

```

1496 static void
1497 CommandAudit(
1498     COMMAND      *command           // IN:
1499 )
1500 {
1501     // If the digest.size is one, it indicates the special case of changing
1502     // the audit hash algorithm. For this case, no audit is done on exit.
1503     // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
1504     // force an update to the NV on exit so that the change in digest will

```

```

1505 // be recorded. So, it is safe to exit here without setting any flags
1506 // because the digest change will be written to NV when this code exits.
1507 if(gr.commandAuditDigest.t.size == 1)
1508 {
1509     gr.commandAuditDigest.t.size = 0;
1510     return;
1511 }
1512 // If the digest size is zero, need to start a new digest and increment
1513 // the audit counter.
1514 if(gr.commandAuditDigest.t.size == 0)
1515 {
1516     gr.commandAuditDigest.t.size = CryptHashGetDigestSize(gp.auditHashAlg);
1517     MemorySet(gr.commandAuditDigest.t.buffer,
1518               0,
1519               gr.commandAuditDigest.t.size);
1520
1521     // Bump the counter and save its value to NV.
1522     gp.auditCounter++;
1523     NV_SYNC_PERSISTENT(auditCounter);
1524 }
1525 UpdateAuditDigest(command, gp.auditHashAlg, &gr.commandAuditDigest);
1526 return;
1527 }
1528 #endif

```

6.4.5.7 UpdateAuditSessionStatus()

Function to update the internal audit related states of a session. It

- a) initializes the session as audit session and sets it to be exclusive if this is the first time it is used for audit or audit reset was requested;
- b) reports exclusive audit session;
- c) extends audit log; and
- d) clears exclusive audit session if no audit session found in the command.

```

1529 static void
1530 UpdateAuditSessionStatus(
1531     COMMAND *command // IN: primary control structure
1532 )
1533 {
1534     UINT32 i;
1535     TPM_HANDLE auditSession = TPM_RH_UNASSIGNED;
1536 //
1537 // Iterate through sessions
1538 for(i = 0; i < command->sessionNum; i++)
1539 {
1540     SESSION *session;
1541 //
1542 // PW session do not have a loaded session and can not be an audit
1543 // session either. Skip it.
1544 if(s_sessionHandles[i] == TPM_RS_PW)
1545     continue;
1546 session = SessionGet(s_sessionHandles[i]);
1547
1548 // If a session is used for audit
1549 if(IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, audit))
1550 {
1551     // An audit session has been found
1552     auditSession = s_sessionHandles[i];
1553
1554     // If the session has not been an audit session yet, or
1555     // the auditSetting bits indicate a reset, initialize it and set

```

```

1556         // it to be the exclusive session
1557         if(session->attributes.isAudit == CLEAR
1558            || IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditReset))
1559         {
1560             InitAuditSession(session);
1561             g_exclusiveAuditSession = auditSession;
1562         }
1563         else
1564         {
1565             // Check if the audit session is the current exclusive audit
1566             // session and, if not, clear previous exclusive audit session.
1567             if(g_exclusiveAuditSession != auditSession)
1568                 g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1569         }
1570         // Report audit session exclusivity.
1571         if(g_exclusiveAuditSession == auditSession)
1572         {
1573             SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1574         }
1575         else
1576         {
1577             CLEAR_ATTRIBUTE(s_attributes[i], TPMA_SESSION, auditExclusive);
1578         }
1579         // Extend audit log.
1580         Audit(command, session);
1581     }
1582 }
1583 // If no audit session is found in the command, and the command allows
1584 // a session then, clear the current exclusive
1585 // audit session.
1586 if(auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(command->index))
1587 {
1588     g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1589 }
1590 return;
1591 }

```

6.4.5.8 ComputeResponseHMAC()

Function to compute HMAC for authorization session in a response.

```

1592 static void
1593 ComputeResponseHMAC(
1594     COMMAND          *command,           // IN: command structure
1595     UINT32            sessionIndex,       // IN: session index to be processed
1596     SESSION           *session,           // IN: loaded session
1597     TPM2B_DIGEST      *hmac,              // OUT: authHMAC
1598 )
1599 {
1600     TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
1601     TPM2B_KEY          key;               // HMAC key
1602     BYTE               marshalBuffer[sizeof(TPMA_SESSION)];
1603     BYTE               *buffer;
1604     UINT32              marshalSize;
1605     HMAC_STATE          hmacState;
1606     TPM2B_DIGEST      *rpHash = ComputeRpHash(command, session->authHashAlg);
1607 //
1608     // Generate HMAC key
1609     MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
1610
1611     // Add the object authValue if required
1612     if(session->attributes.includeAuth == SET)
1613     {
1614         // Note: includeAuth may be SET for a policy that is used in

```

```

1615     // UndefinedSpaceSpecial(). At this point, the Index has been deleted
1616     // so the includeAuth will have no meaning. However, the
1617     // s_associatedHandles[] value for the session is now set to TPM_RH_NULL so
1618     // this will return the authValue associated with TPM_RH_NULL and that is
1619     // and empty buffer.
1620     TPM2B_AUTH          authValue;
1621 //
1622     // Get the authValue with trailing zeros removed
1623     EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
1624
1625     // Add it to the key
1626     MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
1627 }
1628
1629 // if the HMAC key size is 0, the response HMAC is computed according to the
1630 // input HMAC
1631 if(key.t.size == 0
1632    && s_inputAuthValues[sessionIndex].t.size == 0)
1633 {
1634     hmac->t.size = 0;
1635     return;
1636 }
1637 // Start HMAC computation.
1638 hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);
1639
1640 // Add hash components.
1641 CryptDigestUpdate2B(&hmacState.hashState, &rpHash->b);
1642 CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
1643 CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
1644
1645 // Add session attributes.
1646 buffer = marshalBuffer;
1647 marshalSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
1648 CryptDigestUpdate(&hmacState.hashState, marshalSize, marshalBuffer);
1649
1650 // Finalize HMAC.
1651 CryptHmacEnd2B(&hmacState, &hmac->b);
1652
1653 return;
1654 }

```

6.4.5.9 UpdateInternalSession()

Updates internal sessions:

- a) Restarts session time.
- b) Clears a policy session since nonce is rolling.

```

1655 static void
1656 UpdateInternalSession(
1657     SESSION      *session,      // IN: the session structure
1658     UINT32       i,             // IN: session number
1659 )
1660 {
1661     // If nonce is rolling in a policy session, the policy related data
1662     // will be re-initialized.
1663     if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION
1664        && IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1665     {
1666         // When the nonce rolls it starts a new timing interval for the
1667         // policy session.
1668         SessionResetPolicyData(session);
1669         SessionSetStartTime(session);
1670     }

```



```

1671     return;
1672 }

```

6.4.5.10 BuildSingleResponseAuth()

Function to compute response HMAC value for a policy or HMAC session.

```

1673 static TPM2B_NONCE *
1674 BuildSingleResponseAuth(
1675     COMMAND      *command,      // IN: command structure
1676     UINT32        sessionIndex,  // IN: session index to be processed
1677     TPM2B_AUTH    *auth         // OUT: authHMAC
1678 )
1679 {
1680     // Fill in policy/HMAC based session response.
1681     SESSION *session = SessionGet(s_sessionHandles[sessionIndex]);
1682     //
1683     // If the session is a policy session with isPasswordNeeded SET, the
1684     // authorization field is empty.
1685     if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
1686         && session->attributes.isPasswordNeeded == SET)
1687         auth->t.size = 0;
1688     else
1689         // Compute response HMAC.
1690         ComputeResponseHMAC(command, sessionIndex, session, auth);
1691
1692     UpdateInternalSession(session, sessionIndex);
1693     return &session->nonceTPM;
1694 }

```

6.4.5.11 UpdateAllNonceTPM()

Updates TPM nonce for all sessions in command.

```

1695 static void
1696 UpdateAllNonceTPM(
1697     COMMAND *command // IN: controlling structure
1698 )
1699 {
1700     UINT32 i;
1701     SESSION *session;
1702     //
1703     for(i = 0; i < command->sessionNum; i++)
1704     {
1705         // If not a PW session, compute the new nonceTPM.
1706         if(s_sessionHandles[i] != TPM_RS_PW)
1707         {
1708             session = SessionGet(s_sessionHandles[i]);
1709             // Update nonceTPM in both internal session and response.
1710             CryptRandomGenerate(session->nonceTPM.t.size,
1711                                session->nonceTPM.t.buffer);
1712         }
1713     }
1714     return;
1715 }

```

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.

```

1716 void
1717 BuildResponseSession(
1718     COMMAND      *command          // IN: structure that has relevant command
1719                                     // information
1720 )
1721 {
1722     pAssert(command->authSize == 0);
1723
1724     // Reset the parameter buffer to point to the start of the parameters so that
1725     // there is a starting point for any rpHash that might be generated and so there
1726     // is a place where parameter encryption would start
1727     command->parameterBuffer = command->responseBuffer - command->parameterSize;
1728
1729     // Session nonces should be updated before parameter encryption
1730     if(command->tag == TPM_ST_SESSIONS)
1731     {
1732         UpdateAllNonceTPM(command);
1733
1734         // Encrypt first parameter if applicable. Parameter encryption should
1735         // happen after nonce update and before any rpHash is computed.
1736         // If the encrypt session is associated with a handle, the authValue of
1737         // this handle will be concatenated with sessionKey to generate
1738         // encryption key, no matter if the handle is the session bound entity
1739         // or not. The authValue is added to sessionKey only when the authValue
1740         // is available.
1741         if(s_encryptSessionIndex != UNDEFINED_INDEX)
1742         {
1743             UINT32      size;
1744             TPM2B_AUTH   extraKey;
1745
1746             //
1747             extraKey.b.size = 0;
1748             // If this is an authorization session, include the authValue in the
1749             // generation of the encryption key
1750             if(s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED)
1751             {
1752                 EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
1753                                     &extraKey);
1754             }
1755             size = EncryptSize(command->index);
1756             CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
1757                                     &s_nonceCaller[s_encryptSessionIndex].b,
1758                                     (UINT16)size,
1759                                     &extraKey,
1760                                     command->parameterBuffer);
1761         }
1762         // Audit sessions should be processed regardless of the tag because
1763         // a command with no session may cause a change of the exclusivity state.
1764         UpdateAuditSessionStatus(command);
1765         #if CC_GetCommandAuditDigest
1766             // Command Audit
1767             if(CommandAuditIsRequired(command->index))
1768                 CommandAudit(command);
1769         #endif
1770         // Process command with sessions.
1771         if(command->tag == TPM_ST_SESSIONS)
1772         {
1773             UINT32      i;
1774
1775             //
1776             pAssert(command->sessionNum > 0);
1777
1778             // Iterate over each session in the command session area, and create

```

```

1778 // corresponding sessions for response.
1779 for(i = 0; i < command->sessionNum; i++)
1780 {
1781     TPM2B_NONCE    *nonceTPM;
1782     TPM2B_DIGEST    responseAuth;
1783     // Make sure that continueSession is SET on any Password session.
1784     // This makes it marginally easier for the management software
1785     // to keep track of the closed sessions.
1786     if(s_sessionHandles[i] == TPM_RS_PW)
1787     {
1788         SET_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession);
1789         responseAuth.t.size = 0;
1790         nonceTPM = (TPM2B_NONCE *)&responseAuth;
1791     }
1792     else
1793     {
1794         // Compute the response HMAC and get a pointer to the nonce used.
1795         // This function will also update the values if needed. Note, the
1796         nonceTPM = BuildSingleResponseAuth(command, i, &responseAuth);
1797     }
1798     command->authSize += TPM2B_NONCE_Marshal(nonceTPM,
1799                                             &command->responseBuffer,
1800                                             NULL);
1801     command->authSize += TPMA_SESSION_Marshal(&s_attributes[i],
1802                                             &command->responseBuffer,
1803                                             NULL);
1804     command->authSize += TPM2B_DIGEST_Marshal(&responseAuth,
1805                                             &command->responseBuffer,
1806                                             NULL);
1807     if(!IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
1808         SessionFlush(s_sessionHandles[i]);
1809 }
1810 }
1811 return;
1812 }

```

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().

```

1813 void
1814 SessionRemoveAssociationToHandle(
1815     TPM_HANDLE    handle
1816 )
1817 {
1818     UINT32        i;
1819     //
1820     for(i = 0; i < MAX_SESSION_NUM; i++)
1821     {
1822         if(s_associatedHandles[i] == handle)
1823         {
1824             s_associatedHandles[i] = TPM_RH_NULL;
1825         }
1826     }
1827 }

```

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 Returns	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(sizeof(contextBlob->sequence) <= sizeof(sequence2B.t.buffer));
26     MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence,
27               sizeof(contextBlob->sequence));
28
29     // Get handle value in 2B format
30     handle2B.t.size = sizeof(contextBlob->savedHandle);
31     cAssert(sizeof(contextBlob->savedHandle) <= sizeof(handle2B.t.buffer));
32     MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle,
33               sizeof(contextBlob->savedHandle));
34
35     // Get the symmetric encryption key size
36     symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
37     symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;
38     // Get the size of the IV for the algorithm
39     iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);
40
41     // KDFa to generate symmetric key and IV value
42     CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, CONTEXT_KEY, &sequence2B.b,
43               &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL,
44               FALSE);
45
46     // Copy part of the returned value as the key
47     pAssert(symKey->t.size <= sizeof(symKey->t.buffer));
48     MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size);
49 }
```



```

50     // Copy the rest as the IV
51     pAssert(iv->t.size <= sizeof(iv->t.buffer));
52     MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size);
53
54     return;
55 }

```

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.

```

56 void
57 ComputeContextIntegrity(
58     TPMS_CONTEXT *contextBlob,    // IN: context blob
59     TPM2B_DIGEST *integrity       // OUT: integrity
60 )
61 {
62     HMAC_STATE      hmacState;
63     TPM2B_PROOF     *proof;
64     UINT16          integritySize;
65
66     // Get proof value
67     proof = HierarchyGetProof(contextBlob->hierarchy);
68
69     // Start HMAC
70     integrity->t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
71                                         &proof->b);
72
73     // Compute integrity size at the beginning of context blob
74     integritySize = sizeof(integrity->t.size) + integrity->t.size;
75
76     // Adding total reset counter so that the context cannot be
77     // used after a TPM Reset
78     CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
79                         gp.totalResetCount);
80
81     // If this is a ST_CLEAR object, add the clear count
82     // so that this context cannot be loaded after a TPM Restart
83     if(contextBlob->savedHandle == 0x80000002)
84         CryptDigestUpdateInt(&hmacState.hashState, sizeof(gr.clearCount),
85                             gr.clearCount);
86
87     // Adding sequence number to the HMAC to make sure that it doesn't
88     // get changed
89     CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->sequence),
90                         contextBlob->sequence);
91
92     // Protect the handle
93     CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->savedHandle),
94                         contextBlob->savedHandle);
95
96     // Adding sensitive contextData, skip the leading integrity area
97     CryptDigestUpdate(&hmacState.hashState,
98                     contextBlob->contextBlob.t.size - integritySize,
99                     contextBlob->contextBlob.t.buffer + integritySize);
100
101     // Complete HMAC
102     CryptHmacEnd2B(&hmacState, &integrity->b);
103
104     return;
105 }

```

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.

```

106 void
107 SequenceDataExport(
108     HASH_OBJECT      *object,           // IN: an internal hash object
109     HASH_OBJECT_BUFFER *exportObject    // OUT: a sequence context in a buffer
110 )
111 {
112     // If the hash object is not an event, then only one hash context is needed
113     int count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
114
115     for(count--; count >= 0; count--)
116     {
117         HASH_STATE      *hash = &object->state.hashState[count];
118         size_t          offset = (BYTE *)hash - (BYTE *)object;
119         BYTE            *exportHash = &((BYTE *)exportObject)[offset];
120
121         CryptHashExportState(hash, (EXPORT_HASH_STATE *)exportHash);
122     }
123 }

```

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.

```

124 void
125 SequenceDataImport(
126     HASH_OBJECT      *object,           // IN/OUT: an internal hash object
127     HASH_OBJECT_BUFFER *exportObject    // IN/OUT: a sequence context in a buffer
128 )
129 {
130     // If the hash object is not an event, then only one hash context is needed
131     int count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
132
133     for(count--; count >= 0; count--)
134     {
135         HASH_STATE      *hash = &object->state.hashState[count];
136         size_t          offset = (BYTE *)hash - (BYTE *)object;
137         BYTE            *importHash = &((BYTE *)exportObject)[offset];
138
139         CryptHashImportState(hash, (EXPORT_HASH_STATE *)importHash);
140     }
141 }

```

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
88         // Start second hash computation
89         CryptHashStart(&hashState, session->authHashAlg);
90
91         // add policyDigest
92         CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);
93
94         // add policyRef
95         CryptDigestUpdate2B(&hashState, &ref->b);
96
97         // Complete second digest
98         CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
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
110     // update the timeout if it is specified
111     if(policyTimeout != 0)
112     {
113         // If the timeout has not been set, then set it to the new value
114         // than the current timeout then set it to the new value
115         if(session->timeout == 0 || session->timeout > policyTimeout)
116             session->timeout = policyTimeout;
117     }
118     return;
119 }

```

7.4.2.3 ComputeAuthTimeout()

This function is used to determine what the authorization timeout value for the session should be.

```

120 UINT64
121 ComputeAuthTimeout(
122     SESSION      *session,           // IN: the session containing the time
123                                     // values
124     INT32         expiration,        // IN: either the number of seconds from
125                                     // the start of the session or the
126                                     // time in g_timer;
127     TPM2B_NONCE  *nonce             // IN: indicator of the time base
128 )
129 {
130     UINT64        policyTime;
131     // If no expiration, policy time is 0
132     if(expiration == 0)
133         policyTime = 0;
134     else
135     {
136         if(expiration < 0)
137             expiration = -expiration;
138         if(nonce->t.size == 0)
139             // The input time is absolute Time (not Clock), but it is expressed
140             // in seconds. To make sure that we don't time out too early, take the
141             // current value of milliseconds in g_time and add that to the input
142             // seconds value.
143             policyTime = (((UINT64)expiration) * 1000) + g_time % 1000;
144         else
145             // The policy timeout is the absolute value of the expiration in seconds
146             // added to the start time of the policy.
147             policyTime = session->startTime + (((UINT64)expiration) * 1000);
148     }
149     return policyTime;
150 }
151

```

7.4.2.4 PolicyDigestClear()

Function to reset the *policyDigest* of a session

```

152 void
153 PolicyDigestClear(
154     SESSION      *session
155 )
156 {
157     session->u2.policyDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
158     MemorySet(session->u2.policyDigest.t.buffer, 0,
159               session->u2.policyDigest.t.size);
160 }

```

```

161  BOOL
162  PolicySptCheckCondition(
163      TPM_EO      operation,
164      BYTE        *opA,
165      BYTE        *opB,
166      UINT16      size
167  )
168  {
169      // Arithmetic Comparison
170      switch(operation)
171      {
172          case TPM_EO_EQ:
173              // compare A = B
174              return (UnsignedCompareB(size, opA, size, opB) == 0);
175              break;
176          case TPM_EO_NEQ:
177              // compare A != B
178              return (UnsignedCompareB(size, opA, size, opB) != 0);
179              break;
180          case TPM_EO_SIGNED_GT:
181              // compare A > B signed
182              return (SignedCompareB(size, opA, size, opB) > 0);
183              break;
184          case TPM_EO_UNSIGNED_GT:
185              // compare A > B unsigned
186              return (UnsignedCompareB(size, opA, size, opB) > 0);
187              break;
188          case TPM_EO_SIGNED_LT:
189              // compare A < B signed
190              return (SignedCompareB(size, opA, size, opB) < 0);
191              break;
192          case TPM_EO_UNSIGNED_LT:
193              // compare A < B unsigned
194              return (UnsignedCompareB(size, opA, size, opB) < 0);
195              break;
196          case TPM_EO_SIGNED_GE:
197              // compare A >= B signed
198              return (SignedCompareB(size, opA, size, opB) >= 0);
199              break;
200          case TPM_EO_UNSIGNED_GE:
201              // compare A >= B unsigned
202              return (UnsignedCompareB(size, opA, size, opB) >= 0);
203              break;
204          case TPM_EO_SIGNED_LE:
205              // compare A <= B signed
206              return (SignedCompareB(size, opA, size, opB) <= 0);
207              break;
208          case TPM_EO_UNSIGNED_LE:
209              // compare A <= B unsigned
210              return (UnsignedCompareB(size, opA, size, opB) <= 0);
211              break;
212          case TPM_EO_BITSET:
213              // All bits SET in B are SET in A. ((A&B)=B)
214              {
215                  UINT32 i;
216                  for(i = 0; i < size; i++)
217                      if((opA[i] & opB[i]) != opB[i])
218                          return FALSE;
219              }
220              break;
221          case TPM_EO_BITCLEAR:
222              // All bits SET in B are CLEAR in A. ((A&B)=0)
223              {
224                  UINT32 i;
225                  for(i = 0; i < size; i++)
226                      if((opA[i] & opB[i]) != 0)

```

```
227         return FALSE;
228     }
229     break;
230     default:
231         FAIL(FATAL_ERROR_INTERNAL);
232         break;
233 }
234 return TRUE;
235 }
```

DRAFT

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 Returns	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 Returns	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
24     // The IV size is a UINT16 size field plus the block size of the symmetric
25     // algorithm
26     return sizeof(UINT16) + CryptGetSymmetricBlockSize(symAlg, keyBits);
27 }
```

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.

```
28 static void
29 ComputeProtectionKeyParms(
30     OBJECT          *protector,          // IN: the protector object
31     TPM_ALG_ID      hashAlg,            // IN: hash algorithm for KDFa
32     TPM2B           *name,              // IN: name of the object
33     TPM2B           *seedIn,            // IN: optional seed for duplication blob.
34                                         // For non duplication blob, this
35                                         // parameter should be NULL
36     TPM_ALG_ID      *symAlg,            // OUT: the symmetric algorithm
37     UINT16           *keyBits,          // OUT: the symmetric key size in bits
38     TPM2B_SYM_KEY   *symKey            // OUT: the symmetric key
39 )
```

```

40 {
41     const TPM2B          *seed = seedIn;
42
43     // Determine the algorithms for the KDF and the encryption/decryption
44     // For TPM_RH_NULL, using context settings
45     if(protector == NULL)
46     {
47         // Use the context encryption algorithm and key size
48         *symAlg = CONTEXT_ENCRYPT_ALG;
49         symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
50         *keyBits = CONTEXT_ENCRYPT_KEY_BITS;
51     }
52     else
53     {
54         TPMT_SYM_DEF_OBJECT *symDef;
55         symDef = &protector->publicArea.parameters.asymDetail.symmetric;
56         *symAlg = symDef->algorithm;
57         *keyBits = symDef->keyBits.sym;
58         symKey->t.size = (*keyBits + 7) / 8;
59     }
60     // Get seed for KDF
61     if(seed == NULL)
62         seed = GetSeedForKDF(protector);
63     // KDFa to generate symmetric key and IV value
64     CryptKDFa(hashAlg, seed, STORAGE_KEY, name, NULL,
65             symKey->t.size * 8, symKey->t.buffer, NULL, FALSE);
66     return;
67 }

```

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.

```

68 static void
69 ComputeOuterIntegrity(
70     TPM2B          *name,           // IN: the name of the object
71     OBJECT         *protector,     // IN: the object that
72                                     // provides protection. For an object,
73                                     // it is a parent. For a credential, it
74                                     // is the encrypt object. For
75                                     // a Temporary Object, it is NULL
76     TPMI_ALG_HASH  hashAlg,       // IN: algorithm to use for integrity
77     TPM2B          *seedIn,       // IN: an external seed may be provided for
78                                     // duplication blob. For non duplication
79                                     // blob, this parameter should be NULL
80     UINT32         sensitiveSize, // IN: size of the marshaled sensitive data
81     BYTE           *sensitiveData, // IN: sensitive area
82     TPM2B_DIGEST   *integrity,    // OUT: integrity
83 )
84 {
85     HMAC_STATE      hmacState;
86     TPM2B_DIGEST    hmacKey;
87     const TPM2B     *seed = seedIn;
88     //
89     // Get seed for KDF
90     if(seed == NULL)
91         seed = GetSeedForKDF(protector);
92     // Determine the HMAC key bits
93     hmacKey.t.size = CryptHashGetDigestSize(hashAlg);
94
95     // KDFa to generate HMAC key

```

```

96     CryptKDFa(hashAlg, seed, INTEGRITY_KEY, NULL, NULL,
97               hmacKey.t.size * 8, hmacKey.t.buffer, NULL, FALSE);
98     // Start HMAC and get the size of the digest which will become the integrity
99     integrity->t.size = CryptHmacStart2B(&hmacState, hashAlg, &hmacKey.b);
100
101     // Adding the marshaled sensitive area to the integrity value
102     CryptDigestUpdate(&hmacState.hashState, sensitiveSize, sensitiveData);
103
104     // Adding name
105     CryptDigestUpdate2B(&hmacState.hashState, name);
106
107     // Compute HMAC
108     CryptHmacEnd2B(&hmacState, &integrity->b);
109
110     return;
111 }

```

7.6.2.4 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap

```

112 static void
113 ComputeInnerIntegrity(
114     TPM_ALG_ID    hashAlg,        // IN: hash algorithm for inner wrap
115     TPM2B         *name,          // IN: the name of the object
116     UINT16        dataSize,       // IN: the size of sensitive data
117     BYTE          *sensitiveData, // IN: sensitive data
118     TPM2B_DIGEST  *integrity      // OUT: inner integrity
119 )
120 {
121     HASH_STATE     hashState;
122     //
123     // Start hash and get the size of the digest which will become the integrity
124     integrity->t.size = CryptHashStart(&hashState, hashAlg);
125
126     // Adding the marshaled sensitive area to the integrity value
127     CryptDigestUpdate(&hashState, dataSize, sensitiveData);
128
129     // Adding name
130     CryptDigestUpdate2B(&hashState, name);
131
132     // Compute hash
133     CryptHashEnd2B(&hashState, &integrity->b);
134
135     return;
136 }

```

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 integrity at the beginning of the inner buffer. It returns the total size of buffer with the inner wrap.

```

137 static UINT16
138 ProduceInnerIntegrity(
139     TPM2B         *name,          // IN: the name of the object
140     TPM_ALG_ID    hashAlg,       // IN: hash algorithm for inner wrap
141     UINT16        dataSize,       // IN: the size of sensitive data, excluding the
142                                     // leading integrity buffer size
143     BYTE          *innerBuffer    // IN/OUT: inner buffer with sensitive data in
144                                     // it. At input, the leading bytes of this

```

```

145                                     //    buffer is reserved for integrity
146     )
147 {
148     BYTE          *sensitiveData; // pointer to the sensitive data
149     TPM2B_DIGEST  integrity;
150     UINT16        integritySize;
151     BYTE          *buffer;        // Auxiliary buffer pointer
152 //
153 // sensitiveData points to the beginning of sensitive data in innerBuffer
154 integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
155 sensitiveData = innerBuffer + integritySize;
156
157 ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);
158
159 // Add integrity at the beginning of inner buffer
160 buffer = innerBuffer;
161 TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
162
163 return dataSize + integritySize;
164 }

```

7.6.2.6 CheckInnerIntegrity()

This function check integrity of inner blob

Error Returns	Meaning
TPM_RC_INTEGRITY	if the outer blob integrity is bad
unmarshal errors	unmarshal errors while unmarshaling integrity

```

165 static TPM_RC
166 CheckInnerIntegrity(
167     TPM2B          *name,           // IN: the name of the object
168     TPM_ALG_ID     hashAlg,        // IN: hash algorithm for inner wrap
169     UINT16         dataSize,       // IN: the size of sensitive data, including the
170                                     // leading integrity buffer size
171     BYTE          *innerBuffer     // IN/OUT: inner buffer with sensitive data in
172                                     // it
173 )
174 {
175     TPM_RC         result;
176     TPM2B_DIGEST   integrity;
177     TPM2B_DIGEST   integrityToCompare;
178     BYTE          *buffer;          // Auxiliary buffer pointer
179     INT32          size;
180 //
181 // Unmarshal integrity
182 buffer = innerBuffer;
183 size = (INT32)dataSize;
184 result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
185 if(result == TPM_RC_SUCCESS)
186 {
187     // Compute integrity to compare
188     ComputeInnerIntegrity(hashAlg, name, (UINT16)size, buffer,
189                           &integrityToCompare);
190     // Compare outer blob integrity
191     if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
192         result = TPM_RC_INTEGRITY;
193 }
194 return result;
195 }

```


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.

```

196  BOOL
197  AdjustAuthSize(
198      TPM2B_AUTH      *auth,           // IN/OUT: value to adjust
199      TPMI_ALG_HASH   nameAlg         // IN:
200  )
201  {
202      UINT16           digestSize;
203      //
204      // If there is no nameAlg, then this is a LoadExternal and the authVale can
205      // be any size up to the maximum allowed by the
206      digestSize = (nameAlg == TPM_ALG_NULL) ? sizeof(TPMU_HA)
207          : CryptHashGetDigestSize(nameAlg);
208      if(digestSize < MemoryRemoveTrailingZeros(auth))
209          return FALSE;
210      else if(digestSize > auth->t.size)
211          MemoryPad2B(&auth->b, digestSize);
212      auth->t.size = digestSize;
213
214      return TRUE;
215  }

```

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

```

216  BOOL
217  ObjectIsParent(
218      OBJECT      *parentObject      // IN: parent handle
219  )
220  {
221      return parentObject->attributes.isParent;
222  }

```

7.6.3.3 CreateChecks()

Attribute checks that are unique to creation.

Error Returns	Meaning
TPM_RC_ATTRIBUTES	<i>sensitiveDataOrigin</i> is not consistent with the object type
other	returns from PublicAttributesValidation()

```

223  TPM_RC
224  CreateChecks(

```

```

225     OBJECT                *parentObject,
226     TPMT_PUBLIC           *publicArea,
227     UINT16                sensitiveDataSize
228 )
229 {
230     TPMA_OBJECT            attributes = publicArea->objectAttributes;
231     TPM_RC                 result = TPM_RC_SUCCESS;
232 //
233 // If the caller indicates that they have provided the data, then make sure that
234 // they have provided some data.
235 if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
236     && (sensitiveDataSize == 0))
237     return TPM_RCS_ATTRIBUTES;
238 // For an ordinary object, data can only be provided when sensitiveDataOrigin
239 // is CLEAR
240 if((parentObject != NULL)
241     && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
242     && (sensitiveDataSize != 0))
243     return TPM_RCS_ATTRIBUTES;
244 switch(publicArea->type)
245 {
246     case ALG_KEYEDHASH_VALUE:
247         // if this is a data object (sign == decrypt == CLEAR) then the
248         // TPM cannot be the data source.
249         if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
250             && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
251             && IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
252             result = TPM_RC_ATTRIBUTES;
253         // comment out the next line in order to prevent a fixedTPM derivation
254         // parent
255         break;
256 //
257 case ALG_SYMCIPHER_VALUE:
258     // A restricted key symmetric key (SYMCIPHER and KEYEDHASH)
259     // must have sensitiveDataOrigin SET unless it has fixedParent and
260     // fixedTPM CLEAR.
261     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
262         if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
263             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
264                 || IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
265                 result = TPM_RCS_ATTRIBUTES;
266         break;
267     default: // Asymmetric keys cannot have the sensitive portion provided
268         if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, sensitiveDataOrigin))
269             result = TPM_RCS_ATTRIBUTES;
270         break;
271 }
272 if(TPM_RC_SUCCESS == result)
273 {
274     result = PublicAttributesValidation(parentObject, publicArea);
275 }
276 return result;
277 }

```

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 Returns	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_RCS_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 ALG_NULL

```

277 TPM_RC
278 SchemeChecks(
279     OBJECT          *parentObject, // IN: parent (null if primary seed)
280     TPMT_PUBLIC     *publicArea    // IN: public area of the object
281 )
282 {
283     TPMT_SYM_DEF_OBJECT *symAlgs = NULL;
284     TPM_ALG_ID          scheme = TPM_ALG_NULL;
285     TPMA_OBJECT          attributes = publicArea->objectAttributes;
286     TPMU_PUBLIC_PARMS    *parms = &publicArea->parameters;
287     //
288     switch(publicArea->type)
289     {
290     case ALG_SYMCIPHER_VALUE:
291         symAlgs = &parms->symDetail.sym;
292         // If this is a decrypt key, then only the block cipher modes (not
293         // SMAC) are valid. TPM_ALG_NULL is OK too. If this is a 'sign' key,
294         // then any mode that got through the unmarshaling is OK.
295         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
296             && !CryptSymModeIsValid(symAlgs->mode.sym, TRUE))
297             return TPM_RCS_SCHEME;
298         break;
299     case ALG_KEYEDHASH_VALUE:
300         scheme = parms->keyedHashDetail.scheme.scheme;
301         // if both sign and decrypt
302         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
303             == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
304         {
305             // if both sign and decrypt are set or clear, then need
306             // ALG_NULL as scheme
307             if(scheme != TPM_ALG_NULL)
308                 return TPM_RCS_SCHEME;
309         }
310         else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
311             && scheme != TPM_ALG_HMAC)
312             return TPM_RCS_SCHEME;
313         else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
314         {
315             if(scheme != TPM_ALG_XOR)
316                 return TPM_RCS_SCHEME;
317             // If this is a derivation parent, then the KDF needs to be
318             // SP800-108 for this implementation. This is the only derivation
319             // supported by this implementation. Other implementations could
320             // support additional schemes. There is no default.
321             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
322             {
323                 if(parms->keyedHashDetail.scheme.details.xor.kdf
324                     != TPM_ALG_KDF1_SP800_108)
325                     return TPM_RCS_SCHEME;
326                 // Must select a digest.

```

```

327         if(CryptHashGetDigestSize(
328             parms->keyedHashDetail.scheme.details.xor.hashAlg) == 0)
329             return TPM_RCS_HASH;
330     }
331 }
332 break;
333 default: // handling for asymmetric
334     scheme = parms->asymDetail.scheme.scheme;
335     symAlgs = &parms->asymDetail.symmetric;
336     // if the key is both sign and decrypt, then the scheme must be
337     // ALG_NULL because there is no way to specify both a sign and a
338     // decrypt scheme in the key.
339     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
340         == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
341     {
342         // scheme must be TPM_ALG_NULL
343         if(scheme != TPM_ALG_NULL)
344             return TPM_RCS_SCHEME;
345     }
346     else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
347     {
348         // If this is a signing key, see if it has a signing scheme
349         if(CryptIsAsymSignScheme(publicArea->type, scheme))
350         {
351             // if proper signing scheme then it needs a proper hash
352             if(parms->asymDetail.scheme.details.anySig.hashAlg
353                 == TPM_ALG_NULL)
354                 return TPM_RCS_SCHEME;
355         }
356         else
357         {
358             // signing key that does not have a proper signing scheme.
359             // This is OK if the key is not restricted and its scheme
360             // is TPM_ALG_NULL
361             if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
362                 || scheme != TPM_ALG_NULL)
363                 return TPM_RCS_SCHEME;
364         }
365     }
366     else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
367     {
368         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
369         {
370             // for a restricted decryption key (a parent), scheme
371             // is required to be TPM_ALG_NULL
372             if(scheme != TPM_ALG_NULL)
373                 return TPM_RCS_SCHEME;
374         }
375         else
376         {
377             // For an unrestricted decryption key, the scheme has to
378             // be a valid scheme or TPM_ALG_NULL
379             if(scheme != TPM_ALG_NULL &&
380                 !CryptIsAsymDecryptScheme(publicArea->type, scheme))
381                 return TPM_RCS_SCHEME;
382         }
383     }
384     if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
385         || !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
386     {
387         // For an asymmetric key that is not a parent, the symmetric
388         // algorithms must be TPM_ALG_NULL
389         if(symAlgs->algorithm != TPM_ALG_NULL)
390             return TPM_RCS_SYMMETRIC;
391     }
392     // Special checks for an ECC key

```

```

393  #if      ALG_ECC
394      if(publicArea->type == TPM_ALG_ECC)
395      {
396          TPM_ECC_CURVE      curveID;
397          const TPMT_ECC_SCHEME *curveScheme;
398
399          curveID = publicArea->parameters.eccDetail.curveID;
400          curveScheme = CryptGetCurveSignScheme(curveID);
401          // The curveID must be valid or the unmarshaling is busted.
402          pAssert(curveScheme != NULL);
403
404          // If the curveID requires a specific scheme, then the key must
405          // select the same scheme
406          if(curveScheme->scheme != TPM_ALG_NULL)
407          {
408              TPMS_ECC_PARMS *ecc = &publicArea->parameters.eccDetail;
409              if(scheme != curveScheme->scheme)
410                  return TPM_RCS_SCHEME;
411              // The scheme can allow any hash, or not...
412              if(curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
413                 && (ecc->scheme.details.anySig.hashAlg
414                    != curveScheme->details.anySig.hashAlg))
415                  return TPM_RCS_SCHEME;
416          }
417          // For now, the KDF must be TPM_ALG_NULL
418          if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
419              return TPM_RCS_KDF;
420      }
421  #endif
422      break;
423  }
424  // If this is a restricted decryption key with symmetric algorithms, then it
425  // is an ordinary parent (not a derivation parent). It needs to specific
426  // symmetric algorithms other than TPM_ALG_NULL
427  if(symAlgs != NULL
428     && IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
429     && IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
430  {
431      if(symAlgs->algorithm == TPM_ALG_NULL)
432          return TPM_RCS_SYMMETRIC;
433  #if 0      //??
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 Returns	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()

```

454 TPM_RC
455 PublicAttributesValidation(
456     OBJECT      *parentObject, // IN: input parent object
457     TPMT_PUBLIC *publicArea    // IN: public area of the object
458 )
459 {
460     TPMA_OBJECT attributes = publicArea->objectAttributes;
461     TPMA_OBJECT parentAttributes = {0};
462     //
463     if(parentObject != NULL)
464         parentAttributes = parentObject->publicArea.objectAttributes;
465     if(publicArea->nameAlg == TPM_ALG_NULL)
466         return TPM_RCS_HASH;
467     // If there is an authPolicy, it needs to be the size of the digest produced
468     // by the nameAlg of the object
469     if((publicArea->authPolicy.t.size != 0
470         && (publicArea->authPolicy.t.size
471             != CryptHashGetDigestSize(publicArea->nameAlg))))
472         return TPM_RCS_SIZE;
473     // If the parent is fixedTPM (including a Primary Object) the object must have
474     // the same value for fixedTPM and fixedParent
475     if(parentObject == NULL
476        || IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
477     {
478         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
479            != IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
480             return TPM_RCS_ATTRIBUTES;
481     }
482     else
483     {
484         // The parent is not fixedTPM so the object can't be fixedTPM
485         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
486             return TPM_RCS_ATTRIBUTES;
487     }
488     // See if sign and decrypt are the same
489     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
490        == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
491     {
492         // a restricted key cannot have both SET or both CLEAR
493         if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
494             return TPM_RC_ATTRIBUTES;
495         // only a data object may have both sign and decrypt CLEAR
496         // BTW, since we know that decrypt==sign, no need to check both

```



```

497     if(publicArea->type != TPM_ALG_KEYEDHASH
498         && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
499         return TPM_RC_ATTRIBUTES;
500 }
501 // If the object can't be duplicated (directly or indirectly) then there
502 // is no justification for having encryptedDuplication SET
503 if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
504     && IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
505     return TPM_RCS_ATTRIBUTES;
506 // If a parent object has fixedTPM CLEAR, the child must have the
507 // same encryptedDuplication value as its parent.
508 // Primary objects are considered to have a fixedTPM parent (the seeds).
509 if(parentObject != NULL
510     && !IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
511 {
512     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication)
513         != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, encryptedDuplication))
514         return TPM_RCS_ATTRIBUTES;
515 }
516 // Special checks for derived objects
517 if((parentObject != NULL) && (parentObject->attributes.derivation == SET))
518 {
519     // A derived object has the same settings for fixedTPM as its parent
520     if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
521         != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
522         return TPM_RCS_ATTRIBUTES;
523     // A derived object is required to be fixedParent
524     if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent))
525         return TPM_RCS_ATTRIBUTES;
526 }
527 return SchemeChecks(parentObject, publicArea);
528 }

```

7.6.3.6 FillInCreationData()

Fill in creation data for an object.

```

529 void
530 FillInCreationData(
531     TPMI_DH_OBJECT      parentHandle, // IN: handle of parent
532     TPMI_ALG_HASH        nameHashAlg, // IN: name hash algorithm
533     TPMI_PCR_SELECTION   *creationPCR, // IN: PCR selection
534     TPM2B_DATA            *outsideData, // IN: outside data
535     TPM2B_CREATION_DATA   *outCreation, // OUT: creation data for output
536     TPM2B_DIGEST          *creationDigest // OUT: creation digest
537 )
538 {
539     BYTE      creationBuffer[sizeof(TPMS_CREATION_DATA)];
540     BYTE      *buffer;
541     HASH_STATE hashState;
542 //
543 // Fill in TPMS_CREATION_DATA in outCreation
544
545 // Compute PCR digest
546 PCRComputeCurrentDigest(nameHashAlg, creationPCR,
547     &outCreation->creationData.pcrDigest);
548
549 // Put back PCR selection list
550 outCreation->creationData.pcrSelect = *creationPCR;
551
552 // Get locality
553 outCreation->creationData.locality
554     = LocalityGetAttributes(_plat__LocalityGet());
555 outCreation->creationData.parentNameAlg = TPM_ALG_NULL;

```



```

556
557 // If the parent is either a primary seed or TPM_ALG_NULL, then the Name
558 // and QN of the parent are the parent's handle.
559 if(HandleGetType(parentHandle) == TPM_HT_PERMANENT)
560 {
561     buffer = &outCreation->creationData.parentName.t.name[0];
562     outCreation->creationData.parentName.t.size =
563         TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
564     // For a primary or temporary object, the parent name (a handle) and the
565     // parent's QN are the same
566     outCreation->creationData.parentQualifiedName
567         = outCreation->creationData.parentName;
568 }
569 else // Regular object
570 {
571     OBJECT *parentObject = HandleToObject(parentHandle);
572 //
573 // Set name algorithm
574 outCreation->creationData.parentNameAlg = parentObject->publicArea.nameAlg;
575
576 // Copy parent name
577 outCreation->creationData.parentName = parentObject->name;
578
579 // Copy parent qualified name
580 outCreation->creationData.parentQualifiedName = parentObject->qualifiedName;
581 }
582 // Copy outside information
583 outCreation->creationData.outsideInfo = *outsideData;
584
585 // Marshal creation data to canonical form
586 buffer = creationBuffer;
587 outCreation->size = TPMS_CREATION_DATA_Marshal(&outCreation->creationData,
588                                             &buffer, NULL);
589 // Compute hash for creation field in public template
590 creationDigest->t.size = CryptHashStart(&hashState, nameHashAlg);
591 CryptDigestUpdate(&hashState, outCreation->size, creationBuffer);
592 CryptHashEnd2B(&hashState, &creationDigest->b);
593
594 return;
595 }

```

7.6.3.7 GetSeedForKDF()

Get a seed for KDF. The KDF for encryption and HMAC key use the same seed.

```

596 const TPM2B *
597 GetSeedForKDF(
598     OBJECT *protector // IN: the protector handle
599 )
600 {
601     // Get seed for encryption key. Use input seed if provided.
602     // Otherwise, using protector object's seedValue. TPM_RH_NULL is the only
603     // exception that we may not have a loaded object as protector. In such a
604     // case, use nullProof as seed.
605     if(protector == NULL)
606         return &gr.nullProof.b;
607     else
608         return &protector->sensitive.seedValue.b;
609 }

```

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:

- a) Add IV before sensitive area if required
- b) encrypt sensitive data, if iv is required, encrypt by iv. otherwise, encrypted by a NULL iv
- c) add HMAC integrity at the beginning of the buffer It returns the total size of blob with outer wrap

```

610  UINT16
611  ProduceOuterWrap(
612      OBJECT      *protector,      // IN: The handle of the object that provides
613                                   // protection. For object, it is parent
614                                   // handle. For credential, it is the handle
615                                   // of encrypt object.
616      TPM2B       *name,           // IN: the name of the object
617      TPM_ALG_ID   hashAlg,        // IN: hash algorithm for outer wrap
618      TPM2B       *seed,           // IN: an external seed may be provided for
619                                   // duplication blob. For non duplication
620                                   // blob, this parameter should be NULL
621      BOOL         useIV,          // IN: indicate if an IV is used
622      UINT16       dataSize,        // IN: the size of sensitive data, excluding the
623                                   // leading integrity buffer size or the
624                                   // optional iv size
625      BYTE         *outerBuffer     // IN/OUT: outer buffer with sensitive data in
626                                   // it
627  )
628  {
629      TPM_ALG_ID   symAlg;
630      UINT16       keyBits;
631      TPM2B_SYM_KEY symKey;
632      TPM2B_IV     ivRNG;          // IV from RNG
633      TPM2B_IV     *iv = NULL;
634      UINT16       ivSize = 0;     // size of iv area, including the size field
635      BYTE         *sensitiveData; // pointer to the sensitive data
636      TPM2B_DIGEST integrity;
637      UINT16       integritySize;
638      BYTE         *buffer;        // Auxiliary buffer pointer
639  //
640  // Compute the beginning of sensitive data. The outer integrity should
641  // always exist if this function is called to make an outer wrap
642  integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
643  sensitiveData = outerBuffer + integritySize;
644
645  // If iv is used, adjust the pointer of sensitive data and add iv before it
646  if(useIV)
647  {
648      ivSize = GetIV2BSize(protector);
649
650      // Generate IV from RNG. The iv data size should be the total IV area
651      // size minus the size of size field
652      ivRNG.t.size = ivSize - sizeof(UINT16);
653      CryptRandomGenerate(ivRNG.t.size, ivRNG.t.buffer);
654
655      // Marshal IV to buffer
656      buffer = sensitiveData;
657      TPM2B_IV_Marshal(&ivRNG, &buffer, NULL);
658
659      // adjust sensitive data starting after IV area
660      sensitiveData += ivSize;
661  }

```

```

662         // Use iv for encryption
663         iv = &ivRNG;
664     }
665     // Compute symmetric key parameters for outer buffer encryption
666     ComputeProtectionKeyParms(protector, hashAlg, name, seed,
667                               &symAlg, &keyBits, &symKey);
668     // Encrypt inner buffer in place
669     CryptSymmetricEncrypt(sensitiveData, symAlg, keyBits,
670                           symKey.t.buffer, iv, TPM_ALG_CFB, dataSize,
671                           sensitiveData);
672     // Compute outer integrity. Integrity computation includes the optional IV
673     // area
674     ComputeOuterIntegrity(name, protector, hashAlg, seed, dataSize + ivSize,
675                           outerBuffer + integritySize, &integrity);
676     // Add integrity at the beginning of outer buffer
677     buffer = outerBuffer;
678     TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
679
680     // return the total size in outer wrap
681     return dataSize + integritySize + ivSize;
682 }

```

7.6.3.9 UnwrapOuter()

This function remove the outer wrap of a blob containing sensitive data This function performs:

- check integrity of outer blob
- decrypt outer blob

Error Returns	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

```

683 TPM_RC
684 UnwrapOuter(
685     OBJECT          *protector,    // IN: The object that provides
686                                     // protection. For object, it is parent
687                                     // handle. For credential, it is the
688                                     // encrypt object.
689     TPM2B           *name,         // IN: the name of the object
690     TPM_ALG_ID      hashAlg,       // IN: hash algorithm for outer wrap
691     TPM2B           *seed,         // IN: an external seed may be provided for
692                                     // duplication blob. For non duplication
693                                     // blob, this parameter should be NULL.
694     BOOL            useIV,         // IN: indicates if an IV is used
695     UINT16          dataSize,      // IN: size of sensitive data in outerBuffer,
696                                     // including the leading integrity buffer
697                                     // size, and an optional iv area
698     BYTE            *outerBuffer   // IN/OUT: sensitive data
699 )
700 {
701     TPM_RC          result;
702     TPM_ALG_ID      symAlg = TPM_ALG_NULL;
703     TPM2B_SYM_KEY   symKey;
704     UINT16          keyBits = 0;
705     TPM2B_IV        ivIn;         // input IV retrieved from input buffer
706     TPM2B_IV        *iv = NULL;
707     BYTE            *sensitiveData; // pointer to the sensitive data

```

```

708     TPM2B_DIGEST    integrityToCompare;
709     TPM2B_DIGEST    integrity;
710     INT32            size;
711 //
712 // Unmarshal integrity
713 sensitiveData = outerBuffer;
714 size = (INT32) dataSize;
715 result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
716 if(result == TPM_RC_SUCCESS)
717 {
718     // Compute integrity to compare
719     ComputeOuterIntegrity(name, protector, hashAlg, seed,
720                          (UINT16) size, sensitiveData,
721                          &integrityToCompare);
722     // Compare outer blob integrity
723     if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
724         return TPM_RCS_INTEGRITY;
725     // Get the symmetric algorithm parameters used for encryption
726     ComputeProtectionKeyParms(protector, hashAlg, name, seed,
727                               &symAlg, &keyBits, &symKey);
728     // Retrieve IV if it is used
729     if(useIV)
730     {
731         result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
732         if(result == TPM_RC_SUCCESS)
733         {
734             // The input iv size for CFB must match the encryption algorithm
735             // block size
736             if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
737                 result = TPM_RC_VALUE;
738             else
739                 iv = &ivIn;
740         }
741     }
742 }
743 // If no errors, decrypt private in place. Since this function uses CFB,
744 // CryptSymmetricDecrypt() will not return any errors. It may fail but it will
745 // not return an error.
746 if(result == TPM_RC_SUCCESS)
747     CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
748                          symKey.t.buffer, iv, TPM_ALG_CFB,
749                          (UINT16) size, sensitiveData);
750 return result;
751 }

```

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*. It will also make sure that the RSA sensitive contains the right number of values. Returns the size of the marshaled area.

```

752 static UINT16
753 MarshalSensitive(
754     OBJECT            *parent,           // IN: the object parent (optional)
755     BYTE              *buffer,           // OUT: receiving buffer
756     TPMT_SENSITIVE    *sensitive,        // IN: the sensitive area to marshal
757     TPMI_ALG_HASH     nameAlg,           // IN:
758 )
759 {
760     BYTE              *sizeField = buffer; // saved so that size can be
761                                           // marshaled after it is known
762     UINT16            retVal;
763 //
764 // Pad the authValue if needed

```

```

765     MemoryPad2B(&sensitive->authValue.b, CryptHashGetDigestSize(nameAlg));
766     buffer += 2;
767
768     // Marshal the structure
769 #if ALG_RSA
770     // If the sensitive size is the special case for a prime in the type
771     if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) > 0)
772     {
773         UINT16          sizeSave = sensitive->sensitive.rsa.t.size;
774         //
775         // Turn off the flag that indicates that the sensitive->sensitive contains
776         // the CRT form of the exponent.
777         sensitive->sensitive.rsa.t.size &= ~(RSA_prime_flag);
778         // If the parent isn't fixedTPM, then truncate the sensitive data to be
779         // the size of the prime. Otherwise, leave it at the current size which
780         // is the full CRT size.
781         if(parent == NULL
782            || !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
783                           TPMA_OBJECT, fixedTPM))
784             sensitive->sensitive.rsa.t.size /= 5;
785         retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
786         // Restore the flag and the size.
787         sensitive->sensitive.rsa.t.size = sizeSave;
788     }
789     else
790 #endif
791     retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
792
793     // Marshal the size
794     retVal = (UINT16)(retVal + UINT16_Marshal(&retVal, &sizeField, NULL));
795
796     return retVal;
797 }

```

7.6.3.11 SensitiveToPrivate()

This function prepare the private blob for off the chip storage The operations in this function:

- marshal TPM2B_SENSITIVE structure into the buffer of TPM2B_PRIVATE
- apply encryption to the sensitive area.
- apply outer integrity computation.

```

798 void
799 SensitiveToPrivate(
800     TPMT_SENSITIVE *sensitive,          // IN: sensitive structure
801     TPM2B_NAME *name,                  // IN: the name of the object
802     OBJECT *parent,                    // IN: The parent object
803     TPM_ALG_ID nameAlg,                // IN: hash algorithm in public area. This
804                                         // parameter is used when parentHandle is
805                                         // NULL, in which case the object is
806                                         // temporary.
807     TPM2B_PRIVATE *outPrivate           // OUT: output private structure
808 )
809 {
810     BYTE *sensitiveData;                // pointer to the sensitive data
811     UINT16 dataSize;                    // data blob size
812     TPMT_ALG_HASH hashAlg;              // hash algorithm for integrity
813     UINT16 integritySize;
814     UINT16 ivSize;
815     //
816     pAssert(name != NULL && name->t.size != 0);
817
818     // Find the hash algorithm for integrity computation

```

```

819     if(parent == NULL)
820     {
821         // For Temporary Object, using self name algorithm
822         hashAlg = nameAlg;
823     }
824     else
825     {
826         // Otherwise, using parent's name algorithm
827         hashAlg = parent->publicArea.nameAlg;
828     }
829     // Starting of sensitive data without wrappers
830     sensitiveData = outPrivate->t.buffer;
831
832     // Compute the integrity size
833     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
834
835     // Reserve space for integrity
836     sensitiveData += integritySize;
837
838     // Get iv size
839     ivSize = GetIV2BSize(parent);
840
841     // Reserve space for iv
842     sensitiveData += ivSize;
843
844     // Marshal the sensitive area including authValue size adjustments.
845     dataSize = MarshalSensitive(parent, sensitiveData, sensitive, nameAlg);
846
847     //Produce outer wrap, including encryption and HMAC
848     outPrivate->t.size = ProduceOuterWrap(parent, &name->b, hashAlg, NULL,
849                                         TRUE, dataSize, outPrivate->t.buffer);
850     return;
851 }

```

7.6.3.12 PrivateToSensitive()

Unwrap a input private area. Check the integrity, decrypt and retrieve data to a sensitive structure. The operations in this function:

- check the integrity HMAC of the input private area
- decrypt the private buffer
- unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

Error Returns	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

```

852 TPM_RC
853 PrivateToSensitive(
854     TPM2B          *inPrivate,      // IN: input private structure
855     TPM2B          *name,          // IN: the name of the object
856     OBJECT         *parent,        // IN: parent object
857     TPM_ALG_ID     nameAlg,        // IN: hash algorithm in public area. It is
858                                     // passed separately because we only pass
859                                     // name, rather than the whole public area
860                                     // of the object. This parameter is used in
861                                     // the following two cases: 1. primary

```



```

862                                     // objects. 2. duplication blob with inner
863                                     // wrap. In other cases, this parameter
864                                     // will be ignored
865     TPMT_SENSITIVE *sensitive        // OUT: sensitive structure
866 )
867 {
868     TPM_RC      result;
869     BYTE        *buffer;
870     INT32       size;
871     BYTE        *sensitiveData; // pointer to the sensitive data
872     UINT16      dataSize;
873     UINT16      dataSizeInput;
874     TPMI_ALG_HASH hashAlg;      // hash algorithm for integrity
875     UINT16      integritySize;
876     UINT16      ivSize;
877 //
878 // Make sure that name is provided
879 pAssert(name != NULL && name->size != 0);
880
881 // Find the hash algorithm for integrity computation
882 // For Temporary Object (parent == NULL) use self name algorithm;
883 // Otherwise, using parent's name algorithm
884 hashAlg = (parent == NULL) ? nameAlg : parent->publicArea.nameAlg;
885
886 // unwrap outer
887 result = UnwrapOuter(parent, name, hashAlg, NULL, TRUE,
888                     inPrivate->size, inPrivate->buffer);
889 if(result != TPM_RC_SUCCESS)
890     return result;
891 // Compute the inner integrity size.
892 integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
893
894 // Get iv size
895 ivSize = GetIV2BSize(parent);
896
897 // The starting of sensitive data and data size without outer wrapper
898 sensitiveData = inPrivate->buffer + integritySize + ivSize;
899 dataSize = inPrivate->size - integritySize - ivSize;
900
901 // Unmarshal input data size
902 buffer = sensitiveData;
903 size = (INT32)dataSize;
904 result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
905 if(result == TPM_RC_SUCCESS)
906 {
907     if((dataSizeInput + sizeof(UINT16)) != dataSize)
908         result = TPM_RC_SENSITIVE;
909     else
910     {
911         // Unmarshal sensitive buffer to sensitive structure
912         result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
913         if(result != TPM_RC_SUCCESS || size != 0)
914         {
915             result = TPM_RC_SENSITIVE;
916         }
917     }
918 }
919 return result;
920 }

```

7.6.3.13 SensitiveToDuplicate()

This function prepare the duplication blob from the sensitive area. The operations in this function:

- a) marshal TPMT_SENSITIVE structure into the buffer of TPM2B_PRIVATE
- b) apply inner wrap to the sensitive area if required
- c) apply outer wrap if required

```

921 void
922 SensitiveToDuplicate(
923     TPMT_SENSITIVE *sensitive,    // IN: sensitive structure
924     TPM2B *name,                 // IN: the name of the object
925     OBJECT *parent,              // IN: The new parent object
926     TPM_ALG_ID nameAlg,          // IN: hash algorithm in public area. It
927                                   // is passed separately because we
928                                   // only pass name, rather than the
929                                   // whole public area of the object.
930     TPM2B *seed,                 // IN: the external seed. If external
931                                   // seed is provided with size of 0,
932                                   // no outer wrap should be applied
933                                   // to duplication blob.
934     TPMT_SYM_DEF_OBJECT *symDef,  // IN: Symmetric key definition. If the
935                                   // symmetric key algorithm is NULL,
936                                   // no inner wrap should be applied.
937     TPM2B_DATA *innerSymKey,      // IN/OUT: a symmetric key may be
938                                   // provided to encrypt the inner
939                                   // wrap of a duplication blob. May
940                                   // be generated here if needed.
941     TPM2B_PRIVATE *outPrivate     // OUT: output private structure
942 )
943 {
944     BYTE *sensitiveData; // pointer to the sensitive data
945     TPMI_ALG_HASH outerHash = TPM_ALG_NULL; // The hash algorithm for outer wrap
946     TPMI_ALG_HASH innerHash = TPM_ALG_NULL; // The hash algorithm for inner wrap
947     UINT16 dataSize;      // data blob size
948     BOOL doInnerWrap = FALSE;
949     BOOL doOuterWrap = FALSE;
950 //
951 // Make sure that name is provided
952 pAssert(name != NULL && name->size != 0);
953
954 // Make sure symDef and innerSymKey are not NULL
955 pAssert(symDef != NULL && innerSymKey != NULL);
956
957 // Starting of sensitive data without wrappers
958 sensitiveData = outPrivate->t.buffer;
959
960 // Find out if inner wrap is required
961 if(symDef->algorithm != TPM_ALG_NULL)
962 {
963     doInnerWrap = TRUE;
964
965     // Use self nameAlg as inner hash algorithm
966     innerHash = nameAlg;
967
968     // Adjust sensitive data pointer
969     sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(innerHash);
970 }
971 // Find out if outer wrap is required
972 if(seed->size != 0)
973 {
974     doOuterWrap = TRUE;
975
976     // Use parent nameAlg as outer hash algorithm
977     outerHash = parent->publicArea.nameAlg;
978
979     // Adjust sensitive data pointer
980     sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);

```

```

981     }
982     // Marshal sensitive area
983     dataSize = MarshalSensitive(NULL, sensitiveData, sensitive, nameAlg);
984
985     // Apply inner wrap for duplication blob. It includes both integrity and
986     // encryption
987     if(doInnerWrap)
988     {
989         BYTE          *innerBuffer = NULL;
990         BOOL          symKeyInput = TRUE;
991         innerBuffer = outPrivate->t.buffer;
992         // Skip outer integrity space
993         if(doOuterWrap)
994             innerBuffer += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
995         dataSize = ProduceInnerIntegrity(name, innerHash, dataSize,
996                                         innerBuffer);
997         // Generate inner encryption key if needed
998         if(innerSymKey->t.size == 0)
999         {
1000             innerSymKey->t.size = (symDef->keyBits.sym + 7) / 8;
1001             CryptRandomGenerate(innerSymKey->t.size, innerSymKey->t.buffer);
1002
1003             // TPM generates symmetric encryption. Set the flag to FALSE
1004             symKeyInput = FALSE;
1005         }
1006         else
1007         {
1008             // assume the input key size should matches the symmetric definition
1009             pAssert(innerSymKey->t.size == (symDef->keyBits.sym + 7) / 8);
1010         }
1011
1012         // Encrypt inner buffer in place
1013         CryptSymmetricEncrypt(innerBuffer, symDef->algorithm,
1014                               symDef->keyBits.sym, innerSymKey->t.buffer, NULL,
1015                               TPM_ALG_CFB, dataSize, innerBuffer);
1016
1017         // If the symmetric encryption key is imported, clear the buffer for
1018         // output
1019         if(symKeyInput)
1020             innerSymKey->t.size = 0;
1021     }
1022     // Apply outer wrap for duplication blob. It includes both integrity and
1023     // encryption
1024     if(doOuterWrap)
1025     {
1026         dataSize = ProduceOuterWrap(parent, name, outerHash, seed, FALSE,
1027                                     dataSize, outPrivate->t.buffer);
1028     }
1029     // Data size for output
1030     outPrivate->t.size = dataSize;
1031
1032     return;
1033 }

```

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:

- a) check the integrity HMAC of the input private area
- b) decrypt the private buffer
- c) unmarshal TPMT_SENSITIVE structure into the buffer of TPMT_SENSITIVE

Error Returns	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

```

1034 TPM_RC
1035 DuplicateToSensitive(
1036     TPM2B          *inPrivate,      // IN: input private structure
1037     TPM2B          *name,          // IN: the name of the object
1038     OBJECT         *parent,        // IN: the parent
1039     TPM_ALG_ID     nameAlg,        // IN: hash algorithm in public area.
1040     TPM2B          *seed,          // IN: an external seed may be provided.
1041                                     // If external seed is provided with
1042                                     // size of 0, no outer wrap is
1043                                     // applied
1044     TPMT_SYM_DEF_OBJECT *symDef,    // IN: Symmetric key definition. If the
1045                                     // symmetric key algorithm is NULL,
1046                                     // no inner wrap is applied
1047     TPM2B          *innerSymKey,    // IN: a symmetric key may be provided
1048                                     // to decrypt the inner wrap of a
1049                                     // duplication blob.
1050     TPMT_SENSITIVE *sensitive      // OUT: sensitive structure
1051 )
1052 {
1053     TPM_RC      result;
1054     BYTE        *buffer;
1055     INT32       size;
1056     BYTE        *sensitiveData; // pointer to the sensitive data
1057     UINT16      dataSize;
1058     UINT16      dataSizeInput;
1059     //
1060     // Make sure that name is provided
1061     pAssert(name != NULL && name->size != 0);
1062
1063     // Make sure symDef and innerSymKey are not NULL
1064     pAssert(symDef != NULL && innerSymKey != NULL);
1065
1066     // Starting of sensitive data
1067     sensitiveData = inPrivate->buffer;
1068     dataSize = inPrivate->size;
1069
1070     // Find out if outer wrap is applied
1071     if(seed->size != 0)
1072     {
1073         // Use parent nameAlg as outer hash algorithm
1074         TPMI_ALG_HASH outerHash = parent->publicArea.nameAlg;
1075
1076         result = UnwrapOuter(parent, name, outerHash, seed, FALSE,
1077                             dataSize, sensitiveData);
1078         if(result != TPM_RC_SUCCESS)
1079             return result;
1080         // Adjust sensitive data pointer and size
1081         sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1082         dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1083     }
1084     // Find out if inner wrap is applied
1085     if(symDef->algorithm != TPM_ALG_NULL)

```

```

1086 {
1087     // assume the input key size matches the symmetric definition
1088     pAssert(innerSymKey->size == (symDef->keyBits.sym + 7) / 8);
1089
1090     // Decrypt inner buffer in place
1091     CryptSymmetricDecrypt(sensitiveData, symDef->algorithm,
1092                          symDef->keyBits.sym, innerSymKey->buffer, NULL,
1093                          TPM_ALG_CFB, dataSize, sensitiveData);
1094
1095     // Check inner integrity
1096     result = CheckInnerIntegrity(name, nameAlg, dataSize, sensitiveData);
1097     if(result != TPM_RC_SUCCESS)
1098         return result;
1099     // Adjust sensitive data pointer and size
1100     sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1101     dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(nameAlg);
1102 }
1103 // Unmarshal input data size
1104 buffer = sensitiveData;
1105 size = (INT32)dataSize;
1106 result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
1107 if(result == TPM_RC_SUCCESS)
1108 {
1109     if((dataSizeInput + sizeof(UINT16)) != dataSize)
1110         result = TPM_RC_SIZE;
1111     else
1112     {
1113         // Unmarshal sensitive buffer to sensitive structure
1114         result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
1115
1116         // if the results is OK make sure that all the data was unmarshaled
1117         if(result == TPM_RC_SUCCESS && size != 0)
1118             result = TPM_RC_SIZE;
1119     }
1120 }
1121 return result;
1122 }

```

7.6.3.15 SecretToCredential()

This function prepare the credential blob from a secret (a TPM2B_DIGEST) The operations in this function:

- marshal TPM2B_DIGEST structure into the buffer of TPM2B_ID_OBJECT
- encrypt the private buffer, excluding the leading integrity HMAC area
- compute integrity HMAC and append to the beginning of the buffer.
- Set the total size of TPM2B_ID_OBJECT buffer

```

1122 void
1123 SecretToCredential(
1124     TPM2B_DIGEST      *secret,           // IN: secret information
1125     TPM2B             *name,            // IN: the name of the object
1126     TPM2B             *seed,            // IN: an external seed.
1127     OBJECT            *protector,        // IN: the protector
1128     TPM2B_ID_OBJECT    *outIDObject      // OUT: output credential
1129 )
1130 {
1131     BYTE               *buffer;          // Auxiliary buffer pointer
1132     BYTE               *sensitiveData;  // pointer to the sensitive data
1133     TPMI_ALG_HASH      outerHash;        // The hash algorithm for outer wrap
1134     UINT16              dataSize;        // data blob size
1135     //
1136     pAssert(secret != NULL && outIDObject != NULL);

```

```

1137
1138 // use protector's name algorithm as outer hash ???
1139 outerHash = protector->publicArea.nameAlg;
1140
1141 // Marshal secret area to credential buffer, leave space for integrity
1142 sensitiveData = outIDObject->t.credential
1143     + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1144 // Marshal secret area
1145 buffer = sensitiveData;
1146 dataSize = TPM2B_DIGEST_Marshal(secret, &buffer, NULL);
1147
1148 // Apply outer wrap
1149 outIDObject->t.size = ProduceOuterWrap(protector, name, outerHash, seed, FALSE,
1150     dataSize, outIDObject->t.credential);
1151 return;
1152 }

```

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:

- check the integrity HMAC of the input credential area
- decrypt the credential buffer
- unmarshal TPM2B_DIGEST structure into the buffer of TPM2B_DIGEST

Error Returns	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

```

1153 TPM_RC
1154 CredentialToSecret(
1155     TPM2B          *inIDObject, // IN: input credential blob
1156     TPM2B          *name,       // IN: the name of the object
1157     TPM2B          *seed,       // IN: an external seed.
1158     OBJECT         *protector,  // IN: the protector
1159     TPM2B_DIGEST   *secret      // OUT: secret information
1160 )
1161 {
1162     TPM_RC result;
1163     BYTE   *buffer;
1164     INT32  size;
1165     TPMI_ALG_HASH outerHash; // The hash algorithm for outer wrap
1166     BYTE   *sensitiveData; // pointer to the sensitive data
1167     UINT16  dataSize;
1168 //
1169 // use protector's name algorithm as outer hash
1170 outerHash = protector->publicArea.nameAlg;
1171
1172 // Unwrap outer, a TPM_RC_INTEGRITY error may be returned at this point
1173 result = UnwrapOuter(protector, name, outerHash, seed, FALSE,
1174     inIDObject->size, inIDObject->buffer);
1175 if(result == TPM_RC_SUCCESS)
1176 {
1177     // Compute the beginning of sensitive data
1178     sensitiveData = inIDObject->buffer
1179         + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1180     dataSize = inIDObject->size

```

```

1181         - (sizeof(UINT16) + CryptHashGetDigestSize(outerHash));
1182         // Unmarshal secret buffer to TPM2B_DIGEST structure
1183         buffer = sensitiveData;
1184         size = (INT32)dataSize;
1185         result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);
1186
1187         // If there were no other unmarshaling errors, make sure that the
1188         // expected amount of data was recovered
1189         if(result == TPM_RC_SUCCESS && size != 0)
1190             return TPM_RC_SIZE;
1191     }
1192     return result;
1193 }

```

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.

```

1194 UINT16
1195 MemoryRemoveTrailingZeros(
1196     TPM2B_AUTH *auth           // IN/OUT: value to adjust
1197 )
1198 {
1199     while((auth->t.size > 0) && (auth->t.buffer[auth->t.size - 1] == 0))
1200         auth->t.size--;
1201     return auth->t.size;
1202 }

```

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.

```

1203 TPM_RC
1204 SetLabelAndContext(
1205     TPMS_DERIVE *labelContext, // IN/OUT: the recovered label and
1206                               // context
1207     TPM2B_SENSITIVE_DATA *sensitive // IN: the sensitive data
1208 )
1209 {
1210     TPMS_DERIVE sensitiveValue;
1211     TPM_RC result;
1212     INT32 size;
1213     BYTE *buff;
1214     //
1215     // Unmarshal a TPMS_DERIVE from the TPM2B_SENSITIVE_DATA buffer
1216     // If there is something to unmarshal...
1217     if(sensitive->t.size != 0)
1218     {
1219         size = sensitive->t.size;
1220         buff = sensitive->t.buffer;
1221         result = TPMS_DERIVE_Unmarshal(&sensitiveValue, &buff, &size);
1222         if(result != TPM_RC_SUCCESS)
1223             return result;
1224         // If there was a label in the public area leave it there, otherwise, copy
1225         // the new value
1226         if(labelContext->label.t.size == 0)
1227             MemoryCopy2B(&labelContext->label.b, &sensitiveValue.label.b,
1228                         sizeof(labelContext->label.t.buffer));
1229         // if there was a context string in publicArea, it overrides

```



```

1230         if(labelContext->context.t.size == 0)
1231             MemoryCopy2B(&labelContext->context.b, &sensitiveValue.context.b,
1232                         sizeof(labelContext->label.t.buffer));
1233     }
1234     return TPM_RC_SUCCESS;
1235 }

```

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*.

```

1236 TPM_RC
1237 UnmarshalToPublic(
1238     TPMT_PUBLIC          *tOut,          // OUT: output
1239     TPM2B_TEMPLATE       *tIn,          // IN:
1240     BOOL                 derivation,      // IN: indicates if this is for a derivation
1241     TPMS_DERIVE          *labelContext,   // OUT: label and context if derivation
1242 )
1243 {
1244     BYTE                 *buffer = tIn->t.buffer;
1245     INT32                 size = tIn->t.size;
1246     TPM_RC               result;
1247     //
1248     // make sure that tOut is zeroed so that there are no remnants from previous
1249     // uses
1250     MemorySet(tOut, 0, sizeof(TPMT_PUBLIC));
1251     // Unmarshal the components of the TPMT_PUBLIC up to the unique field
1252     result = TPMI_ALG_PUBLIC_Unmarshal(&tOut->type, &buffer, &size);
1253     if(result != TPM_RC_SUCCESS)
1254         return result;
1255     result = TPMI_ALG_HASH_Unmarshal(&tOut->nameAlg, &buffer, &size, FALSE);
1256     if(result != TPM_RC_SUCCESS)
1257         return result;
1258     result = TPMA_OBJECT_Unmarshal(&tOut->objectAttributes, &buffer, &size);
1259     if(result != TPM_RC_SUCCESS)
1260         return result;
1261     result = TPM2B_DIGEST_Unmarshal(&tOut->authPolicy, &buffer, &size);
1262     if(result != TPM_RC_SUCCESS)
1263         return result;
1264     result = TPMU_PUBLIC_PARMS_Unmarshal(&tOut->parameters, &buffer, &size,
1265                                         tOut->type);
1266     if(result != TPM_RC_SUCCESS)
1267         return result;
1268     // Now unmarshal a TPMS_DERIVE if this is for derivation
1269     if(derivation)
1270         result = TPMS_DERIVE_Unmarshal(labelContext, &buffer, &size);
1271     else
1272         // otherwise, unmarshal a TPMU_PUBLIC_ID
1273         result = TPMU_PUBLIC_ID_Unmarshal(&tOut->unique, &buffer, &size,
1274                                         tOut->type);
1275     // Make sure the template was used up
1276     if((result == TPM_RC_SUCCESS) && (size != 0))
1277         result = TPM_RC_SIZE;
1278     return result;
1279 }

```

7.6.3.20 ObjectSetExternal()

Set the external attributes for an object.


```
1280 void
1281 ObjectSetExternal(
1282     OBJECT      *object
1283 )
1284 {
1285     object->attributes.external = SET;
1286 }
```

DRAFT

7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c)

```

1  #include "Tpm.h"
2  #include "EncryptDecrypt_fp.h"
3  #include "EncryptDecrypt_spt_fp.h"
4  #if CC_EncryptDecrypt2

```

Error Returns	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

```

5  TPM_RC
6  EncryptDecryptShared(
7      TPMI_DH_OBJECT      keyHandleIn,
8      TPMI_YES_NO         decryptIn,
9      TPMI_ALG_SYM_MODE   modeIn,
10     TPM2B_IV             *ivIn,
11     TPM2B_MAX_BUFFER     *inData,
12     EncryptDecrypt_Out   *out
13 )
14 {
15     OBJECT      *symKey;
16     UINT16      keySize;
17     UINT16      blockSize;
18     BYTE        *key;
19     TPM_ALG_ID  alg;
20     TPM_ALG_ID  mode;
21     TPM_RC      result;
22     BOOL        OK;
23     // Input Validation
24     symKey = HandleToObject(keyHandleIn);
25     mode = symKey->publicArea.parameters.symDetail.sym.mode.sym;
26
27     // The input key should be a symmetric key
28     if(symKey->publicArea.type != TPM_ALG_SYMCIPHER)
29         return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
30     // The key must be unrestricted and allow the selected operation
31     OK = !IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
32                       TPMA_OBJECT, restricted);
33     if(YES == decryptIn)
34         OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
35                                TPMA_OBJECT, decrypt);
36     else
37         OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
38                                TPMA_OBJECT, sign);
39     if(!OK)
40         return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;
41
42     // Make sure that key is an encrypt/decrypt key and not SMAC
43     if(!CryptSymModeIsValid(mode, TRUE))
44         return TPM_RCS_MODE + RC_EncryptDecrypt_keyHandle;
45
46     // If the key mode is not TPM_ALG_NULL...
47     // or TPM_ALG_NULL
48     if(mode != TPM_ALG_NULL)
49     {
50         // then the input mode has to be TPM_ALG_NULL or the same as the key

```

```

51         if((modeIn != TPM_ALG_NULL) && (modeIn != mode))
52             return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
53     }
54     else
55     {
56         // if the key mode is null, then the input can't be null
57         if(modeIn == TPM_ALG_NULL)
58             return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
59         mode = modeIn;
60     }
61     // The input iv for ECB mode should be an Empty Buffer. All the other modes
62     // should have an iv size same as encryption block size
63     keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
64     alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
65     blockSize = CryptGetSymmetricBlockSize(alg, keySize);
66
67     // reverify the algorithm. This is mainly to keep static analysis tools happy
68     if(blockSize == 0)
69         return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
70
71     // Note: When an algorithm is not supported by a TPM, the TPM_ALG_XXX for that
72     // algorithm is not defined. However, it is assumed that the ALG_XXX_VALUE for
73     // the algorithm is always defined. Both have the same numeric value.
74     // ALG_XXX_VALUE is used here so that the code does not get cluttered with
75     // #ifdef's. Having this check does not mean that the algorithm is supported.
76     // If it was not supported the unmarshaling code would have rejected it before
77     // this function were called. This means that, depending on the implementation,
78     // the check could be redundant but it doesn't hurt.
79     if((mode == ALG_ECB_VALUE) && (ivIn->t.size != 0))
80         || ((mode != ALG_ECB_VALUE) && (ivIn->t.size != blockSize)))
81         return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;
82
83     // The input data size of CBC mode or ECB mode must be an even multiple of
84     // the symmetric algorithm's block size
85     if((mode == ALG_CBC_VALUE) || (mode == ALG_ECB_VALUE))
86         && ((inData->t.size % blockSize) != 0))
87         return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;
88
89     // Copy IV
90     // Note: This is copied here so that the calls to the encrypt/decrypt functions
91     // will modify the output buffer, not the input buffer
92     out->ivOut = *ivIn;
93
94     // Command Output
95     key = symKey->sensitive.sensitive.sym.t.buffer;
96     // For symmetric encryption, the cipher data size is the same as plain data
97     // size.
98     out->outData.t.size = inData->t.size;
99     if(decryptIn == YES)
100     {
101         // Decrypt data to output
102         result = CryptSymmetricDecrypt(out->outData.t.buffer, alg, keySize, key,
103                                     &(out->ivOut), mode, inData->t.size,
104                                     inData->t.buffer);
105     }
106     else
107     {
108         // Encrypt data to output
109         result = CryptSymmetricEncrypt(out->outData.t.buffer, alg, keySize, key,
110                                     &(out->ivOut), mode, inData->t.size,
111                                     inData->t.buffer);
112     }
113     return result;
114 }
115 #endif // CC_EncryptDecrypt

```

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 is 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
142     return more;
143 }

```

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.

```

144 void
145 CommandAuditGetDigest(

```



```
146     TPM2B_DIGEST    *digest          // OUT: command digest
147 )
148 {
149     TPM_CC            commandCode;
150     COMMAND_INDEX     commandIndex;
151     HASH_STATE        hashState;
152
153     // Start hash
154     digest->t.size = CryptHashStart(&hashState, gp.auditHashAlg);
155
156     // Add command code
157     for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
158     {
159         if(CommandAuditIsRequired(commandIndex))
160         {
161             commandCode = GetCommandCode(commandIndex);
162             CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
163         }
164     }
165
166     // Complete hash
167     CryptHashEnd2B(&hashState, &digest->b);
168
169     return;
170 }
```

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      #endif
32  }
```

```

31     s_selfHealTimer = 0;
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
63     // If DA has not been disabled and the previous shutdown is not orderly
64     // failedTries is not already at its maximum then increment 'failedTries'
65     if(gp.recoveryTime != 0
66        && gp.failedTries < gp.maxTries
67        && !IS_ORDERLY(g_prevOrderlyState))
68     {
69 #if USE_DA_USED
70         gp.failedTries += g_daUsed;
71         g_daUsed = FALSE;
72 #else
73         gp.failedTries++;
74 #endif
75         // Record the change to NV
76         NV_SYNC_PERSISTENT(failedTries);
77     }
78     // Before Startup, the TPM will not do clock updates. At startup, need to
79     // do a time update which will do the DA update.
80     TimeUpdate();
81
82     return TRUE;
83 }

```

8.2.3.3 DARegisterFailure()

This function is called when a 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.

```

84 void
85 DARegisterFailure(
86     TPM_HANDLE    handle    // IN: handle for failure
87 )

```

```

88 {
89     // Reset the timer associated with lockout if the handle is the lockoutAuth.
90     if(handle == TPM_RH_LOCKOUT)
91         s_lockoutTimer = g_time;
92     else
93         s_selfHealTimer = g_time;
94     return;
95 }

```

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.

```

96 void
97 DASelfHeal(
98     void
99 )
100 {
101     // Regular authorization self healing logic
102     // If no failed authorization tries, do nothing. Otherwise, try to
103     // decrease failedTries
104     if(gp.failedTries != 0)
105     {
106         // if recovery time is 0, DA logic has been disabled. Clear failed tries
107         // immediately
108         if(gp.recoveryTime == 0)
109         {
110             gp.failedTries = 0;
111             // Update NV record
112             NV_SYNC_PERSISTENT(failedTries);
113         }
114         else
115         {
116             UINT64 decreaseCount;
117             #if 0 // Errata eliminates this code
118                 // In the unlikely event that failedTries should become larger than
119                 // maxTries
120                 if(gp.failedTries > gp.maxTries)
121                     gp.failedTries = gp.maxTries;
122             #endif
123             // How much can failedTries be decreased
124
125             // Cast s_selfHealTimer to an int in case it became negative at
126             // startup
127             decreaseCount = ((g_time - (INT64)s_selfHealTimer) / 1000)
128                 / gp.recoveryTime;
129
130             if(gp.failedTries <= (UINT32)decreaseCount)
131                 // should not set failedTries below zero
132                 gp.failedTries = 0;
133             else
134                 gp.failedTries -= (UINT32)decreaseCount;
135
136             // the cast prevents overflow of the product
137             s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
138             if(decreaseCount != 0)
139                 // If there was a change to the failedTries, record the changes
140                 // to NV
141                 NV_SYNC_PERSISTENT(failedTries);
142         }
143     }

```

```
144
145 // LockoutAuth self healing logic
146 // If lockoutAuth is enabled, do nothing. Otherwise, try to see if we
147 // may enable it
148 if(!gp.lockOutAuthEnabled)
149 {
150     // if lockout authorization recovery time is 0, a reboot is required to
151     // re-enable use of lockout authorization. Self-healing would not
152     // apply in this case.
153     if(gp.lockoutRecovery != 0)
154     {
155         if(((g_time - (INT64)s_lockoutTimer) / 1000) >= gp.lockoutRecovery)
156         {
157             gp.lockOutAuthEnabled = TRUE;
158             // Record the changes to NV
159             NV_SYNC_PERSISTENT(lockOutAuthEnabled);
160         }
161     }
162 }
163 return;
164 }
```

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
83     // nullProof and nullSeed are updated at every TPM_RESET
84     if((type != SU_RESTART) && (type != SU_RESUME))
85     {
86         gr.nullProof.t.size = sizeof(gr.nullProof.t.buffer);
87         CryptRandomGenerate(gr.nullProof.t.size, gr.nullProof.t.buffer);
88         gr.nullSeed.t.size = sizeof(gr.nullSeed.t.buffer);
89         CryptRandomGenerate(gr.nullSeed.t.size, gr.nullSeed.t.buffer);
90     }
91
92     return TRUE;
93 }

```

8.3.3.3 HierarchyGetProof()

This function finds the proof value associated with a hierarchy. It returns a pointer to the proof value.

```

94 TPM2B_PROOF *
95 HierarchyGetProof(
96     TPMI_RH_HIERARCHY    hierarchy    // IN: hierarchy constant

```



```

97     )
98 {
99     TPM2B_PROOF      *proof = NULL;
100
101     switch(hierarchy)
102     {
103         case TPM_RH_PLATFORM:
104             // phProof for TPM_RH_PLATFORM
105             proof = &gp.phProof;
106             break;
107         case TPM_RH_ENDORSEMENT:
108             // ehProof for TPM_RH_ENDORSEMENT
109             proof = &gp.ehProof;
110             break;
111         case TPM_RH_OWNER:
112             // shProof for TPM_RH_OWNER
113             proof = &gp.shProof;
114             break;
115         default:
116             // nullProof for TPM_RH_NULL or anything else
117             proof = &gr.nullProof;
118             break;
119     }
120     return proof;
121 }

```

8.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

```

122 TPM2B_SEED *
123 HierarchyGetPrimarySeed(
124     TPMI_RH_HIERARCHY    hierarchy    // IN: hierarchy
125 )
126 {
127     TPM2B_SEED      *seed = NULL;
128     switch(hierarchy)
129     {
130         case TPM_RH_PLATFORM:
131             seed = &gp.PPSeed;
132             break;
133         case TPM_RH_OWNER:
134             seed = &gp.SPSeed;
135             break;
136         case TPM_RH_ENDORSEMENT:
137             seed = &gp.EPSeed;
138             break;
139         default:
140             seed = &gr.nullSeed;
141             break;
142     }
143     return seed;
144 }

```

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

```

145  BOOL
146  HierarchyIsEnabled(
147      TPMI_RH_HIERARCHY    hierarchy    // IN: hierarchy
148  )
149  {
150      BOOL                enabled = FALSE;
151
152      switch(hierarchy)
153      {
154          case TPM_RH_PLATFORM:
155              enabled = g_phEnable;
156              break;
157          case TPM_RH_OWNER:
158              enabled = gc.shEnable;
159              break;
160          case TPM_RH_ENDORSEMENT:
161              enabled = gc.ehEnable;
162              break;
163          case TPM_RH_NULL:
164              enabled = TRUE;
165              break;
166          default:
167              enabled = FALSE;
168              break;
169      }
170      return enabled;
171  }

```

8.4 NvDynamic.c

8.4.1 Introduction

The NV memory is divided into two area: 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* PERSISTEND_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"
3  #include "PlatformData.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 is no more element in the set, a 0 value is returned indicating the end of traversal.

```
4  static NV_REF
5  NvNext(
6      NV_REF          *iter,           // IN/OUT: the list iterator
7      TPM_HANDLE      *handle         // OUT: the handle of the next item.
8  )
9  {
10     NV_REF          currentAddr;
11     NV_ENTRY_HEADER header;
12     //
```

```

13     // If iterator is at the beginning of list
14     if(*iter == NV_REF_INIT)
15     {
16         // Initialize iterator
17         *iter = NV_USER_DYNAMIC;
18     }
19     // Step over the size field and point to the handle
20     currentAddr = *iter + sizeof(UINT32);
21
22     // read the header of the next entry
23     NvRead(&header, *iter, sizeof(NV_ENTRY_HEADER));
24
25     // if the size field is zero, then we have hit the end of the list
26     if(header.size == 0)
27         // leave the *iter pointing at the end of the list
28         return 0;
29     // advance the header by the size of the entry
30     *iter += header.size;
31
32     if(handle != NULL)
33         *handle = header.handle;
34     return currentAddr;
35 }

```

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

```

36 static NV_REF
37 NvNextByType(
38     TPM_HANDLE *handle,           // OUT: the handle of the found type
39     NV_REF *iter,                // IN: the iterator
40     TPM_HT type                  // IN: the handle type to look for
41 )
42 {
43     NV_REF addr;
44     TPM_HANDLE nvHandle;
45     //
46     while((addr = NvNext(iter, &nvHandle)) != 0)
47     {
48         // addr: the address of the location containing the handle of the value
49         // iter: the next location.
50         if(HandleGetType(nvHandle) == type)
51             break;
52     }
53     if(handle != NULL)
54         *handle = nvHandle;
55     return addr;
56 }

```

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

```

57  #define NvNextIndex(handle, iter)          \
58      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.

```

59  #define NvNextEvict(handle, iter)          \
60      NvNextByType(handle, iter, TPM_HT_PERSISTENT)

```

8.4.3.5 NvGetEnd()

Function to find the end of the NV dynamic data list

```

61  static NV_REF
62  NvGetEnd(
63      void
64  )
65  {
66      NV_REF      iter = NV_REF_INIT;
67      NV_REF      currentAddr;
68  //
69      // Scan until the next address is 0
70      while((currentAddr = NvNext(&iter, NULL)) != 0);
71      return iter;
72  }

```

8.4.3.6 NvGetFreeBytes

This function returns the number of free octets in NV space.

```

73  static UINT32
74  NvGetFreeBytes(
75      void
76  )
77  {
78      // This does not have an overflow issue because NvGetEnd() cannot return a value
79      // that is larger than s_evictNvEnd. This is because there is always a 'stop'
80      // word in the NV memory that terminates the search for the end before the
81      // value can go past s_evictNvEnd.
82      return s_evictNvEnd - NvGetEnd();
83  }

```

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

```

84  static BOOL
85  NvTestSpace(
86      UINT32      size,          // IN: size of the entity to be added
87      BOOL        isIndex,      // IN: TRUE if the entity is an index
88      BOOL        isCounter     // IN: TRUE if the index is a counter
89  )
90  {
91      UINT32      remainBytes = NvGetFreeBytes();
92      UINT32      reserved = sizeof(UINT32)          // size of the forward pointer
93      + sizeof(NV_LIST_TERMINATOR);
94      //
95      // Do a compile time sanity check on the setting for NV_MEMORY_SIZE
96      #if NV_MEMORY_SIZE < 1024
97      #error "NV_MEMORY_SIZE probably isn't large enough"
98      #endif
99
100     // For NV Index, need to make sure that we do not allocate an Index if this
101     // would mean that the TPM cannot allocate the minimum number of evict
102     // objects.
103     if(isIndex)
104     {
105         // Get the number of persistent objects allocated
106         UINT32      persistentNum = NvCapGetPersistentNumber();
107
108         // If we have not allocated the requisite number of evict objects, then we
109         // need to reserve space for them.
110         // NOTE: some of this is not written as simply as it might seem because
111         // the values are all unsigned and subtracting needs to be done carefully
112         // so that an underflow doesn't cause problems.
113         if(persistentNum < MIN_EVICT_OBJECTS)
114             reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
115     }
116     // If this is not an index or is not a counter, reserve space for the
117     // required number of counter indexes
118     if(!isIndex || !isCounter)
119     {
120         // Get the number of counters
121         UINT32      counterNum = NvCapGetCounterNumber();
122
123         // If the required number of counters have not been allocated, reserved
124         // space for the extra needed counters
125         if(counterNum < MIN_COUNTER_INDICES)
126             reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
127     }
128     // Check that the requested allocation will fit after making sure that there
129     // will be no chance of overflow
130     return ((reserved < remainBytes)
131         && (size <= remainBytes)
132         && (size + reserved <= remainBytes));
133 }

```

8.4.3.8 NvWriteNvListEnd()

Function to write the list terminator.

```

134  NV_REF
135  NvWriteNvListEnd(
136      NV_REF      end

```

```

137     )
138 {
139     // Marker is initialized with zeros
140     BYTE      listEndMarker[sizeof(NV_LIST_TERMINATOR)] = {0};
141     UINT64     maxCount = NvReadMaxCount();
142     //
143     // This is a constant check that can be resolved at compile time.
144     cAssert(sizeof(UINT64) <= sizeof(NV_LIST_TERMINATOR) - sizeof(UINT32));
145
146     // Copy the maxCount value to the marker buffer
147     MemoryCopy(&listEndMarker[sizeof(UINT32)], &maxCount, sizeof(UINT64));
148     pAssert(end + sizeof(NV_LIST_TERMINATOR) <= s_evictNvEnd);
149
150     // Write it to memory
151     NvWrite(end, sizeof(NV_LIST_TERMINATOR), &listEndMarker);
152     return end + sizeof(NV_LIST_TERMINATOR);
153 }

```

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.

```

154 static TPM_RC
155 NvAdd(
156     UINT32      totalSize,      // IN: total size needed for this entity For
157                                // evict object, totalSize is the same as
158                                // bufferSize. For NV Index, totalSize is
159                                // bufferSize plus index data size
160     UINT32      bufferSize,    // IN: size of initial buffer
161     TPM_HANDLE  handle,        // IN: optional handle
162     BYTE        *entity         // IN: initial buffer
163 )
164 {
165     NV_REF      newAddr;        // IN: where the new entity will start
166     NV_REF      nextAddr;
167     //
168     RETURN_IF_NV_IS_NOT_AVAILABLE;
169
170     // Get the end of data list
171     newAddr = NvGetEnd();
172
173     // Step over the forward pointer
174     nextAddr = newAddr + sizeof(UINT32);
175
176     // Optionally write the handle. For indexes, the handle is TPM_RH_UNASSIGNED
177     // so that the handle in the nvIndex is used instead of writing this value
178     if(handle != TPM_RH_UNASSIGNED)
179     {
180         NvWrite((UINT32)nextAddr, sizeof(TPM_HANDLE), &handle);
181         nextAddr += sizeof(TPM_HANDLE);
182     }
183     // Write entity data
184     NvWrite((UINT32)nextAddr, bufferSize, entity);
185
186     // Advance the pointer by the amount of the total
187     nextAddr += totalSize;
188
189     // Finish by writing the link value
190
191     // Write the next offset (relative addressing)

```



```

192     totalSize = nextAddr - newAddr;
193
194     // Write link value
195     NvWrite((UINT32)newAddr, sizeof(UINT32), &totalSize);
196
197     // Write the list terminator
198     NvWriteNvListEnd(nextAddr);
199
200     return TPM_RC_SUCCESS;
201 }

```

8.4.3.10 NvDelete()

This function is used to delete an NV Index or persistent object from NV memory.

```

202 static TPM_RC
203 NvDelete(
204     NV_REF          entityRef      // IN: reference to entity to be deleted
205 )
206 {
207     UINT32          entrySize;
208     // adjust entityAddr to back up and point to the forward pointer
209     NV_REF          entryRef = entityRef - sizeof(UINT32);
210     NV_REF          endRef = NvGetEnd();
211     NV_REF          nextAddr; // address of the next entry
212     //
213     RETURN_IF_NV_IS_NOT_AVAILABLE;
214
215     // Get the offset of the next entry. That is, back up and point to the size
216     // field of the entry
217     NvRead(&entrySize, entryRef, sizeof(UINT32));
218
219     // The next entry after the one being deleted is at a relative offset
220     // from the current entry
221     nextAddr = entryRef + entrySize;
222
223     // If this is not the last entry, move everything up
224     if(nextAddr < endRef)
225     {
226         pAssert(nextAddr > entryRef);
227         _plat__NvMemoryMove(nextAddr,
228                             entryRef,
229                             (endRef - nextAddr));
230     }
231     // The end of the used space is now moved up by the amount of space we just
232     // reclaimed
233     endRef -= entrySize;
234
235     // Write the end marker, and make the new end equal to the first byte after
236     // the just added end value. This will automatically update the NV value for
237     // maxCounter.
238     // NOTE: This is the call that sets flag to cause NV to be updated
239     endRef = NvWriteNvListEnd(endRef);
240
241     // Clear the reclaimed memory
242     _plat__NvMemoryClear(endRef, entrySize);
243
244     return TPM_RC_SUCCESS;
245 }

```

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

```

246 static NV_RAM_REF
247 NvRamNext(
248     NV_RAM_REF      *iter,           // IN/OUT: the list iterator
249     TPM_HANDLE      *handle         // OUT: the handle of the next item.
250 )
251 {
252     NV_RAM_REF      currentAddr;
253     NV_RAM_HEADER   header;
254     //
255     // If iterator is at the beginning of list
256     if(*iter == NV_RAM_REF_INIT)
257     {
258         // Initialize iterator
259         *iter = &s_indexOrderlyRam[0];
260     }
261     // if we are going to return what the iter is currently pointing to...
262     currentAddr = *iter;
263
264     // If iterator reaches the end of NV space, then don't advance and return
265     // that we are at the end of the list. The end of the list occurs when
266     // we don't have space for a size and a handle
267     if(currentAddr + sizeof(NV_RAM_HEADER) > RAM_ORDERLY_END)
268         return NULL;
269     // read the header of the next entry
270     MemoryCopy(&header, currentAddr, sizeof(NV_RAM_HEADER));
271
272     // if the size field is zero, then we have hit the end of the list
273     if(header.size == 0)
274         // leave the *iter pointing at the end of the list
275         return NULL;
276     // advance the header by the size of the entry
277     *iter = currentAddr + header.size;
278
279     // pAssert(*iter <= RAM_ORDERLY_END);
280     if(handle != NULL)
281         *handle = header.handle;
282     return currentAddr;
283 }

```

8.4.4.3 NvRamGetEnd()

This routine performs the same function as NvGetEnd() but for the RAM data.

```

284 static NV_RAM_REF
285 NvRamGetEnd(
286     void
287 )

```

```

288 {
289     NV_RAM_REF        iter = NV_RAM_REF_INIT;
290     NV_RAM_REF        currentAddr;
291     //
292     // Scan until the next address is 0
293     while((currentAddr = NvRamNext(&iter, NULL)) != 0);
294     return iter;
295 }

```

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

```

296 static BOOL
297 NvRamTestSpaceIndex(
298     UINT32        size           // IN: size of the data to be added to RAM
299 )
300 {
301     UINT32        remaining = RAM_ORDERLY_END - NvRamGetEnd();
302     UINT32        needed = sizeof(NV_RAM_HEADER) + size;
303     //
304     // NvRamGetEnd points to the next available byte.
305     return remaining >= needed;
306 }

```

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.

```

307 static NV_RAM_REF
308 NvRamGetIndex(
309     TPMI_RH_NV_INDEX    handle           // IN: NV handle
310 )
311 {
312     NV_RAM_REF        iter = NV_RAM_REF_INIT;
313     NV_RAM_REF        currentAddr;
314     TPM_HANDLE        foundHandle;
315     //
316     while((currentAddr = NvRamNext(&iter, &foundHandle)) != 0)
317     {
318         if(handle == foundHandle)
319             break;
320     }
321     return currentAddr;
322 }

```

8.4.4.6 NvUpdateIndexOrderlyData()

This function is used to cause an update of the orderly data to the NV backing store.

```

323 void
324 NvUpdateIndexOrderlyData(
325     void
326 )

```

```

327 {
328     // Write reserved RAM space to NV
329     NvWrite(NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam), s_indexOrderlyRam);
330 }

```

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.

```

331 static void
332 NvAddRAM(
333     TPMS_NV_PUBLIC *index          // IN: the index descriptor
334 )
335 {
336     NV_RAM_HEADER header;
337     NV_RAM_REF end = NvRamGetEnd();
338     //
339     header.size = sizeof(NV_RAM_HEADER) + index->dataSize;
340     header.handle = index->nvIndex;
341     MemoryCopy(&header.attributes, &index->attributes, sizeof(TPMA_NV));
342
343     pAssert(ORDERLY_RAM_ADDRESS_OK(end, header.size));
344
345     // Copy the header to the memory
346     MemoryCopy(end, &header, sizeof(NV_RAM_HEADER));
347
348     // Clear the data area (just in case)
349     MemorySet(end + sizeof(NV_RAM_HEADER), 0, index->dataSize);
350
351     // Step over this new entry
352     end += header.size;
353
354     // If the end marker will fit, add it
355     if(end + sizeof(UINT32) < RAM_ORDERLY_END)
356         MemorySet(end, 0, sizeof(UINT32));
357     // Write reserved RAM space to NV to reflect the newly added NV Index
358     SET_NV_UPDATE(UT_ORDERLY);
359
360     return;
361 }

```

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.

```

362 static void
363 NvDeleteRAM(
364     TPMS_NV_PUBLIC *index          // IN: NV handle
365 )
366 {
367     NV_RAM_REF nodeAddress;
368     NV_RAM_REF nextNode;

```

```

369     UINT32          size;
370     NV_RAM_REF      lastUsed = NvRamGetEnd();
371 //
372     nodeAddress = NvRamGetIndex(handle);
373
374     pAssert(nodeAddress != 0);
375
376     // Get node size
377     MemoryCopy(&size, nodeAddress, sizeof(size));
378
379     // Get the offset of next node
380     nextNode = nodeAddress + size;
381
382     // Copy the data
383     MemoryCopy(nodeAddress, nextNode, lastUsed - nextNode);
384
385     // Clear out the reclaimed space
386     MemorySet(lastUsed - size, 0, size);
387
388     // Write reserved RAM space to NV to reflect the newly delete NV Index
389     SET_NV_UPDATE(UT_ORDERLY);
390
391     return;
392 }

```

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.

```

393 void
394 NvReadNvIndexInfo(
395     NV_REF          ref,           // IN: points to NV where index is located
396     NV_INDEX        *nvIndex      // OUT: place to receive index data
397 )
398 {
399     pAssert(nvIndex != NULL);
400     NvRead(nvIndex, ref, sizeof(NV_INDEX));
401     return;
402 }

```

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.

```

403 void
404 NvReadObject(
405     NV_REF          ref,           // IN: points to NV where index is located
406     OBJECT          *object      // OUT: place to receive the object data
407 )
408 {
409     NvRead(object, (ref + sizeof(TPM_HANDLE)), sizeof(OBJECT));
410     return;
411 }

```

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

```

412 static NV_REF
413 NvFindEvict(
414     TPM_HANDLE    nvHandle,
415     OBJECT        *object
416 )
417 {
418     NV_REF        found = NvFindHandle(nvHandle);
419     //
420     // If we found the handle and the request included an object pointer, fill it in
421     if(found != 0 && object != NULL)
422         NvReadObject(found, object);
423     return found;
424 }

```

8.4.4.12 NvIndexIsDefined()

See if an index is already defined

```

425 BOOL
426 NvIndexIsDefined(
427     TPM_HANDLE    nvHandle    // IN: Index to look for
428 )
429 {
430     return (NvFindHandle(nvHandle) != 0);
431 }

```

8.4.4.13 NvConditionallyWrite()

Function to check if the data to be written has changed and write it if it has

Error Returns	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```

432 static TPM_RC
433 NvConditionallyWrite(
434     NV_REF        entryAddr,    // IN: stating address
435     UINT32        size,        // IN: size of the data to write
436     void          *data        // IN: the data to write
437 )
438 {
439     // If the index data is actually changed, then a write to NV is required
440     if(_plat__NvIsDifferent(entryAddr, size, data))
441     {
442         // Write the data if NV is available
443         if(g_NvStatus == TPM_RC_SUCCESS)
444         {
445             NvWrite(entryAddr, size, data);
446         }
447         return g_NvStatus;
448     }
449     return TPM_RC_SUCCESS;
450 }

```

8.4.4.14 NvReadNvIndexAttributes()

This function returns the attributes of an NV Index.

```

451 static TPMA_NV
452 NvReadNvIndexAttributes(
453     NV_REF          locator          // IN: reference to an NV index
454 )
455 {
456     TPMA_NV          attributes;
457     //
458     NvRead(&attributes,
459         locator + offsetof(NV_INDEX, publicArea.attributes),
460         sizeof(TPMA_NV));
461     return attributes;
462 }

```

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.

```

463 static TPMA_NV
464 NvReadRamIndexAttributes(
465     NV_RAM_REF      ref              // IN: pointer to a NV_RAM_HEADER
466 )
467 {
468     TPMA_NV          attributes;
469     //
470     MemoryCopy(&attributes, ref + offsetof(NV_RAM_HEADER, attributes),
471         sizeof(TPMA_NV));
472     return attributes;
473 }

```

8.4.4.16 NvWriteNvIndexAttributes()

This function is used to write just the attributes of an index to NV.

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```

474 static TPM_RC
475 NvWriteNvIndexAttributes(
476     NV_REF          locator,          // IN: location of the index
477     TPMA_NV          attributes       // IN: attributes to write
478 )
479 {
480     return NvConditionallyWrite(
481         locator + offsetof(NV_INDEX, publicArea.attributes),
482         sizeof(TPMA_NV),
483         &attributes);
484 }

```

8.4.4.17 NvWriteRamIndexAttributes()

This function is used to write the index attributes into an unaligned structure


```

485 static void
486 NvWriteRamIndexAttributes(
487     NV_RAM_REF      ref,           // IN: address of the header
488     TPMA_NV          attributes    // IN: the attributes to write
489 )
490 {
491     MemoryCopy(ref + offsetof(NV_RAM_HEADER, attributes), &attributes,
492               sizeof(TPMA_NV));
493     return;
494 }

```

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 and may reference an owner persistent object either
FALSE(0)	handle does not reference platform persistent object

```

495 BOOL
496 NvIsPlatformPersistentHandle(
497     TPM_HANDLE      handle        // IN: handle
498 )
499 {
500     return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
501 }

```

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

```

502 BOOL
503 NvIsOwnerPersistentHandle(
504     TPM_HANDLE      handle        // IN: handle
505 )
506 {
507     return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
508 }

```

8.4.5.3 NvIndexIsAccessible()

This function validates that a handle references a defined NV Index and that the Index is currently accessible.

Error Returns	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

```

509  TPM_RC
510  NvIndexIsAccessible(
511      TPMI_RH_NV_INDEX    handle        // IN: handle
512  )
513  {
514      NV_INDEX             *nvIndex = NvGetIndexInfo(handle, NULL);
515      //
516      if(nvIndex == NULL)
517          // If index is not found, return TPM_RC_HANDLE
518          return TPM_RC_HANDLE;
519      if(gc.shEnable == FALSE || gc.phEnableNV == FALSE)
520      {
521          // if shEnable is CLEAR, an ownerCreate NV Index should not be
522          // indicated as present
523          if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
524          {
525              if(gc.shEnable == FALSE)
526                  return TPM_RC_HANDLE;
527          }
528          // if phEnableNV is CLEAR, a platform created Index should not
529          // be visible
530          else if(gc.phEnableNV == FALSE)
531              return TPM_RC_HANDLE;
532      }
533      #if 0 // Writelock test for debug
534          // If the Index is write locked and this is an NV Write operation...
535          if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITELOCKED)
536             && IsWriteOperation(commandIndex))
537          {
538              // then return a locked indication unless the command is TPM2_NV_WriteLock
539              if(GetCommandCode(commandIndex) != TPM_CC_NV_WriteLock)
540                  return TPM_RC_NV_LOCKED;
541              return TPM_RC_SUCCESS;
542          }
543      #endif
544      #if 0 // Readlock Test for debug
545          // If the Index is read locked and this is an NV Read operation...
546          if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, READLOCKED)
547             && IsReadOperation(commandIndex))
548          {
549              // then return a locked indication unless the command is TPM2_NV_ReadLock
550              if(GetCommandCode(commandIndex) != TPM_CC_NV_ReadLock)
551                  return TPM_RC_NV_LOCKED;
552          }
553      #endif
554          // NV Index is accessible
555          return TPM_RC_SUCCESS;
556  }

```

8.4.5.4 NvGetEvictObject()

This function is used to dereference an evict object handle and get a pointer to the object.

Error Returns	Meaning
TPM_RC_HANDLE	the handle does not point to an existing persistent object

```

557 TPM_RC
558 NvGetEvictObject(
559     TPM_HANDLE    handle,          // IN: handle
560     OBJECT        *object          // OUT: object data
561 )
562 {
563     NV_REF         entityAddr;      // offset points to the entity
564     //
565     // Find the address of evict object and copy to object
566     entityAddr = NvFindEvict(handle, object);
567
568     // whether there is an error or not, make sure that the evict
569     // status of the object is set so that the slot will get freed on exit
570     // Must do this after NvFindEvict loads the object
571     object->attributes.evict = SET;
572
573     // If handle is not found, return an error
574     if(entityAddr == 0)
575         return TPM_RC_HANDLE;
576     return TPM_RC_SUCCESS;
577 }

```

8.4.5.5 NvIndexCacheInit()

Function to initialize the Index cache

```

578 void
579 NvIndexCacheInit(
580     void
581 )
582 {
583     s_cachedNvRef = NV_REF_INIT;
584     s_cachedNvRamRef = NV_RAM_REF_INIT;
585     s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
586     return;
587 }

```

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.

```

588 void
589 NvGetIndexData(
590     NV_INDEX       *nvIndex,        // IN: the in RAM index descriptor
591     NV_REF         locator,         // IN: where the data is located
592     UINT32         offset,          // IN: offset of NV data
593     UINT16         size,            // IN: number of octets of NV data to read
594     void          *data             // OUT: data buffer
595 )
596 {
597     TPMA_NV        nvAttributes;
598     //
599     pAssert(nvIndex != NULL);
600
601     nvAttributes = nvIndex->publicArea.attributes;

```

```

602
603     pAssert(IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN));
604
605     if(IS_ATTRIBUTE(nvAttributes, TPMA_NV, ORDERLY))
606     {
607         // Get data from RAM buffer
608         NV_RAM_REF ramAddr = NvRamGetIndex(nvIndex->publicArea.nvIndex);
609         pAssert(ramAddr != 0 && (size <=
610             ((NV_RAM_HEADER *)ramAddr->size - sizeof(NV_RAM_HEADER) - offset));
611         MemoryCopy(data, ramAddr + sizeof(NV_RAM_HEADER) + offset, size);
612     }
613     else
614     {
615         // Validate that read falls within range of the index
616         pAssert(offset <= nvIndex->publicArea.dataSize
617             && size <= (nvIndex->publicArea.dataSize - offset));
618         NvRead(data, locator + sizeof(NV_INDEX) + offset, size);
619     }
620     return;
621 }

```

8.4.5.7 NvHashIndexData()

This function adds Index data to a hash. It does this in parts to avoid large stack buffers.

```

622 void
623 NvHashIndexData(
624     HASH_STATE      *hashState,    // IN: Initialized hash state
625     NV_INDEX        *nvIndex,      // IN: Index
626     NV_REF          locator,        // IN: where the data is located
627     UINT32          offset,         // IN: starting offset
628     UINT16          size,           // IN: amount to hash
629 )
630 {
631     #define BUFFER_SIZE 64
632     BYTE buffer[BUFFER_SIZE];
633     if (offset > nvIndex->publicArea.dataSize)
634         return;
635     // Make sure that we don't try to read off the end.
636     if ((offset + size) > nvIndex->publicArea.dataSize)
637         size = nvIndex->publicArea.dataSize - (UINT16)offset;
638     #if BUFFER_SIZE >= MAX_NV_INDEX_SIZE
639         NvGetIndexData(nvIndex, locator, offset, size, buffer);
640         CryptDigestUpdate(hashState, size, buffer);
641     #else
642     {
643         INT16 i;
644         UINT16 readSize;
645         //
646         for (i = size; i > 0; offset += readSize, i -= readSize)
647         {
648             readSize = (i < BUFFER_SIZE) ? i : BUFFER_SIZE;
649             NvGetIndexData(nvIndex, locator, offset, readSize, buffer);
650             CryptDigestUpdate(hashState, readSize, buffer);
651         }
652     }
653     #endif // BUFFER_SIZE >= MAX_NV_INDEX_SIZE
654     #undef BUFFER_SIZE
655 }

```

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.

```

656  UINT64
657  NvGetUINT64Data(
658      NV_INDEX          *nvIndex,          // IN: the in RAM index descriptor
659      NV_REF            locator            // IN: where index exists in NV
660  )
661  {
662      UINT64            intVal;
663      //
664      // Read the value and convert it to internal format
665      NvGetIndexData(nvIndex, locator, 0, 8, &intVal);
666      return BYTE_ARRAY_TO_UINT64((BYTE *)&intVal);
667  }

```

8.4.5.9 NvWriteIndexAttributes()

This function is used to write just the attributes of an index.

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```

668  TPM_RC
669  NvWriteIndexAttributes(
670      TPM_HANDLE        handle,
671      NV_REF            locator,          // IN: location of the index
672      TPMA_NV           attributes       // IN: attributes to write
673  )
674  {
675      TPM_RC            result;
676      //
677      if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
678      {
679          NV_RAM_REF     ram = NvRamGetIndex(handle);
680          NvWriteRamIndexAttributes(ram, attributes);
681          result = TPM_RC_SUCCESS;
682      }
683      else
684      {
685          result = NvWriteNvIndexAttributes(locator, attributes);
686      }
687      return result;
688  }

```

8.4.5.10 NvWriteIndexAuth()

This function is used to write the *authValue* of an index. It is used by TPM2_NV_ChangeAuth()

Error Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```

689  TPM_RC
690  NvWriteIndexAuth(
691      NV_REF            locator,          // IN: location of the index
692      TPM2B_AUTH        *authValue       // IN: the authValue to write
693  )

```

```

694 {
695     TPM_RC          result;
696 //
697 // If the locator is pointing to the cached index value...
698 if(locator == s_cachedNvRef)
699 {
700     // copy the authValue to the cached index so it will be there if we
701     // look for it. This is a safety thing.
702     MemoryCopy2B(&s_cachedNvIndex.authValue.b, &authValue->b,
703                 sizeof(s_cachedNvIndex.authValue.t.buffer));
704 }
705 result = NvConditionallyWrite(
706     locator + offsetof(NV_INDEX, authValue),
707     sizeof(UINT16) + authValue->t.size,
708     authValue);
709 return result;
710 }

```

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

```

711 NV_INDEX *
712 NvGetIndexInfo(
713     TPM_HANDLE      nvHandle,          // IN: the index handle
714     NV_REF          *locator          // OUT: location of the index
715 )
716 {
717     if(s_cachedNvIndex.publicArea.nvIndex != nvHandle)
718     {
719         s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
720         s_cachedNvRamRef = 0;
721         s_cachedNvRef = NvFindHandle(nvHandle);
722         if(s_cachedNvRef == 0)
723             return NULL;
724         NvReadNvIndexInfo(s_cachedNvRef, &s_cachedNvIndex);
725         if(IS_ATTRIBUTE(s_cachedNvIndex.publicArea.attributes, TPMA_NV, ORDERLY))
726         {
727             s_cachedNvRamRef = NvRamGetIndex(nvHandle);
728             s_cachedNvIndex.publicArea.attributes =
729                 NvReadRamIndexAttributes(s_cachedNvRamRef);
730         }
731     }
732     if(locator != NULL)
733         *locator = s_cachedNvRef;
734     return &s_cachedNvIndex;
735 }

```

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 Returns	Meaning
TPM_RC_NV_RATE	NV is rate limiting so retry
TPM_RC_NV_UNAVAILABLE	NV is not available

```

736 TPM_RC
737 NvWriteIndexData(
738     NV_INDEX      *nvIndex,          // IN: the description of the index
739     UINT32         offset,           // IN: offset of NV data
740     UINT32         size,             // IN: size of NV data
741     void           *data,            // IN: data buffer
742 )
743 {
744     TPM_RC          result = TPM_RC_SUCCESS;
745     //
746     pAssert(nvIndex != NULL);
747     // Make sure that this is dealing with the 'default' index.
748     // Note: it is tempting to change the calling sequence so that the 'default' is
749     // presumed.
750     pAssert(nvIndex->publicArea.nvIndex == s_cachedNvIndex.publicArea.nvIndex);
751
752     // Validate that write falls within range of the index
753     pAssert(offset <= nvIndex->publicArea.dataSize
754             && size <= (nvIndex->publicArea.dataSize - offset));
755
756     // Update TPMA_NV_WRITTEN bit if necessary
757     if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
758     {
759         // Update the in memory version of the attributes
760         SET_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN);
761
762         // If this is not orderly, then update the NV version of
763         // the attributes
764         if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
765         {
766             result = NvWriteNvIndexAttributes(s_cachedNvRef,
767                                               nvIndex->publicArea.attributes);
768             if(result != TPM_RC_SUCCESS)
769                 return result;
770             // If this is a partial write of an ordinary index, clear the whole
771             // index.
772             if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes)
773                 && (nvIndex->publicArea.dataSize > size))
774                 _plat__NvMemoryClear(s_cachedNvRef + sizeof(NV_INDEX),
775                                     nvIndex->publicArea.dataSize);
776         }
777     }
778     else
779     {
780         // This is orderly so update the RAM version
781         MemoryCopy(s_cachedNvRamRef + offsetof(NV_RAM_HEADER, attributes),
782                   &nvIndex->publicArea.attributes, sizeof(TPMA_NV));
783         // If setting WRITTEN for an orderly counter, make sure that the
784         // state saved version of the counter is saved
785         if(IsNvCounterIndex(nvIndex->publicArea.attributes))
786             SET_NV_UPDATE(UT_ORDERLY);
787         // If setting the written attribute on an ordinary index, make sure that
788         // the data is all cleared out in case there is a partial write. This
789         // is only necessary for ordinary indexes because all of the other types
790         // are always written in total.
791         else if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes))
792             MemorySet(s_cachedNvRamRef + sizeof(NV_RAM_HEADER),

```



```

792         0, nvIndex->publicArea.dataSize);
793     }
794 }
795 // If this is orderly data, write it to RAM
796 if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
797 {
798     // Note: if this is the first write to a counter, the code above will queue
799     // the write to NV of the RAM data in order to update TPMA_NV_WRITTEN. In
800     // process of doing that write, it will also write the initial counter value
801
802     // Update RAM
803     MemoryCopy(s_cachedNvRamRef + sizeof(NV_RAM_HEADER) + offset, data, size);
804
805     // And indicate that the TPM is no longer orderly
806     g_clearOrderly = TRUE;
807 }
808 else
809 {
810     // Offset into the index to the first byte of the data to be written to NV
811     result = NvConditionallyWrite(s_cachedNvRef + sizeof(NV_INDEX) + offset,
812                                 size, data);
813 }
814 return result;
815 }

```

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)

```

816 TPM_RC
817 NvWriteUINT64Data(
818     NV_INDEX      *nvIndex,      // IN: the description of the index
819     UINT64         intValue       // IN: the value to write
820 )
821 {
822     BYTE          bytes[8];
823     UINT64_TO_BYTE_ARRAY(intValue, bytes);
824     //
825     return NvWriteIndexData(nvIndex, 0, 8, &bytes);
826 }

```

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.

```

827 TPM2B_NAME *
828 NvGetIndexName(
829     NV_INDEX      *nvIndex,      // IN: the index over which the name is to be
830                                 // computed
831     TPM2B_NAME     *name         // OUT: name of the index
832 )
833 {
834     UINT16         dataSize, digestSize;
835     BYTE           marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
836     BYTE           *buffer;
837     HASH_STATE     hashState;

```

```

838 //
839 // Marshal public area
840 buffer = marshalBuffer;
841 dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex->publicArea, &buffer, NULL);
842
843 // hash public area
844 digestSize = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
845 CryptDigestUpdate(&hashState, dataSize, marshalBuffer);
846
847 // Complete digest leaving room for the nameAlg
848 CryptHashEnd(&hashState, digestSize, &name->b.buffer[2]);
849
850 // Include the nameAlg
851 UINT16_TO_BYTE_ARRAY(nvIndex->publicArea.nameAlg, name->b.buffer);
852 name->t.size = digestSize + 2;
853 return name;
854 }

```

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.

```

855 TPM2B_NAME *
856 NvGetNameByIndexHandle(
857     TPMI_RH_NV_INDEX    handle,          // IN: handle of the index
858     TPM2B_NAME          *name,           // OUT: name of the index
859 )
860 {
861     NV_INDEX             *nvIndex = NvGetIndexInfo(handle, NULL);
862     //
863     return NvGetIndexName(nvIndex, name);
864 }

```

8.4.5.16 NvDefineIndex()

This function is used to assign NV memory to an NV Index.

Error Returns	Meaning
TPM_RC_NV_SPACE	insufficient NV space

```

865 TPM_RC
866 NvDefineIndex(
867     TPMS_NV_PUBLIC *publicArea, // IN: A template for an area to create.
868     TPM2B_AUTH     *authValue,  // IN: The initial authorization value
869 )
870 {
871     // The buffer to be written to NV memory
872     NV_INDEX     nvIndex;          // the index data
873     UINT16       entrySize;        // size of entry
874     TPM_RC       result;
875     //
876     entrySize = sizeof(NV_INDEX);
877
878     // only allocate data space for indexes that are going to be written to NV.
879     // Orderly indexes don't need space.
880     if(!IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
881         entrySize += publicArea->dataSize;

```

```

882 // Check if we have enough space to create the NV Index
883 // In this implementation, the only resource limitation is the available NV
884 // space (and possibly RAM space.) Other implementation may have other
885 // limitation on counter or on NV slots
886 if(!NvTestSpace(entrySize, TRUE, IsNvCounterIndex(publicArea->attributes)))
887     return TPM_RC_NV_SPACE;
888
889 // if the index to be defined is RAM backed, check RAM space availability
890 // as well
891 if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY)
892    && !NvRamTestSpaceIndex(publicArea->dataSize))
893     return TPM_RC_NV_SPACE;
894 // Copy input value to nvBuffer
895 nvIndex.publicArea = *publicArea;
896
897 // Copy the authValue
898 nvIndex.authValue = *authValue;
899
900 // Add index to NV memory
901 result = NvAdd(entrySize, sizeof(NV_INDEX), TPM_RH_UNASSIGNED,
902               (BYTE *) &nvIndex);
903 if(result == TPM_RC_SUCCESS)
904 {
905     // If the data of NV Index is RAM backed, add the data area in RAM as well
906     if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
907         NvAddRAM(publicArea);
908 }
909 return result;
910 }

```

8.4.5.17 NvAddEvictObject()

This function is used to assign NV memory to a persistent object.

Error Returns	Meaning
TPM_RC_NV_HANDLE	the requested handle is already in use
TPM_RC_NV_SPACE	insufficient NV space

```

911 TPM_RC
912 NvAddEvictObject(
913     TPMI_DH_OBJECT    evictHandle, // IN: new evict handle
914     OBJECT             *object      // IN: object to be added
915 )
916 {
917     TPM_HANDLE    temp = object->evictHandle;
918     TPM_RC        result;
919 //
920 // Check if we have enough space to add the evict object
921 // An evict object needs 8 bytes in index table + sizeof OBJECT
922 // In this implementation, the only resource limitation is the available NV
923 // space. Other implementation may have other limitation on evict object
924 // handle space
925 if(!NvTestSpace(sizeof(OBJECT) + sizeof(TPM_HANDLE), FALSE, FALSE))
926     return TPM_RC_NV_SPACE;
927
928 // Set evict attribute and handle
929 object->attributes.evict = SET;
930 object->evictHandle = evictHandle;
931
932 // Now put this in NV
933 result = NvAdd(sizeof(OBJECT), sizeof(OBJECT), evictHandle, (BYTE *)object);
934

```

```

935     // Put things back the way they were
936     object->attributes.evict = CLEAR;
937     object->evictHandle = temp;
938
939     return result;
940 }

```

8.4.5.18 NvDeleteIndex()

This function is used to delete an NV Index.

Error Returns	Meaning
TPM_RC_NV_UNAVAILABLE	NV is not accessible
TPM_RC_NV_RATE	NV is rate limiting

```

941 TPM_RC
942 NvDeleteIndex(
943     NV_INDEX      *nvIndex,      // IN: an in RAM index descriptor
944     NV_REF        entityAddr     // IN: location in NV
945 )
946 {
947     TPM_RC        result;
948     //
949     if(nvIndex != NULL)
950     {
951         // Whenever a counter is deleted, make sure that the MaxCounter value is
952         // updated to reflect the value
953         if(IsNvCounterIndex(nvIndex->publicArea.attributes)
954             && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
955             NvUpdateMaxCount(NvGetUINT64Data(nvIndex, entityAddr));
956         result = NvDelete(entityAddr);
957         if(result != TPM_RC_SUCCESS)
958             return result;
959         // If the NV Index is RAM backed, delete the RAM data as well
960         if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
961             NvDeleteRAM(nvIndex->publicArea.nvIndex);
962         NvIndexCacheInit();
963     }
964     return TPM_RC_SUCCESS;
965 }

```

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

```

966 TPM_RC
967 NvDeleteEvict(
968     TPM_HANDLE    handle         // IN: handle of entity to be deleted
969 )
970 {
971     NV_REF        entityAddr = NvFindEvict(handle, NULL); // pointer to entity
972     TPM_RC        result = TPM_RC_SUCCESS;
973     //
974     if(entityAddr != 0)
975         result = NvDelete(entityAddr);
976     return result;
977 }

```

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 Returns	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```

978  TPM_RC
979  NvFlushHierarchy(
980      TPMI_RH_HIERARCHY    hierarchy    // IN: hierarchy to be flushed.
981  )
982  {
983      NV_REF                iter = NV_REF_INIT;
984      NV_REF                currentAddr;
985      TPM_HANDLE            entityHandle;
986      TPM_RC                result = TPM_RC_SUCCESS;
987  //
988      while((currentAddr = NvNext(&iter, &entityHandle)) != 0)
989      {
990          if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
991          {
992              NV_INDEX        nvIndex;
993  //
994              // If flush endorsement or platform hierarchy, no NV Index would be
995              // flushed
996              if(hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
997                  continue;
998              // Get the index information
999              NvReadNvIndexInfo(currentAddr, &nvIndex);
1000
1001              // For storage hierarchy, flush OwnerCreated index
1002              if(!IS_ATTRIBUTE(nvIndex.publicArea.attributes, TPMA_NV,
1003                              PLATFORMCREATE))
1004              {
1005                  // Delete the index (including RAM for orderly)
1006                  result = NvDeleteIndex(&nvIndex, currentAddr);
1007                  if(result != TPM_RC_SUCCESS)
1008                      break;
1009                  // Re-iterate from beginning after a delete
1010                  iter = NV_REF_INIT;
1011              }
1012          }
1013          else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
1014          {
1015              OBJECT_ATTRIBUTES    attributes;
1016  //
1017              NvRead(&attributes,
1018                     (UINT32) (currentAddr
1019                               + sizeof(TPM_HANDLE)
1020                               + offsetof(OBJECT, attributes)),
1021                     sizeof(OBJECT_ATTRIBUTES));
1022              // If the evict object belongs to the hierarchy to be flushed...
1023              if((hierarchy == TPM_RH_PLATFORM && attributes.ppsHierarchy == SET)
1024                 || (hierarchy == TPM_RH_OWNER && attributes.spsHierarchy == SET)
1025                 || (hierarchy == TPM_RH_ENDORSEMENT
1026                     && attributes.epsHierarchy == SET))
1027              {
1028                  // ...then delete the evict object
1029                  result = NvDelete(currentAddr);
1030                  if(result != TPM_RC_SUCCESS)
1031                      break;

```

```

1032         // Re-iterate from beginning after a delete
1033         iter = NV_REF_INIT;
1034     }
1035 }
1036 else
1037 {
1038     FAIL(FATAL_ERROR_INTERNAL);
1039 }
1040 }
1041 return result;
1042 }

```

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 Returns	Meaning
TPM_RC_NV_RATE	NV is unavailable because of rate limit
TPM_RC_NV_UNAVAILABLE	NV is inaccessible

```

1043 TPM_RC
1044 NvSetGlobalLock(
1045     void
1046 )
1047 {
1048     NV_REF          iter = NV_REF_INIT;
1049     NV_RAM_REF      ramIter = NV_RAM_REF_INIT;
1050     NV_REF          currentAddr;
1051     NV_RAM_REF      currentRamAddr;
1052     TPM_RC          result = TPM_RC_SUCCESS;
1053 //
1054 // Check all normal indexes
1055 while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1056 {
1057     TPMA_NV          attributes = NvReadNvIndexAttributes(currentAddr);
1058 //
1059 // See if it should be locked
1060 if(!IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1061    && IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1062 {
1063     SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1064     result = NvWriteNvIndexAttributes(currentAddr, attributes);
1065     if(result != TPM_RC_SUCCESS)
1066         return result;
1067 }
1068 }
1069 // Now search all the orderly attributes
1070 while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1071 {
1072     // See if it should be locked
1073     TPMA_NV          attributes = NvReadRamIndexAttributes(currentRamAddr);
1074     if(IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
1075     {
1076         SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1077         NvWriteRamIndexAttributes(currentRamAddr, attributes);
1078     }
1079 }
1080 return result;
1081 }

```

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

```

1082 static void
1083 InsertSort(
1084     TPML_HANDLE *handleList,    // IN/OUT: sorted handle list
1085     UINT32 count,              // IN: maximum count in the handle list
1086     TPM_HANDLE entityHandle    // IN: handle to be inserted
1087 )
1088 {
1089     UINT32 i, j;
1090     UINT32 originalCount;
1091 //
1092 // For a corner case that the maximum count is 0, do nothing
1093 if(count == 0)
1094     return;
1095 // For empty list, add the handle at the beginning and return
1096 if(handleList->count == 0)
1097 {
1098     handleList->handle[0] = entityHandle;
1099     handleList->count++;
1100     return;
1101 }
1102 // Check if the maximum of the list has been reached
1103 originalCount = handleList->count;
1104 if(originalCount < count)
1105     handleList->count++;
1106 // Insert the handle to the list
1107 for(i = 0; i < originalCount; i++)
1108 {
1109     if(handleList->handle[i] > entityHandle)
1110     {
1111         for(j = handleList->count - 1; j > i; j--)
1112         {
1113             handleList->handle[j] = handleList->handle[j - 1];
1114         }
1115         break;
1116     }
1117 }
1118 // If a slot was found, insert the handle in this position
1119 if(i < originalCount || handleList->count > originalCount)
1120     handleList->handle[i] = entityHandle;
1121 return;
1122 }

```

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

```

1123 TPMI_YES_NO
1124 NvCapGetPersistent(
1125     TPMI_DH_OBJECT handle,    // IN: start handle
1126     UINT32 count,            // IN: maximum number of returned handles

```



```

1127     TPML_HANDLE      *handleList      // OUT: list of handle
1128     )
1129 {
1130     TPMI_YES_NO        more = NO;
1131     NV_REF             iter = NV_REF_INIT;
1132     NV_REF             currentAddr;
1133     TPM_HANDLE         entityHandle;
1134     //
1135     pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
1136
1137     // Initialize output handle list
1138     handleList->count = 0;
1139
1140     // The maximum count of handles we may return is MAX_CAP_HANDLES
1141     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1142
1143     while((currentAddr = NvNextEvict(&entityHandle, &iter)) != 0)
1144     {
1145         // Ignore persistent handles that have values less than the input handle
1146         if(entityHandle < handle)
1147             continue;
1148         // if the handles in the list have reached the requested count, and there
1149         // are still handles need to be inserted, indicate that there are more.
1150         if(handleList->count == count)
1151             more = YES;
1152         // A handle with a value larger than start handle is a candidate
1153         // for return. Insert sort it to the return list. Insert sort algorithm
1154         // is chosen here for simplicity based on the assumption that the total
1155         // number of NV indexes is small. For an implementation that may allow
1156         // large number of NV indexes, a more efficient sorting algorithm may be
1157         // used here.
1158         InsertSort(handleList, count, entityHandle);
1159     }
1160     return more;
1161 }

```

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

```

1162     TPMI_YES_NO
1163     NvCapGetIndex(
1164         TPMI_DH_OBJECT    handle,      // IN: start handle
1165         UINT32            count,      // IN: max number of returned handles
1166         TPML_HANDLE      *handleList  // OUT: list of handle
1167     )
1168 {
1169     TPMI_YES_NO        more = NO;
1170     NV_REF             iter = NV_REF_INIT;
1171     NV_REF             currentAddr;
1172     TPM_HANDLE         nvHandle;
1173     //
1174     pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
1175
1176     // Initialize output handle list
1177     handleList->count = 0;
1178

```

```

1179 // The maximum count of handles we may return is MAX_CAP_HANDLES
1180 if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1181
1182 while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1183 {
1184     // Ignore index handles that have values less than the 'handle'
1185     if(nvHandle < handle)
1186         continue;
1187     // if the count of handles in the list has reached the requested count,
1188     // and there are still handles to report, set more.
1189     if(handleList->count == count)
1190         more = YES;
1191     // A handle with a value larger than start handle is a candidate
1192     // for return. Insert sort it to the return list. Insert sort algorithm
1193     // is chosen here for simplicity based on the assumption that the total
1194     // number of NV indexes is small. For an implementation that may allow
1195     // large number of NV indexes, a more efficient sorting algorithm may be
1196     // used here.
1197     InsertSort(handleList, count, nvHandle);
1198 }
1199 return more;
1200 }

```

8.4.5.25 NvCapGetIndexNumber()

This function returns the count of NV Indexes currently defined.

```

1201 UINT32
1202 NvCapGetIndexNumber(
1203     void
1204 )
1205 {
1206     UINT32    num = 0;
1207     NV_REF    iter = NV_REF_INIT;
1208 //
1209     while(NvNextIndex(NULL, &iter) != 0)
1210         num++;
1211     return num;
1212 }

```

8.4.5.26 NvCapGetPersistentNumber()

Function returns the count of persistent objects currently in NV memory.

```

1213 UINT32
1214 NvCapGetPersistentNumber(
1215     void
1216 )
1217 {
1218     UINT32    num = 0;
1219     NV_REF    iter = NV_REF_INIT;
1220     TPM_HANDLE handle;
1221 //
1222     while(NvNextEvict(&handle, &iter) != 0)
1223         num++;
1224     return num;
1225 }

```

8.4.5.27 NvCapGetPersistentAvail()

This function returns an estimate of the number of additional persistent objects that could be loaded into NV memory.

```

1226  UINT32
1227  NvCapGetPersistentAvail(
1228      void
1229  )
1230  {
1231      UINT32      availNVSpace;
1232      UINT32      counterNum = NvCapGetCounterNumber();
1233      UINT32      reserved = sizeof(NV_LIST_TERMINATOR);
1234  //
1235      // Get the available space in NV storage
1236      availNVSpace = NvGetFreeBytes();
1237
1238      if(counterNum < MIN_COUNTER_INDICES)
1239      {
1240          // Some space has to be reserved for counter objects.
1241          reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
1242          if(reserved > availNVSpace)
1243              availNVSpace = 0;
1244          else
1245              availNVSpace -= reserved;
1246      }
1247      return availNVSpace / NV_EVICT_OBJECT_SIZE;
1248  }

```

8.4.5.28 NvCapGetCounterNumber()

Get the number of defined NV Indexes that are counter indexes.

```

1249  UINT32
1250  NvCapGetCounterNumber(
1251      void
1252  )
1253  {
1254      NV_REF      iter = NV_REF_INIT;
1255      NV_REF      currentAddr;
1256      UINT32      num = 0;
1257  //
1258      while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
1259      {
1260          TPMA_NV      attributes = NvReadNvIndexAttributes(currentAddr);
1261          if(IsNvCounterIndex(attributes))
1262              num++;
1263      }
1264      return num;
1265  }

```

8.4.5.29 NvSetStartupAttributes()

Local function to set the attributes of an Index at TPM Reset and TPM Restart.

```

1266  static TPMA_NV
1267  NvSetStartupAttributes(
1268      TPMA_NV      attributes,          // IN: attributes to change
1269      STARTUP_TYPE  type                // IN: start up type
1270  )
1271  {
1272      // Clear read lock

```

```

1273     CLEAR_ATTRIBUTE(attributes, TPMA_NV, READLOCKED);
1274
1275     // Will change a non counter index to the unwritten state if:
1276     // a) TPMA_NV_CLEAR_STCLEAR is SET
1277     // b) orderly and TPM Reset
1278     if(!IsNvCounterIndex(attributes))
1279     {
1280         if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
1281            || (IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
1282                && (type == SU_RESET)))
1283             CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITTEN);
1284     }
1285     // Unlock any index that is not written or that does not have
1286     // TPMA_NV_WRITEDEFINE SET.
1287     if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
1288        || !IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
1289         CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
1290     return attributes;
1291 }

```

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.

```

1292 BOOL
1293 NvEntityStartup(
1294     STARTUP_TYPE    type           // IN: start up type
1295 )
1296 {
1297     NV_REF            iter = NV_REF_INIT;
1298     NV_RAM_REF        ramIter = NV_RAM_REF_INIT;
1299     NV_REF            currentAddr; // offset points to the current entity
1300     NV_RAM_REF        currentRamAddr;
1301     TPM_HANDLE        nvHandle;
1302     TPMA_NV           attributes;
1303 //
1304     // Restore RAM index data
1305     NvRead(s_indexOrderlyRam, NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam));
1306
1307     // Initialize the max NV counter value
1308     NvSetMaxCount(NvGetMaxCount());
1309
1310     // If recovering from state save, do nothing else
1311     if(type == SU_RESUME)
1312         return TRUE;
1313     // Iterate all the NV Index to clear the locks
1314     while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
1315     {
1316         attributes = NvReadNvIndexAttributes(currentAddr);
1317
1318         // If this is an orderly index, defer processing until loop below
1319         if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
1320             continue;
1321         // Set the attributes appropriate for this startup type
1322         attributes = NvSetStartupAttributes(attributes, type);
1323         NvWriteNvIndexAttributes(currentAddr, attributes);

```

```

1324     }
1325     // Iterate all the orderly indexes to clear the locks and initialize counters
1326     while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1327     {
1328         attributes = NvReadRamIndexAttributes(currentRamAddr);
1329
1330         attributes = NvSetStartupAttributes(attributes, type);
1331
1332         // update attributes in RAM
1333         NvWriteRamIndexAttributes(currentRamAddr, attributes);
1334
1335         // Set the lower bits in an orderly counter to 1 for a non-orderly startup
1336         if(IsNvCounterIndex(attributes)
1337             && (g_prevOrderlyState == SU_NONE_VALUE))
1338         {
1339             UINT64        counter;
1340
1341             // Read the counter value last saved to NV.
1342             counter = BYTE_ARRAY_TO_UINT64(currentRamAddr + sizeof(NV_RAM_HEADER));
1343
1344             // Set the lower bits of counter to 1's
1345             counter |= MAX_ORDERLY_COUNT;
1346
1347             // Write back to RAM
1348             // NOTE: Do not want to force a write to NV here. The counter value will
1349             // stay in RAM until the next shutdown or rollover.
1350             UINT64_TO_BYTE_ARRAY(counter, currentRamAddr + sizeof(NV_RAM_HEADER));
1351         }
1352     }
1353     return TRUE;
1354 }

```

8.4.5.31 NvCapGetCounterAvail()

This function returns an estimate of the number of additional counter type NV indexes that can be defined.

```

1355     UINT32
1356     NvCapGetCounterAvail(
1357         void
1358     )
1359     {
1360         UINT32        availNVSpace;
1361         UINT32        availRAMSpace;
1362         UINT32        persistentNum = NvCapGetPersistentNumber();
1363         UINT32        reserved = sizeof(NV_LIST_TERMINATOR);
1364
1365         // Get the available space in NV storage
1366         availNVSpace = NvGetFreeBytes();
1367
1368         if(persistentNum < MIN_EVICT_OBJECTS)
1369         {
1370             // Some space has to be reserved for evict object. Adjust availNVSpace.
1371             reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
1372             if(reserved > availNVSpace)
1373                 availNVSpace = 0;
1374             else
1375                 availNVSpace -= reserved;
1376         }
1377         // Compute the available space in RAM
1378         availRAMSpace = RAM_ORDERLY_END - NvRamGetEnd();
1379
1380         // Return the min of counter number in NV and in RAM
1381         if(availNVSpace / NV_INDEX_COUNTER_SIZE

```

```

1382     > availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE)
1383     return availRAMSpace / NV_RAM_INDEX_COUNTER_SIZE;
1384 else
1385     return availNVSpace / NV_INDEX_COUNTER_SIZE;
1386 }

```

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.

```

1387 NV_REF
1388 NvFindHandle(
1389     TPM_HANDLE    handle
1390 )
1391 {
1392     NV_REF        addr;
1393     NV_REF        iter = NV_REF_INIT;
1394     TPM_HANDLE    nextHandle;
1395 //
1396 while((addr = NvNext(&iter, &nextHandle)) != 0)
1397 {
1398     if(nextHandle == handle)
1399         break;
1400 }
1401 return addr;
1402 }

```

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 keep 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.

```

1403 UINT64
1404 NvReadMaxCount(
1405     void
1406 )
1407 {
1408     return s_maxCounter;
1409 }

```

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.

```

1410 void
1411 NvUpdateMaxCount(
1412     UINT64          count
1413 )
1414 {
1415     if(count > s_maxCounter)
1416         s_maxCounter = count;
1417 }

```

8.4.6.4 NvSetMaxCount()

This function is used at NV initialization time to set the initial value of the maximum counter.

```

1418 void
1419 NvSetMaxCount(
1420     UINT64          value
1421 )
1422 {
1423     s_maxCounter = value;
1424 }

```

8.4.6.5 NvGetMaxCount()

Function to get the NV max counter value from the end-of-list marker

```

1425 UINT64
1426 NvGetMaxCount(
1427     void
1428 )
1429 {
1430     NV_REF          iter = NV_REF_INIT;
1431     NV_REF          currentAddr;
1432     UINT64          maxCount;
1433     //
1434     // Find the end of list marker and initialize the NV Max Counter value.
1435     while((currentAddr = NvNext(&iter, NULL )) != 0);
1436     // 'iter' should be pointing at the end of list marker so read in the current
1437     // value of the s_maxCounter.
1438     NvRead(&maxCount, iter + sizeof(UINT32), sizeof(maxCount));
1439
1440     return maxCount;
1441 }

```


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 PERSISTEND_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     }
```

```

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  #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.

```
3  void
4  ObjectFlush(
5      OBJECT          *object
6  )
7  {
8      object->attributes.occupied = CLEAR;
9  }
```

8.6.3.2 ObjectSetInUse()

This access function sets the occupied attribute of an object slot.

```
10 void
11 ObjectSetInUse(
12     OBJECT          *object
13 )
14 {
15     object->attributes.occupied = SET;
16 }
```

8.6.3.3 ObjectStartup()

This function is called at TPM2_Startup() to initialize the object subsystem.

```
17 BOOL
18 ObjectStartup(
19     void
20 )
21 {
22     UINT32      i;
23     //
24     // object slots initialization
25     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
26     {
27         //Set the slot to not occupied
28         ObjectFlush(&s_objects[i]);
29     }
```

```

30     return TRUE;
31 }

```

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().

```

32 void
33 ObjectCleanupEvict(
34     void
35 )
36 {
37     UINT32    i;
38     //
39     // This has to be iterated because a command may have two handles
40     // and they may both be persistent.
41     // This could be made to be more efficient so that a search is not needed.
42     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
43     {
44         // If an object is a temporary evict object, flush it from slot
45         OBJECT    *object = &s_objects[i];
46         if(object->attributes.evict == SET)
47             ObjectFlush(object);
48     }
49     return;
50 }

```

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

```

51 BOOL
52 IsObjectPresent(
53     TPMI_DH_OBJECT    handle        // IN: handle to be checked
54 )
55 {
56     UINT32    slotIndex = handle - TRANSIENT_FIRST;
57     // Since the handle is just an index into the array that is zero based, any
58     // handle value outside of the range of:
59     // TRANSIENT_FIRST -- (TRANSIENT_FIRST + MAX_LOADED_OBJECT - 1)
60     // will now be greater than or equal to MAX_LOADED_OBJECTS
61     if(slotIndex >= MAX_LOADED_OBJECTS)
62         return FALSE;
63     // Indicate if the slot is occupied
64     return (s_objects[slotIndex].attributes.occupied == TRUE);
65 }

```

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

```

66  BOOL
67  ObjectIsSequence(
68      OBJECT          *object          // IN: handle to be checked
69  )
70  {
71      pAssert(object != NULL);
72      return (object->attributes.hmacSeq == SET
73              || object->attributes.hashSeq == SET
74              || object->attributes.eventSeq == SET);
75  }

```

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.

```

76  OBJECT*
77  HandleToObject(
78      TPMI_DH_OBJECT  handle          // IN: handle of the object
79  )
80  {
81      UINT32          index;
82      //
83      // Return NULL if the handle references a permanent handle because there is no
84      // associated OBJECT.
85      if(HandleGetType(handle) == TPM_HT_PERMANENT)
86          return NULL;
87      // In this implementation, the handle is determined by the slot occupied by the
88      // object.
89      index = handle - TRANSIENT_FIRST;
90      pAssert(index < MAX_LOADED_OBJECTS);
91      pAssert(s_objects[index].attributes.occupied);
92      return &s_objects[index];
93  }

```

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.

```

94  void
95  GetQualifiedName(
96      TPMI_DH_OBJECT  handle,          // IN: handle of the object
97      TPM2B_NAME      *qualifiedName  // OUT: qualified name of the object
98  )
99  {
100     OBJECT          *object;
101     //
102     switch(HandleGetType(handle))
103     {
104         case TPM_HT_PERMANENT:

```



```

105         qualifiedName->t.size = sizeof(TPM_HANDLE);
106         UINT32_TO_BYTE_ARRAY(handle, qualifiedName->t.name);
107         break;
108     case TPM_HT_TRANSIENT:
109         object = HandleToObject(handle);
110         if(object == NULL || object->publicArea.nameAlg == TPM_ALG_NULL)
111             qualifiedName->t.size = 0;
112         else
113             // Copy the name
114             *qualifiedName = object->qualifiedName;
115         break;
116     default:
117         FAIL(FATAL_ERROR_INTERNAL);
118     }
119     return;
120 }

```

8.6.3.9 ObjectGetHierarchy()

This function returns the handle for the hierarchy of an object.

```

121 TPMI_RH_HIERARCHY
122 ObjectGetHierarchy(
123     OBJECT          *object          // IN :object
124 )
125 {
126     if(object->attributes.spsHierarchy)
127     {
128         return TPM_RH_OWNER;
129     }
130     else if(object->attributes.epsHierarchy)
131     {
132         return TPM_RH_ENDORSEMENT;
133     }
134     else if(object->attributes.ppsHierarchy)
135     {
136         return TPM_RH_PLATFORM;
137     }
138     else
139     {
140         return TPM_RH_NULL;
141     }
142 }

```

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.

```

143 TPMI_RH_HIERARCHY
144 GetHierarchy(
145     TPMI_DH_OBJECT   handle          // IN :object handle
146 )
147 {
148     OBJECT          *object = HandleToObject(handle);
149     //
150     return ObjectGetHierarchy(object);
151 }

```

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

```

152  OBJECT *
153  FindEmptyObjectSlot(
154      TPMI_DH_OBJECT  *handle          // OUT: (optional)
155  )
156  {
157      UINT32          i;
158      OBJECT          *object;
159      //
160      for(i = 0; i < MAX_LOADED_OBJECTS; i++)
161      {
162          object = &s_objects[i];
163          if(object->attributes.occupied == CLEAR)
164          {
165              if(handle)
166                  *handle = i + TRANSIENT_FIRST;
167              // Initialize the object attributes
168              MemorySet(&object->attributes, 0, sizeof(OBJECT_ATTRIBUTES));
169              return object;
170          }
171      }
172      return NULL;
173  }

```

8.6.3.12 ObjectAllocateSlot()

This function is used to allocate a slot in internal object array.

```

174  OBJECT *
175  ObjectAllocateSlot(
176      TPMI_DH_OBJECT  *handle          // OUT: handle of allocated object
177  )
178  {
179      OBJECT          *object = FindEmptyObjectSlot(handle);
180      //
181      if(object != NULL)
182      {
183          // if found, mark as occupied
184          ObjectSetInUse(object);
185      }
186      return object;
187  }

```

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.

```

188  void
189  ObjectSetLoadedAttributes(
190      OBJECT          *object,          // IN: object attributes to finalize
191      TPM_HANDLE      parentHandle      // IN: the parent handle

```

```

192     )
193 {
194     OBJECT                *parent = HandleToObject(parentHandle);
195     TPMA_OBJECT            objectAttributes = object->publicArea.objectAttributes;
196     //
197     // Copy the stClear attribute from the public area. This could be overwritten
198     // if the parent has stClear SET
199     object->attributes.stClear =
200         IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear);
201     // If parent handle is a permanent handle, it is a primary (unless it is NULL
202     if(parent == NULL)
203     {
204         object->attributes.primary = SET;
205         switch(parentHandle)
206         {
207             case TPM_RH_ENDORSEMENT:
208                 object->attributes.epsHierarchy = SET;
209                 break;
210             case TPM_RH_OWNER:
211                 object->attributes.spsHierarchy = SET;
212                 break;
213             case TPM_RH_PLATFORM:
214                 object->attributes.ppsHierarchy = SET;
215                 break;
216             default:
217                 // Treat the temporary attribute as a hierarchy
218                 object->attributes.temporary = SET;
219                 object->attributes.primary = CLEAR;
220                 break;
221         }
222     }
223     else
224     {
225         // is this a stClear object
226         object->attributes.stClear =
227             (IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear)
228              || (parent->attributes.stClear == SET));
229         object->attributes.epsHierarchy = parent->attributes.epsHierarchy;
230         object->attributes.spsHierarchy = parent->attributes.spsHierarchy;
231         object->attributes.ppsHierarchy = parent->attributes.ppsHierarchy;
232         // An object is temporary if its parent is temporary or if the object
233         // is external
234         object->attributes.temporary = parent->attributes.temporary
235             || object->attributes.external;
236     }
237     // If this is an external object, set the QN == name but don't SET other
238     // key properties ('parent' or 'derived')
239     if(object->attributes.external)
240         object->qualifiedName = object->name;
241     else
242     {
243         // check attributes for different types of parents
244         if(IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, restricted)
245            && !object->attributes.publicOnly
246            && IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, decrypt)
247            && object->publicArea.nameAlg != TPM_ALG_NULL)
248         {
249             // This is a parent. If it is not a KEYEDHASH, it is an ordinary parent.
250             // Otherwise, it is a derivation parent.
251             if(object->publicArea.type == TPM_ALG_KEYEDHASH)
252                 object->attributes.derivation = SET;
253             else
254                 object->attributes.isParent = SET;
255         }
256         ComputeQualifiedName(parentHandle, object->publicArea.nameAlg,
257                               &object->name, &object->qualifiedName);

```

```

258     }
259     // Set slot occupied
260     ObjectSetInUse(object);
261     return;
262 }

```

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.

```

263 TPM_RC
264 ObjectLoad(
265     OBJECT      *object,          // IN: pointer to object slot
266                                     // object
267     OBJECT      *parent,          // IN: (optional) the parent object
268     TPMT_PUBLIC *publicArea,      // IN: public area to be installed in the object
269     TPMT_SENSITIVE *sensitive,    // IN: (optional) sensitive area to be
270                                     // installed in the object
271     TPM_RC      blamePublic,      // IN: parameter number to associate with the
272                                     // publicArea errors
273     TPM_RC      blameSensitive,   // IN: parameter number to associate with the
274                                     // sensitive area errors
275     TPM2B_NAME  *name             // IN: (optional)
276 )
277 {
278     TPM_RC      result = TPM_RC_SUCCESS;
279     //
280     // Do validations of public area object descriptions
281     pAssert(publicArea != NULL);
282
283     // Is this public only or a no-name object?
284     if(sensitive == NULL || publicArea->nameAlg == TPM_ALG_NULL)
285     {
286         // Need to have schemes checked so that we do the right thing with the
287         // public key.
288         result = SchemeChecks(NULL, publicArea);
289     }
290     else
291     {
292         // For any sensitive area, make sure that the seedSize is no larger than the
293         // digest size of nameAlg
294         if(sensitive->seedValue.t.size > CryptHashGetDigestSize(publicArea->nameAlg))
295             return TPM_RCS_KEY_SIZE + blameSensitive;
296         // Check attributes and schemes for consistency
297         result = PublicAttributesValidation(parent, publicArea);
298     }
299     if(result != TPM_RC_SUCCESS)
300         return RcSafeAddToResult(result, blamePublic);
301
302     // Sensitive area and binding checks
303
304     // On load, check nothing if the parent is fixedTPM. For all other cases, validate
305     // the keys.
306     if((parent == NULL)
307        || ((parent != NULL) && !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
308                                                TPMA_OBJECT, fixedTPM)))
309     {
310         // Do the cryptographic key validation
311         result = CryptValidateKeys(publicArea, sensitive, blamePublic,
312                                    blameSensitive);
313         if(result != TPM_RC_SUCCESS)

```

```

314         return result;
315     }
316 #if ALG_RSA
317     // If this is an RSA key, then expand the private exponent.
318     // Note: ObjectLoad() is only called by TPM2_Import() if the parent is fixedTPM.
319     // For any key that does not have a fixedTPM parent, the exponent is computed
320     // whenever it is loaded
321     if((publicArea->type == TPM_ALG_RSA) && (sensitive != NULL))
322     {
323         result = CryptRsaLoadPrivateExponent(publicArea, sensitive);
324         if(result != TPM_RC_SUCCESS)
325             return result;
326     }
327 #endif // ALG_RSA
328     // See if there is an object to populate
329     if((result == TPM_RC_SUCCESS) && (object != NULL))
330     {
331         // Initialize public
332         object->publicArea = *publicArea;
333         // Copy sensitive if there is one
334         if(sensitive == NULL)
335             object->attributes.publicOnly = SET;
336         else
337             object->sensitive = *sensitive;
338         // Set the name, if one was provided
339         if(name != NULL)
340             object->name = *name;
341         else
342             object->name.t.size = 0;
343     }
344     return result;
345 }

```

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.

```

346 static HASH_OBJECT *
347 AllocateSequenceSlot(
348     TPM_HANDLE      *newHandle,      // OUT: receives the allocated handle
349     TPM2B_AUTH      *auth           // IN: the authValue for the slot
350 )
351 {
352     HASH_OBJECT      *object = (HASH_OBJECT *)ObjectAllocatesSlot(newHandle);
353     //
354     // Validate that the proper location of the hash state data relative to the
355     // object state data. It would be good if this could have been done at compile
356     // time but it can't so do it in something that can be removed after debug.
357     cAssert(offsetof(HASH_OBJECT, auth) == offsetof(OBJECT, publicArea.authPolicy));
358
359     if(object != NULL)
360     {
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         // This has no name algorithm and the name is the Empty Buffer
370         object->nameAlg = TPM_ALG_NULL;
371     }

```

```

372     // A sequence object is considered to be in the NULL hierarchy so it should
373     // be marked as temporary so that it can't be persisted
374     object->attributes.temporary = SET;
375
376     // A sequence object is DA exempt.
377     SET_ATTRIBUTE(object->objectAttributes, TPMA_OBJECT, noDA);
378
379     // Copy the authorization value
380     if(auth != NULL)
381         object->auth = *auth;
382     else
383         object->auth.t.size = 0;
384 }
385 return object;
386 }
387 #if CC_HMAC_Start || CC_MAC_Start

```

8.6.3.16 ObjectCreateHMACSequence()

This function creates an internal HMAC sequence object.

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```

388 TPM_RC
389 ObjectCreateHMACSequence(
390     TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm
391     OBJECT            *keyObject,     // IN: the object containing the HMAC key
392     TPM2B_AUTH        *auth,         // IN: authValue
393     TPMI_DH_OBJECT    *newHandle     // OUT: HMAC sequence object handle
394 )
395 {
396     HASH_OBJECT        *hmacObject;
397     //
398     // Try to allocate a slot for new object
399     hmacObject = AllocateSequenceSlot(newHandle, auth);
400
401     if(hmacObject == NULL)
402         return TPM_RC_OBJECT_MEMORY;
403     // Set HMAC sequence bit
404     hmacObject->attributes.hmacSeq = SET;
405
406     #if !SMAC_IMPLEMENTED
407     if(CryptHmacStart(&hmacObject->state.hmacState, hashAlg,
408                     keyObject->sensitive.sensitive.bits.b.size,
409                     keyObject->sensitive.sensitive.bits.b.buffer) == 0)
410     #else
411     if(CryptMacStart(&hmacObject->state.hmacState,
412                    &keyObject->publicArea.parameters,
413                    hashAlg, &keyObject->sensitive.sensitive.any.b) == 0)
414     #endif // SMAC_IMPLEMENTED
415         return TPM_RC_FAILURE;
416     return TPM_RC_SUCCESS;
417 }
418 #endif

```

8.6.3.17 ObjectCreateHashSequence()

This function creates a hash sequence object.

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```

419 TPM_RC
420 ObjectCreateHashSequence(
421     TPMI_ALG_HASH    hashAlg,          // IN: hash algorithm
422     TPM2B_AUTH        *auth,           // IN: authValue
423     TPMI_DH_OBJECT    *newHandle       // OUT: sequence object handle
424 )
425 {
426     HASH_OBJECT        *hashObject = AllocateSequenceSlot(newHandle, auth);
427     //
428     // See if slot allocated
429     if(hashObject == NULL)
430         return TPM_RC_OBJECT_MEMORY;
431     // Set hash sequence bit
432     hashObject->attributes.hashSeq = SET;
433
434     // Start hash for hash sequence
435     CryptHashStart(&hashObject->state.hashState[0], hashAlg);
436
437     return TPM_RC_SUCCESS;
438 }

```

8.6.3.18 ObjectCreateEventSequence()

This function creates an event sequence object.

Error Returns	Meaning
TPM_RC_OBJECT_MEMORY	if there is no free slot for an object

```

439 TPM_RC
440 ObjectCreateEventSequence(
441     TPM2B_AUTH        *auth,           // IN: authValue
442     TPMI_DH_OBJECT    *newHandle       // OUT: sequence object handle
443 )
444 {
445     HASH_OBJECT        *hashObject = AllocateSequenceSlot(newHandle, auth);
446     UINT32              count;
447     TPM_ALG_ID          hash;
448     //
449     // See if slot allocated
450     if(hashObject == NULL)
451         return TPM_RC_OBJECT_MEMORY;
452     // Set the event sequence attribute
453     hashObject->attributes.eventSeq = SET;
454
455     // Initialize hash states for each implemented PCR algorithms
456     for(count = 0; (hash = CryptHashGetAlgByIndex(count)) != TPM_ALG_NULL; count++)
457         CryptHashStart(&hashObject->state.hashState[count], hash);
458     return TPM_RC_SUCCESS;
459 }

```

8.6.3.19 ObjectTerminateEvent()

This function is called to close out the event sequence and clean up the hash context states.

```

460 void
461 ObjectTerminateEvent(
462     void

```



```

463     )
464 {
465     HASH_OBJECT      *hashObject;
466     int              count;
467     BYTE             buffer[MAX_DIGEST_SIZE];
468     //
469     hashObject = (HASH_OBJECT *)HandleToObject(g_DRTMHandle);
470
471     // Don't assume that this is a proper sequence object
472     if(hashObject->attributes.eventSeq)
473     {
474         // If it is, close any open hash contexts. This is done in case
475         // the cryptographic implementation has some context values that need to be
476         // cleaned up (hygiene).
477         //
478         for(count = 0; CryptHashGetAlgByIndex(count) != TPM_ALG_NULL; count++)
479         {
480             CryptHashEnd(&hashObject->state.hashState[count], 0, buffer);
481         }
482         // Flush sequence object
483         FlushObject(g_DRTMHandle);
484     }
485     g_DRTMHandle = TPM_RH_UNASSIGNED;
486 }

```

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

```

487 OBJECT *
488 ObjectContextLoad(
489     ANY_OBJECT_BUFFER *object,           // IN: pointer to object structure in saved
490                                           // context
491     TPMI_DH_OBJECT    *handle           // OUT: object handle
492 )
493 {
494     OBJECT      *newObject = ObjectAllocatesSlot(handle);
495     //
496     // Try to allocate a slot for new object
497     if(newObject != NULL)
498     {
499         // Copy the first part of the object
500         MemoryCopy(newObject, object, offsetof(HASH_OBJECT, state));
501         // See if this is a sequence object
502         if(ObjectIsSequence(newObject))
503         {
504             // If this is a sequence object, import the data
505             SequenceDataImport((HASH_OBJECT *)newObject,
506                               (HASH_OBJECT_BUFFER *)object);
507         }
508         else
509         {
510             // Copy input object data to internal structure
511             MemoryCopy(newObject, object, sizeof(OBJECT));
512         }
513     }
514     return newObject;
515 }

```

8.6.3.21 FlushObject()

This function frees an object slot.

This function requires that the object is loaded.

```

516 void
517 FlushObject(
518     TPMI_DH_OBJECT    handle          // IN: handle to be freed
519 )
520 {
521     UINT32    index = handle - TRANSIENT_FIRST;
522     //
523     pAssert(index < MAX_LOADED_OBJECTS);
524     // Clear all the object attributes
525     MemorySet((BYTE*)&(s_objects[index].attributes),
526              0, sizeof(OBJECT_ATTRIBUTES));
527     return;
528 }

```

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.

```

529 void
530 ObjectFlushHierarchy(
531     TPMI_RH_HIERARCHY    hierarchy    // IN: hierarchy to be flush
532 )
533 {
534     UINT16    i;
535     //
536     // iterate object slots
537     for(i = 0; i < MAX_LOADED_OBJECTS; i++)
538     {
539         if(s_objects[i].attributes.occupied)    // If found an occupied slot
540         {
541             switch(hierarchy)
542             {
543                 case TPM_RH_PLATFORM:
544                     if(s_objects[i].attributes.ppsHierarchy == SET)
545                         s_objects[i].attributes.occupied = FALSE;
546                     break;
547                 case TPM_RH_OWNER:
548                     if(s_objects[i].attributes.spsHierarchy == SET)
549                         s_objects[i].attributes.occupied = FALSE;
550                     break;
551                 case TPM_RH_ENDORSEMENT:
552                     if(s_objects[i].attributes.epsHierarchy == SET)
553                         s_objects[i].attributes.occupied = FALSE;
554                     break;
555                 default:
556                     FAIL(FATAL_ERROR_INTERNAL);
557                     break;
558             }
559         }
560     }
561     return;
562 }
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 Returns	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 TPM2B_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 == ALG_RSA_VALUE
705                  || object->publicArea.type == ALG_ECC_VALUE));
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
750     return more;
751 }

```

8.6.3.29 ObjectCapGetTransientAvail()

This function returns an estimate of the number of additional transient objects that could be loaded into the TPM.

```

752 UINT32
753 ObjectCapGetTransientAvail(
754     void
755 )
756 {
757     UINT32    i;
758     UINT32    num = 0;

```

```
759 //
760 // Iterate object slot to get the number of unoccupied slots
761 for(i = 0; i < MAX_LOADED_OBJECTS; i++)
762 {
763     if(s_objects[i].attributes.occupied == FALSE) num++;
764 }
765
766 return num;
767 }
```

8.6.3.30 ObjectGetPublicAttributes()

Returns the attributes associated with an object handles.

```
768 TPMA_OBJECT
769 ObjectGetPublicAttributes(
770     TPM_HANDLE      handle
771 )
772 {
773     return HandleToObject(handle)->publicArea.objectAttributes;
774 }
775 OBJECT_ATTRIBUTES
776 ObjectGetProperties(
777     TPM_HANDLE      handle
778 )
779 {
780     return HandleToObject(handle)->attributes;
781 }
```


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 }
45

```

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 a policy group
FALSE(0)	PCR does not belong a policy group

```

46     BOOL
47     PCRBelongsPolicyGroup(
48         TPMI_DH_PCR    handle,           // IN: handle of PCR
49         UINT32         *groupIndex      // OUT: group index if PCR belongs a group that
50                                         // allows policy. If PCR does not belong to
51                                         // a policy group, the value in this
52                                         // parameter is invalid
53     )
54 {
55     #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
56         // Platform specification decides if a PCR belongs to a policy group and
57         // belongs to which group. In this implementation, we assume there is only
58         // one policy group which contains PCR20-22. If the platform specification
59         // requires differently, the implementation should be changed accordingly
60         if(handle >= 20 && handle <= 22)
61         {
62             *groupIndex = 0;
63             return TRUE;
64         }
65     #endif
66     return FALSE;
67 }

```

8.7.3.3 PCRBelongsTCBGroup()

This function indicates if a PCR belongs to the TCB group.

Return Value	Meaning
TRUE(1)	PCR belongs to TCB group
FALSE(0)	PCR does not belong to TCB group

```

68  static BOOL
69  PCRBelongsTCBGroup(
70      TPMI_DH_PCR    handle          // IN: handle of PCR
71  )
72  {
73      #if ENABLE_PCR_NO_INCREMENT == YES
74          // Platform specification decides if a PCR belongs to a TCB group. In this
75          // implementation, we assume PCR[20-22] belong to TCB group. If the platform
76          // specification requires differently, the implementation should be
77          // changed accordingly
78          if(handle >= 20 && handle <= 22)
79              return TRUE;
80      #endif
81      return FALSE;
82  }
83

```

8.7.3.4 PCRPolicyIsAvailable()

This function indicates if a policy is available for a PCR.

Return Value	Meaning
TRUE(1)	the PCR should be authorized by policy
FALSE(0)	the PCR does not allow policy

```

84  BOOL
85  PCRPolicyIsAvailable(
86      TPMI_DH_PCR    handle          // IN: PCR handle
87  )
88  {
89      UINT32          groupIndex;
90
91      return PCRBelongsPolicyGroup(handle, &groupIndex);
92  }

```

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.

```

93  TPM2B_AUTH *
94  PCRGetAuthValue(
95      TPMI_DH_PCR    handle          // IN: PCR handle
96  )
97  {
98      UINT32          groupIndex;
99
100     if(PCRBelongsAuthGroup(handle, &groupIndex))
101     {
102         return &gc.pcrAuthValues.auth[groupIndex];
103     }
104     else
105     {
106         return NULL;

```

```

107     }
108 }

```

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.

```

109 TPMI_ALG_HASH
110 PCRGetAuthPolicy(
111     TPMI_DH_PCR    handle,           // IN: PCR handle
112     TPM2B_DIGEST   *policy          // OUT: policy of PCR
113 )
114 {
115     UINT32          groupIndex;
116
117     if(PCRBelongsPolicyGroup(handle, &groupIndex))
118     {
119         *policy = gp.pcrPolicies.policy[groupIndex];
120         return gp.pcrPolicies.hashAlg[groupIndex];
121     }
122     else
123     {
124         policy->t.size = 0;
125         return TPM_ALG_NULL;
126     }
127 }

```

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.

```

128 void
129 PCRSimStart(
130     void
131 )
132 {
133     UINT32 i;
134     #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
135     for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
136     {
137         gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
138         gp.pcrPolicies.policy[i].t.size = 0;
139     }
140     #endif
141     #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
142     for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
143     {
144         gp.pcrAuthValues.auth[i].t.size = 0;
145     }
146     #endif
147     // We need to give an initial configuration on allocated PCR before
148     // receiving any TPM2_PCR_Allocate command to change this configuration
149     // When the simulation environment starts, we allocate all the PCRs
150     for(gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
151         gp.pcrAllocated.count++)
152     {
153         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].hash
154             = CryptHashGetAlgByIndex(gp.pcrAllocated.count);
155
156         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].sizeofSelect

```

```

157         = PCR_SELECT_MAX;
158     for(i = 0; i < PCR_SELECT_MAX; i++)
159         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i]
160         = 0xFF;
161     }
162
163     // Store the initial configuration to NV
164     NV_SYNC_PERSISTENT(pcrPolicies);
165     NV_SYNC_PERSISTENT(pcrAllocated);
166
167     return;
168 }

```

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

```

169 static BYTE *
170 GetSavedPcrPointer(
171     TPM_ALG_ID    alg,           // IN: algorithm for bank
172     UINT32         pcrIndex      // IN: PCR index in PCR_SAVE
173 )
174 {
175     BYTE          *retVal;
176     switch(alg)
177     {
178 #if ALG_SHA1
179         case ALG_SHA1_VALUE:
180             retVal = gc.pcrSave.sha1[pcrIndex];
181             break;
182 #endif
183 #if ALG_SHA256
184         case ALG_SHA256_VALUE:
185             retVal = gc.pcrSave.sha256[pcrIndex];
186             break;
187 #endif
188 #if ALG_SHA384
189         case ALG_SHA384_VALUE:
190             retVal = gc.pcrSave.sha384[pcrIndex];
191             break;
192 #endif
193
194 #if ALG_SHA512
195         case ALG_SHA512_VALUE:
196             retVal = gc.pcrSave.sha512[pcrIndex];
197             break;
198 #endif
199 #if ALG_SM3_256
200         case ALG_SM3_256_VALUE:
201             retVal = gc.pcrSave.sm3_256[pcrIndex];
202             break;
203 #endif
204         default:
205             FAIL(FATAL_ERROR_INTERNAL);
206     }
207     return retVal;
208 }

```

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

```

209  BOOL
210  PcrIsAllocated(
211      UINT32      pcr,           // IN: The number of the PCR
212      TPMI_ALG_HASH hashAlg     // IN: The PCR algorithm
213  )
214  {
215      UINT32      i;
216      BOOL      allocated = FALSE;
217
218      if(pcr < IMPLEMENTATION_PCR)
219      {
220          for(i = 0; i < gp.pcrAllocated.count; i++)
221          {
222              if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
223              {
224                  if((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr / 8])
225                      & (1 << (pcr % 8))) != 0)
226                      allocated = TRUE;
227                  else
228                      allocated = FALSE;
229                  break;
230              }
231          }
232      }
233      return allocated;
234  }

```

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

```

235  static BYTE *
236  GetPcrPointer(
237      TPM_ALG_ID      alg,           // IN: algorithm for bank
238      UINT32          pcrNumber     // IN: PCR number
239  )
240  {
241      static BYTE      *pcr = NULL;
242
243      if(!PcrIsAllocated(pcrNumber, alg))
244          return NULL;
245
246      switch(alg)
247      {
248      #if ALG_SHA1
249          case ALG_SHA1_VALUE:
250              pcr = s_pcrs[pcrNumber].sha1Pcr;
251              break;

```

```

252 #endif
253 #if ALG_SHA256
254     case ALG_SHA256_VALUE:
255         pcr = s_pcrs[pcrNumber].sha256Pcr;
256         break;
257 #endif
258 #if ALG_SHA384
259     case ALG_SHA384_VALUE:
260         pcr = s_pcrs[pcrNumber].sha384Pcr;
261         break;
262 #endif
263 #if ALG_SHA512
264     case ALG_SHA512_VALUE:
265         pcr = s_pcrs[pcrNumber].sha512Pcr;
266         break;
267 #endif
268 #if ALG_SM3_256
269     case ALG_SM3_256_VALUE:
270         pcr = s_pcrs[pcrNumber].sm3_256Pcr;
271         break;
272 #endif
273     default:
274         FAIL(FATAL_ERROR_INTERNAL);
275         break;
276 }
277 return pcr;
278 }

```

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

```

279 static BOOL
280 IsPcrSelected(
281     UINT32 pcr, // IN: The number of the PCR
282     TPMS_PCR_SELECTION *selection // IN: The selection structure
283 )
284 {
285     BOOL selected;
286     selected = (pcr < IMPLEMENTATION_PCR
287         && ((selection->pcrSelect[pcr / 8]) & (1 << (pcr % 8))) != 0);
288     return selected;
289 }

```

8.7.3.12 FilterPcr()

This function modifies a PCR selection array based on the implemented PCR.

```

290 static void
291 FilterPcr(
292     TPMS_PCR_SELECTION *selection // IN: input PCR selection
293 )
294 {
295     UINT32 i;
296     TPMS_PCR_SELECTION *allocated = NULL;
297
298     // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR

```



```

299     for(i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
300         selection->pcrSelect[i] = 0;
301
302     // Find the internal configuration for the bank
303     for(i = 0; i < gp.pcrAllocated.count; i++)
304     {
305         if(gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
306         {
307             allocated = &gp.pcrAllocated.pcrSelections[i];
308             break;
309         }
310     }
311
312     for(i = 0; i < selection->sizeofSelect; i++)
313     {
314         if(allocated == NULL)
315         {
316             // If the required bank does not exist, clear input selection
317             selection->pcrSelect[i] = 0;
318         }
319         else
320             selection->pcrSelect[i] &= allocated->pcrSelect[i];
321     }
322
323     return;
324 }

```

8.7.3.13 PcrDrtm()

This function does the DRTM and H-CRTM processing it is called from `_TPM_Hash_End()`.

```

325 void
326 PcrDrtm(
327     const TPMI_DH_PCR    pcrHandle,    // IN: the index of the PCR to be
328                                     // modified
329     const TPMI_ALG_HASH  hash,         // IN: the bank identifier
330     const TPM2B_DIGEST   *digest      // IN: the digest to modify the PCR
331 )
332 {
333     BYTE    *pcrData = GetPcrPointer(hash, pcrHandle);
334
335     if(pcrData != NULL)
336     {
337         // Rest the PCR to zeros
338         MemorySet(pcrData, 0, digest->t.size);
339
340         // if the TPM has not started, then set the PCR to 0...04 and then extend
341         if(!TPMIsStarted())
342         {
343             pcrData[digest->t.size - 1] = 4;
344         }
345         // Now, extend the value
346         PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
347     }
348 }

```

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()`.

```

349 void
350 PCR_ClearAuth(

```

```

351     void
352     )
353 {
354 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
355     int j;
356     for(j = 0; j < NUM_AUTHVALUE_PCR_GROUP; j++)
357     {
358         gc.pcrAuthValues.auth[j].t.size = 0;
359     }
360 #endif
361 }

```

8.7.3.15 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```

362 BOOL
363 PCRStartup(
364     STARTUP_TYPE    type,           // IN: startup type
365     BYTE            locality        // IN: startup locality
366 )
367 {
368     UINT32          pcr, j;
369     UINT32          saveIndex = 0;
370
371     g_pcrReConfig = FALSE;
372
373     // Don't test for SU_RESET because that should be the default when nothing
374     // else is selected
375     if(type != SU_RESUME && type != SU_RESTART)
376     {
377         // PCR generation counter is cleared at TPM_RESET
378         gr.pcrCounter = 0;
379     }
380
381     // Initialize/Restore PCR values
382     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
383     {
384         // On resume, need to know if this PCR had its state saved or not
385         UINT32 stateSaved;
386
387         if(type == SU_RESUME
388            && s_initAttributes[pcr].stateSave == SET)
389         {
390             stateSaved = 1;
391         }
392         else
393         {
394             stateSaved = 0;
395             PCRChanged(pcr);
396         }
397
398         // If this is the H-CRTM PCR and we are not doing a resume and we
399         // had an H-CRTM event, then we don't change this PCR
400         if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
401             continue;
402
403         // Iterate each hash algorithm bank
404         for(j = 0; j < gp.pcrAllocated.count; j++)
405         {
406             TPMI_ALG_HASH hash = gp.pcrAllocated.pcrSelections[j].hash;
407             BYTE          *pcrData = GetPcrPointer(hash, pcr);
408             UINT16         pcrSize = CryptHashGetDigestSize(hash);
409

```

```

410     if(pcrData != NULL)
411     {
412         // if state was saved
413         if(stateSaved == 1)
414         {
415             // Restore saved PCR value
416             BYTE *pcrSavedData;
417             pcrSavedData = GetSavedPcrPointer(
418                 gp.pcrAllocated.pcrSelections[j].hash,
419                 saveIndex);
420             if(pcrSavedData == NULL)
421                 return FALSE;
422             MemoryCopy(pcrData, pcrSavedData, pcrSize);
423         }
424         else
425             // PCR was not restored by state save
426         {
427             // If the reset locality of the PCR is 4, then
428             // the reset value is all one's, otherwise it is
429             // all zero.
430             if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
431                 MemorySet(pcrData, 0xFF, pcrSize);
432             else
433             {
434                 MemorySet(pcrData, 0, pcrSize);
435                 if(pcr == HCRTM_PCR)
436                     pcrData[pcrSize - 1] = locality;
437             }
438         }
439     }
440     saveIndex += stateSaved;
441 }
442 // Reset authValues on TPM2_Startup(CLEAR)
443 if(type != SU_RESUME)
444     PCR_ClearAuth();
445 return TRUE;
446 }

```

8.7.3.16 PCRStateSave()

This function is used to save the PCR values that will be restored on TPM Resume.

```

448 void
449 PCRStateSave(
450     TPM_SU type // IN: startup type
451 )
452 {
453     UINT32 pcr, j;
454     UINT32 saveIndex = 0;
455
456     // if state save CLEAR, nothing to be done. Return here
457     if(type == TPM_SU_CLEAR)
458         return;
459
460     // Copy PCR values to the structure that should be saved to NV
461     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
462     {
463         UINT32 stateSaved = (s_initAttributes[pcr].stateSave == SET) ? 1 : 0;
464
465         // Iterate each hash algorithm bank
466         for(j = 0; j < gp.pcrAllocated.count; j++)
467         {
468             BYTE *pcrData;

```

```

469         UINT32  pcrSize;
470
471         pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);
472
473         if(pcrData != NULL)
474         {
475             pcrSize
476                 = CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
477
478             if(stateSaved == 1)
479             {
480                 // Restore saved PCR value
481                 BYTE    *pcrSavedData;
482                 pcrSavedData
483                     = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash,
484                                         saveIndex);
485                 MemoryCopy(pcrSavedData, pcrData, pcrSize);
486             }
487         }
488     }
489     saveIndex += stateSaved;
490 }
491
492 return;
493 }

```

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

```

494 BOOL
495 PCRIsStateSaved(
496     TPMI_DH_PCR    handle           // IN: PCR handle to be extended
497 )
498 {
499     UINT32          pcr = handle - PCR_FIRST;
500
501     if(s_initAttributes[pcr].stateSave == SET)
502         return TRUE;
503     else
504         return FALSE;
505 }

```

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

```

506 BOOL
507 PCRIsResetAllowed(

```

```

508     TPMI_DH_PCR    handle           // IN: PCR handle to be extended
509 )
510 {
511     UINT8           commandLocality;
512     UINT8           localityBits = 1;
513     UINT32          pcr = handle - PCR_FIRST;
514
515     // Check for the locality
516     commandLocality = _plat__LocalityGet();
517
518     #ifndef DRTM_PCR
519     // For a TPM that does DRTM, Reset is not allowed at locality 4
520     if(commandLocality == 4)
521         return FALSE;
522     #endif
523
524     localityBits = localityBits << commandLocality;
525     if((localityBits & s_initAttributes[pcr].resetLocality) == 0)
526         return FALSE;
527     else
528         return TRUE;
529 }

```

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().

```

530 void
531 PCRChanged(
532     TPM_HANDLE      pcrHandle        // IN: the handle of the PCR that changed.
533 )
534 {
535     // For the reference implementation, the only change that does not cause
536     // increment is a change to a PCR in the TCB group.
537     if((pcrHandle == 0) || !PCRBelongsTCBGroup(pcrHandle))
538     {
539         gr.pcrCounter++;
540         if(gr.pcrCounter == 0)
541             FAIL(FATAL_ERROR_COUNTER_OVERFLOW);
542     }
543 }

```

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

```

544 BOOL
545 PCRIsExtendAllowed(
546     TPMI_DH_PCR    handle           // IN: PCR handle to be extended
547 )
548 {
549     UINT8           commandLocality;

```

```

550     UINT8          localityBits = 1;
551     UINT32         pcr = handle - PCR_FIRST;
552
553     // Check for the locality
554     commandLocality = _plat__LocalityGet();
555     localityBits = localityBits << commandLocality;
556     if((localityBits & s_initAttributes[pcr].extendLocality) == 0)
557         return FALSE;
558     else
559         return TRUE;
560 }

```

8.7.3.21 PCRExtend()

This function is used to extend a PCR in a specific bank.

```

561 void
562 PCRExtend(
563     TPMI_DH_PCR      handle,          // IN: PCR handle to be extended
564     TPMI_ALG_HASH    hash,           // IN: hash algorithm of PCR
565     UINT32           size,           // IN: size of data to be extended
566     BYTE             *data           // IN: data to be extended
567 )
568 {
569     BYTE             *pcrData;
570     HASH_STATE       hashState;
571     UINT16           pcrSize;
572
573     pcrData = GetPcrPointer(hash, handle - PCR_FIRST);
574
575     // Extend PCR if it is allocated
576     if(pcrData != NULL)
577     {
578         pcrSize = CryptHashGetDigestSize(hash);
579         CryptHashStart(&hashState, hash);
580         CryptDigestUpdate(&hashState, pcrSize, pcrData);
581         CryptDigestUpdate(&hashState, size, data);
582         CryptHashEnd(&hashState, pcrSize, pcrData);
583
584         // PCR has changed so update the pcrCounter if necessary
585         PCRChanged(handle);
586     }
587
588     return;
589 }

```

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.

```

590 void
591 PCRComputeCurrentDigest(
592     TPMI_ALG_HASH    hashAlg,        // IN: hash algorithm to compute digest
593     TPML_PCR_SELECTION *selection,    // IN/OUT: PCR selection (filtered on
594                                     // output)
595     TPM2B_DIGEST      *digest        // OUT: digest
596 )
597 {
598     HASH_STATE       hashState;
599     TPMS_PCR_SELECTION *select;
600     BYTE             *pcrData;      // will point to a digest

```

```

601     UINT32                pcrSize;
602     UINT32                pcr;
603     UINT32                i;
604
605     // Initialize the hash
606     digest->t.size = CryptHashStart(&hashState, hashAlg);
607     pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);
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         // Need the size of each digest
617         pcrSize = CryptHashGetDigestSize(selection->pcrSelections[i].hash);
618
619         // Iterate through the selection
620         for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
621         {
622             if(IsPcrSelected(pcr, select)) // Is this PCR selected
623             {
624                 // Get pointer to the digest data for the bank
625                 pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
626                 pAssert(pcrData != NULL);
627                 CryptDigestUpdate(&hashState, pcrSize, pcrData); // add to digest
628             }
629         }
630     }
631     // Complete hash stack
632     CryptHashEnd2B(&hashState, &digest->b);
633
634     return;
635 }

```

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.

```

636 void
637 PCRRead(
638     TPML_PCR_SELECTION *selection, // IN/OUT: PCR selection (filtered on
639                                     // output)
640     TPML_DIGEST         *digest,    // OUT: digest
641     UINT32              *pcrCounter // OUT: the current value of PCR generation
642                                     // number
643 )
644 {
645     TPMS_PCR_SELECTION *select;
646     BYTE               *pcrData; // will point to a digest
647     UINT32              pcr;
648     UINT32              i;
649
650     digest->count = 0;
651
652     // Iterate through the list of PCR selection structures
653     for(i = 0; i < selection->count; i++)
654     {
655         // Point to the current selection
656         select = &selection->pcrSelections[i]; // Point to the current selection
657         FilterPcr(select); // Clear out the bits for unimplemented PCR
658     }

```



```

659 // Iterate through the selection
660 for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
661 {
662     if(IsPcrSelected(pcr, select)) // Is this PCR selected
663     {
664         // Check if number of digest exceed upper bound
665         if(digest->count > 7)
666         {
667             // Clear rest of the current select bitmap
668             while(pcr < IMPLEMENTATION_PCR
669                 // do not round up!
670                 && (pcr / 8) < select->sizeofSelect)
671             {
672                 // do not round up!
673                 select->pcrSelect[pcr / 8] &= (BYTE)~(1 << (pcr % 8));
674                 pcr++;
675             }
676             // Exit inner loop
677             break;
678         }
679         // Need the size of each digest
680         digest->digests[digest->count].t.size =
681             CryptHashGetDigestSize(selection->pcrSelections[i].hash);
682
683         // Get pointer to the digest data for the bank
684         pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
685         pAssert(pcrData != NULL);
686         // Add to the data to digest
687         MemoryCopy(digest->digests[digest->count].t.buffer,
688                 pcrData,
689                 digest->digests[digest->count].t.size);
690         digest->count++;
691     }
692 }
693 // If we exit inner loop because we have exceed the output upper bound
694 if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
695 {
696     // Clear rest of the selection
697     while(i < selection->count)
698     {
699         MemorySet(selection->pcrSelections[i].pcrSelect, 0,
700                 selection->pcrSelections[i].sizeofSelect);
701         i++;
702     }
703     // exit outer loop
704     break;
705 }
706 }
707
708 *pcrCounter = gr.pcrCounter;
709
710 return;
711 }

```

8.7.3.24 PcrWrite()

This function is used by `_TPM_Hash_End()` to set a PCR to the computed hash of the H-CRTM event.

```

712 void
713 PcrWrite(
714     TPMI_DH_PCR    handle, // IN: PCR handle to be extended
715     TPMI_ALG_HASH  hash,   // IN: hash algorithm of PCR
716     TPM2B_DIGEST   *digest // IN: the new value
717 )

```

```

718 {
719     UINT32      pcr = handle - PCR_FIRST;
720     BYTE        *pcrData;
721
722     // Copy value to the PCR if it is allocated
723     pcrData = GetPcrPointer(hash, pcr);
724     if(pcrData != NULL)
725     {
726         MemoryCopy(pcrData, digest->t.buffer, digest->t.size);
727     }
728
729     return;
730 }

```

8.7.3.25 PCRAllocate()

This function is used to change the PCR allocation.

Error Returns	Meaning
TPM_RC_NO_RESULT	allocate failed
TPM_RC_PCR	improper allocation

```

731 TPM_RC
732 PCRAllocate(
733     TPML_PCR_SELECTION *allocate,    // IN: required allocation
734     UINT32              *maxPCR,     // OUT: Maximum number of PCR
735     UINT32              *sizeNeeded, // OUT: required space
736     UINT32              *sizeAvailable // OUT: available space
737 )
738 {
739     UINT32      i, j, k;
740     TPML_PCR_SELECTION newAllocate;
741     // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
742     BOOL        pcrHcrtm = FALSE;
743     BOOL        pcrDrtm = FALSE;
744
745     // Create the expected new PCR allocation based on the existing allocation
746     // and the new input:
747     // 1. if a PCR bank does not appear in the new allocation, the existing
748     // allocation of this PCR bank will be preserved.
749     // 2. if a PCR bank appears multiple times in the new allocation, only the
750     // last one will be in effect.
751     newAllocate = gp.pcrAllocated;
752     for(i = 0; i < allocate->count; i++)
753     {
754         for(j = 0; j < newAllocate.count; j++)
755         {
756             // If hash matches, the new allocation covers the old allocation
757             // for this particular bank.
758             // The assumption is the initial PCR allocation (from manufacture)
759             // has all the supported hash algorithms with an assigned bank
760             // (possibly empty). So there must be a match for any new bank
761             // allocation from the input.
762             if(newAllocate.pcrSelections[j].hash ==
763                allocate->pcrSelections[i].hash)
764             {
765                 newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
766                 break;
767             }
768         }
769         // The j loop must exit with a match.
770         pAssert(j < newAllocate.count);

```

```

771     }
772
773     // Max PCR in a bank is MIN(implemented PCR, PCR with attributes defined)
774     *maxPCR = sizeof(s_initAttributes) / sizeof(PCR_Attributes);
775     if(*maxPCR > IMPLEMENTATION_PCR)
776         *maxPCR = IMPLEMENTATION_PCR;
777
778     // Compute required size for allocation
779     *sizeNeeded = 0;
780     for(i = 0; i < newAllocate.count; i++)
781     {
782         UINT32    digestSize
783             = CryptHashGetDigestSize(newAllocate.pcrSelections[i].hash);
784     #if defined(DRTM_PCR)
785         // Make sure that we end up with at least one DRTM PCR
786         pcrDrtm = pcrDrtm || TestBit(DRTM_PCR,
787                                     newAllocate.pcrSelections[i].pcrSelect,
788                                     newAllocate.pcrSelections[i].sizeofSelect);
789     #else // if DRTM PCR is not required, indicate that the allocation is OK
790         pcrDrtm = TRUE;
791     #endif
792
793     #if defined(HCRTM_PCR)
794         // and one HCRTPM PCR (since this is usually PCR 0...)
795         pcrHcrtm = pcrHcrtm || TestBit(HCRTM_PCR,
796                                       newAllocate.pcrSelections[i].pcrSelect,
797                                       newAllocate.pcrSelections[i].sizeofSelect);
798     #else
799         pcrHcrtm = TRUE;
800     #endif
801
802     for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
803     {
804         BYTE    mask = 1;
805         for(k = 0; k < 8; k++)
806         {
807             if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
808                 *sizeNeeded += digestSize;
809             mask = mask << 1;
810         }
811     }
812 }
813
814 if(!pcrDrtm || !pcrHcrtm)
815     return TPM_RC_PCR;
816
817 // In this particular implementation, we always have enough space to
818 // allocate PCR. Different implementation may return a sizeAvailable less
819 // than the sizeNeed.
820 *sizeAvailable = sizeof(s_pcrs);
821
822 // Save the required allocation to NV. Note that after NV is written, the
823 // PCR allocation in NV is no longer consistent with the RAM data
824 // gp.pcrAllocated. The NV version reflect the allocate after next
825 // TPM_RESET, while the RAM version reflects the current allocation
826 NV_WRITE_PERSISTENT(pcrAllocated, newAllocate);
827
828 return TPM_RC_SUCCESS;
829 }

```

8.7.3.26 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.

```

830 void
831 PCRSetValue(
832     TPM_HANDLE    handle,          // IN: the handle of the PCR to set
833     INT8          initialValue     // IN: the value to set
834 )
835 {
836     int           i;
837     UINT32        pcr = handle - PCR_FIRST;
838     TPMI_ALG_HASH hash;
839     UINT16        digestSize;
840     BYTE          *pcrData;
841
842     // Iterate supported PCR bank algorithms to reset
843     for(i = 0; i < HASH_COUNT; i++)
844     {
845         hash = CryptHashGetAlgByIndex(i);
846         // Prevent runaway
847         if(hash == TPM_ALG_NULL)
848             break;
849
850         // Get a pointer to the data
851         pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
852
853         // If the PCR is allocated
854         if(pcrData != NULL)
855         {
856             // And the size of the digest
857             digestSize = CryptHashGetDigestSize(hash);
858
859             // Set the LSO to the input value
860             pcrData[digestSize - 1] = initialValue;
861
862             // Sign extend
863             if(initialValue >= 0)
864                 MemorySet(pcrData, 0, digestSize - 1);
865             else
866                 MemorySet(pcrData, -1, digestSize - 1);
867         }
868     }
869 }

```

8.7.3.27 PCRResetDynamics

This function is used to reset a dynamic PCR to 0. This function is used in DRTM sequence.

```

870 void
871 PCRResetDynamics(
872     void
873 )
874 {
875     UINT32        pcr, i;
876
877     // Initialize PCR values
878     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
879     {
880         // Iterate each hash algorithm bank
881         for(i = 0; i < gp.pcrAllocated.count; i++)
882         {
883             BYTE    *pcrData;
884             UINT32   pcrSize;
885
886             pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[i].hash, pcr);
887
888             if(pcrData != NULL)

```

```

889         {
890             pcrSize =
891                 CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[i].hash);
892
893             // Reset PCR
894             // Any PCR can be reset by locality 4 should be reset to 0
895             if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
896                 MemorySet(pcrData, 0, pcrSize);
897         }
898     }
899 }
900 return;
901 }

```

8.7.3.28 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

```

902 TPMI_YES_NO
903 PCRCapGetAllocation(
904     UINT32          count,           // IN: count of return
905     TPML_PCR_SELECTION *pcrSelection // OUT: PCR allocation list
906 )
907 {
908     if(count == 0)
909     {
910         pcrSelection->count = 0;
911         return YES;
912     }
913     else
914     {
915         *pcrSelection = gp.pcrAllocated;
916         return NO;
917     }
918 }

```

8.7.3.29 PCRSetSelectBit()

This function sets a bit in a bitmap array.

```

919 static void
920 PCRSetSelectBit(
921     UINT32          pcr,           // IN: PCR number
922     BYTE            *bitmap        // OUT: bit map to be set
923 )
924 {
925     bitmap[pcr / 8] |= (1 << (pcr % 8));
926     return;
927 }

```

8.7.3.30 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

```

928 static BOOL
929 PCRGetProperty(
930     TPM_PT_PCR                property,
931     TPMS_TAGGED_PCR_SELECT *select
932 )
933 {
934     UINT32                pcr;
935     UINT32                groupIndex;
936
937     select->tag = property;
938     // Always set the bitmap to be the size of all PCR
939     select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;
940
941     // Initialize bitmap
942     MemorySet(select->pcrSelect, 0, select->sizeofSelect);
943
944     // Collecting properties
945     for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
946     {
947         switch(property)
948         {
949             case TPM_PT_PCR_SAVE:
950                 if(s_initAttributes[pcr].stateSave == SET)
951                     PCRSetSelectBit(pcr, select->pcrSelect);
952                 break;
953             case TPM_PT_PCR_EXTEND_L0:
954                 if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
955                     PCRSetSelectBit(pcr, select->pcrSelect);
956                 break;
957             case TPM_PT_PCR_RESET_L0:
958                 if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
959                     PCRSetSelectBit(pcr, select->pcrSelect);
960                 break;
961             case TPM_PT_PCR_EXTEND_L1:
962                 if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
963                     PCRSetSelectBit(pcr, select->pcrSelect);
964                 break;
965             case TPM_PT_PCR_RESET_L1:
966                 if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
967                     PCRSetSelectBit(pcr, select->pcrSelect);
968                 break;
969             case TPM_PT_PCR_EXTEND_L2:
970                 if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
971                     PCRSetSelectBit(pcr, select->pcrSelect);
972                 break;
973             case TPM_PT_PCR_RESET_L2:
974                 if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
975                     PCRSetSelectBit(pcr, select->pcrSelect);
976                 break;
977             case TPM_PT_PCR_EXTEND_L3:
978                 if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
979                     PCRSetSelectBit(pcr, select->pcrSelect);
980                 break;
981             case TPM_PT_PCR_RESET_L3:
982                 if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
983                     PCRSetSelectBit(pcr, select->pcrSelect);
984                 break;
985             case TPM_PT_PCR_EXTEND_L4:
986                 if((s_initAttributes[pcr].extendLocality & 0x10) != 0)

```

```

987         PCRSetSelectBit(pcr, select->pcrSelect);
988         break;
989     case TPM_PT_PCR_RESET_L4:
990         if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
991             PCRSetSelectBit(pcr, select->pcrSelect);
992         break;
993     case TPM_PT_PCR_DRTM_RESET:
994         // DRTM reset PCRs are the PCR reset by locality 4
995         if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
996             PCRSetSelectBit(pcr, select->pcrSelect);
997         break;
998 #if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0
999     case TPM_PT_PCR_POLICY:
1000         if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
1001             PCRSetSelectBit(pcr, select->pcrSelect);
1002         break;
1003 #endif
1004 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
1005     case TPM_PT_PCR_AUTH:
1006         if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
1007             PCRSetSelectBit(pcr, select->pcrSelect);
1008         break;
1009 #endif
1010 #if ENABLE_PCR_NO_INCREMENT == YES
1011     case TPM_PT_PCR_NO_INCREMENT:
1012         if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
1013             PCRSetSelectBit(pcr, select->pcrSelect);
1014         break;
1015 #endif
1016     default:
1017         // If property is not supported, stop scanning PCR attributes
1018         // and return.
1019         return FALSE;
1020         break;
1021 }
1022 }
1023 return TRUE;
1024 }

```

8.7.3.31 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

```

1025 TPMI_YES_NO
1026 PCRCapGetProperties(
1027     TPM_PT_PCR                property,           // IN: the starting PCR property
1028     UINT32                    count,              // IN: count of returned properties
1029     TPML_TAGGED_PCR_PROPERTY *select             // OUT: PCR select
1030 )
1031 {
1032     TPMI_YES_NO    more = NO;
1033     UINT32         i;
1034
1035     // Initialize output property list
1036     select->count = 0;
1037
1038     // The maximum count of properties we may return is MAX_PCR_PROPERTIES
1039     if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;

```



```

1040
1041 // TPM_PT_PCR_FIRST is defined as 0 in spec. It ensures that property
1042 // value would never be less than TPM_PT_PCR_FIRST
1043 cAssert(TPM_PT_PCR_FIRST == 0);
1044
1045 // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
1046 // implemented on the TPM.
1047 for(i = property; i <= TPM_PT_PCR_LAST; i++)
1048 {
1049     if(select->count < count)
1050     {
1051         // If we have not filled up the return list, add more properties to it
1052         if(PCRGetProperty(i, &select->pcrProperty[select->count]))
1053             // only increment if the property is implemented
1054             select->count++;
1055     }
1056     else
1057     {
1058         // If the return list is full but we still have properties
1059         // available, report this and stop iterating.
1060         more = YES;
1061         break;
1062     }
1063 }
1064 return more;
1065 }

```

8.7.3.32 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

```

1066 TPMI_YES_NO
1067 PCRCapGetHandles(
1068     TPMI_DH_PCR    handle,        // IN: start handle
1069     UINT32         count,        // IN: count of returned handles
1070     TPML_HANDLE    *handleList   // OUT: list of handle
1071 )
1072 {
1073     TPMI_YES_NO    more = NO;
1074     UINT32         i;
1075
1076     pAssert(HandleGetType(handle) == TPM_HT_PCR);
1077
1078     // Initialize output handle list
1079     handleList->count = 0;
1080
1081     // The maximum count of handles we may return is MAX_CAP_HANDLES
1082     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1083
1084     // Iterate PCR handle range
1085     for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1086     {
1087         if(handleList->count < count)
1088         {
1089             // If we have not filled up the return list, add this PCR
1090             // handle to it
1091             handleList->handle[handleList->count] = i + PCR_FIRST;

```

```
1092         handleList->count++;
1093     }
1094     else
1095     {
1096         // If the return list is full but we still have PCR handle
1097         // available, report this and stop iterating
1098         more = YES;
1099         break;
1100     }
1101 }
1102 return more;
1103 }
```

DRAFT

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 PhysicalPresenceIsRequired()

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

The code in this file is used to manage the session context counter. The scheme implemented here is a "truncated counter". This scheme allows the TPM to not need TPM_SU_CLEAR for a very long period of time and still not have the context count for a session repeated.

The counter (*contextCounter*) in this implementation is a UINT64 but can be smaller. The "tracking array" (*contextArray*) only has 16-bits per context. The tracking array is the data that needs to be saved and restored across TPM_SU_STATE so that sessions are not lost when the system enters the sleep state. Also, when the TPM is active, the tracking array is kept in RAM making it important that the number of bytes for each entry be kept as small as possible.

The TPM prevents **collisions** of these truncated values by not allowing a *contextID* to be assigned if it would be the same as an existing value. Since the array holds 16 bits, after a context has been saved, an additional $2^{16}-1$ contexts may be saved before the count would again match. The normal expectation is that the context will be flushed before its count value is needed again but it is always possible to have long-lived sessions.

The *contextID* is assigned when the context is saved (TPM2_ContextSave()). At that time, the TPM will compare the low-order 16 bits of *contextCounter* to the existing values in *contextArray* and if one matches, the TPM will return TPM_RC_CONTEXT_GAP (by construction, the entry that contains the matching value is the oldest context).

The expected remediation by the TRM is to load the oldest saved session context (the one found by the TPM), and save it. Since loading the oldest session also eliminates its *contextID* value from *contextArray*, there TPM will always be able to load and save the oldest existing context.

In the worst case, software may have to load and save several contexts in order to save an additional one. This should happen very infrequently.

When the TPM searches *contextArray* and finds that none of the *contextIDs* match the low-order 16-bits of *contextCount*, the TPM can copy the low bits to the *contextArray* associated with the session, and increment *contextCount*.

There is one entry in *contextArray* for each of the active sessions allowed by the TPM implementation. This array contains either a context count, an index, or a value indicating the slot is available (0).

The index into the *contextArray* is the handle for the session with the region selector byte of the session set to zero. If an entry in *contextArray* contains 0, then the corresponding handle may be assigned to a session. If the entry contains a value that is less than or equal to the number of loaded sessions for the TPM, then the array entry is the slot in which the context is loaded.

EXAMPLE: If the TPM allows 8 loaded sessions, then the slot numbers would be 1-8 and a *contextArray* value in that range would represent the loaded session.

NOTE: When the TPM firmware determines that the array entry is for a loaded session, it will subtract 1 to create the zero-based slot number.

There is one significant corner case in this scheme. When the *contextCount* is equal to a value in the *contextArray*, the oldest session needs to be recycled or flushed. In order to recycle the session, it must be loaded. To be loaded, there must be an available slot. Rather than require that a spare slot be available all the time, the TPM will check to see if the *contextCount* is equal to some value in the *contextArray* when a session is created. This prevents the last session slot from being used when it is likely that a session will need to be recycled.

If a TPM with both 1.2 and 2.0 functionality uses this scheme for both 1.2 and 2.0 sessions, and the list of active contexts is read with TPM_GetCapability(), the TPM will create 32-bit representations of the list that contains 16-bit values (the TPM2_GetCapability() returns a list of handles for active sessions rather than

a list of *contextID*). The full *contextID* has high-order bits that are either the same as the current *contextCount* or one less. It is one less if the 16-bits of the *contextArray* has a value that is larger than the low-order 16 bits of *contextCount*.

8.9.2 Includes, Defines, and Local Variables

```
1  #define SESSION_C
2  #include "Tpm.h"
```

8.9.3 File Scope Function -- ContextIdSetOldest()

This function is called when the oldest *contextID* is being loaded or deleted. Once a saved context becomes the oldest, it stays the oldest until it is deleted.

Finding the oldest is a bit tricky. It is not just the numeric comparison of values but is dependent on the value of *contextCounter*.

Assume we have a small *contextArray* with 8, 4-bit values with values 1 and 2 used to indicate the loaded context slot number. Also assume that the array contains hex values of (0 0 1 0 3 0 9 F) and that the *contextCounter* is an 8-bit counter with a value of 0x37. Since the low nibble is 7, that means that values above 7 are older than values below it and, in this example, 9 is the oldest value.

Note if we subtract the counter value, from each slot that contains a saved *contextID* we get (- - - B - 2 - 8) and the oldest entry is now easy to find.

```
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 PWAP 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 SessionPCRValuesCurrent()

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 Returns	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
214 // Find an unoccupied entry in the contextArray
215 for(*handle = 0; *handle < MAX_ACTIVE_SESSIONS; (*handle)++)
216 {
217     if(gr.contextArray[*handle] == 0)
218     {
219         // indicate that the session associated with this handle
220         // references a loaded session
221         gr.contextArray[*handle] = (CONTEXT_SLOT)(sessionIndex + 1);
222         return TPM_RC_SUCCESS;
223     }
224 }
225 return TPM_RC_SESSION_HANDLES;
226 }

```

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 Returns	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

```

227 TPM_RC
228 SessionCreate(
229     TPM_SE      sessionType,    // IN: the session type
230     TPMI_ALG_HASH authHash,     // IN: the hash algorithm
231     TPM2B_NONCE *nonceCaller,   // IN: initial nonceCaller
232     TPMT_SYM_DEF *symmetric,    // IN: the symmetric algorithm
233     TPMI_DH_ENTITY bind,        // IN: the bind object
234     TPM2B_DATA *seed,           // IN: seed data
235     TPM_HANDLE *sessionHandle,   // OUT: the session handle
236     TPM2B_NONCE *nonceTpm       // OUT: the session nonce
237 )
238 {
239     TPM_RC      result = TPM_RC_SUCCESS;
240     CONTEXT_SLOT slotIndex;
241     SESSION     *session = NULL;
242
243     pAssert(sessionType == TPM_SE_HMAC
244             || sessionType == TPM_SE_POLICY
245             || sessionType == TPM_SE_TRIAL);
246
247     // If there are no open spots in the session array, then no point in searching
248     if(s_freeSessionSlots == 0)
249         return TPM_RC_SESSION_MEMORY;
250
251     // Find a space for loading a session
252     for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)

```

```

253 {
254     // Is this available?
255     if(s_sessions[slotIndex].occupied == FALSE)
256     {
257         session = &s_sessions[slotIndex].session;
258         break;
259     }
260 }
261 // if no spot found, then this is an internal error
262 if(slotIndex >= MAX_LOADED_SESSIONS)
263     FAIL(FATAL_ERROR_INTERNAL);
264
265 // Call context ID function to get a handle. TPM_RC_SESSION_HANDLE may be
266 // returned from ContextIdHandleAssign()
267 result = ContextIdSessionCreate(sessionHandle, slotIndex);
268 if(result != TPM_RC_SUCCESS)
269     return result;
270
271 /*** Only return from this point on is TPM_RC_SUCCESS
272
273 // Can now indicate that the session array entry is occupied.
274 s_freeSessionSlots--;
275 s_sessions[slotIndex].occupied = TRUE;
276
277 // Initialize the session data
278 MemorySet(session, 0, sizeof(SESSION));
279
280 // Initialize internal session data
281 session->authHashAlg = authHash;
282 // Initialize session type
283 if(sessionType == TPM_SE_HMAC)
284 {
285     *sessionHandle += HMAC_SESSION_FIRST;
286 }
287 else
288 {
289     *sessionHandle += POLICY_SESSION_FIRST;
290
291     // For TPM_SE_POLICY or TPM_SE_TRIAL
292     session->attributes.isPolicy = SET;
293     if(sessionType == TPM_SE_TRIAL)
294         session->attributes.isTrialPolicy = SET;
295
296     SessionSetStartTime(session);
297
298     // Initialize policyDigest. policyDigest is initialized with a string of 0
299     // of session algorithm digest size. Since the session is already clear.
300     // Just need to set the size
301     session->u2.policyDigest.t.size =
302         CryptHashGetDigestSize(session->authHashAlg);
303 }
304 // Create initial session nonce
305 session->nonceTPM.t.size = nonceCaller->t.size;
306 CryptRandomGenerate(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
307 MemoryCopy2B(&nonceTpm->b, &session->nonceTPM.b,
308             sizeof(nonceTpm->t.buffer));
309
310 // Set up session parameter encryption algorithm
311 session->symmetric = *symmetric;
312
313 // If there is a bind object or a session secret, then need to compute
314 // a sessionKey.
315 if(bind != TPM_RH_NULL || seed->t.size != 0)
316 {
317     // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
318     //                      nonceCaller, bits)

```

```

319 // The HMAC key for generating the sessionSecret can be the concatenation
320 // of an authorization value and a seed value
321 TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
322 TPM2B_KEY key;
323
324 // Get hash size, which is also the length of sessionKey
325 session->sessionKey.t.size = CryptHashGetDigestSize(session->authHashAlg);
326
327 // Get authValue of associated entity
328 EntityGetAuthValue(bind, (TPM2B_AUTH *)&key);
329 pAssert(key.t.size + seed->t.size <= sizeof(key.t.buffer));
330
331 // Concatenate authValue and seed
332 MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));
333
334 // Compute the session key
335 CryptKDFa(session->authHashAlg, &key.b, SESSION_KEY, &session->nonceTPM.b,
336           &nonceCaller->b,
337           session->sessionKey.t.size * 8, session->sessionKey.t.buffer,
338           NULL, FALSE);
339 }
340
341 // Copy the name of the entity that the HMAC session is bound to
342 // Policy session is not bound to an entity
343 if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
344 {
345     session->attributes.isBound = SET;
346     SessionComputeBoundEntity(bind, &session->ul.boundEntity);
347 }
348 // If there is a bind object and it is subject to DA, then use of this session
349 // is subject to DA regardless of how it is used.
350 session->attributes.isDaBound = (bind != TPM_RH_NULL)
351     && (IsDAExempted(bind) == FALSE);
352
353 // If the session is bound, then check to see if it is bound to lockoutAuth
354 session->attributes.isLockoutBound = (session->attributes.isDaBound == SET)
355     && (bind == TPM_RH_LOCKOUT);
356 return TPM_RC_SUCCESS;
357 }

```

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 Returns	Meaning
TPM_RC_CONTEXT_GAP	a <i>contextID</i> could not be assigned.
TPM_RC_TOO_MANY_CONTEXTS	the counter maxed out

```

358 TPM_RC
359 SessionContextSave(
360     TPM_HANDLE      handle,           // IN: session handle
361     CONTEXT_COUNTER *contextID       // OUT: assigned contextID
362 )
363 {
364     UINT32          contextIndex;
365     CONTEXT_SLOT    slotIndex;
366
367     pAssert(SessionIsLoaded(handle));

```



```

368
369 // check to see if the gap is already maxed out
370 // Need to have a saved session
371 if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
372    // if the oldest saved session has the same value as the low bits
373    // of the contextCounter, then the GAP is maxed out.
374    && gr.contextArray[s_oldestSavedSession] == (CONTEXT_SLOT)gr.contextCounter)
375     return TPM_RC_CONTEXT_GAP;
376
377 // if the caller wants the context counter, set it
378 if(contextID != NULL)
379     *contextID = gr.contextCounter;
380
381 contextIndex = handle & HR_HANDLE_MASK;
382 pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
383
384 // Extract the session slot number referenced by the contextArray
385 // because we are going to overwrite this with the low order
386 // contextID value.
387 slotIndex = gr.contextArray[contextIndex] - 1;
388
389 // Set the contextID for the contextArray
390 gr.contextArray[contextIndex] = (CONTEXT_SLOT)gr.contextCounter;
391
392 // Increment the counter
393 gr.contextCounter++;
394
395 // In the unlikely event that the 64-bit context counter rolls over...
396 if(gr.contextCounter == 0)
397 {
398     // back it up
399     gr.contextCounter--;
400     // return an error
401     return TPM_RC_TOO_MANY_CONTEXTS;
402 }
403 // if the low-order bits wrapped, need to advance the value to skip over
404 // the values used to indicate that a session is loaded
405 if((CONTEXT_SLOT)gr.contextCounter == 0)
406     gr.contextCounter += MAX_LOADED_SESSIONS + 1;
407
408 // If no other sessions are saved, this is now the oldest.
409 if(s_oldestSavedSession >= MAX_ACTIVE_SESSIONS)
410     s_oldestSavedSession = contextIndex;
411
412 // Mark the session slot as unoccupied
413 s_sessions[slotIndex].occupied = FALSE;
414
415 // and indicate that there is an additional open slot
416 s_freeSessionSlots++;
417
418 return TPM_RC_SUCCESS;
419 }

```

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 Returns	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

```

420 TPM_RC
421 SessionContextLoad(
422     SESSION_BUF    *session,        // IN: session structure from saved context
423     TPM_HANDLE     *handle          // IN/OUT: session handle
424 )
425 {
426     UINT32          contextIndex;
427     CONTEXT_SLOT    slotIndex;
428
429     pAssert(HandleGetType(*handle) == TPM_HT_POLICY_SESSION
430             || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);
431
432     // Don't bother looking if no openings
433     if(s_freeSessionSlots == 0)
434         return TPM_RC_SESSION_MEMORY;
435
436     // Find a free session slot to load the session
437     for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
438         if(s_sessions[slotIndex].occupied == FALSE) break;
439
440     // if no spot found, then this is an internal error
441     pAssert(slotIndex < MAX_LOADED_SESSIONS);
442
443     contextIndex = *handle & HR_HANDLE_MASK; // extract the index
444
445     // If there is only one slot left, and the gap is at maximum, the only session
446     // context that we can safely load is the oldest one.
447     if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
448        && s_freeSessionSlots == 1
449        && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
450        && contextIndex != s_oldestSavedSession)
451         return TPM_RC_CONTEXT_GAP;
452
453     pAssert(contextIndex < MAX_ACTIVE_SESSIONS);
454
455     // set the contextArray value to point to the session slot where
456     // the context is loaded
457     gr.contextArray[contextIndex] = slotIndex + 1;
458
459     // if this was the oldest context, find the new oldest
460     if(contextIndex == s_oldestSavedSession)
461         ContextIdSetOldest();
462
463     // Copy session data to session slot
464     MemoryCopy(&s_sessions[slotIndex].session, session, sizeof(SESSION));
465
466     // Set session slot as occupied
467     s_sessions[slotIndex].occupied = TRUE;
468
469     // Reduce the number of open spots
470     s_freeSessionSlots--;
471
472     return TPM_RC_SUCCESS;
473 }

```

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.

```

474 void
475 SessionFlush(
476     TPM_HANDLE      handle          // IN: loaded or saved session handle
477 )
478 {
479     CONTEXT_SLOT      slotIndex;
480     UINT32            contextIndex;  // Index into contextArray
481
482     pAssert((HandleGetType(handle) == TPM_HT_POLICY_SESSION
483             || HandleGetType(handle) == TPM_HT_HMAC_SESSION
484             )
485             && (SessionIsLoaded(handle) || SessionIsSaved(handle))
486             );
487
488     // Flush context ID of this session
489     // Convert handle to an index into the contextArray
490     contextIndex = handle & HR_HANDLE_MASK;
491
492     pAssert(contextIndex < sizeof(gr.contextArray) / sizeof(gr.contextArray[0]));
493
494     // Get the current contents of the array
495     slotIndex = gr.contextArray[contextIndex];
496
497     // Mark context array entry as available
498     gr.contextArray[contextIndex] = 0;
499
500     // Is this a saved session being flushed
501     if(slotIndex > MAX_LOADED_SESSIONS)
502     {
503         // Flushing the oldest session?
504         if(contextIndex == s_oldestSavedSession)
505             // If so, find a new value for oldest.
506             ContextIdSetOldest();
507     }
508     else
509     {
510         // Adjust slot index to point to session array index
511         slotIndex -= 1;
512
513         // Free session array index
514         s_sessions[slotIndex].occupied = FALSE;
515         s_freeSessionSlots++;
516     }
517
518     return;
519 }

```

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.

```

520 void
521 SessionComputeBoundEntity(

```

```

522     TPMI_DH_ENTITY    entityHandle, // IN: handle of entity
523     TPM2B_NAME        *bind         // OUT: binding value
524 )
525 {
526     TPM2B_AUTH        auth;
527     BYTE              *pAuth = auth.t.buffer;
528     UINT16            i;
529
530     // Get name
531     EntityGetName(entityHandle, bind);
532
533     // // The bound value of a reserved handle is the handle itself
534     // if(bind->t.size == sizeof(TPM_HANDLE)) return;
535
536     // For all the other entities, concatenate the authorization value to the name.
537     // Get a local copy of the authorization value because some overlapping
538     // may be necessary.
539     EntityGetAuthValue(entityHandle, &auth);
540
541     // Make sure that the extra space is zeroed
542     MemorySet(&bind->t.name[bind->t.size], 0, sizeof(bind->t.name) - bind->t.size);
543     // XOR the authValue at the end of the name
544     for(i = sizeof(bind->t.name) - auth.t.size; i < sizeof(bind->t.name); i++)
545         bind->t.name[i] ^= *pAuth++;
546
547     // Set the bind value to the maximum size
548     bind->t.size = sizeof(bind->t.name);
549
550     return;
551 }

```

8.9.6.7 SessionSetStartTime()

This function is used to initialize the session timing

```

552 void
553 SessionSetStartTime(
554     SESSION    *session // IN: the session to update
555 )
556 {
557     session->startTime = g_time;
558     session->epoch = g_timeEpoch;
559     session->timeout = 0;
560 }

```

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.

```

561 void
562 SessionResetPolicyData(
563     SESSION    *session // IN: the session to reset
564 )
565 {
566     SESSION_ATTRIBUTES    oldAttributes;
567     pAssert(session != NULL);
568
569     // Will need later
570     oldAttributes = session->attributes;
571
572     // No command
573     session->commandCode = 0;
574 }

```

```

575 // No locality selected
576 MemorySet(&session->commandLocality, 0, sizeof(session->commandLocality));
577
578 // The cpHash size to zero
579 session->u1.cpHash.b.size = 0;
580
581 // No timeout
582 session->timeout = 0;
583
584 // Reset the pcrCounter
585 session->pcrCounter = 0;
586
587 // Reset the policy hash
588 MemorySet(&session->u2.policyDigest.t.buffer, 0,
589          session->u2.policyDigest.t.size);
590
591 // Reset the session attributes
592 MemorySet(&session->attributes, 0, sizeof(SESSION_ATTRIBUTES));
593
594 // Restore the policy attributes
595 session->attributes.isPolicy = SET;
596 session->attributes.isTrialPolicy = oldAttributes.isTrialPolicy;
597
598 // Restore the bind attributes
599 session->attributes.isDaBound = oldAttributes.isDaBound;
600 session->attributes.isLockoutBound = oldAttributes.isLockoutBound;
601 }

```

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

```

602 TPMI_YES_NO
603 SessionCapGetLoaded(
604     TPMI_SH_POLICY    handle,           // IN: start handle
605     UINT32             count,           // IN: count of returned handles
606     TPML_HANDLE        *handleList      // OUT: list of handle
607 )
608 {
609     TPMI_YES_NO        more = NO;
610     UINT32             i;
611
612     pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);
613
614     // Initialize output handle list
615     handleList->count = 0;
616
617     // The maximum count of handles we may return is MAX_CAP_HANDLES
618     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
619
620     // Iterate session context ID slots to get loaded session handles
621     for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
622     {
623         // If session is active
624         if(gr.contextArray[i] != 0)
625         {
626             // If session is loaded

```

```

627         if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
628         {
629             if(handleList->count < count)
630             {
631                 SESSION          *session;
632
633                 // If we have not filled up the return list, add this
634                 // session handle to it
635                 // assume that this is going to be an HMAC session
636                 handle = i + HMAC_SESSION_FIRST;
637                 session = SessionGet(handle);
638                 if(session->attributes.isPolicy)
639                     handle = i + POLICY_SESSION_FIRST;
640                 handleList->handle[handleList->count] = handle;
641                 handleList->count++;
642             }
643             else
644             {
645                 // If the return list is full but we still have loaded object
646                 // available, report this and stop iterating
647                 more = YES;
648                 break;
649             }
650         }
651     }
652 }
653
654 return more;
655 }

```

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

```

656 TPMI_YES_NO
657 SessionCapGetSaved(
658     TPMI_SH_HMAC    handle,          // IN: start handle
659     UINT32          count,          // IN: count of returned handles
660     TPML_HANDLE     *handleList     // OUT: list of handle
661 )
662 {
663     TPMI_YES_NO     more = NO;
664     UINT32          i;
665
666     #ifdef TPM_HT_SAVED_SESSION
667         pAssert(HandleGetType(handle) == TPM_HT_SAVED_SESSION);
668     #else
669         pAssert(HandleGetType(handle) == TPM_HT_ACTIVE_SESSION);
670     #endif
671
672     // Initialize output handle list
673     handleList->count = 0;
674
675     // The maximum count of handles we may return is MAX_CAP_HANDLES
676     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
677
678     // Iterate session context ID slots to get loaded session handles

```

```

679     for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
680     {
681         // If session is active
682         if(gr.contextArray[i] != 0)
683         {
684             // If session is saved
685             if(gr.contextArray[i] > MAX_LOADED_SESSIONS)
686             {
687                 if(handleList->count < count)
688                 {
689                     // If we have not filled up the return list, add this
690                     // session handle to it
691                     handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
692                     handleList->count++;
693                 }
694                 else
695                 {
696                     // If the return list is full but we still have loaded object
697                     // available, report this and stop iterating
698                     more = YES;
699                     break;
700                 }
701             }
702         }
703     }
704     return more;
705 }
706

```

8.9.6.11 SessionCapGetLoadedNumber()

This function return the number of authorization sessions currently loaded into TPM RAM.

```

707     UINT32
708     SessionCapGetLoadedNumber(
709         void
710     )
711     {
712         return MAX_LOADED_SESSIONS - s_freeSessionSlots;
713     }

```

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.

```

714     UINT32
715     SessionCapGetLoadedAvail(
716         void
717     )
718     {
719         return s_freeSessionSlots;
720     }

```

8.9.6.13 SessionCapGetActiveNumber()

This function returns the number of active authorization sessions currently being tracked by the TPM.


```
721  UINT32
722  SessionCapGetActiveNumber(
723      void
724  )
725  {
726      UINT32          i;
727      UINT32          num = 0;
728
729      // Iterate the context array to find the number of non-zero slots
730      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
731      {
732          if(gr.contextArray[i] != 0) num++;
733      }
734
735      return num;
736  }
```

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.

```
737  UINT32
738  SessionCapGetActiveAvail(
739      void
740  )
741  {
742      UINT32          i;
743      UINT32          num = 0;
744
745      // Iterate the context array to find the number of zero slots
746      for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
747      {
748          if(gr.contextArray[i] == 0) num++;
749      }
750
751      return num;
752  }
```

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 "PlatformData.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
132 return;
133 }

```

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

```

134 UINT16
135 TimeGetMarshaled(
136     TIME_INFO      *dataBuffer    // OUT: result buffer
137 )
138 {
139     TPMS_TIME_INFO  timeInfo;
140
141     // Fill TPMS_TIME_INFO structure
142     timeInfo.time = g_time;
143     TimeFillInfo(&timeInfo.clockInfo);
144
145     // Marshal TPMS_TIME_INFO to canonical form
146     return TPMS_TIME_INFO_Marshal(&timeInfo, (BYTE **)&dataBuffer, NULL);
147 }

```

8.10.3.9 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

```

148 void
149 TimeFillInfo(
150     TPMS_CLOCK_INFO *clockInfo
151 )
152 {
153     clockInfo->clock = go.clock;
154     clockInfo->resetCount = gp.resetCount;
155     clockInfo->restartCount = gr.restartCount;
156
157     // If NV is not available, clock stopped advancing and the value reported is
158     // not "safe".

```

```
159     if(NV_IS_AVAILABLE)
160         clockInfo->safe = go.clockSafe;
161     else
162         clockInfo->safe = NO;
163
164     return;
165 }
```

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  typedef struct
3  {
4      TPM_ALG_ID          algID;
5      TPMA_ALGORITHM      attributes;
6  } ALGORITHM;
7  static const ALGORITHM  s_algorithms[] =
8  {
9      // The entries in this table need to be in ascending order but the table doesn't
10     // need to be full (gaps are allowed). One day, a tool might exist to fill in the
11     // table from the TPM_ALG description
12     #if ALG_RSA
13         {TPM_ALG_RSA,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
14     #endif
15     #if ALG_TDES
16         {TPM_ALG_TDES,         TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
17     #endif
18     #if ALG_SHA1
19         {TPM_ALG_SHA1,         TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
20     #endif
21
22         {TPM_ALG_HMAC,         TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 1, 0, 0, 0)},
23
24     #if ALG_AES
25         {TPM_ALG_AES,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
26     #endif
27     #if ALG_MGF1
28         {TPM_ALG_MGF1,         TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
29     #endif
30
31         {TPM_ALG_KEYEDHASH,    TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 1, 0, 1, 1, 0, 0)},
32
33     #if ALG_XOR
34         {TPM_ALG_XOR,          TPMA_ALGORITHM_INITIALIZER(0, 1, 1, 0, 0, 0, 0, 0, 0)},
35     #endif
36
37     #if ALG_SHA256
38         {TPM_ALG_SHA256,       TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
39     #endif
40     #if ALG_SHA384
41         {TPM_ALG_SHA384,       TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
42     #endif
43     #if ALG_SHA512
44         {TPM_ALG_SHA512,       TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
45     #endif
46     #if ALG_SM3_256
47         {TPM_ALG_SM3_256,      TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
48     #endif
49     #if ALG_SM4
50         {TPM_ALG_SM4,          TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
51     #endif

```



```

52  #if ALG_RSASSA
53      {TPM_ALG_RSASSA,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
54  #endif
55  #if ALG_RSAES
56      {TPM_ALG_RSAES,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
57  #endif
58  #if ALG_RSAPSS
59      {TPM_ALG_RSAPSS,          TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
60  #endif
61  #if ALG_OAEP
62      {TPM_ALG_OAEP,            TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 1, 0, 0)},
63  #endif
64  #if ALG_ECDSA
65      {TPM_ALG_ECDSA,           TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 1, 0)},
66  #endif
67  #if ALG_ECDH
68      {TPM_ALG_ECDH,            TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
69  #endif
70  #if ALG_ECDA
71      {TPM_ALG_ECDA,            TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
72  #endif
73  #if ALG_SM2
74      {TPM_ALG_SM2,             TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 1, 0)},
75  #endif
76  #if ALG_ECSCHNORR
77      {TPM_ALG_ECSCHNORR,       TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)},
78  #endif
79  #if ALG_ECMQV
80      {TPM_ALG_ECMQV,           TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 0, 0, 1, 0)},
81  #endif
82  #if ALG_KDF1_SP800_56A
83      {TPM_ALG_KDF1_SP800_56A, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
84  #endif
85  #if ALG_KDF2
86      {TPM_ALG_KDF2,            TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
87  #endif
88  #if ALG_KDF1_SP800_108
89      {TPM_ALG_KDF1_SP800_108, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)},
90  #endif
91  #if ALG_ECC
92      {TPM_ALG_ECC,              TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 1, 0, 0, 0, 0, 0)},
93  #endif
94
95      {TPM_ALG_SYMCIPHER,        TPMA_ALGORITHM_INITIALIZER(0, 0, 0, 1, 0, 0, 0, 0, 0)},
96
97  #if ALG_CAMELLIA
98      {TPM_ALG_CAMELLIA,        TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
99  #endif
100 #if ALG_CMAC
101     {TPM_ALG_CMAC,             TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 1, 0, 0, 0)},
102 #endif
103 #if ALG_CTR
104     {TPM_ALG_CTR,              TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
105 #endif
106 #if ALG_OFB
107     {TPM_ALG_OFB,              TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
108 #endif
109 #if ALG_CBC
110     {TPM_ALG_CBC,              TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
111 #endif
112 #if ALG_CFB
113     {TPM_ALG_CFB,              TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
114 #endif
115 #if ALG_ECB
116     {TPM_ALG_ECB,              TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 1, 0, 0)},
117 #endif

```

```
118 };
```

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

```
119 TPMI_YES_NO
120 AlgorithmCapGetImplemented(
121     TPM_ALG_ID      algID,      // IN: the starting algorithm ID
122     UINT32           count,      // IN: count of returned algorithms
123     TPML_ALG_PROPERTY *algList  // OUT: algorithm list
124 )
125 {
126     TPMI_YES_NO    more = NO;
127     UINT32         i;
128     UINT32         algNum;
129
130     // initialize output algorithm list
131     algList->count = 0;
132
133     // The maximum count of algorithms we may return is MAX_CAP_ALGS.
134     if(count > MAX_CAP_ALGS)
135         count = MAX_CAP_ALGS;
136
137     // Compute how many algorithms are defined in s_algorithms array.
138     algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);
139
140     // Scan the implemented algorithm list to see if there is a match to 'algID'.
141     for(i = 0; i < algNum; i++)
142     {
143         // If algID is less than the starting algorithm ID, skip it
144         if(s_algorithms[i].algID < algID)
145             continue;
146         if(algList->count < count)
147         {
148             // If we have not filled up the return list, add more algorithms
149             // to it
150             algList->algProperties[algList->count].alg = s_algorithms[i].algID;
151             algList->algProperties[algList->count].algProperties =
152                 s_algorithms[i].attributes;
153             algList->count++;
154         }
155         else
156         {
157             // If the return list is full but we still have algorithms
158             // available, report this and stop scanning.
159             more = YES;
160             break;
161         }
162     }
163
164     return more;
165 }
```

9.1.4 AlgorithmGetImplementedVector()

This function returns the bit vector of the implemented algorithms.

```
166 LIB_EXPORT
167 void
168 AlgorithmGetImplementedVector(
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
177     for(index = (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1;
178         index >= 0;
179         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  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.

```
7  #define _COMMAND_CODE_ATTRIBUTES_
8  #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

```
9  #if !COMPRESSED_LISTS
10 static COMMAND_INDEX
11 NextImplementedIndex(
12     COMMAND_INDEX    commandIndex
13 )
14 {
15     for(;commandIndex < COMMAND_COUNT; commandIndex++)
16     {
17         if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
18             return commandIndex;
19     }
20     return UNIMPLEMENTED_COMMAND_INDEX;
21 }
22 #else
23 #define NextImplementedIndex(x) (x)
24 #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

```

25  COMMAND_INDEX
26  GetClosestCommandIndex(
27      TPM_CC      commandCode    // IN: the command code to start at
28  )
29  {
30      BOOL          vendor = (commandCode & CC_VEND) != 0;
31      COMMAND_INDEX searchIndex = (COMMAND_INDEX)commandCode;
32
33      // The commandCode is a UINT32 and the search index is UINT16. We are going to
34      // search for a match but need to make sure that the commandCode value is not
35      // out of range. To do this, need to clear the vendor bit of the commandCode
36      // (if set) and compare the result to the 16-bit searchIndex value. If it is
37      // out of range, indicate that the command is not implemented
38      if((commandCode & ~CC_VEND) != searchIndex)
39          return UNIMPLEMENTED_COMMAND_INDEX;
40
41      // if there is at least one vendor command, the last entry in the array will
42      // have the v bit set. If the input commandCode is larger than the last
43      // vendor-command, then it is out of range.
44      if(vendor)
45      {
46          #if VENDOR_COMMAND_ARRAY_SIZE > 0
47              COMMAND_INDEX commandIndex;
48              COMMAND_INDEX min;
49              COMMAND_INDEX max;
50              int diff;
51          #if LIBRARY_COMMAND_ARRAY_SIZE == COMMAND_COUNT
52              #error "Constants are not consistent."
53          #endif
54              // Check to see if the value is equal to or below the minimum
55              // entry.
56              // Note: Put this check first so that the typical case of only one vendor-
57              // specific command doesn't waste any more time.
58              if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE], TPMA_CC,
59                  commandIndex) >= searchIndex)
60              {
61                  // the vendor array is always assumed to be packed so there is
62                  // no need to check to see if the command is implemented
63                  return LIBRARY_COMMAND_ARRAY_SIZE;
64              }
65              // See if this is out of range on the top
66              if(GET_ATTRIBUTE(s_ccAttr[COMMAND_COUNT - 1], TPMA_CC, commandIndex)
67                  < searchIndex)
68              {
69                  return UNIMPLEMENTED_COMMAND_INDEX;
70              }
71              commandIndex = UNIMPLEMENTED_COMMAND_INDEX; // Needs initialization to keep
72                                                              // compiler happy
73              min = LIBRARY_COMMAND_ARRAY_SIZE;           // first vendor command
74              max = COMMAND_COUNT - 1;                     // last vendor command
75              diff = 1;                                     // needs initialization to keep
76                                                              // compiler happy
77              while(min <= max)
78              {

```



```

79         commandIndex = (min + max + 1) / 2;
80         diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
81             - searchIndex;
82         if(diff == 0)
83             return commandIndex;
84         if(diff > 0)
85             max = commandIndex - 1;
86         else
87             min = commandIndex + 1;
88     }
89     // didn't find an exact match. commandIndex will be pointing at the last
90     // item tested. If 'diff' is positive, then the last item tested was
91     // larger index of the command code so it is the smallest value
92     // larger than the requested value.
93     if(diff > 0)
94         return commandIndex;
95     // if 'diff' is negative, then the value tested was smaller than
96     // the commandCode index and the next higher value is the correct one.
97     // Note: this will necessarily be in range because of the earlier check
98     // that the index was within range.
99     return commandIndex + 1;
100 #else
101     // If there are no vendor commands so anything with the vendor bit set is out
102     // of range
103     return UNIMPLEMENTED_COMMAND_INDEX;
104 #endif
105 }
106 // Get here if the V-Bit was not set in 'commandCode'
107
108 if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE - 1], TPMA_CC,
109                 commandIndex) < searchIndex)
110 {
111     // requested index is out of the range to the top
112 #if VENDOR_COMMAND_ARRAY_SIZE > 0
113     // If there are vendor commands, then the first vendor command
114     // is the next value greater than the commandCode.
115     // NOTE: we got here if the starting index did not have the V bit but we
116     // reached the end of the array of library commands (non-vendor). Since
117     // there is at least one vendor command, and vendor commands are always
118     // in a compressed list that starts after the library list, the next
119     // index value contains a valid vendor command.
120     return LIBRARY_COMMAND_ARRAY_SIZE;
121 #else
122     // if there are no vendor commands, then this is out of range
123     return UNIMPLEMENTED_COMMAND_INDEX;
124 #endif
125 }
126 // If the request is lower than any value in the array, then return
127 // the lowest value (needs to be an index for an implemented command)
128 if(GET_ATTRIBUTE(s_ccAttr[0], TPMA_CC, commandIndex) >= searchIndex)
129 {
130     return NextImplementedIndex(0);
131 }
132 else
133 {
134 #if COMPRESSED_LISTS
135     COMMAND_INDEX      commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
136     COMMAND_INDEX      min = 0;
137     COMMAND_INDEX      max = LIBRARY_COMMAND_ARRAY_SIZE - 1;
138     int                 diff = 1;
139 #if LIBRARY_COMMAND_ARRAY_SIZE == 0
140 #error "Something is terribly wrong"
141 #endif
142     // The s_ccAttr array contains an extra entry at the end (a zero value).
143     // Don't count this as an array entry. This means that max should start
144     // out pointing to the last valid entry in the array which is - 2

```

```

145     pAssert(max == (sizeof(s_ccAttr) / sizeof(TPMA_CC)
146                  - VENDOR_COMMAND_ARRAY_SIZE - 2));
147     while(min <= max)
148     {
149         commandIndex = (min + max + 1) / 2;
150         diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC,
151                             commandIndex) - searchIndex;
152         if(diff == 0)
153             return commandIndex;
154         if(diff > 0)
155             max = commandIndex - 1;
156         else
157             min = commandIndex + 1;
158     }
159     // didn't find an exact match. commandIndex will be pointing at the
160     // last item tested. If diff is positive, then the last item tested was
161     // larger index of the command code so it is the smallest value
162     // larger than the requested value.
163     if(diff > 0)
164         return commandIndex;
165     // if diff is negative, then the value tested was smaller than
166     // the commandCode index and the next higher value is the correct one.
167     // Note: this will necessarily be in range because of the earlier check
168     // that the index was within range.
169     return commandIndex + 1;
170 #else
171     // The list is not compressed so offset into the array by the command
172     // code value of the first entry in the list. Then go find the first
173     // implemented command.
174     return NextImplementedIndex(searchIndex
175                                - (COMMAND_INDEX)s_ccAttr[0].commandIndex);
176 #endif
177 }
178 }

```

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

```

179 COMMAND_INDEX
180 CommandCodeToCommandIndex(
181     TPM_CC      commandCode    // IN: the command code to look up
182 )
183 {
184     // Extract the low 16-bits of the command code to get the starting search index
185     COMMAND_INDEX searchIndex = (COMMAND_INDEX)commandCode;
186     BOOL          vendor = (commandCode & CC_VEND) != 0;
187     COMMAND_INDEX commandIndex;
188 #if !COMPRESSED_LISTS
189     if(!vendor)
190     {
191         commandIndex = searchIndex - (COMMAND_INDEX)s_ccAttr[0].commandIndex;
192         // Check for out of range or unimplemented.
193         // Note, since a COMMAND_INDEX is unsigned, if searchIndex is smaller than
194         // the lowest value of command, it will become a 'negative' number making
195         // it look like a large unsigned number, this will cause it to fail
196         // the unsigned check below.
197         if(commandIndex >= LIBRARY_COMMAND_ARRAY_SIZE

```

```

198         || (s_commandAttributes[commandIndex] & IS_IMPLEMENTED) == 0)
199         return UNIMPLEMENTED_COMMAND_INDEX;
200     return commandIndex;
201 }
202 #endif
203 // Need this code for any vendor code lookup or for compressed lists
204 commandIndex = GetClosestCommandIndex(commandCode);
205
206 // Look at the returned value from get closest. If it isn't the one that was
207 // requested, then the command is not implemented.
208 if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
209 {
210     if((GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
211         != searchIndex)
212         || (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V)) != vendor)
213         commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
214 }
215 return commandIndex;
216 }

```

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

```

217 COMMAND_INDEX
218 GetNextCommandIndex(
219     COMMAND_INDEX    commandIndex    // IN: the starting index
220 )
221 {
222     while(++commandIndex < COMMAND_COUNT)
223     {
224         #if !COMPRESSED_LISTS
225             if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
226                 #endif
227                 return commandIndex;
228     }
229     return UNIMPLEMENTED_COMMAND_INDEX;
230 }

```

9.3.3.5 GetCommandCode()

This function returns the *commandCode* associated with the command index

```

231 TPM_CC
232 GetCommandCode(
233     COMMAND_INDEX    commandIndex    // IN: the command index
234 )
235 {
236     TPM_CC            commandCode = GET_ATTRIBUTE(s_ccAttr[commandIndex],
237                                                    TPMA_CC, commandIndex);
238     if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
239         commandCode += CC_VEND;
240     return commandCode;
241 }

```

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

```

242  AUTH_ROLE
243  CommandAuthRole(
244      COMMAND_INDEX    commandIndex, // IN: command index
245      UINT32            handleIndex   // IN: handle index (zero based)
246  )
247  {
248      if(0 == handleIndex)
249      {
250          // Any authorization role set?
251          COMMAND_ATTRIBUTES properties = s_commandAttributes[commandIndex];
252
253          if(properties & HANDLE_1_USER)
254              return AUTH_USER;
255          if(properties & HANDLE_1_ADMIN)
256              return AUTH_ADMIN;
257          if(properties & HANDLE_1_DUP)
258              return AUTH_DUP;
259      }
260      else if(1 == handleIndex)
261      {
262          if(s_commandAttributes[commandIndex] & HANDLE_2_USER)
263              return AUTH_USER;
264      }
265      return AUTH_NONE;
266  }

```

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

```

267  int
268  EncryptSize(
269      COMMAND_INDEX    commandIndex // IN: command index
270  )
271  {
272      return ((s_commandAttributes[commandIndex] & ENCRYPT_2) ? 2 :
273              (s_commandAttributes[commandIndex] & ENCRYPT_4) ? 4 : 0);
274  }

```

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

```

275  int
276  DecryptSize(
277      COMMAND_INDEX    commandIndex    // IN: command index
278  )
279  {
280      return ((s_commandAttributes[commandIndex] & DECRYPT_2) ? 2 :
281              (s_commandAttributes[commandIndex] & DECRYPT_4) ? 4 : 0);
282  }
```

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

```

283  BOOL
284  IsSessionAllowed(
285      COMMAND_INDEX    commandIndex    // IN: the command to be checked
286  )
287  {
288      return ((s_commandAttributes[commandIndex] & NO_SESSIONS) == 0);
289  }
```

9.3.3.10 IsHandleInResponse()

This function determines if a command has a handle in the response

```

290  BOOL
291  IsHandleInResponse(
292      COMMAND_INDEX    commandIndex
293  )
294  {
295      return ((s_commandAttributes[commandIndex] & R_HANDLE) != 0);
296  }
```

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

```

297  BOOL
298  IsWriteOperation(
299      COMMAND_INDEX    commandIndex    // IN: Command to check
300  )
```

```

301 {
302 #ifdef WRITE_LOCK
303     return ((s_commandAttributes[commandIndex] & WRITE_LOCK) != 0);
304 #else
305     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
306     {
307         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
308         {
309             case TPM_CC_NV_Write:
310 #if CC_NV_Increment
311                 case TPM_CC_NV_Increment:
312 #endif
313 #if CC_NV_SetBits
314                 case TPM_CC_NV_SetBits:
315 #endif
316 #if CC_NV_Extend
317                 case TPM_CC_NV_Extend:
318 #endif
319 #if CC_AC_Send
320                 case TPM_CC_AC_Send:
321 #endif
322                 // NV write lock counts as a write operation for authorization purposes.
323                 // We check to see if the NV is write locked before we do the
324                 // authorization. If it is locked, we fail the command early.
325                 case TPM_CC_NV_WriteLock:
326                     return TRUE;
327                 default:
328                     break;
329             }
330         }
331     return FALSE;
332 #endif
333 }

```

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.

```

334 BOOL
335 IsReadOperation(
336     COMMAND_INDEX    commandIndex    // IN: Command to check
337 )
338 {
339 #ifdef READ_LOCK
340     return ((s_commandAttributes[commandIndex] & READ_LOCK) != 0);
341 #else
342     if(!IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
343     {
344         switch(GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex))
345         {
346             case TPM_CC_NV_Read:
347             case TPM_CC_PolicyNV:
348             case TPM_CC_NV_Certify:
349                 // NV read lock counts as a read operation for authorization purposes.
350                 // We check to see if the NV is read locked before we do the
351                 // authorization. If it is locked, we fail the command early.
352                 case TPM_CC_NV_ReadLock:
353                     return TRUE;
354                 default:
355                     break;
356             }
357         }
358     }
359     return FALSE;

```

```

360  #endif
361  }

```

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

```

362  TPMI_YES_NO
363  CommandCapGetCCList(
364      TPM_CC      commandCode,    // IN: start command code
365      UINT32      count,          // IN: maximum count for number of entries in
366                                  // 'commandList'
367      TPML_CCA    *commandList    // OUT: list of TPMA_CC
368  )
369  {
370      TPMI_YES_NO    more = NO;
371      COMMAND_INDEX  commandIndex;
372
373      // initialize output handle list count
374      commandList->count = 0;
375
376      for(commandIndex = GetClosestCommandIndex(commandCode);
377          commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
378          commandIndex = GetNextCommandIndex(commandIndex))
379      {
380  #if !COMPRESSED_LISTS
381      // this check isn't needed for compressed lists.
382      if(!(s_commandAttributes[commandIndex] & IS_IMPLEMENTED))
383          continue;
384  #endif
385      if(commandList->count < count)
386      {
387          // If the list is not full, add the attributes for this command.
388          commandList->commandAttributes[commandList->count]
389              = s_ccAttr[commandIndex];
390          commandList->count++;
391      }
392      else
393      {
394          // If the list is full but there are more commands to report,
395          // indicate this and return.
396          more = YES;
397          break;
398      }
399  }
400  return more;
401  }

```

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

```
402  BOOL  
403  IsVendorCommand(  
404      COMMAND_INDEX    commandIndex    // IN: command index to check  
405      )  
406  {  
407      return (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V));  
408  }
```

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 Returns	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         break;
45     default:
46         // handling of the manufacture specific handles
47         if(((TPM_RH)handle >= TPM_RH_AUTH_00)
48             && ((TPM_RH)handle <= TPM_RH_AUTH_FF))
49             // use the value that would have been returned from
50             // unmarshaling if it did the handle filtering
51             result = TPM_RC_VALUE;
52         else
53             FAIL(FATAL_ERROR_INTERNAL);
54         break;
55     }
56     break;
57 case TPM_HT_TRANSIENT:
58     // For a transient object, check if the handle is associated
59     // with a loaded object.
60     if(!IsObjectPresent(handle))
61         result = TPM_RC_REFERENCE_H0;
62     break;
63 case TPM_HT_PERSISTENT:
64     // Persistent object
65     // Copy the persistent object to RAM and replace the handle with the
66     // handle of the assigned slot. A TPM_RC_OBJECT_MEMORY,
67     // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
68     // ObjectLoadEvict()
69     result = ObjectLoadEvict(&command->handles[i], command->index);
70     break;
71 case TPM_HT_HMAC_SESSION:
72     // For an HMAC session, see if the session is loaded
73     // and if the session in the session slot is actually
74     // an HMAC session.
75     if(SessionIsLoaded(handle))
76     {
77         SESSION *session;
78         session = SessionGet(handle);
79         // Check if the session is a HMAC session
80         if(session->attributes.isPolicy == SET)
81             result = TPM_RC_HANDLE;
82     }
83     else
84         result = TPM_RC_REFERENCE_H0;
85     break;
86 case TPM_HT_POLICY_SESSION:
87     // For a policy session, see if the session is loaded
88     // and if the session in the session slot is actually
89     // a policy session.
90     if(SessionIsLoaded(handle))
91     {
92         SESSION *session;
93         session = SessionGet(handle);
94         // Check if the session is a policy session
95         if(session->attributes.isPolicy == CLEAR)
96             result = TPM_RC_HANDLE;
97     }
98     else
99         result = TPM_RC_REFERENCE_H0;
100    break;
101 case TPM_HT_NV_INDEX:

```

```

102         // For an NV Index, use the TPM-specific routine
103         // to search the IN Index space.
104         result = NvIndexIsAccessible(handle);
105         break;
106     case TPM_HT_PCR:
107         // Any PCR handle that is unmarshaled successfully referenced
108         // a PCR that is defined.
109         break;
110 #if CC_AC_Send
111     case TPM_HT_AC:
112         // Use the TPM-specific routine to search for the AC
113         result = AcIsAccessible(handle);
114         break;
115 #endif
116     default:
117         // Any other handle type is a defect in the unmarshaling code.
118         FAIL(FATAL_ERROR_INTERNAL);
119         break;
120 }
121 if(result != TPM_RC_SUCCESS)
122 {
123     if(result == TPM_RC_REFERENCE_H0)
124         result = result + i;
125     else
126         result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
127     break;
128 }
129 }
130 return result;
131 }

```

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

```

132 UINT16
133 EntityGetAuthValue(
134     TPMI_DH_ENTITY    handle,           // IN: handle of entity
135     TPM2B_AUTH         *auth,           // OUT: authValue of the entity
136 )
137 {
138     TPM2B_AUTH         *pAuth = NULL;
139
140     auth->t.size = 0;
141
142     switch(HandleGetType(handle))
143     {
144     case TPM_HT_PERMANENT:
145     {
146         switch(handle)
147         {
148         case TPM_RH_OWNER:
149             // ownerAuth for TPM_RH_OWNER
150             pAuth = &gp.ownerAuth;
151             break;

```

```

152         case TPM_RH_ENDORSEMENT:
153             // endorsementAuth for TPM_RH_ENDORSEMENT
154             pAuth = &gp.endorsementAuth;
155             break;
156         case TPM_RH_PLATFORM:
157             // platformAuth for TPM_RH_PLATFORM
158             pAuth = &gc.platformAuth;
159             break;
160         case TPM_RH_LOCKOUT:
161             // lockoutAuth for TPM_RH_LOCKOUT
162             pAuth = &gp.lockoutAuth;
163             break;
164         case TPM_RH_NULL:
165             // nullAuth for TPM_RH_NULL. Return 0 directly here
166             return 0;
167             break;
168 #ifdef VENDOR_PERMANENT
169         case VENDOR_PERMANENT:
170             // vendor authorization value
171             pAuth = &g_platformUniqueDetails;
172 #endif
173         default:
174             // If any other permanent handle is present it is
175             // a code defect.
176             FAIL(FATAL_ERROR_INTERNAL);
177             break;
178     }
179     break;
180 }
181 case TPM_HT_TRANSIENT:
182     // authValue for an object
183     // A persistent object would have been copied into RAM
184     // and would have an transient object handle here.
185     {
186         OBJECT *object;
187
188         object = HandleToObject(handle);
189         // special handling if this is a sequence object
190         if(ObjectIsSequence(object))
191         {
192             pAuth = &((HASH_OBJECT *)object)->auth;
193         }
194         else
195         {
196             // Authorization is available only when the private portion of
197             // the object is loaded. The check should be made before
198             // this function is called
199             pAssert(object->attributes.publicOnly == CLEAR);
200             pAuth = &object->sensitive.authValue;
201         }
202     }
203     break;
204 case TPM_HT_NV_INDEX:
205     // authValue for an NV index
206     {
207         NV_INDEX *nvIndex = NvGetIndexInfo(handle, NULL);
208         pAssert(nvIndex != NULL);
209         pAuth = &nvIndex->authValue;
210     }
211     break;
212 case TPM_HT_PCR:
213     // authValue for PCR
214     pAuth = PCRGetAuthValue(handle);
215     break;
216 default:
217     // If any other handle type is present here, then there is a defect

```

```

218         // in the unmarshaling code.
219         FAIL(FATAL_ERROR_INTERNAL);
220         break;
221     }
222     // Copy the authValue
223     MemoryCopy2B(&auth->b, &spAuth->b, sizeof(auth->t.buffer));
224     MemoryRemoveTrailingZeros(auth);
225     return auth->t.size;
226 }

```

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.

```

227 TPMI_ALG_HASH
228 EntityGetAuthPolicy(
229     TPMI_DH_ENTITY    handle,           // IN: handle of entity
230     TPM2B_DIGEST      *authPolicy      // OUT: authPolicy of the entity
231 )
232 {
233     TPMI_ALG_HASH      hashAlg = TPM_ALG_NULL;
234     authPolicy->t.size = 0;
235
236     switch(HandleGetType(handle))
237     {
238     case TPM_HT_PERMANENT:
239         switch(handle)
240         {
241         case TPM_RH_OWNER:
242             // ownerPolicy for TPM_RH_OWNER
243             *authPolicy = gp.ownerPolicy;
244             hashAlg = gp.ownerAlg;
245             break;
246         case TPM_RH_ENDORSEMENT:
247             // endorsementPolicy for TPM_RH_ENDORSEMENT
248             *authPolicy = gp.endorsementPolicy;
249             hashAlg = gp.endorsementAlg;
250             break;
251         case TPM_RH_PLATFORM:
252             // platformPolicy for TPM_RH_PLATFORM
253             *authPolicy = gc.platformPolicy;
254             hashAlg = gc.platformAlg;
255             break;
256         case TPM_RH_LOCKOUT:
257             // lockoutPolicy for TPM_RH_LOCKOUT
258             *authPolicy = gp.lockoutPolicy;
259             hashAlg = gp.lockoutAlg;
260             break;
261         default:
262             return TPM_ALG_ERROR;
263             break;
264         }
265         break;
266     case TPM_HT_TRANSIENT:
267         // authPolicy for an object
268         {
269             OBJECT *object = HandleToObject(handle);

```

```

270     *authPolicy = object->publicArea.authPolicy;
271     hashAlg = object->publicArea.nameAlg;
272 }
273 break;
274 case TPM_HT_NV_INDEX:
275     // authPolicy for a NV index
276     {
277         NV_INDEX      *nvIndex = NvGetIndexInfo(handle, NULL);
278         pAssert(nvIndex != 0);
279         *authPolicy = nvIndex->publicArea.authPolicy;
280         hashAlg = nvIndex->publicArea.nameAlg;
281     }
282 break;
283 case TPM_HT_PCR:
284     // authPolicy for a PCR
285     hashAlg = PCRGetAuthPolicy(handle, authPolicy);
286     break;
287 default:
288     // If any other handle type is present it is a code defect.
289     FAIL(FATAL_ERROR_INTERNAL);
290     break;
291 }
292 return hashAlg;
293 }

```

9.4.3.4 EntityGetName()

This function returns the Name associated with a handle.

```

294 TPM2B_NAME *
295 EntityGetName(
296     TPMI_DH_ENTITY    handle,           // IN: handle of entity
297     TPM2B_NAME        *name            // OUT: name of entity
298 )
299 {
300     switch(HandleGetType(handle))
301     {
302     case TPM_HT_TRANSIENT:
303     {
304         // Name for an object
305         OBJECT      *object = HandleToObject(handle);
306         // an object with no nameAlg has no name
307         if(object->publicArea.nameAlg == TPM_ALG_NULL)
308             name->b.size = 0;
309         else
310             *name = object->name;
311         break;
312     }
313     case TPM_HT_NV_INDEX:
314         // Name for a NV index
315         NvGetNameByIndexHandle(handle, name);
316         break;
317     default:
318         // For all other types, the handle is the Name
319         name->t.size = sizeof(TPM_HANDLE);
320         UINT32_TO_BYTE_ARRAY(handle, name->t.name);
321         break;
322     }
323     return name;
324 }

```


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.

```

325 TPMI_RH_HIERARCHY
326 EntityGetHierarchy(
327     TPMI_DH_ENTITY    handle           // IN :handle of entity
328 )
329 {
330     TPMI_RH_HIERARCHY    hierarchy = TPM_RH_NULL;
331
332     switch(HandleGetType(handle))
333     {
334         case TPM_HT_PERMANENT:
335             // hierarchy for a permanent handle
336             switch(handle)
337             {
338                 case TPM_RH_PLATFORM:
339                 case TPM_RH_ENDORSEMENT:
340                 case TPM_RH_NULL:
341                     hierarchy = handle;
342                     break;
343                 // all other permanent handles are associated with the owner
344                 // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
345                 default:
346                     hierarchy = TPM_RH_OWNER;
347                     break;
348             }
349             break;
350         case TPM_HT_NV_INDEX:
351             // hierarchy for NV index
352             {
353                 NV_INDEX    *nvIndex = NvGetIndexInfo(handle, NULL);
354                 pAssert(nvIndex != NULL);
355
356                 // If only the platform can delete the index, then it is
357                 // considered to be in the platform hierarchy, otherwise it
358                 // is in the owner hierarchy.
359                 if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV,
360                                PLATFORMCREATE))
361                     hierarchy = TPM_RH_PLATFORM;
362                 else
363                     hierarchy = TPM_RH_OWNER;
364             }
365             break;
366         case TPM_HT_TRANSIENT:
367             // hierarchy for an object
368             {
369                 OBJECT    *object;
370                 object = HandleToObject(handle);
371                 if(object->attributes.ppsHierarchy)
372                 {
373                     hierarchy = TPM_RH_PLATFORM;
374                 }
375                 else if(object->attributes.epsHierarchy)
376                 {
377                     hierarchy = TPM_RH_ENDORSEMENT;
378                 }
379                 else if(object->attributes.spsHierarchy)

```

```
380         {
381             hierarchy = TPM_RH_OWNER;
382         }
383     }
384     break;
385     case TPM_HT_PCR:
386         hierarchy = TPM_RH_OWNER;
387         break;
388     default:
389         FAIL(FATAL_ERROR_INTERNAL);
390         break;
391 }
392 // this is unreachable but it provides a return value for the default
393 // case which makes the compiler happy
394 return hierarchy;
395 }
```

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"
```

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                 return inHandle;
36                 break;
37             default:
38                 break;
39         }
40     }
```

```

39     }
40 }
41 // Out of range on the top
42 return 0;
43 }

```

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

```

44 TPMI_YES_NO
45 PermanentCapGetHandles(
46     TPM_HANDLE    handle,          // IN: start handle
47     UINT32         count,          // IN: count of returned handles
48     TPML_HANDLE    *handleList     // OUT: list of handle
49 )
50 {
51     TPMI_YES_NO    more = NO;
52     UINT32         i;
53
54     pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
55
56     // Initialize output handle list
57     handleList->count = 0;
58
59     // The maximum count of handles we may return is MAX_CAP_HANDLES
60     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
61
62     // Iterate permanent handle range
63     for(i = NextPermanentHandle(handle);
64         i != 0; i = NextPermanentHandle(i + 1))
65     {
66         if(handleList->count < count)
67         {
68             // If we have not filled up the return list, add this permanent
69             // handle to it
70             handleList->handle[handleList->count] = i;
71             handleList->count++;
72         }
73         else
74         {
75             // If the return list is full but we still have permanent handle
76             // available, report this and stop iterating
77             more = YES;
78             break;
79         }
80     }
81     return more;
82 }

```

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

```

83  TPMI_YES_NO
84  PermanentHandleGetPolicy(
85      TPM_HANDLE      handle,          // IN: start handle
86      UINT32          count,          // IN: max count of returned handles
87      TPML_TAGGED_POLICY *policyList  // OUT: list of handle
88  )
89  {
90      TPMI_YES_NO      more = NO;
91
92      pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
93
94      // Initialize output handle list
95      policyList->count = 0;
96
97      // The maximum count of policies we may return is MAX_TAGGED_POLICIES
98      if(count > MAX_TAGGED_POLICIES)
99          count = MAX_TAGGED_POLICIES;
100
101      // Iterate permanent handle range
102      for(handle = NextPermanentHandle(handle);
103          handle != 0;
104          handle = NextPermanentHandle(handle + 1))
105      {
106          TPM2B_DIGEST  policyDigest;
107          TPM_ALG_ID     policyAlg;
108          // Check to see if this permanent handle has a policy
109          policyAlg = EntityGetAuthPolicy(handle, &policyDigest);
110          if(policyAlg == TPM_ALG_ERROR)
111              continue;
112          if(policyList->count < count)
113          {
114              // If we have not filled up the return list, add this
115              // policy to the list;
116              policyList->policies[policyList->count].handle = handle;
117              policyList->policies[policyList->count].policyHash.hashAlg = policyAlg;
118              MemoryCopy(&policyList->policies[policyList->count].policyHash.digest,
119                      policyDigest.t.buffer, policyDigest.t.size);
120              policyList->count++;
121          }
122          else
123          {
124              // If the return list is full but we still have permanent handle
125              // available, report this and stop iterating
126              more = YES;
127              break;
128          }
129      }
130      return more;
131  }

```

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
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         TpmSizeChecks();
16     #endif
17
18     // If TPM has been manufactured, return indication.
19     if(!firstTime && g_manufactured)
20         return 1;
21
22     // Do power on initializations of the cryptographic libraries.
23     CryptInit();
24
25     s_DAPendingOnNV = FALSE;
26
27     // initialize NV
28     NvManufacture();
29
30     // Clear the magic value in the DRBG state
31     go.drbgState.magic = 0;
32
33     CryptStartup(SU_RESET);
34
35     // default configuration for PCR
36     PCRSimStart();
```

```

37
38 // initialize pre-installed hierarchy data
39 // This should happen after NV is initialized because hierarchy data is
40 // stored in NV.
41 HierarchyPreInstall_Init();
42
43 // initialize dictionary attack parameters
44 DAPreInstall_Init();
45
46 // initialize PP list
47 PhysicalPresencePreInstall_Init();
48
49 // initialize command audit list
50 CommandAuditPreInstall_Init();
51
52 // first start up is required to be Startup(CLEAR)
53 orderlyShutdown = TPM_SU_CLEAR;
54 NV_WRITE_PERSISTENT(orderlyState, orderlyShutdown);
55
56 // initialize the firmware version
57 gp.firmwareV1 = FIRMWARE_V1;
58 #ifdef FIRMWARE_V2
59 gp.firmwareV2 = FIRMWARE_V2;
60 #else
61 gp.firmwareV2 = 0;
62 #endif
63 NV_SYNC_PERSISTENT(firmwareV1);
64 NV_SYNC_PERSISTENT(firmwareV2);
65
66 // initialize the total reset counter to 0
67 gp.totalResetCount = 0;
68 NV_SYNC_PERSISTENT(totalResetCount);
69
70 // initialize the clock stuff
71 go.clock = 0;
72 go.clockSafe = YES;
73
74 NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
75
76 // Commit NV writes. Manufacture process is an artificial process existing
77 // only in simulator environment and it is not defined in the specification
78 // that what should be the expected behavior if the NV write fails at this
79 // point. Therefore, it is assumed the NV write here is always success and
80 // no return code of this function is checked.
81 NvCommit();
82
83 g_manufactured = TRUE;
84
85 return 0;
86 }

```

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

```

87  LIB_EXPORT int
88  TPM_TearDown(
89      void
90  )
91  {
92      g_manufactured = FALSE;
93      return 0;
94  }

```

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.

```

95  LIB_EXPORT void
96  TpmEndSimulation(
97      void
98  )
99  {
100  #if SIMULATION
101      HashLibSimulationEnd();
102      SymLibSimulationEnd();
103      MathLibSimulationEnd();
104  #if ALG_RSA
105      RsaSimulationEnd();
106  #endif
107  #if ALG_ECC
108      EccSimulationEnd();
109  #endif
110  #endif // SIMULATION
111  }

```

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 section 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 <i>nameAlg</i> of the object. NOTE shall be the Empty Buffer if no authorization policy is present
[type]parameters	TPMU_PUBLIC_PARMS	the algorithm or structure details
[type]unique	TPMU_PUBLIC_ID	the unique identifier of the structure For an asymmetric key, this would be the public key.

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

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

```

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;
```

9.11 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 Returns	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 Returns	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 }
```

9.11.2.7 IsNumeric()

Verifies that all the characters are simple numeric (0-9)

```
169  BOOL
170  IsNumeric(
171      TPM2B      *value
172  )
173  {
174      UINT16      i;
175      for(i = 0; i < value->size; i++)
176      {
177          if(value->buffer[i] < '0' || value->buffer[i] > '9')
178              return FALSE;
179      }
180      return TRUE;
181  }
```

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 an 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 = {{0}};
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 = {{0}};
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 = {{0}};
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 typedef struct
12 {
13     BYTE          tag[sizeof(TPM_ST)];
14     BYTE          size[sizeof(UINT32)];
15     BYTE          code[sizeof(TPM_RC)];
16 } PACKED_HEADER;
17 typedef struct
18 {
19     BYTE          size[sizeof(UINT16)];
20     struct
21     {
22         BYTE      function[sizeof(UINT32)];
23         BYTE      line[sizeof(UINT32)];
24         BYTE      code[sizeof(UINT32)];
25     } values;
26     BYTE          returnCode[sizeof(TPM_RC)];
27 } GET_TEST_RESULT_PARAMETERS;
28 typedef struct
29 {
30     BYTE          moreData[sizeof(TPMI_YES_NO)];
31     BYTE          capability[sizeof(TPM_CAP)]; // Always TPM_CAP_TPM_PROPERTIES
32     BYTE          tpmProperty[sizeof(TPML_TAGGED_TPM_PROPERTY)];
33 } GET_CAPABILITY_PARAMETERS;
34 typedef struct
35 {
36     BYTE          header[sizeof(PACKED_HEADER)];
37     BYTE          getTestResult[sizeof(GET_TEST_RESULT_PARAMETERS)];
38 } TEST_RESPONSE;
39 typedef struct
40 {
41     BYTE          header[sizeof(PACKED_HEADER)];
42     BYTE          getCap[sizeof(GET_CAPABILITY_PARAMETERS)];
43 } CAPABILITY_RESPONSE;
44 typedef union
45 {

```

```

46     BYTE          test[sizeof(TEST_RESPONSE)];
47     BYTE          cap[sizeof(CAPABILITY_RESPONSE)];
48 } 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.

```

49 #ifndef __IGNORE_STATE__ // Don't define this value
50 static BYTE response[sizeof(RESPONSES)];
51 #endif

```

9.17.3 Local Functions

9.17.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```

52 static INT32
53 MarshalUint16(
54     UINT16          integer,
55     BYTE            **buffer
56 )
57 {
58     UINT16_TO_BYTE_ARRAY(integer, *buffer);
59     *buffer += 2;
60     return 2;
61 }

```

9.17.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```

62 static INT32
63 MarshalUint32(
64     UINT32          integer,
65     BYTE            **buffer
66 )
67 {
68     UINT32_TO_BYTE_ARRAY(integer, *buffer);
69     *buffer += 4;
70     return 4;
71 }

```

9.17.3.3 Unmarshal32()

```

72 static BOOL Unmarshal32(
73     UINT32          *target,
74     BYTE            **buffer,
75     INT32           *size
76 )
77 {
78     if((*size -= 4) < 0)
79         return FALSE;
80     *target = BYTE_ARRAY_TO_UINT32(*buffer);
81     *buffer += 4;
82     return TRUE;

```

83 }

9.17.3.4 Unmarshal16()

```

84 static BOOL Unmarshal16(
85     UINT16      *target,
86     BYTE        **buffer,
87     INT32       *size
88 )
89 {
90     if((*size -= 2) < 0)
91         return FALSE;
92     *target = BYTE_ARRAY_TO_UINT16(*buffer);
93     *buffer += 2;
94     return TRUE;
95 }
```

9.17.4 Public Functions

9.17.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```

96 #if SIMULATION
97 LIB_EXPORT void
98 SetForceFailureMode(
99     void
100 )
101 {
102     g_forceFailureMode = TRUE;
103     return;
104 }
105 #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.

```

106 void
107 TpmLogFailure(
108 #if FAIL_TRACE
109     const char *function,
110     int line,
111 #endif
112     int code
113 )
114 {
115     // Save the values that indicate where the error occurred.
116     // On a 64-bit machine, this may truncate the address of the string
117     // of the function name where the error occurred.
118 #if FAIL_TRACE
119     s_failFunction = (UINT32)function;
120     s_failLine = line;
121 #else
122     s_failFunction = 0;
123     s_failLine = 0;
124 #endif
125     s_failCode = code;
126
127     // We are in failure mode
```

```

128     g_inFailureMode = TRUE;
129
130     return;
131 }

```

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().

```

132 NORETURN void
133 TpmFail(
134     #if FAIL_TRACE
135     const char    *function,
136     int           line,
137     #endif
138     int           code
139 )
140 {
141     // Save the values that indicate where the error occurred.
142     // On a 64-bit machine, this may truncate the address of the string
143     // of the function name where the error occurred.
144     #if FAIL_TRACE
145     s_failFunction = (UINT32)function;
146     s_failLine = line;
147     #else
148     s_failFunction = (UINT32)NULL;
149     s_failLine = 0;
150     #endif
151     s_failCode = code;
152
153     // We are in failure mode
154     g_inFailureMode = TRUE;
155
156     // if asserts are enabled, then do an assert unless the failure mode code
157     // is being tested.
158     #if SIMULATION
159     #   ifndef NDEBUG
160     assert(g_forceFailureMode);
161     #   endif
162     // Clear this flag
163     g_forceFailureMode = FALSE;
164     #endif
165     // Jump to the failure mode code.
166     // Note: only get here if asserts are off or if we are testing failure mode
167     _plat__Fail();
168 }

```

9.17.4.4 TpmFailureMode

This function is called by the interface code when the platform is in failure mode.

```

169 void
170 TpmFailureMode(
171     unsigned int    inRequestSize,    // IN: command buffer size
172     unsigned char   *inRequest,      // IN: command buffer
173     unsigned int    *outResponseSize, // OUT: response buffer size
174     unsigned char   **outResponse    // OUT: response buffer
175 )
176 {
177     UINT32    marshalSize;
178     UINT32    capability;
179     HEADER    header;    // unmarshaled command header

```

```

180     UINT32          pt;    // unmarshaled property type
181     UINT32          count; // unmarshaled property count
182     UINT8           *buffer = inRequest;
183     INT32           size = inRequestSize;
184
185     // If there is no command buffer, then just return TPM_RC_FAILURE
186     if(inRequestSize == 0 || inRequest == NULL)
187         goto FailureModeReturn;
188     // If the header is not correct for TPM2_GetCapability() or
189     // TPM2_GetTestResult() then just return the in failure mode response;
190     if(! (Unmarshal16(&header.tag, &buffer, &size)
191          && Unmarshal32(&header.size, &buffer, &size)
192          && Unmarshal32(&header.code, &buffer, &size)))
193         goto FailureModeReturn;
194     if(header.tag != TPM_ST_NO_SESSIONS
195        || header.size < 10)
196         goto FailureModeReturn;
197     switch(header.code)
198     {
199     case TPM_CC_GetTestResult:
200         // make sure that the command size is correct
201         if(header.size != 10)
202             goto FailureModeReturn;
203         buffer = &response[10];
204         marshalSize = MarshalUint16(3 * sizeof(UINT32), &buffer);
205         marshalSize += MarshalUint32(s_failFunction, &buffer);
206         marshalSize += MarshalUint32(s_failLine, &buffer);
207         marshalSize += MarshalUint32(s_failCode, &buffer);
208         if(s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
209             marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
210         else
211             marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
212         break;
213     case TPM_CC_GetCapability:
214         // make sure that the size of the command is exactly the size
215         // returned for the capability, property, and count
216         if(header.size != (10 + (3 * sizeof(UINT32))))
217             // also verify that this is requesting TPM properties
218             || !Unmarshal32(&capability, &buffer, &size)
219             || capability != TPM_CAP_TPM_PROPERTIES
220             || !Unmarshal32(&pt, &buffer, &size)
221             || !Unmarshal32(&count, &buffer, &size))
222             goto FailureModeReturn;
223         // If in failure mode because of an unrecoverable read error, and the
224         // property is 0 and the count is 0, then this is an indication to
225         // re-manufacture the TPM. Do the re-manufacture but stay in failure
226         // mode until the TPM is reset.
227         // Note: this behavior is not required by the specification and it is
228         // OK to leave the TPM permanently bricked due to an unrecoverable NV
229         // error.
230         if(count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
231         {
232             g_manufactured = FALSE;
233             TPM_Manufacture(0);
234         }
235         if(count > 0)
236             count = 1;
237         else if(pt > TPM_PT_FIRMWARE_VERSION_2)
238             count = 0;
239         if(pt < TPM_PT_MANUFACTURER)
240             pt = TPM_PT_MANUFACTURER;
241         // set up for return
242         buffer = &response[10];
243         // if the request was for a PT less than the last one
244         // then we indicate more, otherwise, not.
245         if(pt < TPM_PT_FIRMWARE_VERSION_2)

```

```

246         *buffer++ = YES;
247     else
248         *buffer++ = NO;
249     marshalSize = 1;
250
251     // indicate the capability type
252     marshalSize += MarshalUInt32(capability, &buffer);
253     // indicate the number of values that are being returned (0 or 1)
254     marshalSize += MarshalUInt32(count, &buffer);
255     // indicate the property
256     marshalSize += MarshalUInt32(pt, &buffer);
257
258     if(count > 0)
259         switch(pt)
260         {
261             case TPM_PT_MANUFACTURER:
262                 // the vendor ID unique to each TPM manufacturer
263             #ifdef MANUFACTURER
264                 pt = *(UINT32*)MANUFACTURER;
265             #else
266                 pt = 0;
267             #endif
268                 break;
269             case TPM_PT_VENDOR_STRING_1:
270                 // the first four characters of the vendor ID string
271             #ifdef VENDOR_STRING_1
272                 pt = *(UINT32*)VENDOR_STRING_1;
273             #else
274                 pt = 0;
275             #endif
276                 break;
277             case TPM_PT_VENDOR_STRING_2:
278                 // the second four characters of the vendor ID string
279             #ifdef VENDOR_STRING_2
280                 pt = *(UINT32*)VENDOR_STRING_2;
281             #else
282                 pt = 0;
283             #endif
284                 break;
285             case TPM_PT_VENDOR_STRING_3:
286                 // the third four characters of the vendor ID string
287             #ifdef VENDOR_STRING_3
288                 pt = *(UINT32*)VENDOR_STRING_3;
289             #else
290                 pt = 0;
291             #endif
292                 break;
293             case TPM_PT_VENDOR_STRING_4:
294                 // the fourth four characters of the vendor ID string
295             #ifdef VENDOR_STRING_4
296                 pt = *(UINT32*)VENDOR_STRING_4;
297             #else
298                 pt = 0;
299             #endif
300                 break;
301             case TPM_PT_VENDOR_TPM_TYPE:
302                 // vendor-defined value indicating the TPM model
303                 // We just make up a number here
304                 pt = 1;
305                 break;
306             case TPM_PT_FIRMWARE_VERSION_1:
307                 // the more significant 32-bits of a vendor-specific value
308                 // indicating the version of the firmware
309             #ifdef FIRMWARE_V1
310                 pt = FIRMWARE_V1;
311             #else

```

```

312             pt = 0;
313 #endif
314             break;
315         default: // TPM_PT_FIRMWARE_VERSION_2:
316             // the less significant 32-bits of a vendor-specific value
317             // indicating the version of the firmware
318 #ifdef FIRMWARE_V2
319             pt = FIRMWARE_V2;
320 #else
321             pt = 0;
322 #endif
323             break;
324     }
325     marshalSize += MarshalUint32(pt, &buffer);
326     break;
327     default: // default for switch (cc)
328         goto FailureModeReturn;
329 }
330 // Now do the header
331 buffer = response;
332 marshalSize = marshalSize + 10; // Add the header size to the
333                                // stuff already marshaled
334 MarshalUint16(TPM_ST_NO_SESSIONS, &buffer); // structure tag
335 MarshalUint32(marshalSize, &buffer); // responseSize
336 MarshalUint32(TPM_RC_SUCCESS, &buffer); // response code
337
338 *outResponseSize = marshalSize;
339 *outResponse = (unsigned char *)&response;
340 return;
341 FailureModeReturn:
342 buffer = response;
343 marshalSize = MarshalUint16(TPM_ST_NO_SESSIONS, &buffer);
344 marshalSize += MarshalUint32(10, &buffer);
345 marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
346 *outResponseSize = marshalSize;
347 *outResponse = (unsigned char *)response;
348 return;
349 }

```

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...

```

350 void
351 UnmarshalFail(
352     void            *type,
353     BYTE            **buffer,
354     INT32           *size
355 )
356 {
357     NOT_REFERENCED(type);
358     NOT_REFERENCED(buffer);
359     NOT_REFERENCED(size);
360     FAIL(FATAL_ERROR_INTERNAL);
361 }

```


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  #if RADIX_BITS == 64
4  # define RADIX_LOG2 6
5  #elif RADIX_BITS == 32
6  #define RADIX_LOG2 5
7  #else
8  # error "Unsupported radix"
9  #endif
10 #define RADIX_MOD(x) ((x) & ((1 << RADIX_LOG2) - 1))
11 #define RADIX_DIV(x) ((x) >> RADIX_LOG2)
12 #define RADIX_MASK (((crypt_uword_t)1) << RADIX_LOG2) - 1)
13 #define BITS_TO_CRYPT_WORDS(bits) RADIX_DIV((bits) + (RADIX_BITS - 1))
14 #define BYTES_TO_CRYPT_WORDS(bytes) BITS_TO_CRYPT_WORDS(bytes * 8)
15 #define SIZE_IN_CRYPT_WORDS(thing) BYTES_TO_CRYPT_WORDS(sizeof(thing))
16 #if RADIX_BITS == 64
17 #define SWAP_CRYPT_WORD(x) REVERSE_ENDIAN_64(x)
18     typedef uint64_t crypt_uword_t;
19     typedef int64_t crypt_word_t;
20 # define TO_CRYPT_WORD_64 BIG_ENDIAN_BYTES_TO_UINT64
21 # define TO_CRYPT_WORD_32(a, b, c, d) TO_CRYPT_WORD_64(0, 0, 0, 0, a, b, c, d)
22 #elif RADIX_BITS == 32
23 # define SWAP_CRYPT_WORD(x) REVERSE_ENDIAN_32((x))
24     typedef uint32_t crypt_uword_t;
25     typedef int32_t crypt_word_t;
26 # define TO_CRYPT_WORD_64(a, b, c, d, e, f, g, h) \
27     BIG_ENDIAN_BYTES_TO_UINT32(e, f, g, h), \
28     BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d)
29 #endif
30 #define MAX_CRYPT_UWORD (~(crypt_uword_t)0)
31 #define MAX_CRYPT_WORD ((crypt_word_t)(MAX_CRYPT_UWORD >> 1))
32 #define MIN_CRYPT_WORD (~MAX_CRYPT_WORD)
33 #define LARGEST_NUMBER (MAX((ALG_RSA * MAX_RSA_KEY_BYTES), \
34     MAX((ALG_ECC * MAX_ECC_KEY_BYTES), MAX_DIGEST_SIZE)))
35 #define LARGEST_NUMBER_BITS (LARGEST_NUMBER * 8)
36 #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 to much difficulty. For the math performed using these numbers, the value is always positive.

```

37  #define BN_STRUCT_DEF(count) struct {      \
38      crypt_ushort_t    allocated;          \
39      crypt_ushort_t    size;               \
40      crypt_ushort_t    d[count];           \
41  }
42  typedef BN_STRUCT_DEF(1) bignum_t;
43  #ifndef bigNum
44  typedef bignum_t      *bigNum;
45  typedef const bignum_t *bigConst;
46  #endif
47  extern const bignum_t  BnConstZero;

```

The Functions to access the properties of a big number. Get number of allocated words

```

48  #define BnGetAllocated(x)    (unsigned) ((x)->allocated)

```

Get number of words used

```

49  #define BnGetSize(x)        ((x)->size)

```

Get a pointer to the data array

```

50  #define BnGetArray(x)       ((crypt_ushort_t *) &((x)->d[0]))

```

Get the nth word of a BIGNUM (zero-based)

```

51  #define BnGetWord(x, i)     (crypt_ushort_t) ((x)->d[i])

```

Some things that are done often. Test to see if a bignum_t is equal to zero

```

52  #define BnEqualZero(bn)     (BnGetSize(bn) == 0)

```

Test to see if a bignum_t is equal to a word type

```

53  #define BnEqualWord(bn, word) \
54      ((BnGetSize(bn) == 1) && (BnGetWord(bn, 0) == (crypt_ushort_t)word))

```

Determine if a BIGNUM is even. A zero is even. Although the indication that a number is zero is that it's size is zero, all words of the number are 0 so this test works on zero.

```

55  #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*.

```

56  #define BN_INIT(name) \
57      (bigNum) BnInit((bigNum) &(name), \
58      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_*.

```

59  #define BN_ADDRESS(name) (bigNum) &name##_
60  #define BN_STRUCT_ALLOCATION(bits) (BITS_TO_CRYPT_WORDS(bits) + 1)

```

Create a structure of the correct size.

```
61 #define BN_STRUCT(bits) \
62     BN_STRUCT_DEF(BN_STRUCT_ALLOCATION(bits))
```

Define a BIGNUM type with a specific allocation

```
63 #define BN_TYPE(name, bits) \
64     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.

```
65 #define BN_INITIALIZED(name, bits, initializer) \
66     BN_STRUCT(bits) name##_; \
67     bigNum name = BnFrom2B(BN_INIT(name##_), \
68                             (const TPM2B *)initializer)
```

Create a local variable that can hold a number with *bits*

```
69 #define BN_VAR(name, bits) \
70     BN_STRUCT(bits) _##name; \
71     bigNum name = BN_INIT(_##name)
```

Create a type that can hold the largest number defined by the implementation.

```
72 #define BN_MAX(name) BN_VAR(name, LARGEST_NUMBER_BITS) \
73 #define BN_MAX_INITIALIZED(name, initializer) \
74     BN_INITIALIZED(name, LARGEST_NUMBER_BITS, initializer)
```

A word size value is useful

```
75 #define BN_WORD(name) BN_VAR(name, RADIX_BITS)
```

This is used to created a word-size BIGNUM and initialize it with an input parameter to a function.

```
76 #define BN_WORD_INITIALIZED(name, initial) \
77     BN_STRUCT(RADIX_BITS) name##_; \
78     bigNum name = BnInitializeWord((bigNum)&name##_, \
79                                     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

```
80 typedef struct _bn_point_t
81 {
82     bigNum x;
83     bigNum y;
84     bigNum z;
85 } bn_point_t;
86 typedef bn_point_t *bigPoint;
87 typedef const bn_point_t *pointConst;
88 typedef struct constant_point_t
89 {
90     bigConst x;
91     bigConst y;
92     bigConst z;
93 } constant_point_t;
94 #define ECC_BITS (MAX_ECC_KEY_BYTES * 8)
```

```

95 BN_TYPE(ecc, ECC_BITS);
96 #define ECC_NUM(name) BN_VAR(name, ECC_BITS)
97 #define ECC_INITIALIZED(name, initializer) \
98     BN_INITIALIZED(name, ECC_BITS, initializer)
99 #define POINT_INSTANCE(name, bits) \
100     BN_STRUCT (bits) name##_x = \
101         {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}}; \
102     BN_STRUCT ( bits ) name##_y = \
103         {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}}; \
104     BN_STRUCT ( bits ) name##_z = \
105         {BITS_TO_CRYPT_WORDS ( bits ), 0,{0}}; \
106     bn_point_t name##_
107 #define POINT_INITIALIZER(name) \
108     BnInitializePoint(&name##_, (bigNum)&name##_x, \
109         (bigNum)&name##_y, (bigNum)&name##_z)
110 #define POINT_INITIALIZED(name, initValue) \
111     POINT_INSTANCE(name, MAX_ECC_KEY_BITS); \
112     bigPoint name = BnPointFrom2B( \
113         POINT_INITIALIZER(name), \
114         initValue)
115 #define POINT_VAR(name, bits) \
116     POINT_INSTANCE (name, bits); \
117     bigPoint name = POINT_INITIALIZER(name)
118 #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

```

119 typedef struct
120 {
121     bigConst prime; // a prime number
122     bigConst order; // the order of the curve
123     bigConst h; // cofactor
124     bigConst a; // linear coefficient
125     bigConst b; // constant term
126     constant_point_t base; // base point
127 } 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.

```

128 #define CurveGetPrime(C) ((C)->prime)
129 #define CurveGetOrder(C) ((C)->order)
130 #define CurveGetCofactor(C) ((C)->h)
131 #define CurveGet_a(C) ((C)->a)
132 #define CurveGet_b(C) ((C)->b)
133 #define CurveGetG(C) ((pointConst)&((C)->base))
134 #define CurveGetGx(C) ((C)->base.x)
135 #define CurveGetGy(C) ((C)->base.y)

```

Convert bytes in initializers according to the endianness of the system. This is used for CryptEccData.c.

```

136 #define BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d) \
137     ( ((UINT32)(a) << 24) \
138     + ((UINT32)(b) << 16) \
139     + ((UINT32)(c) << 8) \
140     + ((UINT32)(d)) )
141
142 #define BIG_ENDIAN_BYTES_TO_UINT64(a, b, c, d, e, f, g, h) \
143     ( ((UINT64)(a) << 56) \
144     + ((UINT64)(b) << 48) \
145     + ((UINT64)(c) << 40) \
146     + ((UINT64)(d) << 32) )

```

```
147         +      ((UINT64) (e) << 24)           \
148         +      ((UINT64) (f) << 16)           \
149         +      ((UINT64) (g) << 8)            \
150         +      ((UINT64) (h) )                \
151     )
152 #ifndef RADIX_BYTES
153 #   if RADIX_BITS == 32
154 #       define RADIX_BYTES 4
155 #   elif RADIX_BITS == 64
156 #       define RADIX_BYTES 8
157 #   else
158 #       error "RADIX_BITS must either be 32 or 64"
159 #   endif
160 #endif
```

Add implementation dependent definitions for other ECC Values and for linkages.

```
161 #include LIB_INCLUDE(MATH_LIB, Math)
162 #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

This is used to define the macro that may or may not be in the data set for the curve (BnEccData.c). If there is a mismatch, the compiler will warn that there is too much/not enough initialization data in the curve. The macro is used because not all versions of the CryptEccData.c need the curve name.

```
3  #ifndef NAMED_CURVES
4  #define CURVE_NAME(a) , a
5  #define CURVE_NAME_DEF const char *name;
6  #else
7  # define CURVE_NAME(a)
8  # define CURVE_NAME_DEF
9  #endif
10 typedef struct ECC_CURVE
11 {
12     const TPM_ECC_CURVE      curveId;
13     const UINT16              keySizeBits;
14     const TPMT_KDF_SCHEME     kdf;
15     const TPMT_ECC_SCHEME     sign;
16     const ECC_CURVE_DATA      *curveData; // the address of the curve data
17     const BYTE                *OID;
18     CURVE_NAME_DEF
19 } ECC_CURVE;
20 extern const ECC_CURVE eccCurves[ECC_CURVE_COUNT];
21 #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  typedef UINT16(* SMAC_END_METHOD) (
10     union SMAC_STATES      *state,
11     UINT32                  size,
12     BYTE                    *buffer
13 );
14 typedef struct sequenceMethods {
15     SMAC_DATA_METHOD      data;
16     SMAC_END_METHOD       end;
17 } SMAC_METHODS;
18 #define SMAC_IMPLEMENTED (CC_MAC || CC_MAC_Start)
```

These definitions are here because the SMAC state is in the union of hash states.

```
19 typedef struct tpmCmacState {
20     TPM_ALG_ID      symAlg;
21     UINT16           keySizeBits;
22     INT16            bcount; // current count of bytes accumulated in IV
23     TPM2B_IV         iv;    // IV buffer
24     TPM2B_SYM_KEY     symKey;
25 } tpmCmacState_t;
26 typedef union SMAC_STATES {
27     #if ALG_CMIC
28     tpmCmacState_t      cmic;
29     #endif
30     UINT64              pad;
31 } SMAC_STATES;
32 typedef struct SMAC_STATE {
33     SMAC_METHODS        smacMethods;
34     SMAC_STATES          state;
35 } SMAC_STATE;
36 typedef union
37 {
38     #if ALG_SHA1
39     tpmHashStateSHA1_t      Sha1;
40     #endif
41     #if ALG_SHA256
42     tpmHashStateSHA256_t    Sha256;
43     #endif
```



```

44 #if ALG_SHA384
45     tpmHashStateSHA384_t      Sha384;
46 #endif
47 #if ALG_SHA512
48     tpmHashStateSHA512_t      Sha512;
49 #endif
50
51 // Additions for symmetric block cipher MAC
52 #if SMAC_IMPLEMENTED
53     SMAC_STATE                 smac;
54 #endif
55 // to force structure alignment to be no worse than HASH_ALIGNMENT
56 #if HASH_ALIGNMENT == 4
57     uint32_t                   align;
58 #else
59     uint64_t                   align;
60 #endif
61 } ANY_HASH_STATE;
62 typedef ANY_HASH_STATE *PANY_HASH_STATE;
63 typedef const ANY_HASH_STATE *PCANY_HASH_STATE;
64 #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

```

65 #define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
66 #define MAX_HASH_STATE_SIZE_ALIGNED
67     ALIGNED_SIZE(MAX_HASH_STATE_SIZE, HASH_ALIGNMENT)

```

This is an aligned byte array that will hold any of the hash contexts.

```

68 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

```

69 #ifndef HASH_START_METHOD_DEF
70 #   define HASH_START_METHOD_DEF    void (HASH_START_METHOD) (void)
71 #endif
72 #ifndef HASH_DATA_METHOD_DEF
73 #   define HASH_DATA_METHOD_DEF     void (HASH_DATA_METHOD) (void)
74 #endif
75 #ifndef HASH_END_METHOD_DEF
76 #   define HASH_END_METHOD_DEF      void (HASH_END_METHOD) (void)
77 #endif
78 #ifndef HASH_STATE_COPY_METHOD_DEF
79 #   define HASH_STATE_COPY_METHOD_DEF void (HASH_STATE_COPY_METHOD) (void)
80 #endif
81 #ifndef HASH_STATE_EXPORT_METHOD_DEF
82 #   define HASH_STATE_EXPORT_METHOD_DEF void (HASH_STATE_EXPORT_METHOD) (void)
83 #endif
84 #ifndef HASH_STATE_IMPORT_METHOD_DEF
85 #   define HASH_STATE_IMPORT_METHOD_DEF void (HASH_STATE_IMPORT_METHOD) (void)
86 #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.

```

87 typedef HASH_START_METHOD_DEF;
88 typedef HASH_DATA_METHOD_DEF;

```

```

89  typedef HASH_END_METHOD_DEF;
90  typedef HASH_STATE_COPY_METHOD_DEF;
91  typedef HASH_STATE_EXPORT_METHOD_DEF;
92  typedef HASH_STATE_IMPORT_METHOD_DEF;
93  typedef struct _HASH_METHODS
94  {
95      HASH_START_METHOD      *start;
96      HASH_DATA_METHOD      *data;
97      HASH_END_METHOD        *end;
98      HASH_STATE_COPY_METHOD *copy;      // Copy a hash block
99      HASH_STATE_EXPORT_METHOD *copyOut; // Copy a hash block from a hash
100                                     // context
101      HASH_STATE_IMPORT_METHOD *copyIn;  // Copy a hash block to a proper hash
102                                     // context
103  } HASH_METHODS, *PHASH_METHODS;
104  #if ALG_SHA1
105      TPM2B_TYPE(SHA1_DIGEST, SHA1_DIGEST_SIZE);
106  #endif
107  #if ALG_SHA256
108      TPM2B_TYPE(SHA256_DIGEST, SHA256_DIGEST_SIZE);
109  #endif
110  #if ALG_SHA384
111      TPM2B_TYPE(SHA384_DIGEST, SHA384_DIGEST_SIZE);
112  #endif
113  #if ALG_SHA512
114      TPM2B_TYPE(SHA512_DIGEST, SHA512_DIGEST_SIZE);
115  #endif
116  #if ALG_SM3_256
117      TPM2B_TYPE(SM3_256_DIGEST, SM3_256_DIGEST_SIZE);
118  #endif

```

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.

```

119  #if ALG_RSA
120  #define PKCS1_HASH_REF    const BYTE *PKCS1;
121  #define PKCS1_OID(NAME)  , OID_PKCS1_##NAME
122  #else
123  #define PKCS1_HASH_REF
124  #define PKCS1_OID(NAME)
125  #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.

```

126  #if ALG_ECDSA
127  #define ECDSA_HASH_REF    const BYTE *ECDSA;
128  #define ECDSA_OID(NAME)  , OID_ECDSA_##NAME
129  #else
130  #define ECDSA_HASH_REF
131  #define ECDSA_OID(NAME)
132  #endif
133  typedef const struct HASH_DEF
134  {
135      HASH_METHODS      method;
136      uint16_t           blockSize;
137      uint16_t           digestSize;
138      uint16_t           contextSize;
139      uint16_t           hashAlg;
140      const BYTE         *OID;
141      PKCS1_HASH_REF     // PKCS1 OID
142      ECDSA_HASH_REF     // ECDSA OID
143  } 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.

```

144 #define HASH_DEF_TEMPLATE(HASH, Hash)
145 \
146     HASH_DEF    Hash##_Def= {
147         { (HASH_START_METHOD *) &tpmHashStart_##HASH,
148         (HASH_DATA_METHOD *) &tpmHashData_##HASH,
149         (HASH_END_METHOD *) &tpmHashEnd_##HASH,
150         (HASH_STATE_COPY_METHOD *) &tpmHashStateCopy_##HASH,
151         (HASH_STATE_EXPORT_METHOD *) &tpmHashStateExport_##HASH,
152         (HASH_STATE_IMPORT_METHOD *) &tpmHashStateImport_##HASH,
153         },
154         HASH##_BLOCK_SIZE,      /*block size */
155         HASH##_DIGEST_SIZE,     /*data size */
156         sizeof(tpmHashState_##HASH##_t),
157         TPM_ALG_##HASH, OID_##HASH
158         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.

```

158 typedef BYTE    HASH_STATE_TYPE;
159 #define HASH_STATE_EMPTY    ((HASH_STATE_TYPE) 0)
160 #define HASH_STATE_HASH    ((HASH_STATE_TYPE) 1)
161 #define HASH_STATE_HMAC    ((HASH_STATE_TYPE) 2)
162 #if CC_MAC || CC_MAC_Start
163 #define HASH_STATE_SMAC    ((HASH_STATE_TYPE) 3)
164 #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.

```

165 typedef struct _HASH_STATE
166 {
167     HASH_STATE_TYPE    type;                // type of the context
168     TPM_ALG_ID          hashAlg;
169     HASH_DEF            def;
170     ANY_HASH_STATE      state;
171 } HASH_STATE, *PHASH_STATE;
172 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.

```

173 typedef struct hmacState
174 {
175     HASH_STATE          hashState;          // the hash state
176     TPM2B_HASH_BLOCK    hmacKey;           // the HMAC key
177 } 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. There is a run time check that makes sure that this i.

```
178 typedef struct
179 {
180     BYTE                                buffer[sizeof(HASH_STATE)];
181 } EXPORT_HASH_STATE, *PEXPORT_HASH_STATE;
182 typedef const EXPORT_HASH_STATE *PCEXPORT_HASH_STATE;
183 #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,

- a) the key size
- b) the block size (the IV size)
- c) 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  typedef tpmKeyScheduleAES      DRBG_KEY_SCHEDULE;
7  #define DRBG_ENCRYPT_SETUP(key, keySizeInBits, schedule) \
8      TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule) \
9  #define DRBG_ENCRYPT(keySchedule, in, out) \
10     TpmCryptEncryptAES(SWIZZLE(keySchedule, in, out))
11 #if ((DRBG_KEY_SIZE_BITS % RADIX_BITS) != 0) \
12 || ((DRBG_IV_SIZE_BITS % RADIX_BITS) != 0)
13 #error "Key size and IV for DRBG must be even multiples of the radix"
14 #endif
15 #if (DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
16 #error "Key size for DRBG must be even multiple of the cypher block size"
17 #endif
```

Derived values

```
18 #define DRBG_MAX_REQUESTS_PER RESEED (1 << 48)
19 #define DRBG_MAX_REQUEST_SIZE (1 << 32)
20 #define pDRBG_KEY(seed) ((DRBG_KEY *) &(((BYTE *) (seed))[0]))
21 #define pDRBG_IV(seed) ((DRBG_IV *) &(((BYTE *) (seed))[DRBG_KEY_SIZE_BYTES]))
22 #define DRBG_KEY_SIZE_WORDS (BITS_TO_CRYPT_WORDS(DRBG_KEY_SIZE_BITS))
23 #define DRBG_KEY_SIZE_BYTES (DRBG_KEY_SIZE_WORDS * RADIX_BYTES)
24 #define DRBG_IV_SIZE_WORDS (BITS_TO_CRYPT_WORDS(DRBG_IV_SIZE_BITS))
25 #define DRBG_IV_SIZE_BYTES (DRBG_IV_SIZE_WORDS * RADIX_BYTES)
26 #define DRBG_SEED_SIZE_WORDS (DRBG_KEY_SIZE_WORDS + DRBG_IV_SIZE_WORDS)
27 #define DRBG_SEED_SIZE_BYTES (DRBG_KEY_SIZE_BYTES + DRBG_IV_SIZE_BYTES)
28 typedef union
29 {
30     BYTE          bytes[DRBG_KEY_SIZE_BYTES];
31     crypt_uword_t words[DRBG_KEY_SIZE_WORDS];
32 } DRBG_KEY;
33 typedef union
34 {
35     BYTE          bytes[DRBG_IV_SIZE_BYTES];
36     crypt_uword_t words[DRBG_IV_SIZE_WORDS];
37 } DRBG_IV;
38 typedef union
39 {
```

```

40     BYTE          bytes[DRBG_SEED_SIZE_BYTES];
41     crypt_uword_t  words[DRBG_SEED_SIZE_WORDS];
42 } DRBG_SEED;
43 #define CTR_DRBG_MAX_REQUESTS_PER_RESEED      ((UINT64)1 << 20)
44 #define CTR_DRBG_MAX_BYTES_PER_REQUEST        (1 << 16)
45 #define CTR_DRBG_MIN_ENTROPY_INPUT_LENGTH      DRBG_SEED_SIZE_BYTES
46 #define CTR_DRBG_MAX_ENTROPY_INPUT_LENGTH      DRBG_SEED_SIZE_BYTES
47 #define CTR_DRBG_MAX_ADDITIONAL_INPUT_LENGTH  DRBG_SEED_SIZE_BYTES
48 #define TESTING          (1 << 0)
49 #define ENTROPY          (1 << 1)
50 #define TESTED           (1 << 2)
51 #define IsTestStateSet(BIT) ((g_cryptoSelfTestState.rng & BIT) != 0)
52 #define SetTestStateBit(BIT) (g_cryptoSelfTestState.rng |= BIT)
53 #define ClearTestStateBit(BIT) (g_cryptoSelfTestState.rng &= ~BIT)
54 #define IsSelfTest()      IsTestStateSet(TESTING)
55 #define SetSelfTest()     SetTestStateBit(TESTING)
56 #define ClearSelfTest()   ClearTestStateBit(TESTING)
57 #define IsEntropyBad()    IsTestStateSet(ENTROPY)
58 #define SetEntropyBad()   SetTestStateBit(ENTROPY)
59 #define ClearEntropyBad() ClearTestStateBit(ENTROPY)
60 #define IsDrbgTested()    IsTestStateSet(TESTED)
61 #define SetDrbgTested()   SetTestStateBit(TESTED)
62 #define ClearDrbgTested() ClearTestStateBit(TESTED)
63 typedef struct
64 {
65     UINT64      reseedCounter;
66     UINT32      magic;
67     DRBG_SEED   seed; // contains the key and IV for the counter mode DRBG
68     UINT32      lastValue[4]; // used when the TPM does continuous self-test
69                                // for FIPS compliance of DRBG
70 } DRBG_STATE, *pDRBG_STATE;
71 #define DRBG_MAGIC ((UINT32) 0x47425244) // "DRBG" backwards so that it displays
72 typedef struct
73 {
74     UINT64      counter;
75     UINT32      magic;
76     UINT32      limit;
77     TPM2B       *seed;
78     const TPM2B *label;
79     TPM2B       *context;
80     TPM_ALG_ID   hash;
81     TPM_ALG_ID   kdf;
82     UINT16       digestSize;
83     TPM2B_DIGEST residual;
84 } KDF_STATE, *pKDF_STATE;
85 #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

```

86 typedef union
87 {
88     DRBG_STATE drbg;
89     KDF_STATE  kdf;
90 } 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.

```

91 extern RAND_STATE      *s_random;

```

When instrumenting RSA key sieve

```
92  #if RSA_INSTRUMENT
93  #define PRIME_INDEX(x) ((x) == 512 ? 0 : (x) == 1024 ? 1 : 2)
94  #   define INSTRUMENT_SET(a, b) ((a) = (b))
95  #   define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
96  #   define INSTRUMENT_INC(a)    (a) = (a) + 1
97  extern UINT32 PrimeIndex;
98  extern UINT32 failedAtIteration[10];
99  extern UINT32 PrimeCounts[3];
100 extern UINT32 MillerRabinTrials[3];
101 extern UINT32 totalFieldsSieved[3];
102 extern UINT32 bitsInFieldAfterSieve[3];
103 extern UINT32 emptyFieldsSieved[3];
104 extern UINT32 noPrimeFields[3];
105 extern UINT32 primesChecked[3];
106 extern UINT16 lastSievePrime;
107 #else
108 #   define INSTRUMENT_SET(a, b)
109 #   define INSTRUMENT_ADD(a, b)
110 #   define INSTRUMENT_INC(a)
111 #endif
112 #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  #define BN_PRIME(name) BN_VAR(name, (MAX_RSA_KEY_BITS / 2))
8  BN_TYPE(prime, (MAX_RSA_KEY_BITS / 2));
9  #define BN_PRIME_INITIALIZED(name, initializer) \
10     BN_INITIALIZED(name, MAX_RSA_KEY_BITS / 2, initializer)
11 #if !CRT_FORMAT_RSA
12 #   error This version only works with CRT formatted data
13 #endif // !CRT_FORMAT_RSA
14 typedef struct privateExponent
15 {
16     bigNum      P;
17     bigNum      Q;
18     bigNum      dP;
19     bigNum      dQ;
20     bigNum      qInv;
21     bn_prime_t  entries[5];
22 } privateExponent;
23 #define NEW_PRIVATE_EXPONENT(X) \
24     privateExponent _##X; \
25     privateExponent *X = RsaInitializeExponent(&(_##X))
26 #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 ALG_LAST_VALUE is a multiple of 8, ALGORITHM_VECTOR will need to have byte for the single bit in the last byte. So, for example, when ALG_LAST_VECTOR is 8, ALGORITHM_VECTOR will need 2 bytes.

```
3  #define ALGORITHM_VECTOR_BYTES ((ALG_LAST_VALUE + 8) / 8)
4  typedef BYTE    ALGORITHM_VECTOR[ALGORITHM_VECTOR_BYTES];
5  #ifdef TEST_SELF_TEST
6  LIB_EXPORT extern ALGORITHM_VECTOR    LibToTest;
7  #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.

```
8  typedef struct
9  {
10     UINT32    rng;
11     UINT32    hash;
12     UINT32    sym;
13     #if ALG_RSA
14     UINT32    rsa;
15     #endif
16     #if ALG_ECC
17     UINT32    ecc;
18     #endif
19 } CRYPTO_SELF_TEST_STATE;
20 #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,0xe3,0x26,0x73,0xd0,0x17,0x5b,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  TPM2B_TYPE(HASH_TEST_DATA, 256); // Twice the largest block size
13  TPM2B_HASH_TEST_DATA      c_hashTestData = {{256, {
14     0x88,0xac,0xc3,0xe5,0x5f,0x66,0x9d,0x18,0x80,0xc9,0x7a,0x9c,0xa4,0x08,0x90,0x98,
15     0x0f,0x3a,0x53,0x92,0x4c,0x67,0x4e,0xb7,0x37,0xec,0x67,0x87,0xb6,0xbe,0x10,0xca,
16     0x11,0x5b,0x4a,0x0b,0x45,0xc3,0x32,0x68,0x48,0x69,0xce,0x25,0x1b,0xc8,0xaf,0x44,
17     0x79,0x22,0x83,0xc8,0xfb,0xe2,0x63,0x94,0xa2,0x3c,0x59,0x3e,0x3e,0xc6,0x64,0x2c,
18     0x1f,0x8c,0x11,0x93,0x24,0xa3,0x17,0xc5,0x2f,0x37,0xcf,0x95,0x97,0x8e,0x63,0x39,
19     0x68,0xd5,0xca,0xba,0x18,0x37,0x69,0x6e,0x4f,0x19,0xfd,0x8a,0xc0,0x8d,0x87,0x3a,
20     0xbc,0x31,0x42,0x04,0x05,0xef,0xb5,0x02,0xef,0x1e,0x92,0x4b,0xb7,0x73,0x2c,0x8c,
21     0xeb,0x23,0x13,0x81,0x34,0xb9,0xb5,0xc1,0x17,0x37,0x39,0xf8,0x3e,0xe4,0x4c,0x06,
22     0xa8,0x81,0x52,0x2f,0xef,0xc9,0x9c,0x69,0x89,0xbc,0x85,0x9c,0x30,0x16,0x02,0xca,
23     0xe3,0x61,0xd4,0x0f,0xed,0x34,0x1b,0xca,0xc1,0x1b,0xd1,0xfa,0xc1,0xa2,0xe0,0xdf,
24     0x52,0x2f,0x0b,0x4b,0x9f,0x0e,0x45,0x54,0xb9,0x17,0xb6,0xaf,0xd6,0xd5,0xca,0x90,
25     0x29,0x57,0x7b,0x70,0x50,0x94,0x5c,0x8e,0xf6,0x4e,0x21,0x8b,0xc6,0x8b,0xa6,0xbc,
26     0xb9,0x64,0xd4,0x4d,0xf3,0x68,0xd8,0xac,0xde,0xd8,0xd8,0xb5,0x6d,0xcd,0x93,0xeb,
27     0x28,0xa4,0xe2,0x5c,0x44,0xef,0xf0,0xe1,0x6f,0x38,0x1a,0x3c,0xe6,0xef,0xa2,0x9d,
28     0xb9,0xa8,0x05,0x2a,0x95,0xec,0x5f,0xdb,0xb0,0x25,0x67,0x9c,0x86,0x7a,0x8e,0xea,
29     0x51,0xcc,0xc3,0xd3,0xff,0x6e,0xf0,0xed,0xa3,0xae,0xf9,0x5d,0x33,0x70,0xf2,0x11
30     }}};
31  #if ALG_SHA1 == YES
32  TPM2B_TYPE(SHA1, 20);
33  TPM2B_SHA1      c_SHA1_digest = {{20, {
34     0xee,0x2c,0xef,0x93,0x76,0xbd,0xf8,0x91,0xbc,0xe6,0xe5,0x57,0x53,0x77,0x01,0xb5,
35     0x70,0x95,0xe5,0x40
36     }}};
37  #endif
38  #if ALG_SHA256 == YES
39  TPM2B_TYPE(SHA256, 32);
40  TPM2B_SHA256      c_SHA256_digest = {{32, {
41     0x64,0xe8,0xe0,0xc3,0xa9,0xa4,0x51,0x49,0x10,0x55,0x8d,0x31,0x71,0xe5,0x2f,0x69,
42     0x3a,0xdc,0xc7,0x11,0x32,0x44,0x61,0xbd,0x34,0x39,0x57,0xb0,0xa8,0x75,0x86,0x1b
43     }}};
44  #endif
45  #if ALG_SHA384 == YES
46  TPM2B_TYPE(SHA384, 48);
47  TPM2B_SHA384      c_SHA384_digest = {{48, {
48     0x37,0x75,0x29,0xb5,0x20,0x15,0x6e,0xa3,0x7e,0xa3,0x0d,0xcd,0x80,0xa8,0xa3,0x3d,
49     0xeb,0xe8,0xad,0x4e,0x1c,0x77,0x94,0x5a,0xaf,0x6c,0xd0,0xc1,0xfa,0x43,0x3f,0xc7,
50     0xb8,0xf1,0x01,0xc0,0x60,0xbf,0xf2,0x87,0xe8,0x71,0x9e,0x51,0x97,0xa0,0x09,0x8d
51     }}};
52  #endif
53  #if ALG_SHA512 == YES
54  TPM2B_TYPE(SHA512, 64);
55  TPM2B_SHA512      c_SHA512_digest = {{64, {
56     0xe2,0x7b,0x10,0x3d,0x5e,0x48,0x58,0x44,0x67,0xac,0xa3,0x81,0x8c,0x1d,0xc5,0x71,
57     0x66,0x92,0x8a,0x89,0xaa,0xd4,0x35,0x51,0x60,0x37,0x31,0xd7,0xba,0xe7,0x93,0x0b,
58     0x16,0x4d,0xb3,0xc8,0x34,0x98,0x3c,0xd3,0x53,0xde,0x5e,0xe8,0x0c,0xbc,0xaf,0xc9,
59     0x24,0x2c,0xcc,0xed,0xdb,0xde,0xba,0x1f,0x14,0x14,0x5a,0x95,0x80,0xde,0x66,0xbd
60     }}};
61  #endif

```

10.1.8 KdfTestData.h

Hash Test Vectors

```

1  #define TEST_KDF_KEY_SIZE 20
2  TPM2B_TYPE(KDF_TEST_KEY, TEST_KDF_KEY_SIZE);
3  TPM2B_KDF_TEST_KEY c_kdfTestKeyIn = {{TEST_KDF_KEY_SIZE, {
4      0x27, 0x1F, 0xA0, 0x8B, 0xBD, 0xC5, 0x06, 0x0E, 0xC3, 0xDF,
5      0xA9, 0x28, 0xFF, 0x9B, 0x73, 0x12, 0x3A, 0x12, 0xDA, 0x0C }}};
6  TPM2B_TYPE(KDF_TEST_LABEL, 17);
7  TPM2B_KDF_TEST_LABEL c_kdfTestLabel = {{17, {
8      0x4B, 0x44, 0x46, 0x53, 0x45, 0x4C, 0x46, 0x54,
9      0x45, 0x53, 0x54, 0x4C, 0x41, 0x42, 0x45, 0x4C, 0x00 }}};
10 TPM2B_TYPE(KDF_TEST_CONTEXT, 8);
11 TPM2B_KDF_TEST_CONTEXT c_kdfTestContextU = {{8, {
12     0xCE, 0x24, 0x4F, 0x39, 0x5D, 0xCA, 0x73, 0x91 }}};
13 TPM2B_KDF_TEST_CONTEXT c_kdfTestContextV = {{8, {
14     0xDA, 0x50, 0x40, 0x31, 0xDD, 0xF1, 0x2E, 0x83 }}};
15 #if ALG_SHA512 == ALG_YES
16     TPM2B_KDF_TEST_KEY c_kdfTestKeyOut = {{20, {
17         0x8b, 0xe2, 0xc1, 0xb8, 0x5b, 0x78, 0x56, 0x9b, 0x9f, 0xa7,
18         0x59, 0xf5, 0x85, 0x7c, 0x56, 0xd6, 0x84, 0x81, 0x0f, 0xd3 }}};
19     #define KDF_TEST_ALG TPM_ALG_SHA512
20 #elif ALG_SHA384 == ALG_YES
21     TPM2B_KDF_TEST_KEY c_kdfTestKeyOut = {{20, {
22         0x1d, 0xce, 0x70, 0xc9, 0x11, 0x3e, 0xb2, 0xdb, 0xa4, 0x7b,
23         0xd9, 0xcf, 0xc7, 0x2b, 0xf4, 0x6f, 0x45, 0xb0, 0x93, 0x12 }}};
24     #define KDF_TEST_ALG TPM_ALG_SHA384
25 #elif ALG_SHA256 == ALG_YES
26     TPM2B_KDF_TEST_KEY c_kdfTestKeyOut = {{20, {
27         0xbb, 0x02, 0x59, 0xe1, 0xc8, 0xba, 0x60, 0x7e, 0x6a, 0x2c,
28         0xd7, 0x04, 0xb6, 0x9a, 0x90, 0x2e, 0x9a, 0xde, 0x84, 0xc4 }}};
29     #define KDF_TEST_ALG TPM_ALG_SHA256
30 #elif ALG_SHA1 == ALG_YES
31     TPM2B_KDF_TEST_KEY c_kdfTestKeyOut = {{20, {
32         0x55, 0xb5, 0xa7, 0x18, 0x4a, 0xa0, 0x74, 0x23, 0xc4, 0x7d,
33         0xae, 0x76, 0x6c, 0x26, 0xa2, 0x37, 0x7d, 0x7c, 0xf8, 0x51 }}};
34     #define KDF_TEST_ALG TPM_ALG_SHA1
35 #endif

```

10.1.9 RsaTestData.h

RSA Test Vectors

```

1  #define RSA_TEST_KEY_SIZE    256
2  typedef struct
3  {
4      UINT16        size;
5      BYTE          buffer[RSA_TEST_KEY_SIZE];
6  } TPM2B_RSA_TEST_KEY;
7  typedef TPM2B_RSA_TEST_KEY  TPM2B_RSA_TEST_VALUE;
8  typedef struct
9  {
10     UINT16        size;
11     BYTE          buffer[RSA_TEST_KEY_SIZE / 2];
12 } TPM2B_RSA_TEST_PRIME;
13 const TPM2B_RSA_TEST_KEY    c_rsaPublicModulus = {256, {
14     0x91,0x12,0xf5,0x07,0x9d,0x5f,0x6b,0x1c,0x90,0xf6,0xcc,0x87,0xde,0x3a,0x7a,0x15,
15     0xdc,0x54,0x07,0x6c,0x26,0x8f,0x25,0xef,0x7e,0x66,0xc0,0xe3,0x82,0x12,0x2f,0xab,
16     0x52,0x82,0x1e,0x85,0xbc,0x53,0xba,0x2b,0x01,0xad,0x01,0xc7,0x8d,0x46,0x4f,0x7d,
17     0xdd,0x7e,0xdc,0xb0,0xad,0xf6,0x0c,0xa1,0x62,0x92,0x97,0x8a,0x3e,0x6f,0x7e,0x3e,
18     0xf6,0x9a,0xcc,0xf9,0xa9,0x86,0x77,0xb6,0x85,0x43,0x42,0x04,0x13,0x65,0xe2,0xad,
19     0x36,0xc9,0xbf,0xc1,0x97,0x84,0x6f,0xee,0x7c,0xda,0x58,0xd2,0xae,0x07,0x00,0xaf,
20     0xc5,0x5f,0x4d,0x3a,0x98,0xb0,0xed,0x27,0x7c,0xc2,0xc9,0x26,0x5d,0x87,0xe1,0xe3,
21     0xa9,0x69,0x88,0x4f,0x8c,0x08,0x31,0x18,0xae,0x93,0x16,0xe3,0x74,0xde,0xd3,0xf6,
22     0x16,0xaf,0xa3,0xac,0x37,0x91,0x8d,0x10,0xc6,0x6b,0x64,0x14,0x3a,0xd9,0xfc,0xe4,
23     0xa0,0xf2,0xd1,0x01,0x37,0x4f,0x4a,0xeb,0xe5,0xec,0x98,0xc5,0xd9,0x4b,0x30,0xd2,
24     0x80,0x2a,0x5a,0x18,0x5a,0x7d,0xd4,0x3d,0xb7,0x62,0x98,0xce,0x6d,0xa2,0x02,0x6e,
25     0x45,0xaa,0x95,0x73,0xe0,0xaa,0x75,0x57,0xb1,0x3d,0x1b,0x05,0x75,0x23,0x6b,0x20,
26     0x69,0x9e,0x14,0xb0,0x7f,0xac,0xae,0xd2,0xc7,0x48,0x3b,0xe4,0x56,0x11,0x34,0x1e,
27     0x05,0x1a,0x30,0x20,0xef,0x68,0x93,0x6b,0x9d,0x7e,0xdd,0xba,0x96,0x50,0xcc,0x1c,
28     0x81,0xb4,0x59,0xb9,0x74,0x36,0xd9,0x97,0xdc,0x8f,0x17,0x82,0x72,0xb3,0x59,0xf6,
29     0x23,0xfa,0x84,0xf7,0x6d,0xf2,0x05,0xff,0xf1,0xb9,0xcc,0xe9,0xa2,0x82,0x01,0xfb}};
30 const TPM2B_RSA_TEST_PRIME c_rsaPrivatePrime = {RSA_TEST_KEY_SIZE / 2, {
31     0xb7,0xa0,0x90,0xc7,0x92,0x09,0xde,0x71,0x03,0x37,0x4a,0xb5,0x2f,0xda,0x61,0xb8,
32     0x09,0x1b,0xba,0x99,0x70,0x45,0xc1,0x0b,0x15,0x12,0x71,0x8a,0xb3,0x2a,0x4d,0x5a,
33     0x41,0x9b,0x73,0x89,0x80,0x0a,0x8f,0x18,0x4c,0x8b,0xa2,0x5b,0xda,0xbd,0x43,0xbe,
34     0xdc,0x76,0x4d,0x71,0x0f,0xb9,0xfc,0x7a,0x09,0xfe,0x4f,0xac,0x63,0xd9,0x2e,0x50,
35     0x3a,0xa1,0x37,0xc6,0xf2,0xa1,0x89,0x12,0xe7,0x72,0x64,0x2b,0xba,0xc1,0x1f,0xca,
36     0x9d,0xb7,0xaa,0x3a,0xa9,0xd3,0xa6,0x6f,0x73,0x02,0xbb,0x85,0x5d,0x9a,0xb9,0x5c,
37     0x08,0x83,0x22,0x20,0x49,0x91,0x5f,0x4b,0x86,0xbc,0x3f,0x76,0x43,0x08,0x97,0xbf,
38     0x82,0x55,0x36,0x2d,0x8b,0x6e,0x9e,0xfb,0xc1,0x67,0x6a,0x43,0xa2,0x46,0x81,0x71}};
39 const BYTE c_RsaTestValue[RSA_TEST_KEY_SIZE] = {
40     0x2a,0x24,0x3a,0xbb,0x50,0x1d,0xd4,0x2a,0xf9,0x18,0x32,0x34,0xa2,0x0f,0xea,0x5c,
41     0x91,0x77,0xe9,0xe1,0x09,0x83,0xdc,0x5f,0x71,0x64,0x5b,0xeb,0x57,0x79,0xa0,0x41,
42     0xc9,0xe4,0x5a,0x0b,0xf4,0x9f,0xdb,0x84,0x04,0xa6,0x48,0x24,0xf6,0x3f,0x66,0x1f,
43     0xa8,0x04,0x5c,0xf0,0x7a,0x6b,0x4a,0x9c,0x7e,0x21,0xb6,0xda,0x6b,0x65,0x9c,0x3a,
44     0x68,0x50,0x13,0x1e,0xa4,0xb7,0xca,0xec,0xd3,0xcc,0xb2,0x9b,0x8c,0x87,0xa4,0x6a,
45     0xba,0xc2,0x06,0x3f,0x40,0x48,0x7b,0xa8,0xb8,0x2c,0x03,0x14,0x33,0xf3,0x1d,0xe9,
46     0xbd,0x6f,0x54,0x66,0xb4,0x69,0x5e,0xbc,0x80,0x7c,0xe9,0x6a,0x43,0x7f,0xb8,0x6a,
47     0xa0,0x5f,0x5d,0x7a,0x20,0xfd,0x7a,0x39,0xe1,0xea,0x0e,0x94,0x91,0x28,0x63,0x7a,
48     0xac,0xc9,0xa5,0x3a,0x6d,0x31,0x7b,0x7c,0x54,0x56,0x99,0x56,0xbb,0xb7,0xa1,0x2d,
49     0xd2,0x5c,0x91,0x5f,0x1c,0xd3,0x06,0x7f,0x34,0x53,0x2f,0x4c,0xd1,0x8b,0xd2,0x9e,
50     0xdc,0xc3,0x94,0x0a,0xe1,0x0f,0xa5,0x15,0x46,0x2a,0x8e,0x10,0xc2,0xfe,0xb7,0x5e,
51     0x2d,0x0d,0xd1,0x25,0xfc,0xe4,0xf7,0x02,0x19,0xfe,0xb6,0xe4,0x95,0x9c,0x17,0x4a,
52     0x9b,0xdb,0xab,0xc7,0x79,0xe3,0x5e,0x40,0xd0,0x56,0x6d,0x25,0x0a,0x72,0x65,0x80,
53     0x92,0x9a,0xa8,0x07,0x32,0x14,0xfb,0xfe,0x08,0xeb,0x13,0xb4,0x07,0x68,0xb4,
54     0x58,0x39,0xbe,0x8e,0x78,0x3a,0x59,0x3f,0x9c,0x4c,0xe9,0xa8,0x64,0x68,0xf7,0xb9,
55     0x6e,0x20,0xf5,0xcb,0xca,0x47,0xf2,0x17,0xaa,0x8b,0xbc,0x13,0x14,0x84,0xf6,0xab};
56 const TPM2B_RSA_TEST_VALUE c_RsaepKvt = {RSA_TEST_KEY_SIZE, {
57     0x73,0xbd,0x65,0x49,0xda,0x7b,0xb8,0x50,0x9e,0x87,0xf0,0x0a,0x8a,0x9a,0x07,0xb6,
58     0x00,0x82,0x10,0x14,0x60,0xd8,0x01,0xfc,0xc5,0x18,0xea,0x49,0x5f,0x13,0xcf,0x65,
59     0x66,0x30,0x6c,0x60,0x3f,0x24,0x3c,0xfb,0xe2,0x31,0x16,0x99,0x7e,0x31,0x98,0xab,
60     0x93,0xb8,0x07,0x53,0xcc,0xdb,0xf7,0x44,0xd9,0xee,0x5d,0xe8,0x5f,0x97,0x5f,0xe8,
61     0x1f,0x88,0x52,0x24,0x7b,0xac,0x62,0x95,0xb7,0x7d,0xf5,0xf8,0x9f,0x5a,0xa8,0x24,

```

```

62     0x9a,0x76,0x71,0x2a,0x35,0x2a,0xa1,0x08,0xbb,0x95,0xe3,0x64,0xdc,0xdb,0xc2,0x33,
63     0xa9,0x5f,0xbe,0x4c,0xc4,0xcc,0x28,0xc9,0x25,0xff,0xee,0x17,0x15,0x9a,0x50,0x90,
64     0x0e,0x15,0xb4,0xea,0x6a,0x09,0xe6,0xff,0xa4,0xee,0xc7,0x7e,0xce,0xa9,0x73,0xe4,
65     0xa0,0x56,0xbd,0x53,0x2a,0xe4,0xc0,0x2b,0xa8,0x9b,0x09,0x30,0x72,0x62,0x0f,0xf9,
66     0xf6,0xa1,0x52,0xd2,0x8a,0x37,0xee,0xa5,0xc8,0x47,0xe1,0x99,0x21,0x47,0xeb,0xdd,
67     0x37,0xaa,0xe4,0xbd,0x55,0x46,0x5a,0x5a,0x5d,0xfb,0x7b,0xfc,0xff,0xbf,0x26,0x71,
68     0xf6,0x1e,0xad,0xbc,0xbf,0x33,0xca,0xe1,0x92,0x8f,0x2a,0x89,0x6c,0x45,0x24,0xd1,
69     0xa6,0x52,0x56,0x24,0x5e,0x90,0x47,0xe5,0xcb,0x12,0xb0,0x32,0xf9,0xa6,0xbb,0xea,
70     0x37,0xa9,0xbd,0xef,0x23,0xef,0x63,0x07,0x6c,0xc4,0x4e,0x64,0x3c,0xc6,0x11,0x84,
71     0x7d,0x65,0xd6,0x7a,0x17,0x58,0xa5,0xf7,0x74,0x3b,0x42,0xe3,0xd2,0xda,0x5f,
72     0x6f,0xe0,0x1e,0x4b,0xcf,0x46,0xe2,0xdf,0x3e,0x41,0x8e,0x0e,0xb0,0x3f,0x8b,0x65}};
73 #define     OAEP_TEST_LABEL         "OAEP Test Value"
74 #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
75 const TPM2B_RSA_TEST_VALUE     c_OaepKvt = {RSA_TEST_KEY_SIZE, {
76     0x32,0x68,0x84,0x0b,0x9c,0xc9,0x25,0x26,0xd9,0xc0,0xd0,0xb1,0xde,0x60,0x55,0xae,
77     0x33,0xe5,0xcf,0x6c,0x85,0xbe,0xd,0x71,0x11,0xe1,0x45,0x60,0xbb,0x42,0x3d,0xf3,
78     0xb1,0x18,0x84,0x7b,0xc6,0xc6,0x5d,0xce,0x1d,0x5f,0x9a,0x97,0xcf,0xb1,0x97,0x9a,0x85,
79     0x7c,0xa7,0xa1,0x63,0x23,0xb6,0x74,0x0f,0x1a,0xee,0x29,0x51,0xeb,0x50,0x8f,0x3c,
80     0x8e,0x4e,0x31,0x38,0xdc,0x11,0xfc,0x9a,0x4e,0xaf,0x93,0xc9,0x7f,0x6e,0x35,0xf3,
81     0xc9,0xe4,0x89,0x14,0x53,0xe2,0xc2,0x1a,0xf7,0x6b,0x9b,0xf0,0x7a,0xa4,0x69,0x52,
82     0xe0,0x24,0x8f,0xea,0x31,0xa7,0x5c,0x43,0xb0,0x65,0xc9,0xfe,0xba,0xfe,0x80,0x9e,
83     0xa5,0xc0,0xf5,0x8d,0xce,0x41,0xf9,0x83,0x0d,0x8e,0x0f,0xef,0x3d,0x1f,0x6a,0xcc,
84     0x8a,0x3d,0x3b,0xdf,0x22,0x38,0xd7,0x34,0x58,0x7b,0x55,0xc9,0xf6,0xbc,0x7c,0x4c,
85     0x3f,0xd7,0xde,0x4e,0x30,0xa9,0x69,0xf3,0x5f,0x56,0x8f,0xc2,0xe7,0x75,0x79,0xb8,
86     0xa5,0xc8,0x0d,0xc0,0xcd,0xb6,0xc9,0x63,0xad,0x7c,0xe4,0x8f,0x39,0x60,0x4d,0x7d,
87     0xdb,0x34,0x49,0x2a,0x47,0xde,0xc0,0x42,0x4a,0x19,0x94,0x2e,0x50,0x21,0x03,0x47,
88     0xff,0x73,0xb3,0xb7,0x89,0xcc,0x7b,0x2c,0xeb,0x03,0xa7,0x9a,0x06,0xfd,0xed,0x19,
89     0xbb,0x82,0xa0,0x13,0xe9,0xfa,0xac,0x06,0x5f,0xc5,0xa9,0x2b,0xda,0x88,0x23,0xa2,
90     0x5d,0xc2,0x7f,0xda,0xc8,0x5a,0x94,0x31,0xc1,0x21,0xd7,0x1e,0x6b,0xd7,0x89,0xb1,
91     0x93,0x80,0xab,0xd1,0x37,0xf2,0x6f,0x50,0xc0,0x2a,0xea,0xb1,0xc4,0xcd,0xcb,0xb5}};
92 const TPM2B_RSA_TEST_VALUE     c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
93     0x29,0xa4,0x2f,0xbb,0x8a,0x14,0x05,0x1e,0x3c,0x72,0x76,0x77,0x38,0xe7,0x73,0xe3,
94     0x6e,0x24,0x4b,0x38,0xd2,0x1a,0xcf,0x23,0x58,0x78,0x36,0x82,0x23,0x6e,0x6b,0xef,
95     0x2c,0x3d,0xf2,0xe8,0xd6,0xc6,0x87,0x8e,0x78,0x9b,0x27,0x39,0xc0,0xd6,0xef,0x4d,
96     0x0b,0xfc,0x51,0x27,0x18,0xf3,0x51,0x5e,0x4d,0x96,0x3a,0xe2,0x15,0xe2,0x7e,0x42,
97     0xf4,0x16,0xd5,0xc6,0x52,0x5d,0x17,0x44,0x7e,0x09,0x7a,0xcf,0xe3,0x30,0xe3,0x84,
98     0xf6,0x6f,0x3a,0x33,0xfb,0x32,0xd,0x7c,0x1d,0xe6,0x7c,0x80,0x82,0x4f,0xed,0xda,0x87,
99     0x11,0x9c,0xc3,0x7e,0x85,0xbd,0x18,0x58,0x08,0x2b,0x23,0x37,0xe7,0x9d,0xd0,0xd1,
100    0x79,0xe2,0x05,0xbd,0xf5,0x4f,0x0e,0x0f,0xdb,0x4a,0x74,0xeb,0x09,0x01,0xb3,0xca,
101    0xbd,0xa6,0x7b,0x09,0xb1,0x13,0x77,0x30,0x4d,0x87,0x41,0x06,0x57,0x2e,0x5f,0x36,
102    0x6e,0xfc,0x35,0x69,0xfe,0x0a,0x24,0x6c,0x98,0x8c,0xda,0x97,0xf4,0xfb,0xc7,0x83,
103    0x2d,0x3e,0x7d,0xc0,0x5c,0x34,0xfd,0x11,0x2a,0x12,0xa7,0xae,0x4a,0xde,0xc8,0x4e,
104    0xcf,0xf4,0x85,0x63,0x77,0xc6,0x33,0x34,0xe0,0x27,0xe4,0x9e,0x91,0x0b,0x4b,0x85,
105    0xf0,0xb0,0x79,0xaa,0x7c,0xc6,0xff,0x3b,0xbc,0x04,0x73,0xb8,0x95,0xd7,0x31,0x54,
106    0x3b,0x56,0xec,0x52,0x15,0xd7,0x3e,0x62,0xf5,0x82,0x99,0x3e,0x2a,0xc0,0x4b,0x2e,
107    0x06,0x57,0x6d,0x3f,0x3e,0x77,0x1f,0x2b,0x2d,0xc5,0xb9,0x3b,0x68,0x56,0x73,0x70,
108    0x32,0x6b,0x6b,0x65,0x25,0x76,0x45,0x6c,0x45,0xf1,0x6c,0x59,0xfc,0x94,0xa7,0x15}};
109 const TPM2B_RSA_TEST_VALUE     c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
110    0x01,0xfe,0xd5,0x83,0x0b,0x15,0xba,0x90,0x2c,0xdf,0xf7,0x26,0xb7,0x8f,0xb1,0xd7,
111    0x0b,0xfd,0x83,0xf9,0x95,0xd5,0xd7,0xb5,0xc5,0x4a,0xde,0xd5,0xe6,0x20,0x78,
112    0xca,0x73,0x77,0x3d,0x61,0x36,0x48,0xae,0x3e,0x8f,0xee,0x43,0x29,0x96,0xdf,0x3f,
113    0x1c,0x97,0x5a,0xbe,0xe5,0xa2,0x7e,0x5b,0xd0,0xc0,0x29,0x39,0x83,0x81,0x77,0x24,
114    0x43,0xdb,0x3c,0x64,0x4d,0xf0,0x23,0xe4,0xae,0x0f,0x78,0x31,0x8c,0xda,0x0c,0xec,
115    0xf1,0xdf,0x09,0xf2,0x14,0x6a,0x4d,0xaf,0x36,0x81,0x6e,0xbd,0xbe,0x36,0x79,0x88,
116    0x98,0xb6,0x6f,0x5a,0xad,0xcf,0x7c,0xee,0xe0,0xdd,0x00,0xbe,0x59,0x97,0x88,0x00,
117    0x34,0xc0,0x8b,0x48,0x42,0x05,0x04,0x5a,0xb7,0x85,0x38,0xa0,0x35,0xd7,0x3b,0x51,
118    0xb8,0x7b,0x81,0x83,0xee,0xff,0x76,0x6f,0x50,0x39,0x4d,0xab,0x89,0x63,0x07,0x6d,
119    0xf5,0xe5,0x01,0x10,0x56,0xfe,0x93,0x06,0x8f,0xd3,0xc9,0x41,0xab,0xc9,0xdf,0x6e,
120    0x59,0xa8,0xc3,0x1d,0xbf,0x96,0x4a,0x59,0x80,0x3c,0x90,0x3a,0x59,0x56,0x4c,0x6d,
121    0x44,0x6d,0xeb,0xdc,0x73,0xcd,0xc1,0xec,0xb8,0x41,0xbf,0x89,0x8c,0x03,0x69,0x4c,
122    0xaf,0x3f,0xc1,0xc5,0xc7,0xe7,0x7d,0xa7,0x83,0x39,0x70,0xa2,0x6b,0x83,0xbc,0xbe,
123    0xf5,0xbf,0x1c,0xee,0x6e,0xa3,0x22,0x1e,0x25,0x2f,0x16,0x68,0x69,0x5a,0x1d,0xfa,
124    0x2c,0x3a,0x0f,0x67,0xe1,0x77,0x12,0xe8,0x3d,0xba,0xaa,0xef,0x96,0x9c,0x1f,0x64,
125    0x32,0xf4,0xa7,0xb3,0x3f,0x7d,0x61,0xbb,0x9a,0x27,0xad,0xfb,0x2f,0x33,0xc4,0x70}};
126 const TPM2B_RSA_TEST_VALUE     c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
127    0x67,0x4e,0xdd,0xc2,0xd2,0x6d,0xe0,0x03,0xc4,0xc2,0x41,0xd3,0xd4,0x61,0x30,0xd0,

```

```

128     0xe1, 0x68, 0x31, 0x4a, 0xda, 0xd9, 0xc2, 0x5d, 0xaa, 0xa2, 0x7b, 0xfb, 0x44, 0x02, 0xf5, 0xd6,
129     0xd8, 0x2e, 0xcd, 0x13, 0x36, 0xc9, 0x4b, 0xdb, 0x1a, 0x4b, 0x66, 0x1b, 0x4f, 0x9c, 0xb7, 0x17,
130     0xac, 0x53, 0x37, 0x4f, 0x21, 0xbd, 0xc, 0x66, 0xac, 0x06, 0x65, 0x52, 0x9f, 0x04, 0xf6, 0xa5,
131     0x22, 0x5b, 0xf7, 0xe6, 0xd, 0x3c, 0x9f, 0x41, 0x19, 0x09, 0x88, 0x7c, 0x41, 0x4c, 0x2f, 0x9c,
132     0x8b, 0x3c, 0xdd, 0x7c, 0x28, 0x78, 0x24, 0xd2, 0x09, 0xa6, 0x5b, 0xf7, 0x3c, 0x88, 0x7e, 0x73,
133     0x5a, 0x2d, 0x36, 0x02, 0x4f, 0x65, 0xb0, 0xcb, 0xc8, 0xdc, 0xac, 0xa2, 0xda, 0x8b, 0x84, 0x91,
134     0x71, 0xe4, 0x30, 0x8b, 0xb6, 0x12, 0xf2, 0xf0, 0xd0, 0xa0, 0x38, 0xcf, 0x75, 0xb7, 0x20, 0xcb,
135     0x35, 0x51, 0x52, 0x6b, 0xc4, 0xf4, 0x21, 0x95, 0xc2, 0xf7, 0x9a, 0x13, 0xc1, 0x1a, 0x7b, 0x8f,
136     0x77, 0xda, 0x19, 0x48, 0xbb, 0x6d, 0x14, 0x5d, 0xba, 0x65, 0xb4, 0x9e, 0x43, 0x42, 0x58, 0x98,
137     0x0b, 0x91, 0x46, 0xd8, 0x4c, 0xf3, 0x4c, 0xaf, 0x2e, 0x02, 0xa6, 0xb2, 0x49, 0x12, 0x62, 0x43,
138     0x4e, 0xa8, 0xac, 0xbf, 0xfd, 0xfa, 0x37, 0x24, 0xea, 0x69, 0x1c, 0xf5, 0xae, 0xfa, 0x08, 0x82,
139     0x30, 0xc3, 0xc0, 0xf8, 0x9a, 0x89, 0x33, 0xe1, 0x40, 0x6d, 0x18, 0x5c, 0x7b, 0x90, 0x48, 0xbf,
140     0x37, 0xdb, 0xea, 0xfb, 0x0e, 0xd4, 0x2e, 0x11, 0xfa, 0xa9, 0x86, 0xff, 0x00, 0x0b, 0x7b, 0xca,
141     0x09, 0x64, 0x6a, 0x8f, 0xc, 0x0e, 0x09, 0x14, 0x36, 0x4a, 0x74, 0x31, 0x18, 0x5b, 0x18, 0xeb,
142     0xea, 0x83, 0xc3, 0x66, 0x68, 0xa6, 0x7d, 0x43, 0x06, 0x0f, 0x99, 0x60, 0xce, 0x65, 0x08, 0xf6} };
143 #endif // SHA1
144 #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
145 const TPM2B_RSA_TEST_VALUE c_OaepKvt = {RSA_TEST_KEY_SIZE, {
146     0x33, 0x20, 0x6e, 0x21, 0xc3, 0xf6, 0xcd, 0xf8, 0xd7, 0x5d, 0x9f, 0xe9, 0x05, 0x14, 0x8c, 0x7c,
147     0xbb, 0x69, 0x24, 0x9e, 0x52, 0x8f, 0xaf, 0x84, 0x73, 0x21, 0x2c, 0x85, 0xa5, 0x30, 0x4d, 0xb6,
148     0xb8, 0xfa, 0x15, 0x9b, 0xc7, 0x8f, 0xc9, 0x7a, 0x72, 0x4b, 0x85, 0xa4, 0x1c, 0xc5, 0xd8, 0xe4,
149     0x92, 0xb3, 0xec, 0xd9, 0xa8, 0xca, 0x5e, 0x74, 0x73, 0x89, 0x7f, 0xb4, 0xac, 0x7e, 0x68, 0x12,
150     0xb2, 0x53, 0x27, 0x4b, 0xbf, 0xd0, 0x71, 0x69, 0x46, 0x9f, 0xef, 0xf4, 0x70, 0x60, 0xf8, 0xd7,
151     0xae, 0xc7, 0x5a, 0x27, 0x38, 0x25, 0x2d, 0x25, 0xab, 0x96, 0x56, 0x66, 0x3a, 0x23, 0x40, 0xa8,
152     0xdb, 0xbc, 0x86, 0xe8, 0xf3, 0xd2, 0x58, 0x0b, 0x44, 0xfc, 0x94, 0x1e, 0xb7, 0x5d, 0xb4, 0x57,
153     0xb5, 0xf3, 0x56, 0xee, 0x9b, 0xcf, 0x97, 0x91, 0x29, 0x36, 0xe3, 0x06, 0x13, 0xa2, 0xea, 0xd6,
154     0xd6, 0x0b, 0x86, 0x0b, 0x1a, 0x27, 0xe6, 0x22, 0xc4, 0x7b, 0xff, 0xde, 0x0f, 0xbf, 0x79, 0xc8,
155     0x1b, 0xed, 0xf1, 0x27, 0x62, 0xb5, 0x8b, 0xf9, 0xd9, 0x76, 0x90, 0xf6, 0xcc, 0x83, 0x0f, 0xce,
156     0xce, 0x2e, 0x63, 0x7a, 0x9b, 0xf4, 0x48, 0x5b, 0xd7, 0x81, 0x2c, 0x3a, 0xdb, 0x59, 0x0d, 0x4d,
157     0x9e, 0x46, 0xe9, 0x9e, 0x92, 0x22, 0x27, 0x1c, 0xb0, 0x67, 0x8a, 0xe6, 0x8a, 0x16, 0x8a, 0xdf,
158     0x95, 0x76, 0x24, 0x82, 0xad, 0xf1, 0xbc, 0x97, 0xbf, 0xd3, 0x5e, 0x6e, 0x14, 0x0c, 0x5b, 0x25,
159     0xfe, 0x58, 0xfa, 0x64, 0xe5, 0x14, 0x46, 0xb7, 0x58, 0xc6, 0x3f, 0x7f, 0x42, 0xd2, 0x8e, 0x45,
160     0x13, 0x41, 0x85, 0x12, 0x2e, 0x96, 0x19, 0xd0, 0x5e, 0x7d, 0x34, 0x06, 0x32, 0x2b, 0xc8, 0xd9,
161     0xd, 0x6c, 0x06, 0x36, 0xa0, 0xff, 0x47, 0x57, 0x2c, 0x25, 0xbc, 0x8a, 0xa5, 0xe2, 0xc7, 0xe3} };
162 const TPM2B_RSA_TEST_VALUE c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
163     0x39, 0xfc, 0x10, 0x5d, 0xf4, 0x45, 0x3d, 0x94, 0x53, 0x06, 0x89, 0x24, 0xe7, 0xe8, 0xfd, 0x03,
164     0xac, 0xfd, 0xbd, 0xb2, 0x28, 0xd3, 0x4a, 0x52, 0xc5, 0xd4, 0xdb, 0x17, 0xd4, 0x24, 0x05, 0xc4,
165     0xeb, 0x6a, 0xce, 0x1d, 0xbb, 0x37, 0xcb, 0x09, 0xd8, 0x6c, 0x83, 0x19, 0x93, 0xd4, 0xe2, 0x88,
166     0x88, 0x9b, 0xaf, 0x92, 0x16, 0xc4, 0x15, 0xbd, 0x49, 0x13, 0x22, 0xb7, 0x84, 0xcf, 0x23, 0xf2,
167     0x6f, 0x0c, 0x3e, 0x8f, 0xde, 0x04, 0x09, 0x31, 0x2d, 0x99, 0xdf, 0xe6, 0x74, 0x70, 0x30, 0xde,
168     0x8c, 0xad, 0x32, 0x86, 0xe2, 0x7c, 0x12, 0x90, 0x21, 0xf3, 0x86, 0xb7, 0xe2, 0x64, 0xca, 0x98,
169     0xcc, 0x64, 0x4b, 0xef, 0x57, 0x4f, 0x5a, 0x16, 0x6e, 0xd7, 0x2f, 0x5b, 0xf6, 0x07, 0xad, 0x33,
170     0xb4, 0x8f, 0x3b, 0x3a, 0x8b, 0xd9, 0x06, 0x2b, 0xed, 0x3c, 0x3c, 0x76, 0xf6, 0x21, 0x31, 0xe3,
171     0xfb, 0x2c, 0x45, 0x61, 0x42, 0xba, 0xe0, 0xc3, 0x72, 0x63, 0xd0, 0x6b, 0x8f, 0x36, 0x26, 0xfb,
172     0x9e, 0x89, 0x0e, 0x44, 0x9a, 0xc1, 0x84, 0x5e, 0x84, 0x8d, 0xb6, 0xea, 0xf1, 0x0d, 0x66, 0xc7,
173     0xdb, 0x44, 0xbd, 0x19, 0x7c, 0x05, 0xbe, 0xc4, 0xab, 0x88, 0x32, 0xbe, 0xc7, 0x63, 0x31, 0xe6,
174     0x38, 0xd4, 0xe5, 0xb8, 0x4b, 0xf5, 0x0e, 0x55, 0x9a, 0x3a, 0xe6, 0x0a, 0xec, 0xee, 0xe2, 0xa8,
175     0x88, 0x04, 0xf2, 0xb8, 0xaa, 0x5a, 0xd8, 0x97, 0x5d, 0xa0, 0xa8, 0x42, 0xfb, 0xd9, 0xde, 0x80,
176     0xae, 0x4c, 0xb3, 0xa1, 0x90, 0x47, 0x57, 0x03, 0x10, 0x78, 0xa6, 0x8f, 0x11, 0xba, 0x4b, 0xce,
177     0x2d, 0x56, 0xa4, 0xe1, 0xbd, 0xf8, 0xa0, 0xa4, 0xd5, 0x48, 0x3c, 0x63, 0x20, 0x00, 0x38, 0xa0,
178     0xd1, 0xe6, 0x12, 0xe9, 0x1d, 0xd8, 0x49, 0xe3, 0xd5, 0x24, 0xb5, 0xc5, 0x3a, 0x1f, 0xb0, 0xd4} };
179 const TPM2B_RSA_TEST_VALUE c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
180     0x74, 0x89, 0x29, 0x3e, 0x1b, 0xac, 0xc6, 0x85, 0xca, 0xf0, 0x63, 0x43, 0x30, 0x7d, 0x1c, 0x9b,
181     0x2f, 0xbd, 0x4d, 0x69, 0x39, 0x5e, 0x85, 0xe2, 0xef, 0x86, 0x0a, 0xc6, 0x6b, 0xa6, 0x08, 0x19,
182     0x6c, 0x56, 0x38, 0x24, 0x55, 0x92, 0x84, 0x9b, 0x1b, 0x8b, 0x04, 0xcf, 0x24, 0x14, 0x24, 0x13,
183     0x0e, 0x8b, 0x82, 0x6f, 0x96, 0xc8, 0x9a, 0x68, 0xfc, 0x4c, 0x02, 0xf0, 0xdc, 0xcd, 0x36, 0x25,
184     0x31, 0xd5, 0x82, 0xcf, 0xc9, 0x69, 0x72, 0xf6, 0x1d, 0xab, 0x68, 0x20, 0x2e, 0x2d, 0x19, 0x49,
185     0xf0, 0x2e, 0xad, 0xd2, 0xda, 0xaf, 0xff, 0xb6, 0x92, 0x83, 0x5b, 0x8a, 0x06, 0x2d, 0x0c, 0x32,
186     0x11, 0x32, 0x3b, 0x77, 0x17, 0xf6, 0x50, 0xfb, 0xf8, 0x57, 0xc9, 0xc7, 0x9b, 0x9e, 0xc6, 0xd1,
187     0xa9, 0x55, 0xf0, 0x22, 0x35, 0xda, 0xca, 0x3c, 0x8e, 0xc6, 0x9a, 0xd8, 0x25, 0xc8, 0x5e, 0x93,
188     0xd, 0xaa, 0xa7, 0x06, 0xaf, 0x11, 0x29, 0x99, 0xe7, 0x7c, 0xee, 0x49, 0x82, 0x30, 0xba, 0x2c,
189     0xe2, 0x40, 0x8f, 0x0a, 0xa6, 0x7b, 0x24, 0x75, 0xc5, 0xcd, 0x03, 0x12, 0xf4, 0xb2, 0x4b, 0x3a,
190     0xd1, 0x91, 0x3c, 0x20, 0x0e, 0x58, 0x2b, 0x31, 0xf8, 0x8b, 0xee, 0xbc, 0x1f, 0x95, 0x35, 0x58,
191     0x6a, 0x73, 0xee, 0x99, 0xb0, 0x01, 0x42, 0x4f, 0x66, 0xc0, 0x66, 0xbb, 0x35, 0x86, 0xeb, 0xd9,
192     0x7b, 0x55, 0x77, 0x2d, 0x54, 0x78, 0x19, 0x49, 0xe8, 0xcc, 0xfd, 0xb1, 0xcb, 0x49, 0xc9, 0xea,
193     0x20, 0xab, 0xed, 0xb5, 0xed, 0xfe, 0xb2, 0xb5, 0xa8, 0xcf, 0x05, 0x06, 0xd5, 0x7d, 0x2b, 0xb,

```



```
194     0x0b,0x65,0x6b,0x2b,0x6d,0x55,0x95,0x85,0x44,0x8b,0x12,0x05,0xf3,0x4b,0xd4,0x8e,  
195     0x3d,0x68,0x2d,0x29,0x9c,0x05,0x79,0xd6,0xfc,0x72,0x90,0x6a,0xab,0x46,0x38,0x81}};  
196 const TPM2B_RSA_TEST_VALUE c_RsassaKvt = {RSA_TEST_KEY_SIZE, {  
197     0xfa,0xb1,0x0a,0xb5,0xe4,0x02,0xf7,0xdd,0x45,0x2a,0xcc,0x2b,0x6b,0x8c,0x0e,0x9a,  
198     0x92,0x4f,0x9b,0xc5,0xe4,0x8b,0x82,0xb9,0xb0,0xd9,0x87,0x8c,0xcb,0xf0,0xb0,0x59,  
199     0xa5,0x92,0x21,0xa0,0xa7,0x61,0x5c,0xed,0xa8,0x6e,0x22,0x29,0x46,0xc7,0x86,0x37,  
200     0x4b,0x1b,0x1e,0x94,0x93,0xc8,0x4c,0x17,0x7a,0xae,0x59,0x91,0xf8,0x83,0x84,0xc4,  
201     0x8c,0x38,0xc2,0x35,0x0e,0x7e,0x50,0x67,0x76,0xe7,0xd3,0xec,0x6f,0x0d,0xa0,0x5c,  
202     0x2f,0x0a,0x80,0x28,0xd3,0xc5,0x7d,0x2d,0x1a,0x0b,0x96,0xd6,0xe5,0x98,0x05,0x8c,  
203     0x4d,0xa0,0x1f,0x8c,0xf9,0xfb,0xb1,0xcf,0xe9,0xcb,0x38,0x27,0x60,0xff,0xfe,0xca,  
204     0xf4,0x8b,0x61,0xb7,0x1d,0xb6,0x20,0x9d,0x40,0x2a,0x1c,0xfd,0x55,0x40,0x4b,0x95,  
205     0x39,0x52,0x18,0x3b,0xab,0x44,0xe8,0x83,0x4b,0x7c,0x47,0xfb,0xed,0x06,0x9c,0xcd,  
206     0x4f,0xba,0x81,0xd6,0xb7,0x31,0xcf,0x5c,0x23,0xf8,0x25,0xab,0x95,0x77,0x0a,0x8f,  
207     0x46,0xef,0xfb,0x59,0xb8,0x04,0xd7,0x1e,0xf5,0xaf,0x6a,0x1a,0x26,0x9b,0xae,0xf4,  
208     0xf5,0x7f,0x84,0x6f,0x3c,0xed,0xf8,0x24,0x0b,0x43,0xd1,0xba,0x74,0x89,0x4e,0x39,  
209     0xfe,0xab,0xa5,0x16,0xa5,0x28,0xee,0x96,0x84,0x3e,0x16,0x6d,0x5f,0x4e,0x0b,0x7d,  
210     0x94,0x16,0x1b,0x8c,0xf9,0xaa,0x9b,0xc0,0x02,0x4c,0x3e,0x62,0xff,0xfe,0xa2,  
211     0x20,0x33,0x5e,0xa6,0xdd,0xda,0x15,0x2d,0xb7,0xcd,0xda,0xff,0xb1,0x0b,0x45,0x7b,  
212     0xd3,0xa0,0x42,0x29,0xab,0xa9,0x73,0xe9,0xa4,0xd9,0x8d,0xac,0xa1,0x88,0x2c,0x2d}};  
213 #endif // SHA256  
214 #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH  
215 const TPM2B_RSA_TEST_VALUE c_OaepKvt = {RSA_TEST_KEY_SIZE, {  
216     0x0f,0x3c,0x42,0x4d,0x8c,0x91,0x96,0x05,0x3c,0xfd,0x59,0x3b,0x7f,0x29,0xbc,0x03,  
217     0x67,0xc1,0xff,0x74,0x09,0xf4,0x13,0x45,0xbe,0x13,0x1d,0xc9,0x86,0x94,0xfe,  
218     0xed,0xa6,0xe8,0x3a,0xcb,0x89,0x4d,0xec,0x86,0x63,0x4c,0xdb,0xf1,0x95,0xee,0xc1,  
219     0x46,0xc5,0x3b,0xd8,0xf8,0xa2,0x41,0x6a,0x60,0x8b,0x9e,0x5e,0x7f,0x20,0x16,0xe3,  
220     0x69,0xb6,0x2d,0x92,0xfc,0x60,0xa2,0x74,0x88,0xd5,0xc7,0xa6,0xd1,0xff,0xe3,0x45,  
221     0x02,0x51,0x39,0xd9,0xf3,0x56,0x0b,0x91,0x80,0xe0,0x6c,0xa8,0xc3,0x78,0xef,0x34,  
222     0x22,0x8c,0xf5,0xfb,0x47,0x98,0x5d,0x57,0x8e,0x3a,0xb9,0xff,0x92,0x04,0xc7,0xc2,  
223     0x6e,0xfa,0x14,0xc1,0xb9,0x68,0x15,0x5c,0x12,0xe8,0xa8,0xbe,0xea,0xe8,0x9b,  
224     0x48,0x28,0x35,0xdb,0x4b,0x52,0xc1,0x2d,0x85,0x47,0x83,0xd0,0xe9,0xae,0x90,0x6e,  
225     0x65,0xd4,0x34,0x7f,0x81,0xce,0x69,0xf0,0x96,0x62,0xf7,0xec,0x41,0xd5,0xc2,0xe3,  
226     0x4b,0xba,0x9c,0x8a,0x02,0xce,0xf0,0x5d,0x14,0xf7,0x09,0x42,0x8e,0x4a,0x27,0xfe,  
227     0x3e,0x66,0x42,0x99,0x03,0xe1,0x69,0xbd,0xdb,0x7f,0x9b,0x70,0xeb,0x4e,0x9c,0xac,  
228     0x45,0x67,0x91,0x9f,0x75,0x10,0xc6,0xfc,0x14,0xe1,0x28,0xc1,0x0e,0xe0,0x7e,0xc0,  
229     0x5c,0x1d,0xee,0xe8,0xff,0x45,0x79,0x51,0x86,0x08,0xe6,0x39,0xac,0xb5,0xfd,0xb8,  
230     0xf1,0xdd,0x2e,0xf4,0xb2,0x1a,0x69,0x0d,0xd9,0x98,0x8e,0xdb,0x85,0x61,0x70,0x20,  
231     0x82,0x91,0x26,0x87,0x80,0xc4,0x6a,0xd8,0x3b,0x91,0x4d,0xd3,0x33,0x84,0xad,0xb7}};  
232 const TPM2B_RSA_TEST_VALUE c_RsaesKvt = {RSA_TEST_KEY_SIZE, {  
233     0x44,0xd5,0x9f,0xbc,0x48,0x03,0x3d,0x9f,0x22,0x91,0x2a,0xab,0x3c,0x31,0x71,0xab,  
234     0x86,0x3f,0x0f,0x6f,0x59,0x5b,0x93,0x27,0xbc,0xbc,0xcd,0x29,0x38,0x43,0x2a,0x3b,  
235     0x3b,0xd2,0xb3,0x45,0x40,0xba,0x15,0xb4,0x45,0xe3,0x56,0xab,0xff,0xb3,0x20,0x26,  
236     0x39,0xc0,0x48,0xc5,0x5d,0x41,0x0d,0x2f,0x57,0x7f,0x9d,0x16,0x2e,0x57,0xc7,  
237     0x6b,0xf3,0x36,0x54,0xbd,0xb6,0x1d,0x46,0x4e,0x13,0x50,0xd7,0x61,0x9d,0x8d,0x7b,  
238     0xeb,0x21,0x9f,0x79,0xf3,0xfd,0xe0,0x1b,0xa8,0xed,0x6d,0x29,0x33,0x0d,0x65,0x94,  
239     0x24,0x1e,0x62,0x88,0x6b,0x2b,0x4e,0x39,0xf5,0x80,0x39,0xca,0x76,0x95,0xbc,0x7c,  
240     0x27,0x1d,0xdd,0x3a,0x11,0xf1,0x3e,0x54,0x03,0xb7,0x43,0x91,0x99,0x33,0xfe,0x9d,  
241     0x14,0x2c,0x87,0x9a,0x95,0x18,0x1f,0x02,0x04,0x6a,0xe2,0xb7,0x81,0x14,0x13,0x45,  
242     0x16,0xfb,0xe4,0xb7,0x8f,0xab,0x2b,0xd7,0x60,0x34,0x8a,0x55,0xbc,0x01,0x8c,0x49,  
243     0x02,0x29,0xf1,0x9c,0x94,0x98,0x44,0xd0,0x94,0xcb,0xd4,0x85,0x4c,0x3b,0x77,0x72,  
244     0x99,0xd5,0x4b,0xc6,0x3b,0xe4,0xd2,0xc8,0xe9,0x6a,0x23,0x18,0x3b,0x3b,0x5e,0x32,  
245     0xec,0x70,0x84,0x5d,0xbb,0x6a,0x8f,0x0c,0x5f,0x55,0xa5,0x30,0x34,0x48,0xbb,0xc2,  
246     0xdf,0x12,0xb9,0x81,0xad,0x36,0x3f,0xf0,0x24,0x16,0x48,0x04,0x4a,0x7f,0xfd,0x9f,  
247     0x4c,0xea,0xfe,0x1d,0x83,0xd0,0x81,0xad,0x25,0x6c,0x5f,0x45,0x36,0x91,0xf0,0xd5,  
248     0x8b,0x53,0x0a,0xdf,0xec,0x9f,0x04,0x58,0xc4,0x35,0xa0,0x78,0x1f,0x68,0xe0,0x22}};  
249 const TPM2B_RSA_TEST_VALUE c_RsapssKvt = {RSA_TEST_KEY_SIZE, {  
250     0x3f,0x3a,0x82,0x6d,0x42,0xe3,0x8b,0x4f,0x45,0x9c,0xda,0x6c,0xbe,0xcd,0x00,  
251     0x98,0xfb,0xbe,0x59,0x30,0xc6,0x3c,0xaa,0xb3,0x06,0x27,0xb5,0xda,0xfa,0xb2,0xc3,  
252     0x43,0xb7,0xbd,0xe9,0xd3,0x23,0xed,0x80,0xce,0x74,0xb3,0xb8,0x77,0x8d,0xe6,0x8d,  
253     0x3c,0xe5,0xf5,0xd7,0x80,0xcf,0x38,0x55,0x76,0xd7,0x87,0xa8,0xd6,0x3a,0xcf,0xfd,  
254     0xd8,0x91,0x65,0xab,0x43,0x66,0x50,0xb7,0x9a,0x13,0x6b,0x45,0x80,0x76,0x86,0x22,  
255     0x27,0x72,0xf7,0xbb,0x65,0x22,0x5c,0x55,0x60,0xd8,0x84,0x9f,0xf2,0x61,0x52,0xac,  
256     0xf2,0x4f,0x5b,0x7b,0x21,0xe1,0xf5,0x4b,0x8f,0x01,0xf2,0x4b,0xcf,0xd3,0xfb,0x74,  
257     0x5e,0x6e,0x96,0xb4,0xa8,0x0f,0x01,0x9b,0x26,0x54,0x0a,0x70,0x55,0x26,0xb7,0x0b,  
258     0xe8,0x01,0x68,0x66,0x0d,0x6f,0xb5,0xfc,0x66,0xbd,0x9e,0x44,0xed,0x6a,0x1e,0x3c,  
259     0x3b,0x61,0x5d,0xe8,0xdb,0x99,0x5b,0x67,0xbf,0x94,0xfb,0xe6,0x8c,0x4b,0x07,0xcb,
```

```

260     0x43,0x3a,0x0d,0xb1,0x1b,0x10,0x66,0x81,0xe2,0x0d,0xe7,0xd1,0xca,0x85,0xa7,0x50,
261     0x82,0x2d,0xbf,0xed,0xcf,0x43,0x6d,0xdb,0x2c,0x7b,0x73,0x20,0xfe,0x73,0x3f,0x19,
262     0xc6,0xdb,0x69,0xb8,0xc3,0xd3,0xf4,0xe5,0x64,0xf8,0x36,0x8e,0xd5,0xd8,0x09,0x2a,
263     0x5f,0x26,0x70,0xa1,0xd9,0x5b,0x14,0xf8,0x22,0xe9,0x9d,0x22,0x51,0xf4,0x52,0xc1,
264     0x6f,0x53,0xf5,0xca,0x0d,0xda,0x39,0x8c,0x29,0x42,0xe8,0x58,0x89,0xbb,0xd1,0x2e,
265     0xc5,0xdb,0x86,0x8d,0xaf,0xec,0x58,0x36,0x8d,0x8d,0x57,0x23,0xd5,0xdd,0xb9,0x24}};
266 const TPM2B_RSA_TEST_VALUE c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
267     0x39,0x10,0x58,0x7d,0x6d,0xa8,0xd5,0x90,0x07,0xd6,0x2b,0x13,0xe9,0xd8,0x93,0x7e,
268     0xf3,0x5d,0x71,0xe0,0xf0,0x33,0x3a,0x4a,0x22,0xf3,0xe6,0x95,0xd3,0x8e,0x8c,0x41,
269     0xe7,0xb3,0x13,0xde,0x4a,0x45,0xd3,0xd1,0xfb,0xb1,0x3f,0x9b,0x39,0xa5,0x50,0x58,
270     0xef,0xb6,0x3a,0x43,0xdd,0x54,0xab,0xda,0x9d,0x32,0x49,0xe4,0x57,0x96,0xe5,0x1b,
271     0x1d,0x8f,0x33,0x8e,0x07,0x67,0x56,0x14,0xc1,0x18,0x78,0xa2,0x52,0xe6,0x2e,0x07,
272     0x81,0xbe,0xd8,0xca,0x76,0x63,0x68,0xc5,0x47,0xa2,0x92,0x5e,0x4c,0xfd,0x14,0xc7,
273     0x46,0x14,0xbe,0xc7,0x85,0xef,0xe6,0xb8,0x46,0xcb,0x3a,0x67,0x66,0x89,0xc6,0xee,
274     0x9d,0x64,0xf5,0x0d,0x09,0x80,0x9a,0x6f,0x0e,0xeb,0xe4,0xb9,0xe9,0xab,0x90,0x4f,
275     0xe7,0x5a,0xc8,0xca,0xf6,0x16,0x0a,0x82,0xbd,0xb7,0x76,0x59,0x08,0x2d,0xd9,0x40,
276     0x5d,0xaa,0xa5,0xef,0xfb,0xe3,0x81,0x2c,0x2c,0x5c,0xa8,0x16,0xbd,0x63,0xd5,0xc2,
277     0x4d,0x3b,0x51,0xaa,0x62,0x1f,0x06,0xe5,0xbb,0x78,0x44,0x04,0x0c,0x5c,0xe1,0x1b,
278     0x6b,0x9d,0x21,0x10,0xaf,0x48,0x48,0x98,0x97,0x77,0xc2,0x73,0xb4,0x98,0x64,0xcc,
279     0x94,0x2c,0x29,0x28,0x45,0x36,0xd1,0xc5,0xd0,0x2f,0x97,0x27,0x92,0x65,0x22,0xbb,
280     0x63,0x79,0xea,0xf5,0xff,0x77,0x0f,0x4b,0x56,0x8a,0x9f,0xad,0x1a,0x97,0x67,0x39,
281     0x69,0xb8,0x4c,0x6c,0xc2,0x56,0xc5,0x7a,0xa8,0x14,0x5a,0x24,0x7a,0xa4,0x6e,0x55,
282     0xb2,0x86,0x1d,0xf4,0x62,0x5a,0x2d,0x87,0x6d,0xde,0x99,0x78,0x2d,0xef,0xd7,0xdc}};
283 #endif // SHA384
284 #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
285 const TPM2B_RSA_TEST_VALUE c_OaepKvt = {RSA_TEST_KEY_SIZE, {
286     0x48,0x45,0xa7,0x70,0xb2,0x41,0xb7,0x48,0x5e,0x79,0x8c,0xdf,0x1c,0xc6,0x7e,0xbb,
287     0x11,0x80,0x82,0x52,0xbf,0x40,0x3d,0x90,0x03,0x6e,0x20,0x3a,0xb9,0x65,0xc8,0x51,
288     0x4c,0xbd,0x9c,0xa9,0x43,0x89,0xd0,0x57,0x0c,0xa3,0x69,0x22,0x7e,0x82,0x2a,0x1c,
289     0x1d,0x5a,0x80,0x84,0x81,0xbb,0x5e,0x5e,0x0c,0xc1,0x66,0x9a,0xac,0x00,0xba,0x14,
290     0xa2,0xe9,0xd0,0x3a,0x89,0x5a,0x63,0xe2,0xec,0x92,0x05,0xf4,0x47,0x66,0x12,0x7f,
291     0xdb,0xa7,0x3c,0x5b,0x67,0xe1,0x55,0xca,0x0a,0x27,0xbf,0x39,0x89,0x11,0x05,0xba,
292     0x9b,0x5a,0x9b,0x65,0x44,0xad,0x78,0xcf,0x8f,0x94,0xf6,0x9a,0xb4,0x52,0x39,0x0e,
293     0x00,0xba,0xbc,0xe0,0xbd,0x6f,0x81,0x2d,0x76,0x42,0x66,0x70,0x07,0x77,0xbf,0x09,
294     0x88,0x2a,0x0c,0xb1,0x56,0x3e,0xee,0xfd,0xdc,0xb6,0x3c,0x0d,0xc5,0xa4,0x0d,0x10,
295     0x32,0x80,0x3e,0x1e,0xfe,0x36,0x8f,0xb5,0x42,0xc1,0x21,0x7b,0xdf,0xdf,0x4a,0xd2,
296     0x68,0x0c,0x01,0x9f,0x4a,0xf0,0xd4,0xec,0xf7,0x49,0x06,0xab,0xed,0xc6,0xd5,0x1b,
297     0x63,0x76,0x38,0xc8,0x6c,0xc7,0x4f,0xcb,0x29,0x8a,0x0e,0x6f,0x33,0xaf,0x69,0x31,
298     0x8e,0xa7,0xdd,0x9a,0x36,0xde,0x9b,0xf1,0x0b,0xfb,0x20,0xa0,0x6d,0x33,0x31,0xc9,
299     0x9e,0xb4,0x2e,0xc5,0x40,0x0e,0x60,0x71,0x36,0x75,0x05,0xf9,0x37,0xe0,0xca,0x8e,
300     0x8f,0x56,0xe0,0xea,0x9b,0xeb,0x17,0xf3,0xca,0x40,0xc3,0x48,0x01,0xba,0xdc,0xc6,
301     0x4b,0x2b,0x5b,0x7b,0x5c,0x81,0xa6,0xbb,0xc7,0x43,0xc0,0xbe,0xc0,0x30,0x7b,0x55}};
302 const TPM2B_RSA_TEST_VALUE c_RsaesKvt = {RSA_TEST_KEY_SIZE, {
303     0x74,0x83,0xfa,0x52,0x65,0x50,0x68,0xd0,0x82,0x05,0x72,0x70,0x78,0x1c,0xac,0x10,
304     0x23,0xc5,0x07,0xf8,0x93,0xd2,0xeb,0x65,0x87,0xbb,0x47,0xc2,0xfb,0x30,0x9e,0x61,
305     0x4c,0xac,0x04,0x57,0x5a,0x7c,0xeb,0x29,0x08,0x84,0x86,0x89,0x1e,0x8f,0x07,0x32,
306     0xa3,0x8b,0x70,0xe7,0xa2,0x9f,0x9c,0x42,0x71,0x3d,0x23,0x59,0x82,0x5e,0x8a,0xde,
307     0xd6,0xfb,0xd8,0xc5,0x8b,0xc0,0xdb,0x10,0x38,0x87,0xd3,0xbf,0x04,0xb0,0x66,0xb9,
308     0x85,0x81,0x54,0x4c,0x69,0xdc,0xba,0x78,0xf3,0x4a,0xdb,0x25,0xa2,0xf2,0x34,0x55,
309     0xdd,0xaa,0xa5,0xc4,0xed,0x55,0x06,0x0e,0x2a,0x30,0x77,0xab,0x82,0x79,0xf0,0xcd,
310     0x9d,0x6f,0x09,0xa0,0xc8,0x82,0xc9,0xe0,0x61,0xda,0x40,0xcd,0x17,0x59,0xc0,0xef,
311     0x95,0x6d,0xa3,0x6d,0x1c,0x2b,0xee,0x24,0xef,0xd8,0x4a,0x55,0x6c,0xd6,0x26,0x42,
312     0x32,0x17,0xfd,0x6a,0xb3,0x4f,0xde,0x07,0x2f,0x10,0xd4,0xac,0x14,0xea,0x89,0x68,
313     0xcc,0xd3,0x07,0xb7,0xcf,0xba,0x39,0x20,0x63,0x20,0x7b,0x44,0x8b,0x48,0x60,0x5d,
314     0x3a,0x2a,0x0a,0xe9,0x68,0xab,0x15,0x46,0x27,0x64,0xb5,0x82,0x06,0x29,0xe7,0x25,
315     0xca,0x46,0x48,0x6e,0x2a,0x34,0x57,0x4b,0x81,0x75,0xae,0xb6,0xfd,0x6f,0x51,0x5f,
316     0x04,0x59,0xc7,0x15,0x1f,0xe0,0x68,0xf7,0x36,0x2d,0xdf,0xc8,0x9d,0x05,0x27,0x2d,
317     0x3f,0x2b,0x59,0x5d,0xcb,0xf3,0xc4,0x92,0x6e,0x00,0xa8,0x8d,0xd0,0x69,0xe5,0x59,
318     0xda,0xba,0x4f,0x38,0xf5,0xa0,0x8b,0xf1,0x73,0xe9,0x0d,0xee,0x64,0xe5,0xa2,0xd8}};
319 const TPM2B_RSA_TEST_VALUE c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
320     0x1b,0xca,0x8b,0x18,0x15,0x3b,0x95,0x5b,0xa0,0x89,0x10,0x03,0x7f,0x7c,0xa0,0xc9,
321     0x66,0x57,0x86,0x6a,0xc9,0xeb,0x82,0x71,0xf3,0x8d,0x6f,0xa9,0xa4,0x2d,0xd0,0x22,
322     0xdf,0xe9,0xc6,0x71,0x5b,0xf4,0x27,0x38,0x5b,0x2c,0x8a,0x54,0xcc,0x85,0x11,0x69,
323     0x6d,0x6f,0x42,0xe7,0x22,0xcb,0xd6,0xad,0x1a,0xc5,0xab,0x6a,0xa5,0xfc,0xa5,0x70,
324     0x72,0x4a,0x62,0x25,0xd0,0xa2,0x16,0x61,0xab,0xac,0x31,0xa0,0x46,0x24,0x4f,0xdd,
325     0x9a,0x36,0x55,0xb6,0x00,0x9e,0x23,0x50,0x0d,0x53,0x01,0xb3,0x46,0x56,0xb2,0x1d,

```

```

326     0x33,0x5b,0xca,0x41,0x7f,0x65,0x7e,0x00,0x5c,0x12,0xff,0x0a,0x70,0x5d,0x8c,0x69,
327     0x4a,0x02,0xee,0x72,0x30,0xa7,0x5c,0xa4,0xbb,0xbe,0x03,0x0c,0xe4,0x5f,0x33,0xb6,
328     0x78,0x91,0x9d,0xd8,0xec,0x34,0x03,0x2e,0x63,0x32,0xc7,0x2a,0x36,0x50,0xd5,0x8b,
329     0x0e,0x7f,0x54,0x4e,0xf4,0x29,0x11,0x1b,0xcd,0x0f,0x37,0xa5,0xbc,0x61,0x83,0x50,
330     0xfa,0x18,0x75,0xd9,0xfe,0xa7,0xe8,0x9b,0xc1,0x4f,0x96,0x37,0x81,0x71,0xdf,0x71,
331     0x8b,0x89,0x81,0xf4,0x95,0xb5,0x29,0x66,0x41,0x0c,0x73,0xd7,0x0b,0x21,0xb4,0xfb,
332     0xf9,0x63,0x2f,0xe9,0x7b,0x38,0xaa,0x20,0xc3,0x96,0xcc,0xb7,0xb2,0x24,0xa1,0xe0,
333     0x59,0x9c,0x10,0x9e,0x5a,0xf7,0xe3,0x02,0xe6,0x23,0xe2,0x44,0x21,0x3f,0x6e,0x5e,
334     0x79,0xb2,0x93,0x7d,0xce,0xed,0xe2,0xe1,0xab,0x98,0x07,0xa7,0xbd,0xbc,0xd8,0xf7,
335     0x06,0xeb,0xc5,0xa6,0x37,0x18,0x11,0x88,0xf7,0x63,0x39,0xb9,0x57,0x29,0xdc,0x03} };
336 const TPM2B_RSA_TEST_VALUE    c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
337     0x05,0x55,0x00,0x62,0x01,0xc6,0x04,0x31,0x55,0x73,0x3f,0x2a,0xf9,0xd4,0x0f,0xc1,
338     0x2b,0xeb,0xd8,0xc8,0xdb,0xb2,0xab,0x6c,0x26,0xde,0x2d,0x89,0xc2,0x2d,0x36,0x62,
339     0xc8,0x22,0x5d,0x58,0x03,0xb1,0x46,0x14,0xa5,0xd4,0xbc,0x25,0x6b,0x7f,0x8f,0x14,
340     0x7e,0x03,0x2f,0x3d,0xb8,0x39,0xa5,0x79,0x13,0x7e,0x22,0x2a,0xb9,0x3e,0x8f,0xaa,
341     0x01,0x7c,0x03,0x12,0x21,0x6c,0x2a,0xb4,0x39,0x98,0x6d,0xff,0x08,0x6c,0x59,0x2d,
342     0xdc,0xc6,0xf1,0x77,0x62,0x10,0xa6,0xcc,0xe2,0x71,0x8e,0x97,0x00,0x87,0x5b,0x0e,
343     0x20,0x00,0x3f,0x18,0x63,0x83,0xf0,0xe4,0x0a,0x64,0x8c,0xe9,0x8c,0x91,0xe7,0x89,
344     0x04,0x64,0x2c,0x8b,0x41,0xc8,0xac,0xf6,0x5a,0x75,0xe6,0xa5,0x76,0x43,0xcb,0xa5,
345     0x33,0x8b,0x07,0xc9,0x73,0x0f,0x45,0xa4,0xc3,0xac,0xc1,0xc3,0xe6,0xe7,0x21,0x66,
346     0x1c,0xba,0xbf,0xea,0x3e,0x39,0xfa,0xb2,0xe2,0x8f,0xfe,0x9c,0xb4,0x85,0x89,0x33,
347     0x2a,0x0c,0xc8,0x5d,0x58,0xe1,0x89,0x12,0xe9,0x4d,0x42,0xb3,0x1f,0x99,0x0c,0x3e,
348     0xd8,0xb2,0xeb,0xf5,0x88,0xfb,0xe1,0x4b,0x8e,0xdc,0xd3,0xa8,0xda,0xbe,0x04,0x45,
349     0xbf,0x56,0xc6,0x54,0x70,0x00,0xb8,0x66,0x46,0x3a,0xa3,0x1e,0xb6,0xeb,0x1a,0xa0,
350     0x0b,0xd3,0x9a,0x9a,0x52,0xda,0x60,0x69,0xb7,0xef,0x93,0x47,0x38,0xab,0x1a,0xa0,
351     0x22,0x6e,0x76,0x06,0xb6,0x74,0xaf,0x74,0x8f,0x51,0xc0,0x89,0x5a,0x4b,0xbe,0x6a,
352     0x91,0x18,0x25,0x7d,0xa6,0x77,0xe6,0xfd,0xc2,0x62,0x36,0x07,0xc6,0xef,0x79,0xc9} };
353 #endif // SHA512

```

10.1.10 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  const BYTE key_AES128 [] = {
3      0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
4      0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c};
5  const BYTE dataIn_AES128 [] = {
6      0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
7      0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
8      0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
9      0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
10 const BYTE dataOut_AES128_ECB [] = {
11     0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60,
12     0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xef, 0x97,
13     0xf5, 0xd3, 0xd5, 0x85, 0x03, 0xb9, 0x69, 0x9d,
14     0xe7, 0x85, 0x89, 0x5a, 0x96, 0xfd, 0xba, 0xaf};
15 const BYTE dataOut_AES128_CBC [] = {
16     0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46,
17     0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
18     0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xee,
19     0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2};
20 const BYTE dataOut_AES128_CFB [] = {
21     0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
22     0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
23     0xc8, 0xa6, 0x45, 0x37, 0xa0, 0xb3, 0xa9, 0x3f,
24     0xcd, 0xe3, 0xcd, 0xad, 0x9f, 0x1c, 0xe5, 0x8b};
25 const BYTE dataOut_AES128_OFB [] = {
26     0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
27     0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
28     0x77, 0x89, 0x50, 0x8d, 0x16, 0x91, 0x8f, 0x03,
29     0xf5, 0x3c, 0x52, 0xda, 0xc5, 0x4e, 0xd8, 0x25};
30 const BYTE dataOut_AES128_CTR [] = {
31     0x87, 0x4d, 0x61, 0x91, 0xb6, 0x20, 0xe3, 0x26,
32     0x1b, 0xef, 0x68, 0x64, 0x99, 0x0d, 0xb6, 0xce,
33     0x98, 0x06, 0xf6, 0x6b, 0x79, 0x70, 0xfd, 0xff,
34     0x86, 0x17, 0x18, 0x7b, 0xb9, 0xff, 0xfd, 0xff};
35 #endif
36 #if AES_192
37 const BYTE key_AES192 [] = {
38     0x8e, 0x73, 0xb0, 0xf7, 0xda, 0x0e, 0x64, 0x52,
39     0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
40     0x62, 0xf8, 0xea, 0xd2, 0x52, 0x2c, 0x6b, 0x7b};
41 const BYTE dataIn_AES192 [] = {
42     0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
43     0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
44     0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
45     0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
46 const BYTE dataOut_AES192_ECB [] = {
47     0xbd, 0x33, 0x4f, 0x1d, 0x6e, 0x45, 0xf2, 0x5f,
48     0xf7, 0x12, 0xa2, 0x14, 0x57, 0x1f, 0xa5, 0xcc,
49     0x97, 0x41, 0x04, 0x84, 0x6d, 0x0a, 0xd3, 0xad,
50     0x77, 0x34, 0xec, 0xb3, 0xec, 0xee, 0x4e, 0xef};
51 const BYTE dataOut_AES192_CBC [] = {
52     0x4f, 0x02, 0x1d, 0xb2, 0x43, 0xbc, 0x63, 0x3d,
53     0x71, 0x78, 0x18, 0x3a, 0x9f, 0xa0, 0x71, 0xe8,
54     0xb4, 0xd9, 0xad, 0xa9, 0xad, 0x7d, 0xed, 0xf4,
55     0xe5, 0xe7, 0x38, 0x76, 0x3f, 0x69, 0x14, 0x5a};
56 const BYTE dataOut_AES192_CFB [] = {
57     0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,

```

```

58         0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
59         0x67, 0xce, 0x7f, 0x7f, 0x81, 0x17, 0x36, 0x21,
60         0x96, 0x1a, 0x2b, 0x70, 0x17, 0x1d, 0x3d, 0x7a};
61 const BYTE dataOut_AES192_OFB [] = {
62         0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
63         0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
64         0xfc, 0xc2, 0x8b, 0x8d, 0x4c, 0x63, 0x83, 0x7c,
65         0x09, 0xe8, 0x17, 0x00, 0xc1, 0x10, 0x04, 0x01};
66 const BYTE dataOut_AES192_CTR [] = {
67         0x1a, 0xbc, 0x93, 0x24, 0x17, 0x52, 0x1c, 0xa2,
68         0x4f, 0x2b, 0x04, 0x59, 0xfe, 0x7e, 0x6e, 0x0b,
69         0x09, 0x03, 0x39, 0xec, 0x0a, 0xa6, 0xfa, 0xef,
70         0xd5, 0xcc, 0xc2, 0xc6, 0xf4, 0xce, 0x8e, 0x94};
71 #endif
72 #if AES_256
73 const BYTE key_AES256 [] = {
74         0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
75         0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
76         0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
77         0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4};
78 const BYTE dataIn_AES256 [] = {
79         0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
80         0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
81         0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
82         0x9e, 0xb7, 0x6f, 0xac, 0x45, 0xaf, 0x8e, 0x51};
83 const BYTE dataOut_AES256_ECB [] = {
84         0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c,
85         0x06, 0x4b, 0x5a, 0x7e, 0x3d, 0xb1, 0x81, 0xf8,
86         0x59, 0x1c, 0xcb, 0x10, 0xd4, 0x10, 0xed, 0x26,
87         0xdc, 0x5b, 0xa7, 0x4a, 0x31, 0x36, 0x28, 0x70};
88 const BYTE dataOut_AES256_CBC [] = {
89         0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0xba,
90         0x77, 0x9e, 0xab, 0xfb, 0x5f, 0x7b, 0xfb, 0xd6,
91         0x9c, 0xfc, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d,
92         0x67, 0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d};
93 const BYTE dataOut_AES256_CFB [] = {
94         0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
95         0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
96         0x39, 0xff, 0xed, 0x14, 0x3b, 0x28, 0xb1, 0xc8,
97         0x32, 0x11, 0x3c, 0x63, 0x31, 0xe5, 0x40, 0x7b};
98 const BYTE dataOut_AES256_OFB [] = {
99         0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
100        0x7e, 0xcd, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
101        0x4f, 0xeb, 0xdc, 0x67, 0x40, 0xd2, 0x0b, 0x3a,
102        0xc8, 0x8f, 0x6a, 0xd8, 0x2a, 0x4f, 0xb0, 0x8d};
103 const BYTE dataOut_AES256_CTR [] = {
104        0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0xa5,
105        0xb7, 0xa7, 0xf5, 0x04, 0xbb, 0xf3, 0xd2, 0x28,
106        0xf4, 0x43, 0xe3, 0xca, 0x4d, 0x62, 0xb5, 0x9a,
107        0xca, 0x84, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5};
108 #endif

```

10.1.11 SymmetricTest.h

10.1.11.1 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  #ifndef      _SYMMETRIC_TEST_H
5  #define      _SYMMETRIC_TEST_H
6  #include     "SymmetricTestData.h"

```

10.1.11.2 Symmetric Test Structures

```

7  const SYMMETRIC_TEST_VECTOR  c_symTestValues[NUM_SYMS + 1] = {
8  #if ALG_AES && AES_128
9      {ALG_AES_VALUE, 128, key_AES128, 16, sizeof(dataIn_AES128), dataIn_AES128,
10      {dataOut_AES128_CTR, dataOut_AES128_OFB, dataOut_AES128_CBC,
11      dataOut_AES128_CFB, dataOut_AES128_ECB}},
12  #endif
13  #if ALG_AES && AES_192
14      {ALG_AES_VALUE, 192, key_AES192, 16, sizeof(dataIn_AES192), dataIn_AES192,
15      {dataOut_AES192_CTR, dataOut_AES192_OFB, dataOut_AES192_CBC,
16      dataOut_AES192_CFB, dataOut_AES192_ECB}},
17  #endif
18  #if ALG_AES && AES_256
19      {ALG_AES_VALUE, 256, key_AES256, 16, sizeof(dataIn_AES256), dataIn_AES256,
20      {dataOut_AES256_CTR, dataOut_AES256_OFB, dataOut_AES256_CBC,
21      dataOut_AES256_CFB, dataOut_AES256_ECB}},
22  #endif
23  #if ALG_SM4 && SM4_128
24      {ALG_SM4_VALUE, 128, key_SM4128, 16, sizeof(dataIn_SM4128), dataIn_SM4128,
25      {dataOut_SM4128_CTR, dataOut_SM4128_OFB, dataOut_SM4128_CBC,
26      dataOut_SM4128_CFB, dataOut_AES128_ECB}},
27  #endif
28      {0}
29  };
30  #endif // _SYMMETRIC_TEST_H

```


10.1.12 EccTestData.h

This file contains the parameter data for ECC testing.

```
1  #ifndef SELF_TEST_DATA
2  TPM2B_TYPE(EC_TEST, 32);
3  const TPM_ECC_CURVE      c_testCurve = 00003;
```

The **static** key

```
4  const TPM2B_EC_TEST      c_ecTestKey_ds = {{32, {
5      0xdf,0x8d,0xa4,0xa3,0x88,0xf6,0x76,0x96,0x89,0xfc,0x2f,0x2d,0xa1,0xb4,0x39,0x7a,
6      0x78,0xc4,0x7f,0x71,0x8c,0xa6,0x91,0x85,0xc0,0xbf,0xf3,0x54,0x20,0x91,0x2f,0x73}}};
7  const TPM2B_EC_TEST      c_ecTestKey_QsX = {{32, {
8      0x17,0xad,0x2f,0xcb,0x18,0xd4,0xdb,0x3f,0x2c,0x53,0x13,0x82,0x42,0x97,0xff,0x8d,
9      0x99,0x50,0x16,0x02,0x35,0xa7,0x06,0xae,0x1f,0xda,0xe2,0x9c,0x12,0x77,0xc0,0xf9}}};
10 const TPM2B_EC_TEST      c_ecTestKey_QsY = {{32, {
11     0xa6,0xca,0xf2,0x18,0x45,0x96,0x6e,0x58,0xe6,0x72,0x34,0x12,0x89,0xcd,0xaa,0xad,
12     0xcb,0x68,0xb2,0x51,0xdc,0x5e,0xd1,0x6d,0x38,0x20,0x35,0x57,0xb2,0xfd,0xc7,0x52}}};
```

The **ephemeral** key

```
13 const TPM2B_EC_TEST      c_ecTestKey_de = {{32, {
14     0xb6,0xb5,0x33,0x5c,0xd1,0xee,0x52,0x07,0x99,0xea,0x2e,0x8f,0x8b,0x19,0x18,0x07,
15     0xc1,0xf8,0xdf,0xdd,0xb8,0x77,0x00,0xc7,0xd6,0x53,0x21,0xed,0x02,0x53,0xee,0xac}}};
16 const TPM2B_EC_TEST      c_ecTestKey_QeX = {{32, {
17     0xa5,0x1e,0x80,0xd1,0x76,0x3e,0x8b,0x96,0xce,0xcc,0x21,0x82,0xc9,0xa2,0xa2,0xed,
18     0x47,0x21,0x89,0x53,0x44,0xe9,0xc7,0x92,0xe7,0x31,0x48,0x38,0xe6,0xea,0x93,0x47}}};
19 const TPM2B_EC_TEST      c_ecTestKey_QeY = {{32, {
20     0x30,0xe6,0x4f,0x97,0x03,0xa1,0xcb,0x3b,0x32,0x2a,0x70,0x39,0x94,0xeb,0x4e,0xea,
21     0x55,0x88,0x81,0x3f,0xb5,0x00,0xb8,0x54,0x25,0xab,0xd4,0xda,0xfd,0x53,0x7a,0x18}}};
```

ECDH test results

```
22 const TPM2B_EC_TEST      c_ecTestEcdh_X = {{32, {
23     0x64,0x02,0x68,0x92,0x78,0xdb,0x33,0x52,0xed,0x3b,0xfa,0x3b,0x74,0xa3,0x3d,0x2c,
24     0x2f,0x9c,0x59,0x03,0x07,0xf8,0x22,0x90,0xed,0xe3,0x45,0xf8,0x2a,0x0a,0xd8,0x1d}}};
25 const TPM2B_EC_TEST      c_ecTestEcdh_Y = {{32, {
26     0x58,0x94,0x05,0x82,0xbe,0x5f,0x33,0x02,0x25,0x90,0x3a,0x33,0x90,0x89,0xe3,0xe5,
27     0x10,0x4a,0xbc,0x78,0xa5,0xc5,0x07,0x64,0xaf,0x91,0xbc,0xe6,0xff,0x85,0x11,0x40}}};
28 TPM2B_TYPE(TEST_VALUE, 64);
29 const TPM2B_TEST_VALUE    c_ecTestValue = {{64, {
30     0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
31     0xf4,0x45,0x7f,0x13,0xd8,0x13,0x81,0x0b,0xe5,0x76,0xbe,0xaa,0xb6,0x3f,0x8d,0x4d,
32     0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xcb,0x0c,0xc7,0x11,0x8d,0xc3,0xff,
33     0x62,0x69,0xa2,0xbe,0x46,0x90,0xe7,0xf4,0x81,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}}};
34 #if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
35 const TPM2B_EC_TEST      c_TestEcDsa_r = {{32, {
36     0x57,0xf3,0x36,0xb7,0xec,0xc2,0xdd,0x76,0x0e,0xe2,0x81,0x21,0x49,0xc5,0x66,0x11,
37     0x4b,0x8a,0x4f,0x17,0x62,0x82,0xcc,0x06,0xf6,0x64,0x78,0xef,0x6b,0x7c,0xf2,0x6c}}};
38 const TPM2B_EC_TEST      c_TestEcDsa_s = {{32, {
39     0x1b,0xed,0x23,0x72,0x8f,0x17,0x5f,0x47,0x2e,0xa7,0x97,0x2c,0x51,0x57,0x20,0x70,
40     0x6f,0x89,0x74,0x8a,0xa8,0xf4,0x26,0xf4,0x96,0xa1,0xb8,0x3e,0xe5,0x35,0xc5,0x94}}};
41 const TPM2B_EC_TEST      c_TestEcSchnorr_r = {{32,{
42     0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x1b,0x08,0x9f,0xde,
43     0xef,0x62,0xe3,0xf1,0x14,0xcb,0x54,0x28,0x13,0x76,0xfc,0x6d,0x69,0x22,0xb5,0x3e}}};
44 const TPM2B_EC_TEST      c_TestEcSchnorr_s = {{32,{
45     0xd9,0xd3,0x20,0xfb,0x4d,0x16,0xf2,0xe6,0xe2,0x45,0x07,0x45,0x1c,0x92,0x92,0x92,
46     0xa9,0x6b,0xa8,0xf8,0xd1,0x98,0x29,0x4d,0xd3,0x8f,0x56,0xf2,0xbb,0x2e,0x22,0x3b}}};
47 #endif // SHA1
48 #if ALG_SHA256_VALUE == DEFAULT_TEST_HASH
49 const TPM2B_EC_TEST      c_TestEcDsa_r = {{32, {
50     0x04,0x7d,0x54,0xeb,0x04,0x6f,0x56,0xec,0xa2,0x6c,0x38,0x8c,0xeb,0x43,0x0b,0x71,
```



```
51     0xf8,0xf2,0xf4,0xa5,0xe0,0x1d,0x3c,0xa2,0x39,0x31,0xe4,0xe7,0x36,0x3b,0xb5,0x5f}}};
52 const TPM2B_EC_TEST    c_TestEcDsa_s = {{32, {
53     0x8f,0xd0,0x12,0xd9,0x24,0x75,0xf6,0xc4,0x3b,0xb5,0x46,0x75,0x3a,0x41,0x8d,0x80,
54     0x23,0x99,0x38,0xd7,0xe2,0x40,0xca,0x9a,0x19,0x2a,0xfc,0x54,0x75,0xd3,0x4a,0x6e}}};
55 const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32, {
56     0xf7,0xb9,0x15,0x4c,0x34,0xf6,0x41,0x19,0xa3,0xd2,0xf1,0xbd,0xf4,0x13,0x6a,0x4f,
57     0x63,0xb8,0x4d,0xb5,0xc8,0xcd,0xde,0x85,0x95,0xa5,0x39,0x0a,0x14,0x49,0x3d,0x2f}}};
58 const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32,{
59     0xfe,0xbe,0x17,0xaa,0x31,0x22,0x9f,0xd0,0xd2,0xf5,0x25,0x04,0x92,0xb0,0xaa,0x4e,
60     0xcc,0x1c,0xb6,0x79,0xd6,0x42,0xb3,0x4e,0x3f,0xbb,0xfe,0x5f,0xd0,0xd0,0x8b,0xc3}}};
61 #endif // SHA256
62 #if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
63 const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
64     0xf5,0x74,0x6d,0xd6,0xc6,0x56,0x86,0xbb,0xba,0x1c,0xba,0x75,0x65,0xee,0x64,0x31,
65     0xce,0x04,0xe3,0x9f,0x24,0x3f,0xbd,0xfe,0x04,0xcd,0xab,0x7e,0xfe,0xad,0xcb,0x82}}};
66 const TPM2B_EC_TEST    c_TestEcDsa_s = {{32, {
67     0xc2,0x4f,0x32,0xa1,0x06,0xc0,0x85,0x4f,0xc6,0xd8,0x31,0x66,0x91,0x9f,0x79,0xcd,
68     0x5b,0xe5,0x7b,0x94,0xa1,0x91,0x38,0xac,0xd4,0x20,0xa2,0x10,0xf0,0xd5,0x9d,0xbf}}};
69 const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32, {
70     0x1e,0xb8,0xe1,0xbf,0xa1,0x9e,0x39,0x1e,0x58,0xa2,0xe6,0x59,0xd0,0x1a,0x6a,0x03,
71     0x6a,0x1f,0x1c,0x4f,0x36,0x19,0xc1,0xec,0x30,0xa4,0x85,0x1b,0xe9,0x74,0x35,0x66}}};
72 const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32,{
73     0xb9,0xe6,0xe3,0x7e,0xcb,0xb9,0xea,0xf1,0xcc,0xf4,0x48,0x44,0x4a,0xda,0xc8,0xd7,
74     0x87,0xb4,0xba,0x40,0xfe,0x5b,0x68,0x11,0x14,0xcf,0xa0,0x0e,0x85,0x46,0x99,0x01}}};
75 #endif // SHA384
76 #if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
77 const TPM2B_EC_TEST    c_TestEcDsa_r = {{32, {
78     0xc9,0x71,0xa6,0xb4,0xaf,0x46,0x26,0x8c,0x27,0x00,0x06,0x3b,0x00,0x0f,0xa3,0x17,
79     0x72,0x48,0x40,0x49,0x4d,0x51,0x4f,0xa4,0xcb,0x7e,0x86,0xe9,0xe7,0xb4,0x79,0xb2}}};
80 const TPM2B_EC_TEST    c_TestEcDsa_s = {{32,{
81     0x87,0xbc,0xc0,0xed,0x74,0x60,0x9e,0xfa,0x4e,0xe8,0x16,0xf3,0xf9,0x6b,0x26,0x07,
82     0x3c,0x74,0x31,0x7e,0xf0,0x62,0x46,0xdc,0xd6,0x45,0x22,0x47,0x3e,0x0c,0xa0,0x02}}};
83 const TPM2B_EC_TEST    c_TestEcSchnorr_r = {{32,{
84     0xcc,0x07,0xad,0x65,0x91,0xdd,0xa0,0x10,0x23,0xae,0x53,0xec,0xdf,0xf1,0x50,0x90,
85     0x16,0x96,0xf4,0x45,0x09,0x73,0x9c,0x84,0xb5,0x5c,0x5f,0x08,0x51,0xcb,0x60,0x01}}};
86 const TPM2B_EC_TEST    c_TestEcSchnorr_s = {{32,{
87     0x55,0x20,0x21,0x54,0xe2,0x49,0x07,0x47,0x71,0xf4,0x99,0x15,0x54,0xf3,0xab,0x14,
88     0xdb,0x8e,0xda,0x79,0xb6,0x02,0x0e,0xe3,0x5e,0x6f,0x2c,0xb6,0x05,0xbd,0x14,0x10}}};
89 #endif // SHA512
90 #endif // SELF_TEST_DATA
```

10.1.13 CryptSym.h

10.1.13.1 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.

10.1.13.2 Includes, Defines, and Typedefs

```

1  #ifndef CRYPT_SYM_H
2  #define CRYPT_SYM_H
3  typedef union tpmCryptKeySchedule_t {
4  #if ALG_AES
5      tpmKeyScheduleAES          AES;
6  #endif
7  #if ALG_SM4
8      tpmKeyScheduleSM4          SM4;
9  #endif
10 #if ALG_CAMELLIA
11     tpmKeyScheduleCAMELLIA      CAMELLIA;
12 #endif
13
14 #if ALG_TDES
15     tpmKeyScheduleTDES          TDES[3];
16 #endif
17 #if SYMMETRIC_ALIGNMENT == 8
18     uint64_t                    alignment;
19 #else
20     uint32_t                    alignment;
21 #endif
22 } 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.

```

23 # define ENCRYPT(keySchedule, in, out)          \
24     encrypt(SWIZZLE(keySchedule, in, out))
25 # define DECRYPT(keySchedule, in, out)          \
26     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.

```

27 #define ENCRYPT_CASE(ALG)                        \
28     case TPM_ALG_##ALG:                        \
29         TpmCryptSetEncryptKey##ALG(key, keySizeInBits, &keySchedule.ALG); \
30         encrypt = (TpmCryptSetSymKeyCall_t)TpmCryptEncrypt##ALG; \
31         break;
32 #define DECRYPT_CASE(ALG)                        \
33     case TPM_ALG_##ALG:                        \
34         TpmCryptSetDecryptKey##ALG(key, keySizeInBits, &keySchedule.ALG); \
35         decrypt = (TpmCryptSetSymKeyCall_t)TpmCryptDecrypt##ALG; \
36         break;
37 #if ALG_AES
38 #define ENCRYPT_CASE_AES    ENCRYPT_CASE(AES)

```

```

39  #define DECRYPT_CASE_AES      DECRYPT_CASE (AES)
40  #else
41  #define ENCRYPT_CASE_AES
42  #define DECRYPT_CASE_AES
43  #endif
44  #if ALG_SM4
45  #define ENCRYPT_CASE_SM4      ENCRYPT_CASE (SM4)
46  #define DECRYPT_CASE_SM4      DECRYPT_CASE (SM4)
47  #else
48  #define ENCRYPT_CASE_SM4
49  #define DECRYPT_CASE_SM4
50  #endif
51  #if ALG_CAMELLIA
52  #define ENCRYPT_CASE_CAMELLIA  ENCRYPT_CASE (CAMELLIA)
53  #define DECRYPT_CASE_CAMELLIA  DECRYPT_CASE (CAMELLIA)
54  #else
55  #define ENCRYPT_CASE_CAMELLIA
56  #define DECRYPT_CASE_CAMELLIA
57  #endif
58  #if ALG_TDES
59  #define ENCRYPT_CASE_TDES      ENCRYPT_CASE (TDES)
60  #define DECRYPT_CASE_TDES      DECRYPT_CASE (TDES)
61  #else
62  #define ENCRYPT_CASE_TDES
63  #define DECRYPT_CASE_TDES
64  #endif

```

For each algorithm the case will either be defined or null.

```

65  #define      SELECT(direction)
66  switch(algorithm)
67  {
68      direction##_CASE_AES
69      direction##_CASE_SM4
70      direction##_CASE_CAMELLIA
71      direction##_CASE_TDES
72      default:
73          FAIL(FATAL_ERROR_INTERNAL);
74  }
75  #endif // CRYPT_SYM_H

```

10.1.14 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).

```

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

```

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.

```

5  #ifndef ALG_SHA1
6  #   define ALG_SHA1 NO
7  #endif
8  #if ALG_SHA1
9  #define SHA1_OID(NAME) MAKE_OID(NAME##SHA1)
10 #else
11 #define SHA1_OID(NAME)
12 #endif
13 #ifndef ALG_SHA256
14 #   define ALG_SHA256 NO
15 #endif
16 #if ALG_SHA256
17 #define SHA256_OID(NAME) MAKE_OID(NAME##SHA256)
18 #else
19 #define SHA256_OID(NAME)
20 #endif
21 #ifndef ALG_SHA384
22 #   define ALG_SHA384 NO
23 #endif
24 #if ALG_SHA384
25 #define SHA384_OID(NAME) MAKE_OID(NAME##SHA384)
26 #else
27 #define SHA384_OID(NAME)
28 #endif
29 #ifndef ALG_SHA512
30 #   define ALG_SHA512 NO
31 #endif
32 #if ALG_SHA512
33 #define SHA512_OID(NAME) MAKE_OID(NAME##SHA512)
34 #else
35 #define SHA512_OID(NAME)
36 #endif
37 #ifndef ALG_SM3_256
38 #   define ALG_SM3_256 NO
39 #endif
40 #if ALG_SM3_256
41 #define SM3_256_OID(NAME) MAKE_OID(NAME##SM3_256)
42 #else
43 #define SM3_256_OID(NAME)
44 #endif
45 #ifndef ALG_SHA3_256
46 #   define ALG_SHA3_256 NO
47 #endif
48 #if ALG_SHA3_256
49 #define SHA3_256_OID(NAME) MAKE_OID(NAME##SHA3_256)

```

```

50  #else
51  #define SHA3_256_OID(NAME)
52  #endif
53  #ifndef ALG_SHA3_384
54  #   define ALG_SHA3_384 NO
55  #endif
56  #if ALG_SHA3_384
57  #define SHA3_384_OID(NAME) MAKE_OID(NAME##SHA3_384)
58  #else
59  #define SHA3_384_OID(NAME)
60  #endif
61  #ifndef ALG_SHA3_512
62  #   define ALG_SHA3_512 NO
63  #endif
64  #if ALG_SHA3_512
65  #define SSHA3_512_OID(NAME) MAKE_OID(NAME##SHA3_512)
66  #else
67  #define SHA3_512_OID(NAME)
68  #endif

```

These are encoded to take one additional byte of algorithm selector

```

69  #define NIST_HASH      0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 2
70  #define NIST_SIG       0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 3

```

These hash OIDs used in a lot of places.

```

71  #define OID_SHA1_VALUE      0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A
72  SHA1_OID(_);              // Expands to
73                             // MAKE_OID(_SHA1)
74                             // which expands to:
75                             // extern BYTE      OID_SHA1[]
76                             // or
77                             // const BYTE      OID_SHA1[] = {OID_SHA1_VALUE}
78                             // which is:
79                             // const BYTE      OID_SHA1[] = {0x06, 0x05, 0x2B, 0x0E,
80                             //                                0x03, 0x02, 0x1A}
81  #define OID_SHA256_VALUE    NIST_HASH, 1
82  SHA256_OID(_);
83  #define OID_SHA384_VALUE    NIST_HASH, 2
84  SHA384_OID(_);
85  #define OID_SHA512_VALUE    NIST_HASH, 3
86  SHA512_OID(_);
87  #define OID_SM3_256_VALUE    0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, \
88                                0x83, 0x11
89  SM3_256_OID(_);           // (1.2.156.10197.1.401)
90  #define OID_SHA3_256_VALUE    NIST_HASH, 8
91  SHA3_256_OID(_);
92  #define OID_SHA3_384_VALUE    NIST_HASH, 9
93  SHA3_384_OID(_);
94  #define OID_SHA3_512_VALUE    NIST_HASH, 10
95  SHA3_512_OID(_);

```

These are used for RSA-PSS

```

96  #if ALG_RSA
97  #define OID_MGF1_VALUE      0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
98                                0x01, 0x01, 0x08
99  MAKE_OID(_MGF1);
100  #define OID_RSAPSS_VALUE    0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
101                                0x01, 0x01, 0x0A
102  MAKE_OID(_RSAPSS);

```

This is the OID to designate the public part of an RSA key.

```

103 #define OID_PKCS1_PUB_VALUE      0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \
104                                   0x01, 0x01, 0x01
105 MAKE_OID( _PKCS1_PUB );

```

These are used for RSA PKCS1 signature Algorithms

```

106 #define OID_PKCS1_SHA1_VALUE      0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,      \
107                                   0x0D, 0x01, 0x01, 0x05
108 SHA1_OID( _PKCS1_ );      // (1.2.840.113549.1.1.5)
109 #define OID_PKCS1_SHA256_VALUE    0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,      \
110                                   0x0D, 0x01, 0x01, 0x0B
111 SHA256_OID( _PKCS1_ );    // (1.2.840.113549.1.1.11)
112 #define OID_PKCS1_SHA384_VALUE    0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,      \
113                                   0x0D, 0x01, 0x01, 0x0C
114 SHA384_OID( _PKCS1_ );    // (1.2.840.113549.1.1.12)
115 #define OID_PKCS1_SHA512_VALUE    0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7,      \
116                                   0x0D, 0x01, 0x01, 0x0D
117 SHA512_OID( _PKCS1_ );    // (1.2.840.113549.1.1.13)
118 #define OID_PKCS1_SM3_256_VALUE   0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55,      \
119                                   0x01, 0x83, 0x78
120 SM3_256_OID( _PKCS1_ );    // 1.2.156.10197.1.504
121 #define OID_PKCS1_SHA3_256_VALUE  NIST_SIG, 14
122 SHA3_256_OID( _PKCS1_ );
123 #define OID_PKCS1_SHA3_384_VALUE  NIST_SIG, 15
124 SHA3_384_OID( _PKCS1_ );
125 #define OID_PKCS1_SHA3_512_VALUE  NIST_SIG, 16
126 SHA3_512_OID( _PKCS1_ );
127 #endif // ALG_RSA
128 #if ALG_ECDSA
129 #define OID_ECDSA_SHA1_VALUE      0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
130                                   0x01
131 SHA1_OID( _ECDSA_ );      // (1.2.840.10045.4.1) SHA1 digest signed by an ECDSA key.
132 #define OID_ECDSA_SHA256_VALUE    0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
133                                   0x03, 0x02
134 SHA256_OID( _ECDSA_ );    // (1.2.840.10045.4.3.2) SHA256 digest signed by an ECDSA key.
135 #define OID_ECDSA_SHA384_VALUE    0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
136                                   0x03, 0x03
137 SHA384_OID( _ECDSA_ );    // (1.2.840.10045.4.3.3) SHA384 digest signed by an ECDSA key.
138 #define OID_ECDSA_SHA512_VALUE    0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x04, \
139                                   0x03, 0x04
140 SHA512_OID( _ECDSA_ );    // (1.2.840.10045.4.3.4) SHA512 digest signed by an ECDSA key.
141 #define OID_ECDSA_SM3_256_VALUE   0x00
142 SM3_256_OID( _ECDSA_ );
143 #define OID_ECDSA_SHA3_256_VALUE  NIST_SIG, 10
144 SHA3_256_OID( _ECDSA_ );
145 #define OID_ECDSA_SHA3_384_VALUE  NIST_SIG, 11
146 SHA3_384_OID( _ECDSA_ );
147 #define OID_ECDSA_SHA3_512_VALUE  NIST_SIG, 12
148 SHA3_512_OID( _ECDSA_ );
149 #endif // ALG_ECDSA
150 #if ALG_ECC
151 #define OID_ECC_PUBLIC_VALUE      0x06, 0x07, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x02, \
152                                   0x01
153 MAKE_OID( _ECC_PUBLIC );
154 #define OID_ECC_NIST_P192_VALUE   0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, \
155                                   0x01, 0x01
156 #if ECC_NIST_P192
157 MAKE_OID( _ECC_NIST_P192 );    // (1.2.840.10045.3.1.1) 'nistP192'
158 #endif // ECC_NIST_P192
159 #define OID_ECC_NIST_P224_VALUE   0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x21
160 #if ECC_NIST_P224
161 MAKE_OID( _ECC_NIST_P224 );    // (1.3.132.0.33) 'nistP224'
162 #endif // ECC_NIST_P224
163 #define OID_ECC_NIST_P256_VALUE   0x06, 0x08, 0x2A, 0x86, 0x48, 0xCE, 0x3D, 0x03, \
164                                   0x01, 0x07

```

```

165  #if ECC_NIST_P256
166  MAKE_OID(_ECC_NIST_P256);    // (1.2.840.10045.3.1.7)  'nistP256'
167  #endif // ECC_NIST_P256
168  #define OID_ECC_NIST_P384_VALUE    0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x22
169  #if ECC_NIST_P384
170  MAKE_OID(_ECC_NIST_P384);    // (1.3.132.0.34)          'nistP384'
171  #endif // ECC_NIST_P384
172  #define OID_ECC_NIST_P521_VALUE    0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x23
173  #if ECC_NIST_P521
174  MAKE_OID(_ECC_NIST_P521);    // (1.3.132.0.35)          'nistP521'
175  #endif // ECC_NIST_P521

```

No OIDs defined for these anonymous curves

```

176  #define OID_ECC_BN_P256_VALUE    0x00
177  #if ECC_BN_P256
178  MAKE_OID(_ECC_BN_P256);
179  #endif // ECC_BN_P256
180  #define OID_ECC_BN_P638_VALUE    0x00
181  #if ECC_BN_P638
182  MAKE_OID(_ECC_BN_P638);
183  #endif // ECC_BN_P638
184  #define OID_ECC_SM2_P256_VALUE    0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, \
185                                     0x82, 0x2D
186  #if ECC_SM2_P256
187  MAKE_OID(_ECC_SM2_P256);    // Don't know where I found this OID. It needs checking
188  #endif // ECC_SM2_P256
189  #if ECC_BN_P256
190  #define OID_ECC_BN_P256    NULL
191  #endif // ECC_BN_P256
192  #endif // ALG_ECC
193  #undef MAKE_OID
194  #define OID_SIZE(OID)    (OID[1] + 2)
195  #endif // !_OIDS_H

```


10.1.15 TpmAsn1.h

10.1.15.1 Introduction

This file contains the macro and structure definitions for the X509 commands and functions.

```
1  #ifndef _TPMASN1_H_
2  #define _TPMASN1_H_
```

10.1.15.2 Includes

```
3  #include "Tpm.h"
4  #include "OIDS.h"
```

10.1.15.3 Defined Constants

10.1.15.3.1 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 #define ASN1_APPLICATION_SPECIFIC 0xA0
35 #define ASN1_CONSTRUCTED_SEQUENCE (ASN1_SEQUENCE + ASN1_CONSTRUCTED)
36 #define MAX_DEPTH                10    // maximum push depth for marshaling context.
```

10.1.15.4 Macros

10.1.15.4.1 Unmarshaling Macros

```
37 #ifndef VERIFY
38 #define VERIFY(_X_) {if(!(_X_)) goto Error; }
39 #endif
```

Checks the validity of the size making sure that there is no wrap around

```

40 #define CHECK_SIZE(context, length) \
41     VERIFY( ((length) + (context)->offset) >= (context)->offset) \
42             && ((length) + (context)->offset) <= (context)->size))
43 #define NEXT_OCTET(context) ((context)->buffer[(context)->offset++])
44 #define PEEK_NEXT(context) ((context)->buffer[(context)->offset])

```

10.1.15.4.2 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 (when the

```

45 #define CHECK_SPACE(context, length)    VERIFY(context->offset > length)

```

10.1.15.5 Structures

```

46 typedef struct ASN1UnmarshalContext {
47     BYTE        *buffer;    // pointer to the buffer
48     INT16       size;       // size of the buffer (a negative number indicates
49                             // a parsing failure).
50     INT16       offset;     // current offset into the buffer (a negative number
51                             // indicates a parsing failure). Not used
52     BYTE        tag;        // The last unmarshaled tag
53 } ASN1UnmarshalContext;
54 typedef struct ASN1MarshalContext {
55     BYTE        *buffer;    // pointer to the start of the buffer
56     INT16       offset;     // place on the top where the last entry was added
57                             // items are added from the bottom up.
58     INT16       end;        // the end offset of the current value
59     INT16       depth;      // how many pushed end values.
60     INT16       ends[MAX_DEPTH];
61 } ASN1MarshalContext;
62 #endif // _TPMASN1_H_

```

10.1.16 X509.h

10.1.16.1 Introduction

This file contains the macro and structure definitions for the X509 commands and functions.

```
1  #ifndef _X509_H_
2  #define _X509_H_
```

10.1.16.2 Includes

```
3  #include "Tpm.h"
4  #include "TpmASN1.h"
```

10.1.16.3 Defined Constants

10.1.16.3.1 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)
19 #undef MAKE_OID
20 #ifdef _X509_SPT_
21 #   define MAKE_OID(NAME)        \
22         const BYTE    OID##NAME[] = {OID##NAME##_VALUE}
23 #else
24 #   define MAKE_OID(NAME)        \
25         extern const BYTE    OID##NAME[]
26 #endif
```

10.1.16.4 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.

```
27 typedef struct stringRef
28 {
29     BYTE        *buf;
30     INT16       len;
31 } stringRef;
32 typedef union x509KeyUsageUnion {
33     TPMA_X509_KEY_USAGE    x509;
34     UINT32                 integer;
```

```
35 } x509KeyUsageUnion;
```

10.1.16.5 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

```
36 #define OID_KEY_USAGE_EXTENSTION_VALUE 0x06, 0x03, 0x55, 0x1D, 0x0F
37 MAKE_OID(_KEY_USAGE_EXTENSTION);
```

This is the DER-encoded value for the TCG-defined TPMA_OBJECT OID (2.23.133.10.1.1.1)

```
38 #define OID_TCG_TPMA_OBJECT_VALUE      0x06, 0x07, 0x67, 0x81, 0x05, 0x0a, 0x01, \
39                                         0x01, 0x01
40 MAKE_OID(_TCG_TPMA_OBJECT);
41 #ifdef _X509_SPT_
42 const x509KeyUsageUnion keyUsageSign = TPMA_X509_KEY_USAGE_INITIALIZER(
43     /* digitalsignature */ 1, /* nonrepudiation */ 0,
44     /* keyencipherment */ 0, /* dataencipherment */ 0,
45     /* keyagreement */ 0, /* keycertsign */ 1,
46     /* crlsign */ 1, /* encipheronly */ 0,
47     /* decipheronly */ 0, /* bits_at_9 */ 0);
48 const x509KeyUsageUnion keyUsageDecrypt = TPMA_X509_KEY_USAGE_INITIALIZER(
49     /* digitalsignature */ 0, /* nonrepudiation */ 0,
50     /* keyencipherment */ 1, /* dataencipherment */ 1,
51     /* keyagreement */ 1, /* keycertsign */ 0,
52     /* crlsign */ 0, /* encipheronly */ 1,
53     /* decipheronly */ 1, /* bits_at_9 */ 0);
54 #else
55 extern x509KeyUsageUnion keyUsageSign;
56 extern x509KeyUsageUnion keyUsageDecrypt;
57 #endif
58 #undef MAKE_OID
59 #endif // _X509_H_
```

10.1.17 MinMax.h

```
1  #ifndef _MIN_MAX_H_
2  #define _MIN_MAX_H_
3  #ifndef MAX
4  #define MAX(a, b) ((a) > (b) ? (a) : (b))
5  #endif
6  #ifndef MIN
7  #define MIN(a, b) ((a) < (b) ? (a) : (b))
8  #endif
9  #endif // _MIN_MAX_H_
```

10.1.18 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_
```

Table 2:3 - Definition of Base Types Base Types are in BaseTypes.h

```
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)
```

Table 0:6 - 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
```

Table 1:12 - Defines for SHA1 Hash Values

```
22 #define SHA1_DIGEST_SIZE 20
23 #define SHA1_BLOCK_SIZE 64
```

Table 1:13 - Defines for SHA256 Hash Values

```
24 #define SHA256_DIGEST_SIZE 32
25 #define SHA256_BLOCK_SIZE 64
```

Table 1:14 - Defines for SHA384 Hash Values

```
26 #define SHA384_DIGEST_SIZE 48
27 #define SHA384_BLOCK_SIZE 128
```

Table 1:15 - Defines for SHA512 Hash Values

```
28 #define SHA512_DIGEST_SIZE 64
29 #define SHA512_BLOCK_SIZE 128
```

Table 1:16 - Defines for SM3_256 Hash Values

```
30 #define SM3_256_DIGEST_SIZE 32
31 #define SM3_256_BLOCK_SIZE 64
```

Table 1:16 - Defines for SHA3_256 Hash Values

```
32 #define SHA3_256_DIGEST_SIZE 32
```

```
33 #define SHA3_256_BLOCK_SIZE      136
```

Table 1:16 - Defines for SHA3_384 Hash Values

```
34 #define SHA3_384_DIGEST_SIZE      48
35 #define SHA3_384_BLOCK_SIZE      104
```

Table 1:16 - Defines for SHA3_512 Hash Values

```
36 #define SHA3_512_DIGEST_SIZE      64
37 #define SHA3_512_BLOCK_SIZE      72
```

Table 1:00 - Defines for RSA Asymmetric Cipher Algorithm Constants

```
38 #define RSA_KEY_SIZES_BITS        \
39     (1024 * RSA_1024), (2048 * RSA_2048), (3072 * RSA_3072), \
40     (4096 * RSA_4096)
41 #if RSA_4096
42 #   define RSA_MAX_KEY_SIZE_BITS  4096
43 #elif RSA_3072
44 #   define RSA_MAX_KEY_SIZE_BITS  3072
45 #elif RSA_2048
46 #   define RSA_MAX_KEY_SIZE_BITS  2048
47 #elif RSA_1024
48 #   define RSA_MAX_KEY_SIZE_BITS  1024
49 #else
50 #   define RSA_MAX_KEY_SIZE_BITS  0
51 #endif
52 #define MAX_RSA_KEY_BITS          RSA_MAX_KEY_SIZE_BITS
53 #define MAX_RSA_KEY_BYTES         ((RSA_MAX_KEY_SIZE_BITS + 7) / 8)
```

Table 1:17 - Defines for AES Symmetric Cipher Algorithm Constants

```
54 #define AES_KEY_SIZES_BITS        \
55     (128 * AES_128), (192 * AES_192), (256 * AES_256)
56 #if AES_256
57 #   define AES_MAX_KEY_SIZE_BITS  256
58 #elif AES_192
59 #   define AES_MAX_KEY_SIZE_BITS  192
60 #elif AES_128
61 #   define AES_MAX_KEY_SIZE_BITS  128
62 #else
63 #   define AES_MAX_KEY_SIZE_BITS  0
64 #endif
65 #define MAX_AES_KEY_BITS          AES_MAX_KEY_SIZE_BITS
66 #define MAX_AES_KEY_BYTES         ((AES_MAX_KEY_SIZE_BITS + 7) / 8)
67 #define AES_128_BLOCK_SIZE_BYTES (AES_128 * 16)
68 #define AES_192_BLOCK_SIZE_BYTES (AES_192 * 16)
69 #define AES_256_BLOCK_SIZE_BYTES (AES_256 * 16)
70 #define AES_BLOCK_SIZES          \
71     AES_128_BLOCK_SIZE_BYTES, AES_192_BLOCK_SIZE_BYTES, \
72     AES_256_BLOCK_SIZE_BYTES
73 #if ALG_AES
74 #   define AES_MAX_BLOCK_SIZE     16
75 #else
76 #   define AES_MAX_BLOCK_SIZE     0
77 #endif
78 #define MAX_AES_BLOCK_SIZE_BYTES AES_MAX_BLOCK_SIZE
```

Table 1:18 - Defines for SM4 Symmetric Cipher Algorithm Constants

```
79 #define SM4_KEY_SIZES_BITS        (128 * SM4_128)
80 #if SM4_128
```



```

81 # define SM4_MAX_KEY_SIZE_BITS 128
82 #else
83 # define SM4_MAX_KEY_SIZE_BITS 0
84 #endif
85 #define MAX_SM4_KEY_BITS SM4_MAX_KEY_SIZE_BITS
86 #define MAX_SM4_KEY_BYTES ((SM4_MAX_KEY_SIZE_BITS + 7) / 8)
87 #define SM4_128_BLOCK_SIZE_BYTES (SM4_128 * 16)
88 #define SM4_BLOCK_SIZES SM4_128_BLOCK_SIZE_BYTES
89 #if ALG_SM4
90 # define SM4_MAX_BLOCK_SIZE 16
91 #else
92 # define SM4_MAX_BLOCK_SIZE 0
93 #endif
94 #define MAX_SM4_BLOCK_SIZE_BYTES SM4_MAX_BLOCK_SIZE

```

Table 1:19 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```

95 #define CAMELLIA_KEY_SIZES_BITS \
96     (128 * CAMELLIA_128), (192 * CAMELLIA_192), (256 * CAMELLIA_256)
97 #if CAMELLIA_256
98 # define CAMELLIA_MAX_KEY_SIZE_BITS 256
99 #elif CAMELLIA_192
100 # define CAMELLIA_MAX_KEY_SIZE_BITS 192
101 #elif CAMELLIA_128
102 # define CAMELLIA_MAX_KEY_SIZE_BITS 128
103 #else
104 # define CAMELLIA_MAX_KEY_SIZE_BITS 0
105 #endif
106 #define MAX_CAMELLIA_KEY_BITS CAMELLIA_MAX_KEY_SIZE_BITS
107 #define MAX_CAMELLIA_KEY_BYTES ((CAMELLIA_MAX_KEY_SIZE_BITS + 7) / 8)
108 #define CAMELLIA_128_BLOCK_SIZE_BYTES (CAMELLIA_128 * 16)
109 #define CAMELLIA_192_BLOCK_SIZE_BYTES (CAMELLIA_192 * 16)
110 #define CAMELLIA_256_BLOCK_SIZE_BYTES (CAMELLIA_256 * 16)
111 #define CAMELLIA_BLOCK_SIZES \
112     CAMELLIA_128_BLOCK_SIZE_BYTES, CAMELLIA_192_BLOCK_SIZE_BYTES, \
113     CAMELLIA_256_BLOCK_SIZE_BYTES
114 #if ALG_CAMELLIA
115 # define CAMELLIA_MAX_BLOCK_SIZE 16
116 #else
117 # define CAMELLIA_MAX_BLOCK_SIZE 0
118 #endif
119 #define MAX_CAMELLIA_BLOCK_SIZE_BYTES CAMELLIA_MAX_BLOCK_SIZE

```

Table 1:17 - Defines for TDES Symmetric Cipher Algorithm Constants

```

120 #define TDES_KEY_SIZES_BITS (128 * TDES_128), (192 * TDES_192)
121 #if TDES_192
122 # define TDES_MAX_KEY_SIZE_BITS 192
123 #elif TDES_128
124 # define TDES_MAX_KEY_SIZE_BITS 128
125 #else
126 # define TDES_MAX_KEY_SIZE_BITS 0
127 #endif
128 #define MAX_TDES_KEY_BITS TDES_MAX_KEY_SIZE_BITS
129 #define MAX_TDES_KEY_BYTES ((TDES_MAX_KEY_SIZE_BITS + 7) / 8)
130 #define TDES_128_BLOCK_SIZE_BYTES (TDES_128 * 8)
131 #define TDES_192_BLOCK_SIZE_BYTES (TDES_192 * 8)
132 #define TDES_BLOCK_SIZES \
133     TDES_128_BLOCK_SIZE_BYTES, TDES_192_BLOCK_SIZE_BYTES
134 #if ALG_TDES
135 # define TDES_MAX_BLOCK_SIZE 8
136 #else
137 # define TDES_MAX_BLOCK_SIZE 0
138 #endif
139 #define MAX_TDES_BLOCK_SIZE_BYTES TDES_MAX_BLOCK_SIZE

```

Additional values for benefit of code

```

140 #define TPM_CC_FIRST          0x0000011F
141 #define TPM_CC_LAST          0x00000197
142 #if COMPRESSED_LISTS
143 #define ADD_FILL              0
144 #else
145 #define ADD_FILL              1
146 #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)

```

147 #define LIBRARY_COMMAND_ARRAY_SIZE (0 \
148 + (ADD_FILL || CC_NV_UndefineSpaceSpecial) /* 0x0000011F */ \
149 + (ADD_FILL || CC_EvictControl) /* 0x00000120 */ \
150 + (ADD_FILL || CC_HierarchyControl) /* 0x00000121 */ \
151 + (ADD_FILL || CC_NV_UndefineSpace) /* 0x00000122 */ \
152 + ADD_FILL /* 0x00000123 */ \
153 + (ADD_FILL || CC_ChangeEPS) /* 0x00000124 */ \
154 + (ADD_FILL || CC_ChangePPS) /* 0x00000125 */ \
155 + (ADD_FILL || CC_Clear) /* 0x00000126 */ \
156 + (ADD_FILL || CC_ClearControl) /* 0x00000127 */ \
157 + (ADD_FILL || CC_ClockSet) /* 0x00000128 */ \
158 + (ADD_FILL || CC_HierarchyChangeAuth) /* 0x00000129 */ \
159 + (ADD_FILL || CC_NV_DefineSpace) /* 0x0000012A */ \
160 + (ADD_FILL || CC_PCR_Allocate) /* 0x0000012B */ \
161 + (ADD_FILL || CC_PCR_SetAuthPolicy) /* 0x0000012C */ \
162 + (ADD_FILL || CC_PP_Commands) /* 0x0000012D */ \
163 + (ADD_FILL || CC_SetPrimaryPolicy) /* 0x0000012E */ \
164 + (ADD_FILL || CC_FieldUpgradeStart) /* 0x0000012F */ \
165 + (ADD_FILL || CC_ClockRateAdjust) /* 0x00000130 */ \
166 + (ADD_FILL || CC_CreatePrimary) /* 0x00000131 */ \
167 + (ADD_FILL || CC_NV_GlobalWriteLock) /* 0x00000132 */ \
168 + (ADD_FILL || CC_GetCommandAuditDigest) /* 0x00000133 */ \
169 + (ADD_FILL || CC_NV_Increment) /* 0x00000134 */ \
170 + (ADD_FILL || CC_NV_SetBits) /* 0x00000135 */ \
171 + (ADD_FILL || CC_NV_Extend) /* 0x00000136 */ \
172 + (ADD_FILL || CC_NV_Write) /* 0x00000137 */ \
173 + (ADD_FILL || CC_NV_WriteLock) /* 0x00000138 */ \
174 + (ADD_FILL || CC_DictionaryAttackLockReset) /* 0x00000139 */ \
175 + (ADD_FILL || CC_DictionaryAttackParameters) /* 0x0000013A */ \
176 + (ADD_FILL || CC_NV_ChangeAuth) /* 0x0000013B */ \
177 + (ADD_FILL || CC_PCR_Event) /* 0x0000013C */ \
178 + (ADD_FILL || CC_PCR_Reset) /* 0x0000013D */ \
179 + (ADD_FILL || CC_SequenceComplete) /* 0x0000013E */ \
180 + (ADD_FILL || CC_SetAlgorithmSet) /* 0x0000013F */ \
181 + (ADD_FILL || CC_SetCommandCodeAuditStatus) /* 0x00000140 */ \
182 + (ADD_FILL || CC_FieldUpgradeData) /* 0x00000141 */ \
183 + (ADD_FILL || CC_IncrementalSelfTest) /* 0x00000142 */ \
184 + (ADD_FILL || CC_SelfTest) /* 0x00000143 */ \
185 + (ADD_FILL || CC_Startup) /* 0x00000144 */ \
186 + (ADD_FILL || CC_Shutdown) /* 0x00000145 */ \
187 + (ADD_FILL || CC_StirRandom) /* 0x00000146 */ \
188 + (ADD_FILL || CC_ActivateCredential) /* 0x00000147 */ \
189 + (ADD_FILL || CC_Certify) /* 0x00000148 */ \
190 + (ADD_FILL || CC_PolicyNV) /* 0x00000149 */ \
191 + (ADD_FILL || CC_CertifyCreation) /* 0x0000014A */ \
192 + (ADD_FILL || CC_Duplicate) /* 0x0000014B */ \
193 + (ADD_FILL || CC_GetTime) /* 0x0000014C */ \
194 + (ADD_FILL || CC_GetSessionAuditDigest) /* 0x0000014D */ \
195 + (ADD_FILL || CC_NV_Read) /* 0x0000014E */ \
196 + (ADD_FILL || CC_NV_ReadLock) /* 0x0000014F */ \
197 + (ADD_FILL || CC_ObjectChangeAuth) /* 0x00000150 */ \
198 + (ADD_FILL || CC_PolicySecret) /* 0x00000151 */ \

```

199	+ (ADD_FILL CC_Rewrap)	/* 0x00000152 */	\
200	+ (ADD_FILL CC_Create)	/* 0x00000153 */	\
201	+ (ADD_FILL CC_ECDH_ZGen)	/* 0x00000154 */	\
202	+ (ADD_FILL CC_HMAC CC_MAC)	/* 0x00000155 */	\
203	+ (ADD_FILL CC_Import)	/* 0x00000156 */	\
204	+ (ADD_FILL CC_Load)	/* 0x00000157 */	\
205	+ (ADD_FILL CC_Quote)	/* 0x00000158 */	\
206	+ (ADD_FILL CC_RSA_Decrypt)	/* 0x00000159 */	\
207	+ ADD_FILL	/* 0x0000015A */	\
208	+ (ADD_FILL CC_HMAC_Start CC_MAC_Start)	/* 0x0000015B */	\
209	+ (ADD_FILL CC_SequenceUpdate)	/* 0x0000015C */	\
210	+ (ADD_FILL CC_Sign)	/* 0x0000015D */	\
211	+ (ADD_FILL CC_Unseal)	/* 0x0000015E */	\
212	+ ADD_FILL	/* 0x0000015F */	\
213	+ (ADD_FILL CC_PolicySigned)	/* 0x00000160 */	\
214	+ (ADD_FILL CC_ContextLoad)	/* 0x00000161 */	\
215	+ (ADD_FILL CC_ContextSave)	/* 0x00000162 */	\
216	+ (ADD_FILL CC_ECDH_KeyGen)	/* 0x00000163 */	\
217	+ (ADD_FILL CC_EncryptDecrypt)	/* 0x00000164 */	\
218	+ (ADD_FILL CC_FlushContext)	/* 0x00000165 */	\
219	+ ADD_FILL	/* 0x00000166 */	\
220	+ (ADD_FILL CC_LoadExternal)	/* 0x00000167 */	\
221	+ (ADD_FILL CC_MakeCredential)	/* 0x00000168 */	\
222	+ (ADD_FILL CC_NV_ReadPublic)	/* 0x00000169 */	\
223	+ (ADD_FILL CC_PolicyAuthorize)	/* 0x0000016A */	\
224	+ (ADD_FILL CC_PolicyAuthValue)	/* 0x0000016B */	\
225	+ (ADD_FILL CC_PolicyCommandCode)	/* 0x0000016C */	\
226	+ (ADD_FILL CC_PolicyCounterTimer)	/* 0x0000016D */	\
227	+ (ADD_FILL CC_PolicyCpHash)	/* 0x0000016E */	\
228	+ (ADD_FILL CC_PolicyLocality)	/* 0x0000016F */	\
229	+ (ADD_FILL CC_PolicyNameHash)	/* 0x00000170 */	\
230	+ (ADD_FILL CC_PolicyOR)	/* 0x00000171 */	\
231	+ (ADD_FILL CC_PolicyTicket)	/* 0x00000172 */	\
232	+ (ADD_FILL CC_ReadPublic)	/* 0x00000173 */	\
233	+ (ADD_FILL CC_RSA_Encrypt)	/* 0x00000174 */	\
234	+ ADD_FILL	/* 0x00000175 */	\
235	+ (ADD_FILL CC_StartAuthSession)	/* 0x00000176 */	\
236	+ (ADD_FILL CC_VerifySignature)	/* 0x00000177 */	\
237	+ (ADD_FILL CC_ECC_Parameters)	/* 0x00000178 */	\
238	+ (ADD_FILL CC_FirmwareRead)	/* 0x00000179 */	\
239	+ (ADD_FILL CC_GetCapability)	/* 0x0000017A */	\
240	+ (ADD_FILL CC_GetRandom)	/* 0x0000017B */	\
241	+ (ADD_FILL CC_GetTestResult)	/* 0x0000017C */	\
242	+ (ADD_FILL CC_Hash)	/* 0x0000017D */	\
243	+ (ADD_FILL CC_PCR_Read)	/* 0x0000017E */	\
244	+ (ADD_FILL CC_PolicyPCR)	/* 0x0000017F */	\
245	+ (ADD_FILL CC_PolicyRestart)	/* 0x00000180 */	\
246	+ (ADD_FILL CC_ReadClock)	/* 0x00000181 */	\
247	+ (ADD_FILL CC_PCR_Extend)	/* 0x00000182 */	\
248	+ (ADD_FILL CC_PCR_SetAuthValue)	/* 0x00000183 */	\
249	+ (ADD_FILL CC_NV_Certify)	/* 0x00000184 */	\
250	+ (ADD_FILL CC_EventSequenceComplete)	/* 0x00000185 */	\
251	+ (ADD_FILL CC_HashSequenceStart)	/* 0x00000186 */	\
252	+ (ADD_FILL CC_PolicyPhysicalPresence)	/* 0x00000187 */	\
253	+ (ADD_FILL CC_PolicyDuplicationSelect)	/* 0x00000188 */	\
254	+ (ADD_FILL CC_PolicyGetDigest)	/* 0x00000189 */	\
255	+ (ADD_FILL CC_TestParms)	/* 0x0000018A */	\
256	+ (ADD_FILL CC_Commit)	/* 0x0000018B */	\
257	+ (ADD_FILL CC_PolicyPassword)	/* 0x0000018C */	\
258	+ (ADD_FILL CC_ZGen_2Phase)	/* 0x0000018D */	\
259	+ (ADD_FILL CC_EC_Ephemeral)	/* 0x0000018E */	\
260	+ (ADD_FILL CC_PolicyNvWritten)	/* 0x0000018F */	\
261	+ (ADD_FILL CC_PolicyTemplate)	/* 0x00000190 */	\
262	+ (ADD_FILL CC_CreateLoaded)	/* 0x00000191 */	\
263	+ (ADD_FILL CC_PolicyAuthorizeNV)	/* 0x00000192 */	\
264	+ (ADD_FILL CC_EncryptDecrypt2)	/* 0x00000193 */	\

```

265     + (ADD_FILL || CC_AC_GetCapability)                /* 0x00000194 */ \
266     + (ADD_FILL || CC_AC_Send)                        /* 0x00000195 */ \
267     + (ADD_FILL || CC_Policy_AC_SendSelect)           /* 0x00000196 */ \
268     + (ADD_FILL || CC_CertifyX509)                   /* 0x00000197 */ \
269 )
270 #define VENDOR_COMMAND_ARRAY_SIZE (0 + CC_Vendor_TCG_Test)
271 #define COMMAND_COUNT (LIBRARY_COMMAND_ARRAY_SIZE + VENDOR_COMMAND_ARRAY_SIZE)
272 #define HASH_COUNT \
273     (ALG_SHA1 + ALG_SHA256 + ALG_SHA384 + ALG_SHA3_256 + \
274     ALG_SHA3_384 + ALG_SHA3_512 + ALG_SHA512 + ALG_SM3_256)
275 #define MAX_HASH_BLOCK_SIZE \
276     (MAX(ALG_SHA1 * SHA1_BLOCK_SIZE, \
277     MAX(ALG_SHA256 * SHA256_BLOCK_SIZE, \
278     MAX(ALG_SHA384 * SHA384_BLOCK_SIZE, \
279     MAX(ALG_SHA3_256 * SHA3_256_BLOCK_SIZE, \
280     MAX(ALG_SHA3_384 * SHA3_384_BLOCK_SIZE, \
281     MAX(ALG_SHA3_512 * SHA3_512_BLOCK_SIZE, \
282     MAX(ALG_SHA512 * SHA512_BLOCK_SIZE, \
283     MAX(ALG_SM3_256 * SM3_256_BLOCK_SIZE, \
284     0))))))
285 #define MAX_DIGEST_SIZE \
286     (MAX(ALG_SHA1 * SHA1_DIGEST_SIZE, \
287     MAX(ALG_SHA256 * SHA256_DIGEST_SIZE, \
288     MAX(ALG_SHA384 * SHA384_DIGEST_SIZE, \
289     MAX(ALG_SHA3_256 * SHA3_256_DIGEST_SIZE, \
290     MAX(ALG_SHA3_384 * SHA3_384_DIGEST_SIZE, \
291     MAX(ALG_SHA3_512 * SHA3_512_DIGEST_SIZE, \
292     MAX(ALG_SHA512 * SHA512_DIGEST_SIZE, \
293     MAX(ALG_SM3_256 * SM3_256_DIGEST_SIZE, \
294     0))))))
295 #if MAX_DIGEST_SIZE == 0 || MAX_HASH_BLOCK_SIZE == 0
296 #error "Hash data not valid"
297 #endif

```

Define the 2B structure that would hold any hash block

```

298 TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);

```

Following typedef is for some old code

```

299 typedef TPM2B_MAX_HASH_BLOCK TPM2B_HASH_BLOCK;

```

Additional symmetric constants

```

300 #define MAX_SYM_KEY_BITS \
301     (MAX(AES_MAX_KEY_SIZE_BITS, MAX(CAMELLIA_MAX_KEY_SIZE_BITS, \
302     MAX(SM4_MAX_KEY_SIZE_BITS, MAX(TDES_MAX_KEY_SIZE_BITS, \
303     0))))
304 #define MAX_SYM_KEY_BYTES ((MAX_SYM_KEY_BITS + 7) / 8)
305 #define MAX_SYM_BLOCK_SIZE \
306     (MAX(AES_MAX_BLOCK_SIZE, MAX(CAMELLIA_MAX_BLOCK_SIZE, \
307     MAX(SM4_MAX_BLOCK_SIZE, MAX(TDES_MAX_BLOCK_SIZE, \
308     0))))
309 #if MAX_SYM_KEY_BITS == 0 || MAX_SYM_BLOCK_SIZE == 0
310 #error Bad size for MAX_SYM_KEY_BITS or MAX_SYM_BLOCK
311 #endif
312 #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  #define     SELF_TEST_DATA
3  #if SELF_TEST
```

These includes pull in the data structures. They contain data definitions for the various tests.

```
4  #include    "SelfTest.h"
5  #include    "SymmetricTest.h"
6  #include    "RsaTestData.h"
7  #include    "EccTestData.h"
8  #include    "HashTestData.h"
9  #include    "KdfTestData.h"
10 #define TEST_DEFAULT_TEST_HASH(vector) \
11     if(TEST_BIT(DEFAULT_TEST_HASH, g_toTest)) \
12         TestHash(DEFAULT_TEST_HASH, vector);
```

Make sure that the algorithm has been tested

```
13 #define CLEAR_BOTH(alg)    {    CLEAR_BIT(alg, *toTest); \
14                             if(toTest != &g_toTest) \
15                                 CLEAR_BIT(alg, g_toTest); }
16 #define SET_BOTH(alg)     {    SET_BIT(alg, *toTest); \
17                             if(toTest != &g_toTest) \
18                                 SET_BIT(alg, g_toTest); }
19 #define TEST_BOTH(alg)    ((toTest != &g_toTest) \
20 ? TEST_BIT(alg, *toTest) || TEST_BIT(alg, g_toTest) \
21 : TEST_BIT(alg, *toTest))
```

Can only cancel if doing a list.

```
22 #define CHECK_CANCELED \
23     if(_plat_IsCanceled() && toTest != &g_toTest) \
24         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.

```

25 static TPM_RC
26 TestHash(
27     TPM_ALG_ID      hashAlg,
28     ALGORITHM_VECTOR *toTest
29 )
30 {
31     static TPM2B_DIGEST    computed; // value computed
32     static HMAC_STATE      state;
33     UINT16                 digestSize;
34     const TPM2B             *testDigest = NULL;
35     // TPM2B_TYPE(HMAC_BLOCK, DEFAULT_TEST_HASH_BLOCK_SIZE);
36
37     pAssert(hashAlg != ALG_NULL_VALUE);
38     switch(hashAlg)
39     {
40 #if ALG_SHA1
41         case ALG_SHA1_VALUE:
42             testDigest = &c_SHA1_digest.b;
43             break;
44 #endif
45 #if ALG_SHA256
46         case ALG_SHA256_VALUE:
47             testDigest = &c_SHA256_digest.b;
48             break;
49 #endif
50 #if ALG_SHA384
51         case ALG_SHA384_VALUE:
52             testDigest = &c_SHA384_digest.b;
53             break;
54 #endif
55 #if ALG_SHA512
56         case ALG_SHA512_VALUE:
57             testDigest = &c_SHA512_digest.b;
58             break;
59 #endif
60 #if ALG_SM3_256
61         case ALG_SM3_256_VALUE:
62             testDigest = &c_SM3_256_digest.b;
63             break;
64 #endif
65         default:
66             FAIL(FATAL_ERROR_INTERNAL);
67     }
68     // Clear the to-test bits
69     CLEAR_BOTH(hashAlg);
70
71     // Set the HMAC key to twice the digest size
72     digestSize = CryptHashGetDigestSize(hashAlg);
73     CryptHmacStart(&state, hashAlg, digestSize * 2,
74         (BYTE *)c_hashTestKey.t.buffer);
75     CryptDigestUpdate(&state.hashState, 2 * CryptHashGetBlockSize(hashAlg),
76         (BYTE *)c_hashTestData.t.buffer);
77     computed.t.size = digestSize;
78     CryptHmacEnd(&state, digestSize, computed.t.buffer);
79     if((testDigest->size != computed.t.size)
80         || (memcmp(testDigest->buffer, computed.t.buffer, computed.b.size) != 0))
81         SELF_TEST_FAILURE;
82     return TPM_RC_SUCCESS;
83 }

```


10.2.1.4 Symmetric Test Functions

10.2.1.4.1 MakeIv()

Internal function to make the appropriate IV depending on the mode.

```

84  static UINT32
85  MakeIv(
86      TPM_ALG_ID    mode,      // IN: symmetric mode
87      UINT32        size,      // IN: block size of the algorithm
88      BYTE          *iv        // OUT: IV to fill in
89  )
90  {
91      BYTE          i;
92
93      if(mode == ALG_ECB_VALUE)
94          return 0;
95      if(mode == ALG_CTR_VALUE)
96      {
97          // The test uses an IV that has 0xff in the last byte
98          for(i = 1; i <= size; i++)
99              *iv++ = 0xff - (BYTE)(size - i);
100     }
101     else
102     {
103         for(i = 0; i < size; i++)
104             *iv++ = i;
105     }
106     return size;
107 }

```

10.2.1.4.2 TestSymmetricAlgorithm()

Function to test a specific algorithm, key size, and mode.

```

108  static void
109  TestSymmetricAlgorithm(
110      const SYMMETRIC_TEST_VECTOR *test,      //
111      TPM_ALG_ID                  mode        //
112  )
113  {
114      static BYTE                  encrypted[MAX_SYM_BLOCK_SIZE * 2];
115      static BYTE                  decrypted[MAX_SYM_BLOCK_SIZE * 2];
116      static TPM2B_IV              iv;
117      //
118      // Get the appropriate IV
119      iv.t.size = (UINT16)MakeIv(mode, test->ivSize, iv.t.buffer);
120
121      // Encrypt known data
122      CryptSymmetricEncrypt(encrypted, test->alg, test->keyBits, test->key, &iv,
123                          mode, test->dataInOutSize, test->dataIn);
124      // Check that it matches the expected value
125      if(!MemoryEqual(encrypted, test->dataOut[mode - ALG_CTR_VALUE],
126                     test->dataInOutSize))
127          SELF_TEST_FAILURE;
128      // Reinitialize the iv for decryption
129      MakeIv(mode, test->ivSize, iv.t.buffer);
130      CryptSymmetricDecrypt(decrypted, test->alg, test->keyBits, test->key, &iv,
131                          mode, test->dataInOutSize,
132                          test->dataOut[mode - ALG_CTR_VALUE]);
133      // Make sure that it matches what we started with
134      if(!MemoryEqual(decrypted, test->dataIn, test->dataInOutSize))
135          SELF_TEST_FAILURE;

```


136 }

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

```

137 static BOOL
138 AllSymsAreDone(
139     ALGORITHM_VECTOR    *toTest
140 )
141 {
142     return (!TEST_BOTH(ALG_AES_VALUE) && !TEST_BOTH(ALG_SM4_VALUE));
143 }
```

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

```

144 static BOOL
145 AllModesAreDone(
146     ALGORITHM_VECTOR    *toTest
147 )
148 {
149     TPM_ALG_ID          alg;
150     for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
151         if(TEST_BOTH(alg))
152             return FALSE;
153     return TRUE;
154 }
```

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.

```

155 static TPM_RC
156 TestSymmetric(
157     TPM_ALG_ID          alg,
158     ALGORITHM_VECTOR    *toTest
159 )
160 {
161     SYM_INDEX            index;
162     TPM_ALG_ID            mode;
163     //
164     if(!TEST_BIT(alg, *toTest))
165         return TPM_RC_SUCCESS;
166     if(alg == ALG_AES_VALUE || alg == ALG_SM4_VALUE || alg == ALG_CAMELLIA_VALUE)
167     {
```

```

168     // Will test the algorithm for all modes and key sizes
169     CLEAR_BOTH(alg);
170
171     // A test this algorithm for all modes
172     for(index = 0; index < NUM_SYMS; index++)
173     {
174         if(c_symTestValues[index].alg == alg)
175         {
176             for(mode = TPM_SYM_MODE_FIRST;
177                 mode <= TPM_SYM_MODE_LAST;
178                 mode++)
179             {
180                 if(TEST_BIT(mode, *toTest))
181                     TestSymmetricAlgorithm(&c_symTestValues[index], mode);
182             }
183         }
184     }
185     // if all the symmetric tests are done
186     if(AllSymsAreDone(toTest))
187     {
188         // all symmetric algorithms tested so no modes should be set
189         for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
190             CLEAR_BOTH(alg);
191     }
192 }
193 else if(TPM_SYM_MODE_FIRST <= alg && alg <= TPM_SYM_MODE_LAST)
194 {
195     // Test this mode for all key sizes and algorithms
196     for(index = 0; index < NUM_SYMS; index++)
197     {
198         // The mode testing only comes into play when doing self tests
199         // by command. When doing self tests by command, the block ciphers are
200         // tested first. That means that all of their modes would have been
201         // tested for all key sizes. If there is no block cipher left to
202         // test, then clear this mode bit.
203         if(!TEST_BIT(ALG_AES_VALUE, *toTest)
204            && !TEST_BIT(ALG_SM4_VALUE, *toTest))
205         {
206             CLEAR_BOTH(alg);
207         }
208     }
209     else
210     {
211         for(index = 0; index < NUM_SYMS; index++)
212         {
213             if(TEST_BIT(c_symTestValues[index].alg, *toTest))
214                 TestSymmetricAlgorithm(&c_symTestValues[index], alg);
215         }
216         // have tested this mode for all algorithms
217         CLEAR_BOTH(alg);
218     }
219 }
220 if(AllModesAreDone(toTest))
221 {
222     CLEAR_BOTH(ALG_AES_VALUE);
223     CLEAR_BOTH(ALG_SM4_VALUE);
224 }
225 else
226     pAssert(alg == 0 && alg != 0);
227 return TPM_RC_SUCCESS;
228 }

```

10.2.1.5 RSA Tests

```
229 #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.

```

230 static void
231 RsaKeyInitialize(
232     OBJECT          *testObject
233 )
234 {
235     MemoryCopy2B(&testObject->publicArea.unique.rsa.b, (P2B)&c_rsaPublicModulus,
236                 sizeof(c_rsaPublicModulus));
237     MemoryCopy2B(&testObject->sensitive.sensitive.rsa.b, (P2B)&c_rsaPrivatePrime,
238                 sizeof(testObject->sensitive.sensitive.rsa.t.buffer));
239     testObject->publicArea.parameters.rsaDetail.keyBits = RSA_TEST_KEY_SIZE * 8;
240     // Use the default exponent
241     testObject->publicArea.parameters.rsaDetail.exponent = 0;
242 }

```

10.2.1.5.3 TestRsaEncryptDecrypt()

These test are for an public key encryption that uses a random value

```

243 static TPM_RC
244 TestRsaEncryptDecrypt(
245     TPM_ALG_ID      scheme,           // IN: the scheme
246     ALGORITHM_VECTOR *toTest          //
247 )
248 {
249     static TPM2B_PUBLIC_KEY_RSA testInput;
250     static TPM2B_PUBLIC_KEY_RSA testOutput;
251     static OBJECT              testObject;
252     const TPM2B_RSA_TEST_KEY   *kvtValue = NULL;
253     TPM_RC                      result = TPM_RC_SUCCESS;
254     const TPM2B                *testLabel = NULL;
255     TPMT_RSA_DECRYPT            rsaScheme;
256     //
257     // Don't need to initialize much of the test object
258     RsaKeyInitialize(&testObject);
259     rsaScheme.scheme = scheme;
260     rsaScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
261     CLEAR_BOTH(scheme);
262     CLEAR_BOTH(ALG_NULL_VALUE);
263     if(scheme == ALG_NULL_VALUE)
264     {

```

```

265 // This is an encryption scheme using the private key without any encoding.
266 memcpy(testInput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue));
267 testInput.t.size = sizeof(c_RsaTestValue);
268 if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
269                                     &testObject, &rsaScheme, NULL, NULL))
270     SELF_TEST_FAILURE;
271 if(!MemoryEqual(testOutput.t.buffer, c_RsaepKvt.buffer, c_RsaepKvt.size))
272     SELF_TEST_FAILURE;
273 MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
274 if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
275                                     &testObject, &rsaScheme, NULL))
276     SELF_TEST_FAILURE;
277 if(!MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
278                 sizeof(c_RsaTestValue)))
279     SELF_TEST_FAILURE;
280 }
281 else
282 {
283     // ALG_RSAES_VALUE:
284     // This is an decryption scheme using padding according to
285     // PKCS#1v2.1, 7.2. This padding uses random bits. To test a public
286     // key encryption that uses random data, encrypt a value and then
287     // decrypt the value and see that we get the encrypted data back.
288     // The hash is not used by this encryption so it can be TPM_ALG_NULL
289
290     // ALG_OAEP_VALUE:
291     // This is also an decryption scheme and it also uses a
292     // pseudo-random
293     // value. However, this also uses a hash algorithm. So, we may need
294     // to test that algorithm before use.
295     if(scheme == ALG_OAEP_VALUE)
296     {
297         TEST_DEFAULT_TEST_HASH(toTest);
298         kvtValue = &c_OaepKvt;
299         testLabel = OAEP_TEST_STRING;
300     }
301     else if(scheme == ALG_RSAES_VALUE)
302     {
303         kvtValue = &c_RsaesKvt;
304         testLabel = NULL;
305     }
306     else
307     {
308         SELF_TEST_FAILURE;
309     }
310     // Only use a digest-size portion of the test value
311     memcpy(testInput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
312     testInput.t.size = DEFAULT_TEST_DIGEST_SIZE;
313
314     // See if the encryption works
315     if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b,
316                                         &testObject, &rsaScheme, testLabel,
317                                         NULL))
318     {
319         SELF_TEST_FAILURE;
320     }
321     MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
322     // see if we can decrypt this value and get the original data back
323     if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,
324                                         &testObject, &rsaScheme, testLabel))
325     {
326         SELF_TEST_FAILURE;
327     }
328     // See if the results compare
329     if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
330        || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
331                        DEFAULT_TEST_DIGEST_SIZE))
332     {
333         SELF_TEST_FAILURE;
334     }
335     // Now check that the decryption works on a known value
336     MemoryCopy2B(&testInput.b, (P2B)kvtValue,
337                 sizeof(testInput.t.buffer));
338     if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b,

```

```

331                                     &testObject, &rsaScheme, testLabel))
332         SELF_TEST_FAILURE;
333         if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE
334            || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue,
335                           DEFAULT_TEST_DIGEST_SIZE))
336             SELF_TEST_FAILURE;
337     }
338     return result;
339 }

```

10.2.1.5.4 TestRsaSignAndVerify()

This function does the testing of the RSA sign and verification functions. This test does a KVT.

```

340 static TPM_RC
341 TestRsaSignAndVerify(
342     TPM_ALG_ID          scheme,
343     ALGORITHM_VECTOR    *toTest
344 )
345 {
346     TPM_RC              result = TPM_RC_SUCCESS;
347     static OBJECT       testObject;
348     static TPM2B_DIGEST testDigest;
349     static TPMT_SIGNATURE testSig;
350
351     // Do a sign and signature verification.
352     // RSASSA:
353     // This is a signing scheme according to PKCS#1-v2.1 8.2. It does not
354     // use random data so there is a KVT for the signing operation. On
355     // first use of the scheme for signing, use the TPM's RSA key to
356     // sign a portion of c_RsaTestData and compare the results to c_RsassaKvt. Then
357     // decrypt the data to see that it matches the starting value. This verifies
358     // the signature with a KVT
359
360     // Clear the bits indicating that the function has not been checked. This is to
361     // prevent looping
362     CLEAR_BOTH(scheme);
363     CLEAR_BOTH(ALG_NULL_VALUE);
364     CLEAR_BOTH(ALG_RSA_VALUE);
365
366     RsaKeyInitialize(&testObject);
367     memcpy(testDigest.t.buffer, (BYTE *)c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
368     testDigest.t.size = DEFAULT_TEST_DIGEST_SIZE;
369     testSig.sigAlg = scheme;
370     testSig.signature.rsapss.hash = DEFAULT_TEST_HASH;
371
372     // RSAPSS:
373     // This is a signing scheme according to PKCS#1-v2.2 8.1 it uses
374     // random data in the signature so there is no KVT for the signing
375     // operation. To test signing, the TPM will use the TPM's RSA key
376     // to sign a portion of c_RsaTestValue and then it will verify the
377     // signature. For verification, c_RsapssKvt is verified before the
378     // user signature blob is verified. The worst case for testing of this
379     // algorithm is two private and one public key operation.
380
381     // The process is to sign known data. If RSASSA is being done, verify that the
382     // signature matches the precomputed value. For both, use the signed value and
383     // see that the verification says that it is a good signature. Then
384     // if testing RSAPSS, do a verify of a known good signature. This ensures that
385     // the validation function works.
386
387     if(TPM_RC_SUCCESS != CryptRsaSign(&testSig, &testObject, &testDigest, NULL))
388         SELF_TEST_FAILURE;
389     // For RSASSA, make sure the results is what we are looking for

```

```

390     if(testSig.sigAlg == ALG_RSASSA_VALUE)
391     {
392         if(testSig.signature.rsassa.sig.t.size != RSA_TEST_KEY_SIZE
393            || !MemoryEqual(c_RsassaKvt.buffer,
394                           testSig.signature.rsassa.sig.t.buffer,
395                           RSA_TEST_KEY_SIZE))
396             SELF_TEST_FAILURE;
397     }
398     // See if the TPM will validate its own signatures
399     if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
400                                                    &testDigest))
401         SELF_TEST_FAILURE;
402     // If this is RSAPSS, check the verification with known signature
403     // Have to copy because CryptRsaValidateSignature() eats the signature
404     if(ALG_RSAPSS_VALUE == scheme)
405     {
406         MemoryCopy2B(&testSig.signature.rsapss.sig.b, (P2B)&c_RsapssKvt,
407                     sizeof(testSig.signature.rsapss.sig.t.buffer));
408         if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
409                                                        &testDigest))
410             SELF_TEST_FAILURE;
411     }
412     return result;
413 }

```

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*

```

414 static TPM_RC
415 TestRsa(
416     TPM_ALG_ID          alg,
417     ALGORITHM_VECTOR    *toTest
418 )
419 {
420     TPM_RC              result = TPM_RC_SUCCESS;
421     //
422     switch(alg)
423     {
424         case ALG_NULL_VALUE:
425             // This is the RSAEP/RSADP function. If we are processing a list, don't
426             // need to test these now because any other test will validate
427             // RSAEP/RSADP. Can tell this is list of test by checking to see if
428             // 'toTest' is pointing at g_toTest. If so, this is an isolated test
429             // an need to go ahead and do the test;
430             if((toTest == &g_toTest)
431                || (!TEST_BIT(ALG_RSASSA_VALUE, *toTest)
432                    && !TEST_BIT(ALG_RSAES_VALUE, *toTest)
433                    && !TEST_BIT(ALG_RSAPSS_VALUE, *toTest)
434                    && !TEST_BIT(ALG_OAEP_VALUE, *toTest)))
435                 // Not running a list of tests or no other tests on the list
436                 // so run the test now
437                 result = TestRsaEncryptDecrypt(alg, toTest);
438             // if not running the test now, leave the bit on, just in case things
439             // get interrupted
440             break;
441         case ALG_OAEP_VALUE:
442         case ALG_RSAES_VALUE:
443             result = TestRsaEncryptDecrypt(alg, toTest);
444             break;
445         case ALG_RSAPSS_VALUE:
446         case ALG_RSASSA_VALUE:
447             result = TestRsaSignAndVerify(alg, toTest);

```

```

448         break;
449     default:
450         SELF_TEST_FAILURE;
451     }
452     return result;
453 }
454 #endif // ALG_RSA

```

10.2.1.6 ECC Tests

```

455 #if ALG_ECC

```

10.2.1.6.1 LoadEccParameter()

This function is mostly for readability and type checking

```

456 static void
457 LoadEccParameter(
458     TPM2B_ECC_PARAMETER      *to,          // target
459     const TPM2B_EC_TEST      *from         // source
460 )
461 {
462     MemoryCopy2B(&to->b, &from->b, sizeof(to->t.buffer));
463 }

```

10.2.1.6.2 LoadEccPoint()

```

464 static void
465 LoadEccPoint(
466     TPMS_ECC_POINT           *point,        // target
467     const TPM2B_EC_TEST      *x,           // source
468     const TPM2B_EC_TEST      *y
469 )
470 {
471     MemoryCopy2B(&point->x.b, (TPM2B *)x, sizeof(point->x.t.buffer));
472     MemoryCopy2B(&point->y.b, (TPM2B *)y, sizeof(point->y.t.buffer));
473 }

```

10.2.1.6.3 TestECDH()

This test does a KVT on a point multiply.

```

474 static TPM_RC
475 TestECDH(
476     TPM_ALG_ID               scheme,        // IN: for consistency
477     ALGORITHM_VECTOR         *toTest       // IN/OUT: modified after test is run
478 )
479 {
480     static TPMS_ECC_POINT     Z;
481     static TPMS_ECC_POINT     Qe;
482     static TPM2B_ECC_PARAMETER ds;
483     TPM_RC                    result = TPM_RC_SUCCESS;
484     //
485     NOT_REFERENCED(scheme);
486     CLEAR_BOTH(ALG_ECDH_VALUE);
487     LoadEccParameter(&ds, &c_ecTestKey_ds);
488     LoadEccPoint(&Qe, &c_ecTestKey_QeX, &c_ecTestKey_QeY);
489     if(TPM_RC_SUCCESS != CryptEccPointMultiply(&Z, c_testCurve, &Qe, &ds,
490                                                NULL, NULL))
491         SELF_TEST_FAILURE;
492     if(!MemoryEqual2B(&c_ecTestEcdh_X.b, &Z.x.b)

```



```

493         || !MemoryEqual2B(&c_ecTestEcdh_Y.b, &Z.y.b))
494         SELF_TEST_FAILURE;
495     return result;
496 }

```

10.2.1.6.4 TestEccSignAndVerify()

```

497 static TPM_RC
498 TestEccSignAndVerify(
499     TPM_ALG_ID          scheme,
500     ALGORITHM_VECTOR    *toTest
501 )
502 {
503     static OBJECT        testObject;
504     static TPMT_SIGNATURE testSig;
505     static TPMT_ECC_SCHEME eccScheme;
506
507     testSig.sigAlg = scheme;
508     testSig.signature.ecdsa.hash = DEFAULT_TEST_HASH;
509
510     eccScheme.scheme = scheme;
511     eccScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
512
513     CLEAR_BOTH(scheme);
514     CLEAR_BOTH(ALG_ECDH_VALUE);
515
516     // ECC signature verification testing uses a KVT.
517     switch(scheme)
518     {
519     case ALG_ECDSA_VALUE:
520         LoadEccParameter(&testSig.signature.ecdsa.signatureR, &c_TestEcDsa_r);
521         LoadEccParameter(&testSig.signature.ecdsa.signatureS, &c_TestEcDsa_s);
522         break;
523     case ALG_ECSCHNORR_VALUE:
524         LoadEccParameter(&testSig.signature.ecschnorr.signatureR,
525                         &c_TestEcSchnorr_r);
526         LoadEccParameter(&testSig.signature.ecschnorr.signatureS,
527                         &c_TestEcSchnorr_s);
528         break;
529     case ALG_SM2_VALUE:
530         // don't have a test for SM2
531         return TPM_RC_SUCCESS;
532     default:
533         SELF_TEST_FAILURE;
534         break;
535     }
536     TEST_DEFAULT_TEST_HASH(toTest);
537
538     // Have to copy the key. This is because the size used in the test vectors
539     // is the size of the ECC parameter for the test key while the size of a point
540     // is TPM dependent
541     MemoryCopy2B(&testObject.sensitive.sensitive.ecc.b, &c_ecTestKey_ds.b,
542                 sizeof(testObject.sensitive.sensitive.ecc.t.buffer));
543     LoadEccPoint(&testObject.publicArea.unique.ecc, &c_ecTestKey_QsX,
544                 &c_ecTestKey_QsY);
545     testObject.publicArea.parameters.eccDetail.curveID = c_testCurve;
546
547     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
548                                                    (TPM2B_DIGEST *)&c_ecTestValue.b))
549     {
550         SELF_TEST_FAILURE;
551     }
552     CHECK_CANCELED;
553
554     // Now sign and verify some data

```

```

555     if(TPM_RC_SUCCESS != CryptEccSign(&testSig, &testObject,
556                                     (TPM2B_DIGEST *)&c_ecTestValue,
557                                     &eccScheme, NULL))
558         SELF_TEST_FAILURE;
559
560     CHECK_CANCELED;
561
562     if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
563                                                  (TPM2B_DIGEST *)&c_ecTestValue))
564         SELF_TEST_FAILURE;
565
566     CHECK_CANCELED;
567
568     return TPM_RC_SUCCESS;
569 }

```

10.2.1.6.5 TestKDFa()

```

570 static TPM_RC
571 TestKDFa(
572     ALGORITHM_VECTOR      *toTest
573 )
574 {
575     static TPM2B_KDF_TEST_KEY    keyOut;
576     UINT32                      counter = 0;
577     //
578     CLEAR_BOTH(ALG_KDF1_SP800_108_VALUE);
579
580     keyOut.t.size = CryptKDFa(KDF_TEST_ALG, &c_kdfTestKeyIn.b, &c_kdfTestLabel.b,
581                             &c_kdfTestContextU.b, &c_kdfTestContextV.b,
582                             TEST_KDF_KEY_SIZE * 8, keyOut.t.buffer,
583                             &counter, FALSE);
584     if ( keyOut.t.size != TEST_KDF_KEY_SIZE
585         || !MemoryEqual(keyOut.t.buffer, c_kdfTestKeyOut.t.buffer,
586                         TEST_KDF_KEY_SIZE))
587         SELF_TEST_FAILURE;
588
589     return TPM_RC_SUCCESS;
590 }

```

10.2.1.6.6 TestEcc()

```

591 static TPM_RC
592 TestEcc(
593     TPM_ALG_ID            alg,
594     ALGORITHM_VECTOR      *toTest
595 )
596 {
597     TPM_RC                result = TPM_RC_SUCCESS;
598     NOT_REFERENCED(toTest);
599     switch(alg)
600     {
601         case ALG_ECC_VALUE:
602         case ALG_ECDH_VALUE:
603             // If this is in a loop then see if another test is going to deal with
604             // this.
605             // If toTest is not a self-test list
606             if((toTest == &q_toTest)
607                 // or this is the only ECC test in the list
608                 || !(TEST_BIT(ALG_ECDSA_VALUE, *toTest)
609                     || TEST_BIT(ALG_ECSCNORR, *toTest)
610                     || TEST_BIT(ALG_SM2_VALUE, *toTest)))
611             {
612                 result = TestECDH(alg, toTest);

```

```

613         }
614         break;
615     case ALG_ECDSA_VALUE:
616     case ALG_ECSCNORR_VALUE:
617     case ALG_SM2_VALUE:
618         result = TestEccSignAndVerify(alg, toTest);
619         break;
620     default:
621         SELF_TEST_FAILURE;
622         break;
623     }
624     return result;
625 }
626 #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 Returns	Meaning
TPM_RC_CANCELED	test was canceled

```

627 LIB_EXPORT
628 TPM_RC
629 TestAlgorithm(
630     TPM_ALG_ID      alg,
631     ALGORITHM_VECTOR *toTest
632 )
633 {
634     TPM_ALG_ID      first = (alg == ALG_ERROR_VALUE) ? ALG_FIRST_VALUE : alg;
635     TPM_ALG_ID      last  = (alg == ALG_ERROR_VALUE) ? ALG_LAST_VALUE : alg;
636     BOOL            doTest = (alg != ALG_ERROR_VALUE);
637     TPM_RC          result = TPM_RC_SUCCESS;
638
639     if(toTest == NULL)
640         toTest = &g_toTest;
641
642     // This is kind of strange. This function will either run a test of the selected
643     // algorithm or just clear a bit if there is no test for the algorithm. So,
644     // either this loop will be executed once for the selected algorithm or once for
645     // each of the possible algorithms. If it is executed more than once ('alg' ==
646     // ALG_ERROR), then no test will be run but bits will be cleared for
647     // unimplemented algorithms. This was done this way so that there is only one
648     // case statement with all of the algorithms. It was easier to have one case
649     // statement than to have multiple ones to manage whenever an algorithm ID is
650     // added.
651     for(alg = first; (alg <= last); alg++)
652     {
653         // if 'alg' was TPM_ALG_ERROR, then we will be cycling through
654         // values, some of which may not be implemented. If the bit in toTest
655         // happens to be set, then we could either generated an assert, or just
656         // silently CLEAR it. Decided to just clear.
657         if(!TEST_BIT(alg, g_implementedAlgorithms))

```

```

658     {
659         CLEAR_BIT(alg, *toTest);
660         continue;
661     }
662     // Process whatever is left.
663     // NOTE: since this switch will only be called if the algorithm is
664     // implemented, it is not necessary to modify this list except to comment
665     // out the algorithms for which there is no test
666     switch(alg)
667     {
668         // Symmetric block ciphers
669     #if ALG_AES
670         case ALG_AES_VALUE:
671     #endif // ALG_AES
672     #if ALG_SM4
673         // if SM4 is implemented, its test is like other block ciphers but there
674         // aren't any test vectors for it yet
675         // case ALG_SM4_VALUE:
676     #endif // ALG_SM4
677     #if ALG_CAMELLIA
678         // no test vectors for camellia
679         // case ALG_CAMELLIA_VALUE:
680     #endif
681         // Symmetric modes
682     #if !ALG_CFB
683     # error CFB is required in all TPM implementations
684     #endif // !ALG_CFB
685         case ALG_CFB_VALUE:
686             if(doTest)
687                 result = TestSymmetric(alg, toTest);
688             break;
689     #if ALG_CTR
690         case ALG_CTR_VALUE:
691     #endif // ALG_CTR
692     #if ALG_OFB
693         case ALG_OFB_VALUE:
694     #endif // ALG_OFB
695     #if ALG_CBC
696         case ALG_CBC_VALUE:
697     #endif // ALG_CBC
698     #if ALG_ECB
699         case ALG_ECB_VALUE:
700     #endif
701             if(doTest)
702                 result = TestSymmetric(alg, toTest);
703             else
704                 // If doing the initialization of g_toTest vector, only need
705                 // to test one of the modes for the symmetric algorithms. If
706                 // initializing for a SelfTest(FULL_TEST), allow all the modes.
707                 if(toTest == &g_toTest)
708                     CLEAR_BIT(alg, *toTest);
709             break;
710     #if !ALG_HMAC
711     # error HMAC is required in all TPM implementations
712     #endif
713         case ALG_HMAC_VALUE:
714             // Clear the bit that indicates that HMAC is required because
715             // HMAC is used as the basic test for all hash algorithms.
716             CLEAR_BOTH(alg);
717             // Testing HMAC means test the default hash
718             if(doTest)
719                 TestHash(DEFAULT_TEST_HASH, toTest);
720             else
721                 // If not testing, then indicate that the hash needs to be
722                 // tested because this uses HMAC
723                 SET_BOTH(DEFAULT_TEST_HASH);

```

```

724         break;
725 #if ALG_SHA1
726     case ALG_SHA1_VALUE:
727 #endif // ALG_SHA1
728 #if ALG_SHA256
729     case ALG_SHA256_VALUE:
730 #endif // ALG_SHA256
731 #if ALG_SHA384
732     case ALG_SHA384_VALUE:
733 #endif // ALG_SHA384
734 #if ALG_SHA512
735     case ALG_SHA512_VALUE:
736 #endif // ALG_SHA512
737     // if SM3 is implemented its test is like any other hash, but there
738     // aren't any test vectors yet.
739 #if ALG_SM3_256
740     case ALG_SM3_256_VALUE:
741 #endif // ALG_SM3_256
742     if(doTest)
743         result = TestHash(alg, toTest);
744     break;
745 // RSA-dependent
746 #if ALG_RSA
747     case ALG_RSA_VALUE:
748         CLEAR_BOTH(alg);
749         if(doTest)
750             result = TestRsa(ALG_NULL_VALUE, toTest);
751         else
752             SET_BOTH(ALG_NULL_VALUE);
753         break;
754     case ALG_RSASSA_VALUE:
755     case ALG_RSAES_VALUE:
756     case ALG_RSAPSS_VALUE:
757     case ALG_OAEP_VALUE:
758     case ALG_NULL_VALUE: // used or RSADP
759         if(doTest)
760             result = TestRsa(alg, toTest);
761         break;
762 #endif // ALG_RSA
763 #if ALG_KDF1_SP800_108
764     case ALG_KDF1_SP800_108_VALUE:
765         if(doTest)
766             result = TestKDFa(toTest);
767         break;
768 #endif // ALG_KDF1_SP800_108
769 #if ALG_ECC
770     // ECC dependent but no tests
771     // case ALG_ECDSA_VALUE:
772     // case ALG_ECMQV_VALUE:
773     // case ALG_KDF1_SP800_56a_VALUE:
774     // case ALG_KDF2_VALUE:
775     // case ALG_MGF1_VALUE:
776     case ALG_ECC_VALUE:
777         CLEAR_BOTH(alg);
778         if(doTest)
779             result = TestEcc(ALG_ECDH_VALUE, toTest);
780         else
781             SET_BOTH(ALG_ECDH_VALUE);
782         break;
783     case ALG_ECDSA_VALUE:
784     case ALG_ECDH_VALUE:
785     case ALG_ECSCHNORR_VALUE:
786 // case ALG_SM2_VALUE:
787     if(doTest)
788         result = TestEcc(alg, toTest);
789     break;

```

```
790 #endif // ALG_ECC
791     default:
792         CLEAR_BIT(alg, *toTest);
793         break;
794     }
795     if(result != TPM_RC_SUCCESS)
796         break;
797 }
798 return result;
799 }
800 #endif // SELF_TESTS
```

DRAFT

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 = 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_uword_t    requiredSize;
110     BYTE             *pFrom;
111     BYTE             *pTo;
112     crypt_uword_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_uword_t    a = op1[i];
17         crypt_uword_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_uword_t    *result,
31     const crypt_uword_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_uword_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 = AddSame(result->d, n1->d, n2->d, stop);
75  if(n1->size > stop)
76      carry = CarryProp(&result->d[stop], &n1->d[stop], n1->size - stop, carry);
77  CarryResolve(result, 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_ushort_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], op->size - 1, carry);
91      CarryResolve(result, 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_ushort_t    *result,
97      const crypt_ushort_t *op1,
98      const crypt_ushort_t *op2,
99      int               count
100  )
101  {
102      int             borrow = 0;
103      int             i;
104      for(i = 0; i < count; i++)
105      {
106          crypt_ushort_t a = op1[i];
107          crypt_ushort_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_UWORD) && 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     crypt_ushort_t stop = 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 > stop)
140         borrow = BorrowProp(&result->d[stop], &op1->d[stop], 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], 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 = 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
<i>n</i>	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 += (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_uword_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] |= (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, 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_ushort_t allocated, // IN:
43      crypt_ushort_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_ushort_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 CryptMac.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  #if ALG_CMAC

```

10.2.5.3 Functions

10.2.5.3.1 CryptMacStart()

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.

```

5  UINT16
6  CryptMacStart(
7      SMAC_STATE          *state,
8      TPMU_PUBLIC_PARMS  *keyParms,
9      TPM_ALG_ID          macAlg,
10     TPM2B                *key
11 )
12 {
13     tpmCmacState_t        *cState = &state->state.cmac;
14     TPMT_SYM_DEF_OBJECT *def = &keyParms->symDetail.sym;
15     //
16     if(macAlg != TPM_ALG_CMAC)
17         return 0;
18     // set up the encryption algorithm and parameters
19     cState->symAlg = def->algorithm;
20     cState->keySizeBits = def->keyBits.sym;
21     cState->iv.t.size = CryptGetSymmetricBlockSize(def->algorithm,
22                                                     def->keyBits.sym);
23     MemoryCopy2B(&cState->symKey.b, key, sizeof(cState->symKey.t.buffer));
24
25     // Set up the dispatch methods for the CMAC
26     state->smacMethods.data = CryptCmacData;
27     state->smacMethods.end = CryptCmacEnd;
28     return cState->iv.t.size;
29 }

```

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.

```

30 void
31 CryptCmacData(
32     SMAC_STATES      *state,
33     UINT32            size,
34     const BYTE        *buffer
35 )
36 {
37     tpmCmacState_t      *cmacState = &state->cmac;
38     TPM_ALG_ID          algorithm = cmacState->symAlg;
39     BYTE                *key = cmacState->symKey.t.buffer;
40     UINT16              keySizeInBits = cmacState->keySizeBits;
41     tpmCryptKeySchedule_t keySchedule;
42     TpmCryptSetSymKeyCall_t encrypt;
43     //
44     SELECT(ENCRYPT);
45     while(size > 0)
46     {
47         if(cmacState->bcount == cmacState->iv.t.size)
48         {
49             ENCRYPT(&keySchedule, cmacState->iv.t.buffer, cmacState->iv.t.buffer);
50             cmacState->bcount = 0;
51         }
52         for(;;(size > 0) && (cmacState->bcount < cmacState->iv.t.size);
53             size--, cmacState->bcount++)
54         {
55             cmacState->iv.t.buffer[cmacState->bcount] ^= *buffer++;
56         }
57     }
58 }

```

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.

```

59 UINT16
60 CryptCmacEnd(
61     SMAC_STATES      *state,
62     UINT32            outSize,
63     BYTE              *outBuffer
64 )
65 {
66     tpmCmacState_t      *cState = &state->cmac;
67     // Need to set algorithm, key, and keySizeInBits in the local context so that
68     // the SELECT and ENCRYPT macros will work here
69     TPM_ALG_ID          algorithm = cState->symAlg;
70     BYTE                *key = cState->symKey.t.buffer;
71     UINT16              keySizeInBits = cState->keySizeBits;
72     tpmCryptKeySchedule_t keySchedule;
73     TpmCryptSetSymKeyCall_t encrypt;
74     TPM2B_IV            subkey = {{0, {0}}};
75     BOOL                xorVal;
76     UINT16              i;
77
78     subkey.t.size = cState->iv.t.size;
79     // Encrypt a block of zero
80     SELECT(ENCRYPT);
81     ENCRYPT(&keySchedule, subkey.t.buffer, subkey.t.buffer);
82
83     // shift left by 1 and XOR with 0x0...87 if the MSb was 0
84     xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
85     ShiftLeft(&subkey.b);
86     subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
87     // this is a sanity check to make sure that the algorithm is working properly.

```

```
88     // remove this check when debug is done
89     pAssert(cState->bcount <= cState->iv.t.size);
90     // If the buffer is full then no need to compute subkey 2.
91     if(cState->bcount < cState->iv.t.size)
92     {
93         //Pad the data
94         cState->iv.t.buffer[cState->bcount++] ^= 0x80;
95         // The rest of the data is a pad of zero which would simply be XORed
96         // with the iv value so nothing to do...
97         // Now compute K2
98         xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
99         ShiftLeft(&subkey.b);
100         subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
101     }
102     // XOR the subkey into the IV
103     for(i = 0; i < subkey.t.size; i++)
104         cState->iv.t.buffer[i] ^= subkey.t.buffer[i];
105     ENCRYPT(&keySchedule, cState->iv.t.buffer, cState->iv.t.buffer);
106     i = (UINT16)MIN(cState->iv.t.size, outSize);
107     MemoryCopy(outBuffer, cState->iv.t.buffer, i);
108
109     return i;
110 }
111 #endif
8
```


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 Returns	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 Returns	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 != ALG_HMAC_VALUE)
31         || (signature->signature.hmac.hashAlg == ALG_NULL_VALUE))
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 != ALG_NULL_VALUE)
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 Returns	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                  hashBlockSize;
68         UINT16                  digestSize;
69
70         scheme = &publicArea->parameters.keyedHashDetail.scheme;
71
72         if(publicArea->type != ALG_KEYEDHASH_VALUE)
73             return TPM_RC_FAILURE;
74
75         // Pick the limiting hash algorithm
76         if(scheme->scheme == ALG_NULL_VALUE)
77             hashAlg = publicArea->nameAlg;
78         else if(scheme->scheme == ALG_XOR_VALUE)

```

```

79     hashAlg = scheme->details.xor.hashAlg;
80     else
81         hashAlg = scheme->details.hmac.hashAlg;
82     hashBlockSize = CryptHashGetBlockSize(hashAlg);
83     digestSize = CryptHashGetDigestSize(hashAlg);
84
85     // if this is a signing or a decryption key, then the limit
86     // for the data size is the block size of the hash. This limit
87     // is set because larger values have lower entropy because of the
88     // HMAC function. The lower limit is 1/2 the size of the digest
89     //
90     // If the user provided the key, check that it is a proper size
91     if(sensitiveCreate->data.t.size != 0)
92     {
93         if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
94            || IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
95         {
96             if(sensitiveCreate->data.t.size > hashBlockSize)
97                 return TPM_RC_SIZE;
98 #if 0 // May make this a FIPS-mode requirement
99             if(sensitiveCreate->data.t.size < (digestSize / 2))
100                 return TPM_RC_SIZE;
101 #endif
102         }
103         // If this is a data blob, then anything that will get past the unmarshaling
104         // is OK
105         MemoryCopy2B(&sensitive->sensitive.bits.b, &sensitiveCreate->data.b,
106                     sizeof(sensitive->sensitive.bits.t.buffer));
107     }
108     else
109     {
110         // The TPM is going to generate the data so set the size to be the
111         // size of the digest of the algorithm
112         sensitive->sensitive.bits.t.size =
113             DRBG_Generate(rand, sensitive->sensitive.bits.t.buffer, digestSize);
114         if(sensitive->sensitive.bits.t.size == 0)
115             return (g_inFailureMode) ? TPM_RC_FAILURE : TPM_RC_NO_RESULT;
116     }
117     return TPM_RC_SUCCESS;
118 }

```

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 ECDA. ECDA can be used to do things like U-Prove.

```

119 BOOL
120 CryptIsSchemeAnonymous(
121     TPM_ALG_ID    scheme           // IN: the scheme algorithm to test
122 )
123 {
124     return scheme == ALG_ECDA_VALUE;
125 }

```

10.2.6.4 Symmetric Functions

10.2.6.4.1 ParmDecryptSym()

This function performs parameter decryption using symmetric block cipher.

```

126 void
127 ParmDecryptSym(

```

```

128     TPM_ALG_ID      symAlg,          // IN: the symmetric algorithm
129     TPM_ALG_ID      hash,           // IN: hash algorithm for KDFa
130     UINT16           keySizeInBits,  // IN: the key size in bits
131     TPM2B            *key,           // IN: KDF HMAC key
132     TPM2B            *nonceCaller,   // IN: nonce caller
133     TPM2B            *nonceTpm,      // IN: nonce TPM
134     UINT32           dataSize,       // IN: size of parameter buffer
135     BYTE             *data           // OUT: buffer to be decrypted
136 )
137 {
138     // KDF output buffer
139     // It contains parameters for the CFB encryption
140     // From MSB to LSB, they are the key and iv
141     BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
142     // Symmetric key size in byte
143     UINT16           keySize = (keySizeInBits + 7) / 8;
144     TPM2B_IV         iv;
145
146     iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
147     // If there is decryption to do...
148     if(iv.t.size > 0)
149     {
150         // Generate key and iv
151         CryptKDFa(hash, key, CFB_KEY, nonceCaller, nonceTpm,
152                 keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
153         MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
154
155         CryptSymmetricDecrypt(data, symAlg, keySizeInBits, symParmString,
156                             &iv, ALG_CFB_VALUE, dataSize, data);
157     }
158     return;
159 }

```

10.2.6.4.2 ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```

160 void
161 ParmEncryptSym(
162     TPM_ALG_ID      symAlg,          // IN: symmetric algorithm
163     TPM_ALG_ID      hash,           // IN: hash algorithm for KDFa
164     UINT16           keySizeInBits,  // IN: symmetric key size in bits
165     TPM2B            *key,           // IN: KDF HMAC key
166     TPM2B            *nonceCaller,   // IN: nonce caller
167     TPM2B            *nonceTpm,      // IN: nonce TPM
168     UINT32           dataSize,       // IN: size of parameter buffer
169     BYTE             *data           // OUT: buffer to be encrypted
170 )
171 {
172     // KDF output buffer
173     // It contains parameters for the CFB encryption
174     BYTE             symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
175
176     // Symmetric key size in bytes
177     UINT16           keySize = (keySizeInBits + 7) / 8;
178
179     TPM2B_IV         iv;
180
181     iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
182     // See if there is any encryption to do
183     if(iv.t.size > 0)
184     {
185         // Generate key and iv
186         CryptKDFa(hash, key, CFB_KEY, nonceTpm, nonceCaller,

```

```

187         keySizeInBits + (iv.t.size * 8), symParmString, NULL, FALSE);
188     MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
189
190     CryptSymmetricEncrypt(data, symAlg, keySizeInBits, symParmString, &iv,
191         ALG_CFB_VALUE, dataSize, data);
192 }
193 return;
194 }

```

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 Returns	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

```

195 static TPM_RC
196 CryptGenerateKeySymmetric(
197     TPMT_PUBLIC *publicArea,           // IN/OUT: The public area template
198                                           // for the new key.
199     TPMT_SENSITIVE *sensitive,         // OUT: sensitive area
200     TPMS_SENSITIVE_CREATE *sensitiveCreate, // IN: sensitive creation data
201     RAND_STATE *rand                   // IN: the "entropy" source for
202 )
203 {
204     UINT16 keyBits = publicArea->parameters.symDetail.sym.keyBits.sym;
205     TPM_RC result;
206     //
207     // only do multiples of RADIX_BITS
208     if((keyBits % RADIX_BITS) != 0)
209         return TPM_RC_KEY_SIZE;
210     // If this is not a new key, then the provided key data must be the right size
211     if(sensitiveCreate->data.t.size != 0)
212     {
213         result = CryptSymKeyValidate(&publicArea->parameters.symDetail.sym,
214             (TPM2B_SYM_KEY *)&sensitiveCreate->data);
215         if(result == TPM_RC_SUCCESS)
216             MemoryCopy2B(&sensitive->sensitive.sym.b, &sensitiveCreate->data.b,
217                 sizeof(sensitive->sensitive.sym.t.buffer));
218     }
219     #if ALG_TDES
220     else if(publicArea->parameters.symDetail.sym.algorithm == ALG_TDES_VALUE)
221     {
222         result = CryptGenerateKeyDes(publicArea, sensitive, rand);
223     }
224     #endif
225     else
226     {
227         sensitive->sensitive.sym.t.size =
228             DRBG_Generate(rand, sensitive->sensitive.sym.t.buffer,
229                 BITS_TO_BYTES(keyBits));
230         if(g_inFailureMode)
231             result = TPM_RC_FAILURE;
232         else if(sensitive->sensitive.sym.t.size == 0)
233             result = TPM_RC_NO_RESULT;
234         else
235             result = TPM_RC_SUCCESS;
236     }

```

```

237     return result;
238 }

```

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.

```

239 void
240 CryptXORObfuscation(
241     TPM_ALG_ID    hash,           // IN: hash algorithm for KDF
242     TPM2B         *key,           // IN: KDF key
243     TPM2B         *contextU,      // IN: contextU
244     TPM2B         *contextV,      // IN: contextV
245     UINT32        dataSize,       // IN: size of data buffer
246     BYTE          *data           // IN/OUT: data to be XORed in place
247 )
248 {
249     BYTE          mask[MAX_DIGEST_SIZE]; // Allocate a digest sized buffer
250     BYTE          *pm;
251     UINT32        i;
252     UINT32        counter = 0;
253     UINT16        hLen = CryptHashGetDigestSize(hash);
254     UINT32        requestSize = dataSize * 8;
255     INT32         remainBytes = (INT32)dataSize;
256
257     pAssert((key != NULL) && (data != NULL) && (hLen != 0));
258
259     // Call KDFa to generate XOR mask
260     for(; remainBytes > 0; remainBytes -= hLen)
261     {
262         // Make a call to KDFa to get next iteration
263         CryptKDFa(hash, key, XOR_KEY, contextU, contextV,
264                 requestSize, mask, &counter, TRUE);
265
266         // XOR next piece of the data
267         pm = mask;
268         for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
269             *data++ ^= *pm++;
270     }
271     return;
272 }

```

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

```

273 BOOL
274 CryptInit(
275     void

```

```

276     )
277 {
278     BOOL        ok;
279     // Initialize the vector of implemented algorithms
280     AlgorithmGetImplementedVector(&g_implementedAlgorithms);
281
282     // Indicate that all test are necessary
283     CryptInitializeToTest();
284
285     // Do any library initializations that are necessary. If any fails,
286     // the caller should go into failure mode;
287     ok = SupportLibInit();
288     ok = ok && CryptSymInit();
289     ok = ok && CryptRandInit();
290     ok = ok && CryptHashInit();
291 #if ALG_RSA
292     ok = ok && CryptRsaInit();
293 #endif // ALG_RSA
294 #if ALG_ECC
295     ok = ok && CryptEccInit();
296 #endif // ALG_ECC
297     return ok;
298 }

```

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

```

299  BOOL
300  CryptStartup(
301      STARTUP_TYPE    type           // IN: the startup type
302  )
303  {
304      BOOL        OK;
305      NOT_REFERENCED(type);
306
307      OK = CryptSymStartup() && CryptRandStartup() && CryptHashStartup()
308 #if ALG_RSA
309     && CryptRsaStartup()
310 #endif // ALG_RSA
311 #if ALG_ECC
312     && CryptEccStartup()
313 #endif // ALG_ECC
314     ;
315 #if ALG_ECC
316     // Don't directly check for SU_RESET because that is the default
317     if(OK && (type != SU_RESTART) && (type != SU_RESUME))
318     {
319         // If the shutdown was orderly, then the values recovered from NV will
320         // be OK to use.
321         // Get a new random commit nonce
322         gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
323         CryptRandomGenerate(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
324         // Reset the counter and commit array
325         gr.commitCounter = 0;
326         MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));

```



```

327     }
328 #endif // ALG_ECC
329     return OK;
330 }

```

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

```

331  BOOL
332  CryptIsAsymAlgorithm(
333      TPM_ALG_ID      algID          // IN: algorithm ID
334  )
335  {
336      switch(algID)
337      {
338      #if ALG_RSA
339          case ALG_RSA_VALUE:
340      #endif
341      #if ALG_ECC
342          case ALG_ECC_VALUE:
343      #endif
344          return TRUE;
345          break;
346      default:
347          break;
348      }
349      return FALSE;
350  }

```

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 Returns	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

```

351  TPM_RC

```

```

352 CryptSecretEncrypt(
353     OBJECT *encryptKey, // IN: encryption key object
354     const TPM2B *label, // IN: a null-terminated string as L
355     TPM2B_DATA *data, // OUT: secret value
356     TPM2B_ENCRYPTED_SECRET *secret // OUT: secret structure
357 )
358 {
359     TPMT_RSA_DECRYPT scheme;
360     TPM_RC result = TPM_RC_SUCCESS;
361     //
362     if(data == NULL || secret == NULL)
363         return TPM_RC_FAILURE;
364
365     // The output secret value has the size of the digest produced by the nameAlg.
366     data->t.size = CryptHashGetDigestSize(encryptKey->publicArea.nameAlg);
367     // The encryption scheme is OAEP using the nameAlg of the encrypt key.
368     scheme.scheme = ALG_OAEP_VALUE;
369     scheme.details.anySig.hashAlg = encryptKey->publicArea.nameAlg;
370
371     if(!IS_ATTRIBUTE(encryptKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
372         return TPM_RC_ATTRIBUTES;
373     switch(encryptKey->publicArea.type)
374     {
375 #if ALG_RSA
376         case ALG_RSA_VALUE:
377         {
378             // Create secret data from RNG
379             CryptRandomGenerate(data->t.size, data->t.buffer);
380
381             // Encrypt the data by RSA OAEP into encrypted secret
382             result = CryptRsaEncrypt((TPM2B_PUBLIC_KEY_RSA *)secret, &data->b,
383                                     encryptKey, &scheme, label, NULL);
384         }
385         break;
386 #endif // ALG_RSA
387
388 #if ALG_ECC
389         case ALG_ECC_VALUE:
390         {
391             TPMS_ECC_POINT eccPublic;
392             TPM2B_ECC_PARAMETER eccPrivate;
393             TPMS_ECC_POINT eccSecret;
394             BYTE *buffer = secret->t.secret;
395
396             // Need to make sure that the public point of the key is on the
397             // curve defined by the key.
398             if(!CryptEccIsPointOnCurve(
399                 encryptKey->publicArea.parameters.eccDetail.curveID,
400                 &encryptKey->publicArea.unique.ecc))
401                 result = TPM_RC_KEY;
402             else
403             {
404                 // Call crypto engine to create an auxiliary ECC key
405                 // We assume crypt engine initialization should always success.
406                 // Otherwise, TPM should go to failure mode.
407
408                 CryptEccNewKeyPair(&eccPublic, &eccPrivate,
409                                     encryptKey->publicArea.parameters.eccDetail.curveID);
410                 // Marshal ECC public to secret structure. This will be used by the
411                 // recipient to decrypt the secret with their private key.
412                 secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
413
414                 // Compute ECDH shared secret which is R = [d]Q where d is the
415                 // private part of the ephemeral key and Q is the public part of a
416                 // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
417                 // because the auxiliary ECC key is just created according to the

```

```

418         // parameters of input ECC encrypt key.
419         if (CryptEccPointMultiply(&eccSecret,
420             encryptKey->publicArea.parameters.eccDetail.curveID,
421             &encryptKey->publicArea.unique.ecc, &eccPrivate,
422             NULL, NULL)
423             != TPM_RC_SUCCESS)
424             result = TPM_RC_KEY;
425         else
426         {
427             // The secret value is computed from Z using KDFe as:
428             // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
429             // Where:
430             // HashID the nameAlg of the decrypt key
431             // Z the x coordinate (Px) of the product (P) of the point
432             // (Q) of the secret and the private x coordinate (de,V)
433             // of the decryption key
434             // Use a null-terminated string containing "SECRET"
435             // PartyUInfo the x coordinate of the point in the secret
436             // (Qe,U )
437             // PartyVInfo the x coordinate of the public key (Qs,V )
438             // bits the number of bits in the digest of HashID
439             // Retrieve seed from KDFe
440             CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
441                 label, &eccPublic.x.b,
442                 &encryptKey->publicArea.unique.ecc.x.b,
443                 data->t.size * 8, data->t.buffer);
444         }
445     }
446 }
447 break;
448 #endif // ALG_ECC
449 default:
450     FAIL(FATAL_ERROR_INTERNAL);
451     break;
452 }
453 return result;
454 }

```

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 Returns	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 <i>keyedHash</i> or symmetric key, the secret is larger than the size of the digest produced by the name algorithm.
TPM_RC_FAILURE	internal error

```

455 TPM_RC
456 CryptSecretDecrypt(
457     OBJECT                *decryptKey,    // IN: decrypt key
458     TPM2B_NONCE            *nonceCaller,    // IN: nonceCaller. It is needed for
459                                     // symmetric decryption. For
460                                     // asymmetric decryption, this
461                                     // parameter is NULL
462     const TPM2B            *label,          // IN: a value for L
463     TPM2B_ENCRYPTED_SECRET *secret,         // IN: input secret
464     TPM2B_DATA             *data           // OUT: decrypted secret value
465 )
466 {
467     TPM_RC result = TPM_RC_SUCCESS;
468
469     // Decryption for secret
470     switch(decryptKey->publicArea.type)
471     {
472     #if ALG_RSA
473     case ALG_RSA_VALUE:
474     {
475         TPMT_RSA_DECRYPT    scheme;
476         TPMT_RSA_SCHEME    *keyScheme
477             = &decryptKey->publicArea.parameters.rsaDetail.scheme;
478         UINT16             digestSize;
479
480         scheme = *(TPMT_RSA_DECRYPT *)keyScheme;
481         // If the key scheme is ALG_NULL_VALUE, set the scheme to OAEP and
482         // set the algorithm to the name algorithm.
483         if(scheme.scheme == ALG_NULL_VALUE)
484         {
485             // Use OAEP scheme
486             scheme.scheme = ALG_OAEP_VALUE;
487             scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
488         }
489         // use the digestSize as an indicator of whether or not the scheme
490         // is using a supported hash algorithm.
491         // Note: depending on the scheme used for encryption, a hashAlg might
492         // not be needed. However, the return value has to have some upper
493         // limit on the size. In this case, it is the size of the digest of the
494         // hash algorithm. It is checked after the decryption is done but, there
495         // is no point in doing the decryption if the size is going to be
496         // 'wrong' anyway.
497         digestSize = CryptHashGetDigestSize(scheme.details.oaep.hashAlg);
498         if(scheme.scheme != ALG_OAEP_VALUE || digestSize == 0)
499             return TPM_RC_SCHEME;

```

```

500
501 // Set the output buffer capacity
502 data->t.size = sizeof(data->t.buffer);
503
504 // Decrypt seed by RSA OAEP
505 result = CryptRsaDecrypt(&data->b, &secret->b,
506                        decryptKey, &scheme, label);
507 if((result == TPM_RC_SUCCESS) && (data->t.size > digestSize))
508     result = TPM_RC_VALUE;
509 }
510 break;
511 #endif // ALG_RSA
512 #if ALG_ECC
513     case ALG_ECC_VALUE:
514     {
515         TPMS_ECC_POINT      eccPublic;
516         TPMS_ECC_POINT      eccSecret;
517         BYTE                *buffer = secret->t.secret;
518         INT32               size = secret->t.size;
519
520         // Retrieve ECC point from secret buffer
521         result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
522         if(result == TPM_RC_SUCCESS)
523         {
524             result = CryptEccPointMultiply(&eccSecret,
525                                           decryptKey->publicArea.parameters.eccDetail.curveID,
526                                           &eccPublic, &decryptKey->sensitive.sensitive.ecc,
527                                           NULL, NULL);
528             if(result == TPM_RC_SUCCESS)
529             {
530                 // Set the size of the "recovered" secret value to be the size
531                 // of the digest produced by the nameAlg.
532                 data->t.size =
533                     CryptHashGetDigestSize(decryptKey->publicArea.nameAlg);
534
535                 // The secret value is computed from Z using KDFe as:
536                 // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
537                 // Where:
538                 // HashID -- the nameAlg of the decrypt key
539                 // Z -- the x coordinate (Px) of the product (P) of the point
540                 //      (Q) of the secret and the private x coordinate (de,V)
541                 //      of the decryption key
542                 // Use -- a null-terminated string containing "SECRET"
543                 // PartyUInfo -- the x coordinate of the point in the secret
544                 //      (Qe,U )
545                 // PartyVInfo -- the x coordinate of the public key (Qs,V )
546                 // bits -- the number of bits in the digest of HashID
547                 // Retrieve seed from KDFe
548                 CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
549                          &eccPublic.x.b,
550                          &decryptKey->publicArea.unique.ecc.x.b,
551                          data->t.size * 8, data->t.buffer);
552             }
553         }
554     }
555     break;
556 #endif // ALG_ECC
557 #if !ALG_KEYEDHASH
558 #error "KEYEDHASH support is required"
559 #endif
560     case ALG_KEYEDHASH_VALUE:
561         // The seed size can not be bigger than the digest size of nameAlg
562         if(secret->t.size >
563            CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
564             result = TPM_RC_VALUE;
565         else

```

```

566     {
567         // Retrieve seed by XOR Obfuscation:
568         // seed = XOR(secret, hash, key, nonceCaller, nullNonce)
569         // where:
570         // secret the secret parameter from the TPM2_StartAuthHMAC
571         // command that contains the seed value
572         // hash nameAlg of tpmKey
573         // key the key or data value in the object referenced by
574         // entityHandle in the TPM2_StartAuthHMAC command
575         // nonceCaller the parameter from the TPM2_StartAuthHMAC command
576         // nullNonce a zero-length nonce
577         // XOR Obfuscation in place
578         CryptXORObfuscation(decryptKey->publicArea.nameAlg,
579                             &decryptKey->sensitive.sensitive.bits.b,
580                             &nonceCaller->b, NULL,
581                             secret->t.size, secret->t.secret);
582         // Copy decrypted seed
583         MemoryCopy2B(&data->b, &secret->b, sizeof(data->t.buffer));
584     }
585     break;
586 case ALG_SYMCIPHER_VALUE:
587     {
588         TPM2B_IV iv = {{0}};
589         TPMT_SYM_DEF_OBJECT *symDef;
590         // The seed size can not be bigger than the digest size of nameAlg
591         if(secret->t.size >
592            CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
593             result = TPM_RC_VALUE;
594         else
595         {
596             symDef = &decryptKey->publicArea.parameters.symDetail.sym;
597             iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
598                                                     symDef->keyBits.sym);
599             if(iv.t.size == 0)
600                 return TPM_RC_FAILURE;
601             if(nonceCaller->t.size >= iv.t.size)
602             {
603                 MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
604             }
605             else
606             {
607                 if(nonceCaller->t.size > sizeof(iv.t.buffer))
608                     return TPM_RC_FAILURE;
609                 MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
610                             nonceCaller->t.size);
611             }
612             // make sure secret will fit
613             if(secret->t.size > data->t.size)
614                 return TPM_RC_FAILURE;
615             data->t.size = secret->t.size;
616             // CFB decrypt, using nonceCaller as iv
617             CryptSymmetricDecrypt(data->t.buffer, symDef->algorithm,
618                                   symDef->keyBits.sym,
619                                   decryptKey->sensitive.sensitive.sym.t.buffer,
620                                   &iv, ALG_CFB_VALUE, secret->t.size,
621                                   secret->t.secret);
622         }
623     }
624     break;
625 default:
626     FAIL(FATAL_ERROR_INTERNAL);
627     break;
628 }
629 return result;
630 }

```

10.2.6.6.5 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

```

631 void
632 CryptParameterEncryption(
633     TPM_HANDLE    handle,           // IN: encrypt session handle
634     TPM2B         *nonceCaller,     // IN: nonce caller
635     UINT16        leadingSizeInByte, // IN: the size of the leading size field in
636                                     // bytes
637     TPM2B_AUTH    *extraKey,        // IN: additional key material other than
638                                     // sessionAuth
639     BYTE          *buffer            // IN/OUT: parameter buffer to be encrypted
640 )
641 {
642     SESSION        *session = SessionGet(handle); // encrypt session
643     TPM2B_TYPE(TEMP_KEY, (sizeof(extraKey->t.buffer)
644                           + sizeof(session->sessionKey.t.buffer)));
645     TPM2B_TEMP_KEY key;              // encryption key
646     UINT32         cipherSize = 0;   // size of cipher text
647 //
648 // Retrieve encrypted data size.
649 if(leadingSizeInByte == 2)
650 {
651     // Extract the first two bytes as the size field as the data size
652     // encrypt
653     cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
654     // advance the buffer
655     buffer = &buffer[2];
656 }
657 #ifdef TPM4B
658 else if(leadingSizeInByte == 4)
659 {
660     // use the first four bytes to indicate the number of bytes to encrypt
661     cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
662     //advance pointer
663     buffer = &buffer[4];
664 }
665 #endif
666 else
667 {
668     FAIL(FATAL_ERROR_INTERNAL);
669 }
670
671 // Compute encryption key by concatenating sessionKey with extra key
672 MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
673 MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
674
675 if(session->symmetric.algorithm == ALG_XOR_VALUE)
676 {
677     // XOR parameter encryption formulation:
678     // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
679     CryptXORObfuscation(session->authHashAlg, &(key.b),
680                         &(session->nonceTPM.b),
681                         nonceCaller, cipherSize, buffer);
682 }
683 else
684 {
685     ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
686                   session->symmetric.keyBits.aes, &(key.b),
687                   nonceCaller, &(session->nonceTPM.b),
688                   cipherSize, buffer);
689 }
690 return;
691 }

```


10.2.6.6.6 CryptParameterDecryption()

This function does in-place decryption of a command parameter.

Error Returns	Meaning
TPM_RC_SIZE	The number of bytes in the input buffer is less than the number of bytes to be decrypted.

```

689  TPM_RC
690  CryptParameterDecryption(
691      TPM_HANDLE      handle,          // IN: encrypted session handle
692      TPM2B           *nonceCaller,    // IN: nonce caller
693      UINT32          bufferSize,      // IN: size of parameter buffer
694      UINT16           leadingSizeInByte, // IN: the size of the leading size field in
695                                          // byte
696      TPM2B_AUTH       *extraKey,      // IN: the authValue
697      BYTE             *buffer         // IN/OUT: parameter buffer to be decrypted
698  )
699  {
700      SESSION          *session = SessionGet(handle); // encrypt session
701      // The HMAC key is going to be the concatenation of the session key and any
702      // additional key material (like the authValue). The size of both of these
703      // is the size of the buffer which can contain a TPMT_HA.
704      TPM2B_TYPE(HMAC_KEY, (sizeof(extraKey->t.buffer)
705                             + sizeof(session->sessionKey.t.buffer)));
706      TPM2B_HMAC_KEY   key;            // decryption key
707      UINT32            cipherSize = 0; // size of cipher text
708  //
709  // Retrieve encrypted data size.
710  if(leadingSizeInByte == 2)
711  {
712      // The first two bytes of the buffer are the size of the
713      // data to be decrypted
714      cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
715      buffer = &buffer[2]; // advance the buffer
716  }
717  #ifdef TPM4B
718  else if(leadingSizeInByte == 4)
719  {
720      // the leading size is four bytes so get the four byte size field
721      cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
722      buffer = &buffer[4]; //advance pointer
723  }
724  #endif
725  else
726  {
727      FAIL(FATAL_ERROR_INTERNAL);
728  }
729  if(cipherSize > bufferSize)
730      return TPM_RC_SIZE;
731
732  // Compute decryption key by concatenating sessionAuth with extra input key
733  MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
734  MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
735
736  if(session->symmetric.algorithm == ALG_XOR_VALUE)
737      // XOR parameter decryption formulation:
738      // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
739      // Call XOR obfuscation function
740      CryptXORObfuscation(session->authHashAlg, &key.b, nonceCaller,
741                          &(session->nonceTPM.b), cipherSize, buffer);
742  else
743      // Assume that it is one of the symmetric block ciphers.
744      ParmDecryptSym(session->symmetric.algorithm, session->authHashAlg,

```

```

745         session->symmetric.keyBits.sym,
746         &key.b, nonceCaller, &session->nonceTPM.b,
747         cipherSize, buffer);
748
749     return TPM_RC_SUCCESS;
750 }

```

10.2.6.6.7 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```

751 void
752 CryptComputeSymmetricUnique(
753     TPMT_PUBLIC *publicArea,    // IN: the object's public area
754     TPMT_SENSITIVE *sensitive,  // IN: the associated sensitive area
755     TPM2B_DIGEST *unique        // OUT: unique buffer
756 )
757 {
758     // For parents (symmetric and derivation), use an HMAC to compute
759     // the 'unique' field
760     if (IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
761         && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt))
762     {
763         // Unique field is HMAC(sensitive->seedValue, sensitive->sensitive)
764         HMAC_STATE hmacState;
765         unique->b.size = CryptHmacStart2B(&hmacState, publicArea->nameAlg,
766                                           &sensitive->seedValue.b);
767         CryptDigestUpdate2B(&hmacState.hashState,
768                             &sensitive->sensitive.any.b);
769         CryptHmacEnd2B(&hmacState, &unique->b);
770     }
771     else
772     {
773         HASH_STATE hashState;
774         // Unique := Hash(sensitive->seedValue || sensitive->sensitive)
775         unique->t.size = CryptHashStart(&hashState, publicArea->nameAlg);
776         CryptDigestUpdate2B(&hashState, &sensitive->seedValue.b);
777         CryptDigestUpdate2B(&hashState, &sensitive->sensitive.any.b);
778         CryptHashEnd2B(&hashState, &unique->b);
779     }
780     return;
781 }

```

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 Returns	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

```

782  TPM_RC
783  CryptCreateObject(
784      OBJECT                *object,                // IN: new object structure pointer
785      TPMS_SENSITIVE_CREATE *sensitiveCreate,       // IN: sensitive creation
786      RAND_STATE            *rand                    // IN: the random number generator
787                                              // to use
788  )
789  {
790      TPMT_PUBLIC      *publicArea = &object->publicArea;
791      TPMT_SENSITIVE   *sensitive = &object->sensitive;
792      TPM_RC           result = TPM_RC_SUCCESS;
793  //
794  // Set the sensitive type for the object
795  sensitive->sensitiveType = publicArea->type;
796
797  // For all objects, copy the initial authorization data
798  sensitive->authValue = sensitiveCreate->userAuth;
799
800  // If the TPM is the source of the data, set the size of the provided data to
801  // zero so that there's no confusion about what to do.
802  if(IS_ATTRIBUTE(publicArea->objectAttributes,
803                  TPMA_OBJECT, sensitiveDataOrigin))
804      sensitiveCreate->data.t.size = 0;
805
806  // Generate the key and unique fields for the asymmetric keys and just the
807  // sensitive value for symmetric object
808  switch(publicArea->type)
809  {
810  #if ALG_RSA
811      // Create RSA key
812      case ALG_RSA_VALUE:
813          // RSA uses full object so that it has a place to put the private
814          // exponent
815          result = CryptRsaGenerateKey(publicArea, sensitive, rand);
816          break;
817  #endif // ALG_RSA
818
819  #if ALG_ECC
820      // Create ECC key
821      case ALG_ECC_VALUE:
822          result = CryptEccGenerateKey(publicArea, sensitive, rand);
823          break;
824  #endif // ALG_ECC
825      case ALG_SYMCIPHER_VALUE:

```

```

826         result = CryptGenerateKeySymmetric(publicArea, sensitive,
827                                           sensitiveCreate, rand);
828         break;
829     case ALG_KEYEDHASH_VALUE:
830         result = CryptGenerateKeyedHash(publicArea, sensitive,
831                                       sensitiveCreate, rand);
832         break;
833     default:
834         FAIL(FATAL_ERROR_INTERNAL);
835         break;
836 }
837 if(result != TPM_RC_SUCCESS)
838     return result;
839 // Create the sensitive seed value
840 // If this is a primary key in the endorsement hierarchy, stir the DRBG state
841 // This implementation uses both shProof and ehProof to make sure that there
842 // is no leakage of either.
843 if(object->attributes.primary && object->attributes.epsHierarchy)
844 {
845     DRBG_AdditionalData((DRBG_STATE *)rand, &gp.shProof.b);
846     DRBG_AdditionalData((DRBG_STATE *)rand, &gp.ehProof.b);
847 }
848 // Generate a seedValue that is the size of the digest produced by nameAlg
849 sensitive->seedValue.t.size =
850     DRBG_Generate(rand, sensitive->seedValue.t.buffer,
851                  CryptHashGetDigestSize(publicArea->nameAlg));
852 if(g_inFailureMode)
853     return TPM_RC_FAILURE;
854 else if(sensitive->seedValue.t.size == 0)
855     return TPM_RC_NO_RESULT;
856 // For symmetric objects, need to compute the unique value for the public area
857 if(publicArea->type == ALG_SYMCIPHER_VALUE
858    || publicArea->type == ALG_KEYEDHASH_VALUE)
859 {
860     CryptComputeSymmetricUnique(publicArea, sensitive, &publicArea->unique.sym);
861 }
862 else
863 {
864     // if this is an asymmetric key and it isn't a parent, then
865     // get rid of the seed.
866     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
867        || !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
868         memset(&sensitive->seedValue, 0, sizeof(sensitive->seedValue));
869 }
870 // Compute the name
871 PublicMarshalAndComputeName(publicArea, &object->name);
872 return result;
873 }

```

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

```

874 TPMI_ALG_HASH
875 CryptGetSignHashAlg(
876     TPMT_SIGNATURE *auth          // IN: signature
877 )
878 {
879     if(auth->sigAlg == ALG_NULL_VALUE)
880         FAIL(FATAL_ERROR_INTERNAL);
881
882     // Get authHash algorithm based on signing scheme
883     switch(auth->sigAlg)

```

```

884     {
885     #if ALG_RSA
886     // If RSA is supported, both RSASSA and RSAPSS are required
887     # if !defined ALG_RSASSA_VALUE || !defined ALG_RSAPSS_VALUE
888     # error "RSASSA and RSAPSS are required for RSA"
889     # endif
890     case ALG_RSASSA_VALUE:
891         return auth->signature.rsassa.hash;
892     case ALG_RSAPSS_VALUE:
893         return auth->signature.rsapss.hash;
894     #endif // ALG_RSA
895
896     #if ALG_ECC
897     // If ECC is defined, ECDSA is mandatory
898     # if !ALG_ECDSA
899     # error "ECDSA is required for ECC"
900     # endif
901     case ALG_ECDSA_VALUE:
902     // SM2 and ECSCHNORR are optional
903
904     # if ALG_SM2
905     case ALG_SM2_VALUE:
906     # endif
907     # if ALG_ECSCHNORR
908     case ALG_ECSCHNORR_VALUE:
909     # endif
910     //all ECC signatures look the same
911     return auth->signature.ecdsa.hash;
912
913     # if ALG_ECDA
914     // Don't know how to verify an ECDA signature
915     case ALG_ECDA_VALUE:
916     break;
917     # endif
918     #endif // ALG_ECC
919
920     case ALG_HMAC_VALUE:
921     return auth->signature.hmac.hashAlg;
922
923     default:
924     break;
925     }
926     return ALG_NULL_VALUE;
927 }

```

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().

```

929     BOOL
930     CryptIsSplitSign(
931         TPM_ALG_ID    scheme           // IN: the algorithm selector
932     )
933     {
934         switch(scheme)
935         {
936         # if ALG_ECDA
937         case ALG_ECDA_VALUE:
938             return TRUE;
939             break;
940         # endif // ALG_ECDA
941         default:

```

```

942         return FALSE;
943         break;
944     }
945 }

```

10.2.6.6.11 CryptIsAsymSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```

946 BOOL
947 CryptIsAsymSignScheme(
948     TPMI_ALG_PUBLIC      publicKey,          // IN: Type of the object
949     TPMI_ALG_ASYNC_SCHEME scheme            // IN: the scheme
950 )
951 {
952     BOOL          isSignScheme = TRUE;
953
954     switch(publicType)
955     {
956     #if ALG_RSA
957         case ALG_RSA_VALUE:
958             switch(scheme)
959             {
960             # if !ALG_RSASSA || !ALG_RSAPSS
961             #     error "RSASSA and PSAPSS required if RSA used."
962             #     endif
963
964                 case ALG_RSASSA_VALUE:
965                 case ALG_RSAPSS_VALUE:
966                     break;
967                 default:
968                     isSignScheme = FALSE;
969                     break;
970             }
971             break;
972     #endif // ALG_RSA
973
974     #if ALG_ECC
975         // If ECC is implemented ECDSA is required
976         case ALG_ECC_VALUE:
977             switch(scheme)
978             {
979             // Support for ECDSA is required for ECC
980             case ALG_ECDSA_VALUE:
981             #if ALG_ECDAE // ECDAE is optional
982             case ALG_ECDAE_VALUE:
983             #endif
984             #if ALG_ECSCHNORR // Schnorr is also optional
985             case ALG_ECSCHNORR_VALUE:
986             #endif
987             #if ALG_SM2 // SM2 is optional
988             case ALG_SM2_VALUE:
989             #endif
990                 break;
991             default:
992                 isSignScheme = FALSE;
993                 break;
994             }
995             break;
996     #endif // ALG_ECC
997     default:
998         isSignScheme = FALSE;
999         break;
1000 }
1001 return isSignScheme;

```

```
1001 }
```

10.2.6.6.12 CryptIsAsymDecryptScheme()

This function indicate if a scheme algorithm is a decrypt algorithm.

```
1002 BOOL
1003 CryptIsAsymDecryptScheme(
1004     TPMI_ALG_PUBLIC      publicKey,      // IN: Type of the object
1005     TPMI_ALG_ASYNC_SCHEME  scheme        // IN: the scheme
1006 )
1007 {
1008     BOOL      isDecryptScheme = TRUE;
1009
1010     switch(publicType)
1011     {
1012 #if ALG_RSA
1013         case ALG_RSA_VALUE:
1014             switch(scheme)
1015             {
1016                 case ALG_RSAES_VALUE:
1017                 case ALG_OAEP_VALUE:
1018                     break;
1019                 default:
1020                     isDecryptScheme = FALSE;
1021                     break;
1022             }
1023             break;
1024 #endif // ALG_RSA
1025
1026 #if ALG_ECC
1027     // If ECC is implemented ECDH is required
1028     case ALG_ECC_VALUE:
1029         switch(scheme)
1030         {
1031             #if !ALG_ECDH
1032             # error "ECDH is required for ECC"
1033             #endif
1034             case ALG_ECDH_VALUE:
1035 #if ALG_SM2
1036             case ALG_SM2_VALUE:
1037             #endif
1038 #if ALG_ECMQV
1039             case ALG_ECMQV_VALUE:
1040             #endif
1041                 break;
1042             default:
1043                 isDecryptScheme = FALSE;
1044                 break;
1045         }
1046         break;
1047 #endif // ALG_ECC
1048     default:
1049         isDecryptScheme = FALSE;
1050         break;
1051     }
1052     return isDecryptScheme;
1053 }
```


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>

```

1054  BOOL
1055  CryptSelectSignScheme (
1056      OBJECT          *signObject,      // IN: signing key
1057      TPMT_SIG_SCHEME *scheme           // IN/OUT: signing scheme
1058  )
1059  {
1060      TPMT_SIG_SCHEME *objectScheme;
1061      TPMT_PUBLIC      *publicArea;
1062      BOOL             OK;
1063
1064      // If the signHandle is TPM_RH_NULL, then the NULL scheme is used, regardless
1065      // of the setting of scheme
1066      if(signObject == NULL)
1067      {
1068          OK = TRUE;
1069          scheme->scheme = ALG_NULL_VALUE;
1070          scheme->details.any.hashAlg = ALG_NULL_VALUE;
1071      }
1072      else
1073      {
1074          // assignment to save typing.
1075          publicArea = &signObject->publicArea;
1076
1077          // A symmetric cipher can be used to encrypt and decrypt but it can't
1078          // be used for signing
1079          if(publicArea->type == ALG_SYMCIPHER_VALUE)
1080              return FALSE;
1081          // Point to the scheme object
1082          if(CryptIsAsymAlgorithm(publicArea->type))
1083              objectScheme =
1084                  (TPMT_SIG_SCHEME *) &publicArea->parameters.asymDetail.scheme;
1085          else
1086              objectScheme =
1087                  (TPMT_SIG_SCHEME *) &publicArea->parameters.keyedHashDetail.scheme;
1088
1089          // If the object doesn't have a default scheme, then use the
1090          // input scheme.
1091          if(objectScheme->scheme == ALG_NULL_VALUE)
1092          {
1093              // Input and default can't both be NULL
1094              OK = (scheme->scheme != ALG_NULL_VALUE);
1095              // Assume that the scheme is compatible with the key. If not,
1096              // an error will be generated in the signing operation.
1097          }
1098          else if(scheme->scheme == ALG_NULL_VALUE)

```

```

1099     {
1100         // input scheme is NULL so use default
1101
1102         // First, check to see if the default requires that the caller
1103         // provided scheme data
1104         OK = !CryptIsSplitSign(objectScheme->scheme);
1105         if(OK)
1106         {
1107             // The object has a scheme and the input is TPM_ALG_NULL so copy
1108             // the object scheme as the final scheme. It is better to use a
1109             // structure copy than a copy of the individual fields.
1110             *scheme = *objectScheme;
1111         }
1112     }
1113     else
1114     {
1115         // Both input and object have scheme selectors
1116         // If the scheme and the hash are not the same then...
1117         // NOTE: the reason that there is no copy here is that the input
1118         // might contain extra data for a split signing scheme and that
1119         // data is not in the object so, it has to be preserved.
1120         OK = (objectScheme->scheme == scheme->scheme)
1121             && (objectScheme->details.any.hashAlg
1122                == scheme->details.any.hashAlg);
1123     }
1124 }
1125 return OK;
1126 }

```

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 Returns	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 in <i>signScheme</i> (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)

```

1127 TPM_RC
1128 CryptSign(
1129     OBJECT            *signKey,        // IN: signing key
1130     TPMT_SIG_SCHEME   *signScheme,     // IN: sign scheme.
1131     TPM2B_DIGEST      *digest,         // IN: The digest being signed
1132     TPMT_SIGNATURE     *signature      // OUT: signature
1133 )
1134 {
1135     TPM_RC            result = TPM_RC_SCHEME;
1136
1137     // Initialize signature scheme
1138     signature->sigAlg = signScheme->scheme;
1139
1140     // If the signature algorithm is TPM_ALG_NULL or the signing key is NULL,
1141     // then we are done
1142     if((signature->sigAlg == ALG_NULL_VALUE) || (signKey == NULL))
1143         return TPM_RC_SUCCESS;
1144
1145     // Initialize signature hash

```

```

1146 // Note: need to do the check for TPM_ALG_NULL first because the null scheme
1147 // doesn't have a hashAlg member.
1148 signature->signature.any.hashAlg = signScheme->details.any.hashAlg;
1149
1150 // perform sign operation based on different key type
1151 switch(signKey->publicArea.type)
1152 {
1153 #if ALG_RSA
1154     case ALG_RSA_VALUE:
1155         result = CryptRsaSign(signature, signKey, digest, NULL);
1156         break;
1157 #endif // ALG_RSA
1158 #if ALG_ECC
1159     case ALG_ECC_VALUE:
1160         // The reason that signScheme is passed to CryptEccSign but not to the
1161         // other signing methods is that the signing for ECC may be split and
1162         // need the 'r' value that is in the scheme but not in the signature.
1163         result = CryptEccSign(signature, signKey, digest,
1164                               (TPMT_ECC_SCHEME *)signScheme, NULL);
1165         break;
1166 #endif // ALG_ECC
1167     case ALG_KEYEDHASH_VALUE:
1168         result = CryptHmacSign(signature, signKey, digest);
1169         break;
1170     default:
1171         FAIL(FATAL_ERROR_INTERNAL);
1172         break;
1173 }
1174 return result;
1175 }

```

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 Returns	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

```

1176 TPM_RC
1177 CryptValidateSignature(
1178     TPMI_DH_OBJECT    keyHandle,        // IN: The handle of sign key
1179     TPM2B_DIGEST       *digest,         // IN: The digest being validated
1180     TPMT_SIGNATURE     *signature       // IN: signature
1181 )
1182 {
1183     // NOTE: HandleToObject will either return a pointer to a loaded object or
1184     // will assert. It will never return a non-valid value. This makes it save
1185     // to initialize 'publicArea' with the return value from HandleToObject()
1186     // without checking it first.
1187     OBJECT *signObject = HandleToObject(keyHandle);
1188     TPMT_PUBLIC *publicArea = &signObject->publicArea;
1189     TPM_RC result = TPM_RC_SCHEME;
1190 }

```

```

1191     // The input unmarshaling should prevent any input signature from being
1192     // a NULL signature, but just in case
1193     if(signature->sigAlg == ALG_NULL_VALUE)
1194         return TPM_RC_SIGNATURE;
1195
1196     switch(publicArea->type)
1197     {
1198     #if ALG_RSA
1199         case ALG_RSA_VALUE:
1200             {
1201                 //
1202                 // Call RSA code to verify signature
1203                 result = CryptRsaValidateSignature(signature, signObject, digest);
1204                 break;
1205             }
1206     #endif // ALG_RSA
1207
1208     #if ALG_ECC
1209         case ALG_ECC_VALUE:
1210             result = CryptEccValidateSignature(signature, signObject, digest);
1211             break;
1212     #endif // ALG_ECC
1213
1214         case ALG_KEYEDHASH_VALUE:
1215             if(signObject->attributes.publicOnly)
1216                 result = TPM_RCS_HANDLE;
1217             else
1218                 result = CryptHMACVerifySignature(signObject, digest, signature);
1219             break;
1220         default:
1221             break;
1222     }
1223     return result;
1224 }

```

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().

```

1225 TPM_RC
1226 CryptGetTestResult(
1227     TPM2B_MAX_BUFFER *outData    // OUT: test result data
1228 )
1229 {
1230     outData->t.size = 0;
1231     return TPM_RC_SUCCESS;
1232 }

```

10.2.6.6.17 CryptIsUniqueSizeValid()

This function validates that the unique values are consistent.

NOTE: This is not a comprehensive test of the public key.

Return Value	Meaning
TRUE(1)	sizes are consistent
FALSE(0)	sizes are not consistent

```

1233  BOOL
1234  CryptIsUniqueSizeValid(
1235      TPMT_PUBLIC      *publicArea      // IN: the public area to check
1236  )
1237  {
1238      BOOL              consistent = FALSE;
1239      UINT16            keySizeInBytes;
1240
1241      switch(publicArea->type)
1242      {
1243      #if ALG_RSA
1244          case ALG_RSA_VALUE:
1245              keySizeInBytes = BITS_TO_BYTES(
1246                  publicArea->parameters.rsaDetail.keyBits);
1247              consistent = publicArea->unique.rsa.t.size == keySizeInBytes;
1248              break;
1249      #endif // ALG_RSA
1250      #if ALG_ECC
1251          case ALG_ECC_VALUE:
1252              {
1253                  keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(
1254                      publicArea->parameters.eccDetail.curveID));
1255                  consistent = keySizeInBytes > 0
1256                      && publicArea->unique.ecc.x.t.size <= keySizeInBytes
1257                      && publicArea->unique.ecc.y.t.size <= keySizeInBytes;
1258              }
1259              break;
1260      #endif // ALG_ECC
1261          default:
1262              // For SYMCIPHER and KEYDEDHASH objects, the unique field is the size
1263              // of the nameAlg digest.
1264              consistent = publicArea->unique.sym.t.size
1265                  == CryptHashGetDigestSize(publicArea->nameAlg);
1266              break;
1267      }
1268      return consistent;
1269  }

```

10.2.6.6.18 CryptIsSensitiveSizeValid()

This function is used by TPM2_LoadExternal() to validate that the sensitive area contains a *sensitive* value that is consistent with the values in the public area.

```

1270  BOOL
1271  CryptIsSensitiveSizeValid(
1272      TPMT_PUBLIC      *publicArea,      // IN: the object's public part
1273      TPMT_SENSITIVE    *sensitiveArea   // IN: the object's sensitive part
1274  )
1275  {
1276      BOOL              consistent;
1277      UINT16            keySizeInBytes;
1278
1279      switch(publicArea->type)
1280      {
1281      #if ALG_RSA

```

```

1282     case ALG_RSA_VALUE:
1283         // sensitive prime value has to be half the size of the public modulus
1284         keySizeInBytes = BITS_TO_BYTES(publicArea->parameters.rsaDetail.keyBits);
1285         consistent =
1286             ((sensitiveArea->sensitive.rsa.t.size * 2) == keySizeInBytes);
1287         break;
1288     #endif
1289     #if ALG_ECC
1290     case ALG_ECC_VALUE:
1291         keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(
1292             publicArea->parameters.eccDetail.curveID));
1293         consistent = (keySizeInBytes > 0)
1294             && (sensitiveArea->sensitive.ecc.t.size == keySizeInBytes);
1295         break;
1296     #endif
1297     case ALG_SYMCIPHER_VALUE:
1298         keySizeInBytes =
1299             BITS_TO_BYTES(publicArea->parameters.symDetail.sym.keyBits.sym);
1300         consistent = keySizeInBytes == sensitiveArea->sensitive.sym.t.size;
1301         break;
1302     case ALG_KEYEDHASH_VALUE:
1303         keySizeInBytes = CryptHashGetBlockSize(publicArea->nameAlg);
1304         // if the block size is 0, then the algorithm is TPM_ALG_NULL and the
1305         // size of the private part is limited to 128. If the algorithm block
1306         // size is over 128 bytes, then the size is limited to 128 bytes for
1307         // interoperability reasons.
1308         if((keySizeInBytes == 0) || (keySizeInBytes > 128))
1309             keySizeInBytes = 128;
1310         consistent = sensitiveArea->sensitive.bits.t.size <= keySizeInBytes;
1311         break;
1312     default:
1313         consistent = TRUE;
1314         break;
1315 }
1316 return consistent;
1317 }
1318

```

10.2.6.6.19 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 Returns	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

```

1319     TPM_RC
1320     CryptValidateKeys(
1321         TPMT_PUBLIC      *publicArea,
1322         TPMT_SENSITIVE    *sensitive,

```

```

1323     TPM_RC          blamePublic,
1324     TPM_RC          blameSensitive
1325 )
1326 {
1327     TPM_RC          result;
1328     UINT16          keySizeInBytes;
1329     UINT16          digestSize = CryptHashGetDigestSize(publicArea->nameAlg);
1330     TPMU_PUBLIC_PARMS *params = &publicArea->parameters;
1331     TPMU_PUBLIC_ID    *unique = &publicArea->unique;
1332
1333     if(sensitive != NULL)
1334     {
1335         // Make sure that the types of the public and sensitive are compatible
1336         if(publicArea->type != sensitive->sensitiveType)
1337             return TPM_RCS_TYPE + blameSensitive;
1338         // Make sure that the authValue is not bigger than allowed
1339         // If there is no name algorithm, then the size just needs to be less than
1340         // the maximum size of the buffer used for authorization. That size check
1341         // was made during unmarshaling of the sensitive area
1342         if((sensitive->authValue.t.size) > digestSize && (digestSize > 0))
1343             return TPM_RCS_SIZE + blameSensitive;
1344     }
1345     switch(publicArea->type)
1346     {
1347 #if ALG_RSA
1348         case ALG_RSA_VALUE:
1349             keySizeInBytes = BITS_TO_BYTES(params->rsaDetail.keyBits);
1350
1351             // Regardless of whether there is a sensitive area, the public modulus
1352             // needs to have the correct size. Otherwise, it can't be used for
1353             // any public key operation nor can it be used to compute the private
1354             // exponent.
1355             // NOTE: This implementation only supports key sizes that are multiples
1356             // of 1024 bits which means that the MSb of the 0th byte will always be
1357             // SET in any prime and in the public modulus.
1358             if((unique->rsa.t.size != keySizeInBytes)
1359                || (unique->rsa.t.buffer[0] < 0x80))
1360                 return TPM_RCS_KEY + blamePublic;
1361             if(params->rsaDetail.exponent != 0
1362                && params->rsaDetail.exponent < 7)
1363                 return TPM_RCS_VALUE + blamePublic;
1364             if(sensitive != NULL)
1365             {
1366                 // If there is a sensitive area, it has to be the correct size
1367                 // including having the correct high order bit SET.
1368                 if(((sensitive->sensitive.rsa.t.size * 2) != keySizeInBytes)
1369                    || (sensitive->sensitive.rsa.t.buffer[0] < 0x80))
1370                     return TPM_RCS_KEY_SIZE + blameSensitive;
1371             }
1372             break;
1373 #endif
1374 #if ALG_ECC
1375         case ALG_ECC_VALUE:
1376         {
1377             TPMI_ECC_CURVE    curveId;
1378             curveId = params->eccDetail.curveID;
1379             keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(curveId));
1380             if(sensitive == NULL)
1381             {
1382                 // Validate the public key size
1383                 if(unique->ecc.x.t.size != keySizeInBytes
1384                    || unique->ecc.y.t.size != keySizeInBytes)
1385                     return TPM_RCS_KEY + blamePublic;
1386                 if(publicArea->nameAlg != ALG_NULL_VALUE)
1387                 {
1388                     if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))

```



```

1389         return TPM_RCS_ECC_POINT + blamePublic;
1390     }
1391 }
1392 else
1393 {
1394     // If the nameAlg is TPM_ALG_NULL, then only verify that the
1395     // private part of the key is OK.
1396     if(!CryptEccIsValidPrivateKey(&sensitive->sensitive.ecc,
1397                                   curveId))
1398         return TPM_RCS_KEY_SIZE;
1399     if(publicArea->nameAlg != ALG_NULL_VALUE)
1400     {
1401         // Full key load, verify that the public point belongs to the
1402         // private key.
1403         TPMS_ECC_POINT toCompare;
1404         result = CryptEccPointMultiply(&toCompare, curveId, NULL,
1405                                         &sensitive->sensitive.ecc,
1406                                         NULL, NULL);
1407         if(result != TPM_RC_SUCCESS)
1408             return TPM_RCS_BINDING;
1409         else
1410         {
1411             // Make sure that the private key generated the public key.
1412             // The input values and the values produced by the point
1413             // multiply may not be the same size so adjust the computed
1414             // value to match the size of the input value by adding or
1415             // removing zeros.
1416             AdjustNumberB(&toCompare.x.b, unique->ecc.x.t.size);
1417             AdjustNumberB(&toCompare.y.b, unique->ecc.y.t.size);
1418             if(!MemoryEqual2B(&unique->ecc.x.b, &toCompare.x.b)
1419                || !MemoryEqual2B(&unique->ecc.y.b, &toCompare.y.b))
1420                 return TPM_RCS_BINDING;
1421         }
1422     }
1423 }
1424 break;
1425 }
1426 #endif
1427 default:
1428     // Checks for SYMCIPHER and KEYEDHASH are largely the same
1429     // If public area has a nameAlg, then validate the public area size
1430     // and if there is also a sensitive area, validate the binding
1431
1432     // For consistency, if the object is public-only just make sure that
1433     // the unique field is consistent with the name algorithm
1434     if(sensitive == NULL)
1435     {
1436         if(unique->sym.t.size != digestSize)
1437             return TPM_RCS_KEY + blamePublic;
1438     }
1439     else
1440     {
1441         // Make sure that the key size in the sensitive area is consistent.
1442         if(publicArea->type == ALG_SYMCIPHER_VALUE)
1443         {
1444             result = CryptSymKeyValidate(&params->symDetail.sym,
1445                                         &sensitive->sensitive.sym);
1446             if(result != TPM_RC_SUCCESS)
1447                 return result + blameSensitive;
1448         }
1449         else
1450         {
1451             // For a keyed hash object, the key has to be less than the
1452             // smaller of the block size of the hash used in the scheme or
1453             // 128 bytes. The worst case value is limited by the
1454             // unmarshaling code so the only thing left to be checked is

```

```

1455         // that it does not exceed the block size of the hash.
1456         // by the hash algorithm of the scheme.
1457         TPMT_KEYEDHASH_SCHEME      *scheme;
1458         UINT16                      maxSize;
1459         scheme = &params->keyedHashDetail.scheme;
1460         if(scheme->scheme == ALG_XOR_VALUE)
1461         {
1462             maxSize = CryptHashGetBlockSize(scheme->details.xor.hashAlg);
1463         }
1464         else if(scheme->scheme == ALG_HMAC_VALUE)
1465         {
1466             maxSize = CryptHashGetBlockSize(scheme->details.hmac.hashAlg);
1467         }
1468         else if(scheme->scheme == ALG_NULL_VALUE)
1469         {
1470             // Not signing or xor so must be a data block
1471             maxSize = 128;
1472         }
1473         else
1474             return TPM_RCS_SCHEME + blamePublic;
1475         if(sensitive->sensitive.bits.t.size > maxSize)
1476             return TPM_RCS_KEY_SIZE + blameSensitive;
1477     }
1478     // If there is a nameAlg, check the binding
1479     if(publicArea->nameAlg != ALG_NULL_VALUE)
1480     {
1481         TPM2B_DIGEST                compare;
1482         if(sensitive->seedValue.t.size != digestSize)
1483             return TPM_RCS_KEY_SIZE + blameSensitive;
1484
1485         CryptComputeSymmetricUnique(publicArea, sensitive, &compare);
1486         if(!MemoryEqual2B(&unique->sym.b, &compare.b))
1487             return TPM_RC_BINDING;
1488     }
1489 }
1490 break;
1491 }
1492 // For a parent, need to check that the seedValue is the correct size for
1493 // protections. It should be at least half the size of the nameAlg
1494 if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
1495    && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
1496    && sensitive != NULL
1497    && publicArea->nameAlg != ALG_NULL_VALUE)
1498 {
1499     if((sensitive->seedValue.t.size < (digestSize / 2))
1500        || (sensitive->seedValue.t.size > digestSize))
1501         return TPM_RCS_SIZE + blameSensitive;
1502 }
1503 return TPM_RC_SUCCESS;
1504 }

```

10.2.6.6.20 CryptAlgSetImplemented()

This function initializes the bit vector with one bit for each implemented algorithm. This function is called from `_TPM_Init()`. The vector of implemented algorithms should be generated by the part 2 parser so that the `g_implementedAlgorithms` vector can be a constant. That's not how it is now

```

1505 void
1506 CryptAlgsSetImplemented(
1507     void
1508 )
1509 {
1510     AlgorithmGetImplementedVector(&g_implementedAlgorithms);
1511 }

```

10.2.6.6.21 CryptSelectMac()

This function is used to set the MAC scheme based on the key parameters and the input scheme.

Error Returns	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

```

1512 TPM_RC
1513 CryptSelectMac(
1514     TPMT_PUBLIC          *publicArea,
1515     TPMI_ALG_MAC_SCHEME  *inMac
1516 )
1517 {
1518     TPM_ALG_ID            macAlg = ALG_NULL_VALUE;
1519     switch(publicArea->type)
1520     {
1521         case ALG_KEYEDHASH_VALUE:
1522         {
1523             // Local value to keep lines from getting too long
1524             TPMT_KEYEDHASH_SCHEME *scheme;
1525             scheme = &publicArea->parameters.keyedHashDetail.scheme;
1526             // Expect that the scheme is either HMAC or NULL
1527             if(scheme->scheme != ALG_NULL_VALUE)
1528                 macAlg = scheme->details.hmac.hashAlg;
1529             break;
1530         }
1531         case ALG_SYMCIPHER_VALUE:
1532         {
1533             TPMT_SYM_DEF_OBJECT *scheme;
1534             scheme = &publicArea->parameters.symDetail.sym;
1535             // Expect that the scheme is either valid symmetric cipher or NULL
1536             if(scheme->algorithm != ALG_NULL_VALUE)
1537                 macAlg = scheme->mode.sym;
1538             break;
1539         }
1540         default:
1541             return TPM_RCS_TYPE;
1542     }
1543     // If the input value is not TPM_ALG_NULL ...
1544     if(*inMac != ALG_NULL_VALUE)
1545     {
1546         // ... then either the scheme in the key must be TPM_ALG_NULL or the input
1547         // value must match
1548         if((macAlg != ALG_NULL_VALUE) && (*inMac != macAlg))
1549             return TPM_RCS_VALUE;
1550     }
1551     else
1552     {
1553         // Since the input value is TPM_ALG_NULL, then the key value can't be
1554         // TPM_ALG_NULL
1555         if(macAlg == ALG_NULL_VALUE)
1556             return TPM_RCS_VALUE;
1557         *inMac = macAlg;
1558     }
1559     if(!CryptMacIsValidForKey(publicArea->type, *inMac, FALSE))
1560         return TPM_RCS_SCHEME;
1561     return TPM_RC_SUCCESS;
1562 }

```

10.2.6.6.22 CryptMacIsValidForKey()

Check to see if the key type is compatible with the mac type

```

1563  BOOL
1564  CryptMacIsValidForKey(
1565      TPM_ALG_ID      keyType,
1566      TPM_ALG_ID      macAlg,
1567      BOOL           flag
1568  )
1569  {
1570      switch(keyType)
1571      {
1572          case ALG_KEYEDHASH_VALUE:
1573              return CryptHashIsValidAlg(macAlg, flag);
1574              break;
1575          case ALG_SYMCIPHER_VALUE:
1576              return CryptSmacIsValidAlg(macAlg, flag);
1577              break;
1578          default:
1579              break;
1580      }
1581      return FALSE;
1582  }

```

10.2.6.6.23 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.

```

1583  BOOL
1584  CryptSmacIsValidAlg(
1585      TPM_ALG_ID      alg,
1586      BOOL           FLAG      // IN: Indicates if TPM_ALG_NULL is valid
1587  )
1588  {
1589      switch (alg)
1590      {
1591          #if ALG_CMACE
1592              case ALG_CMACE_VALUE:
1593                  return TRUE;
1594                  break;
1595          #endif
1596              case ALG_NULL_VALUE:
1597                  return FLAG;
1598                  break;
1599              default:
1600                  return FALSE;
1601      }
1602  }

```

10.2.6.6.24 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

```

1603  BOOL
1604  CryptSymModeIsValid(
1605      TPM_ALG_ID      mode,
1606      BOOL           flag
1607  )
1608  {

```

```
1609     switch(mode)
1610     {
1611     #if ALG_CTR
1612         case ALG_CTR_VALUE:
1613     #endif // ALG_CTR
1614     #if ALG_OFB
1615         case ALG_OFB_VALUE:
1616     #endif // ALG_OFB
1617     #if ALG_CBC
1618         case ALG_CBC_VALUE:
1619     #endif // ALG_CBC
1620     #if ALG_CFB
1621         case ALG_CFB_VALUE:
1622     #endif // ALG_CFB
1623     #if ALG_ECB
1624         case ALG_ECB_VALUE:
1625     #endif // ALG_ECB
1626         return TRUE;
1627         case ALG_NULL_VALUE:
1628             return flag;
1629             break;
1630         default:
1631             break;
1632     }
1633     return FALSE;
1634 }
```

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 Returns	Meaning
TPM_RC_CANCELED	if the command is canceled

```

22  LIB_EXPORT
23  TPM_RC
24  CryptSelfTest(
25      TPML_YES_NO      fullTest      // IN: if full test is required
26  )
27  {
28  #if SIMULATION
29      if(g_forceFailureMode)
30          FAIL(FATAL_ERROR_FORCED);
31  #endif
32
33      // If the caller requested a full test, then reset the toTest vector so that
34      // all the tests will be run
35      if(fullTest == YES)
36      {
37          MemoryCopy(g_toTest,
38                      g_implementedAlgorithms,
39                      sizeof(g_toTest));
40      }
41      return CryptRunSelfTests(&g_toTest);
42  }

```

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 Returns	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

```

43  TPM_RC
44  CryptIncrementalSelfTest(
45      TPML_ALG          *toTest,      // IN: list of algorithms to be tested
46      TPML_ALG          *toDoList    // OUT: list of algorithms needing test
47  )
48  {
49      ALGORITHM_VECTOR   toTestVector = {0};
50      TPM_ALG_ID         alg;
51      UINT32             i;
52
53      pAssert(toTest != NULL && toDoList != NULL);
54      if(toTest->count > 0)
55      {
56          // Transcribe the toTest list into the toTestVector
57          for(i = 0; i < toTest->count; i++)
58          {
59              alg = toTest->algorithms[i];
60
61              // make sure that the algorithm value is not out of range
62              if((alg > TPM_ALG_LAST) || !TEST_BIT(alg, g_implementedAlgorithms))

```



```

63         return TPM_RC_VALUE;
64         SET_BIT(alg, toTestVector);
65     }
66     // Run the test
67     if(CryptRunSelfTests(&toTestVector) == TPM_RC_CANCELED)
68         return TPM_RC_CANCELED;
69 }
70 // Fill in the toDoList with the algorithms that are still untested
71 toDoList->count = 0;
72
73 for(alg = TPM_ALG_FIRST;
74 toDoList->count < MAX_ALG_LIST_SIZE && alg <= TPM_ALG_LAST;
75     alg++)
76 {
77     if(TEST_BIT(alg, g_toTest))
78         toDoList->algorithms[toDoList->count++] = alg;
79 }
80 return TPM_RC_SUCCESS;
81 }

```

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

```

82 void
83 CryptInitializeToTest(
84     void
85 )
86 {
87     // Indicate that nothing has been tested
88     memset(&g_cryptoSelfTestState, 0, sizeof(g_cryptoSelfTestState));
89
90     // Copy the implemented algorithm vector
91     MemoryCopy(g_toTest, g_implementedAlgorithms, sizeof(g_toTest));
92
93     // Setting the algorithm to null causes the test function to just clear
94     // out any algorithms for which there is no test.
95     CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);
96
97     return;
98 }

```

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 Returns	Meaning
TPM_RC_CANCELED	test was canceled

```

99 LIB_EXPORT
100 TPM_RC
101 CryptTestAlgorithm(
102     TPM_ALG_ID      alg,
103     ALGORITHM_VECTOR *toTest

```

```
104     )
105 {
106     TPM_RC          result;
107 #if SELF_TEST
108     result = TestAlgorithm(alg, toTest);
109 #else
110     // If this is an attempt to determine the algorithms for which there is a
111     // self test, pretend that all of them do. We do that by not clearing any
112     // of the algorithm bits. When/if this function is called to run tests, it
113     // will over report. This can be changed so that any call to check on which
114     // algorithms have tests, 'toTest' can be cleared.
115     if(alg != TPM_ALG_ERROR)
116     {
117         CLEAR_BIT(alg, g_toTest);
118         if(toTest != NULL)
119             CLEAR_BIT(alg, *toTest);
120     }
121     result = TPM_RC_SUCCESS;
122 #endif
123     return result;
124 }
```

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  #if USE_BN_ECC_DATA
5  #   define TO_ECC_64                                TO_CRYPT_WORD_64
6  #   define TO_ECC_56(a, b, c, d, e, f, g)            TO_ECC_64(0, a, b, c, d, e, f, g)
7  #   define TO_ECC_48(a, b, c, d, e, f)              TO_ECC_64(0, 0, a, b, c, d, e, f)
8  #   define TO_ECC_40(a, b, c, d, e)                 TO_ECC_64(0, 0, 0, a, b, c, d, e)
9  #   if RADIX_BITS > 32
10 #   define TO_ECC_32(a, b, c, d)                     TO_ECC_64(0, 0, 0, 0, a, b, c, d)
11 #   define TO_ECC_24(a, b, c)                       TO_ECC_64(0, 0, 0, 0, 0, a, b, c)
12 #   define TO_ECC_16(a, b)                          TO_ECC_64(0, 0, 0, 0, 0, 0, a, b)
13 #   define TO_ECC_8(a)                              TO_ECC_64(0, 0, 0, 0, 0, 0, 0, a)
14 #   else // RADIX_BITS == 32
15 #   define TO_ECC_32                                BIG_ENDIAN_BYTES_TO_UINT32
16 #   define TO_ECC_24(a, b, c)                       TO_ECC_32(0, a, b, c)
17 #   define TO_ECC_16(a, b)                          TO_ECC_32(0, 0, a, b)
18 #   define TO_ECC_8(a)                              TO_ECC_32(0, 0, 0, a)
19 #   endif
20 #else // TPM2B
21 #   define TO_ECC_64(a, b, c, d, e, f, g, h)          a, b, c, d, e, f, g, h
22 #   define TO_ECC_56(a, b, c, d, e, f, g)            a, b, c, d, e, f, g
23 #   define TO_ECC_48(a, b, c, d, e, f)              a, b, c, d, e, f
24 #   define TO_ECC_40(a, b, c, d, e)                 a, b, c, d, e
25 #   define TO_ECC_32(a, b, c, d)                    a, b, c, d
26 #   define TO_ECC_24(a, b, c)                      a, b, c
27 #   define TO_ECC_16(a, b)                         a, b
28 #   define TO_ECC_8(a)                             a
29 #endif
30 #if USE_BN_ECC_DATA
31 #define BN_MIN_ALLOC(bytes)                          \
32     (BYTES_TO_CRYPT_WORDS(bytes) == 0) ? 1 : BYTES_TO_CRYPT_WORDS(bytes) \
33 # define ECC_CONST(NAME, bytes, initializer)          \
34     const struct {                                     \
35         crypt_uword_t  allocate, size, d[BN_MIN_ALLOC(bytes)]; \
36         } NAME = {BN_MIN_ALLOC(bytes), BYTES_TO_CRYPT_WORDS(bytes), {initializer}} \
37 ECC_CONST(ECC_ZERO, 0, 0);
38 #else
39 # define ECC_CONST(NAME, bytes, initializer)          \
40     const TPM2B_##bytes##_BYTE_VALUE NAME = {bytes, {initializer}} \

```

Have to special case ECC_ZERO

```

41 TPM2B_BYTE_VALUE(1);
42 TPM2B_1_BYTE_VALUE ECC_ZERO = {1, {0}};
43 #endif
44 ECC_CONST(ECC_ONE, 1, 1);
45 #if !USE_BN_ECC_DATA
46 TPM2B_BYTE_VALUE(24);
47 #define TO_ECC_192(a, b, c)  a, b, c
48 TPM2B_BYTE_VALUE(28);
49 #define TO_ECC_224(a, b, c, d)  a, b, c, d
50 TPM2B_BYTE_VALUE(32);
51 #define TO_ECC_256(a, b, c, d)  a, b, c, d
52 TPM2B_BYTE_VALUE(48);

```

```

53 #define TO_ECC_384(a, b, c, d, e, f)      a, b, c, d, e, f
54 TPM2B_BYTE_VALUE(66);
55 #define TO_ECC_528(a, b, c, d, e, f, g, h, i)      a, b, c, d, e, f, g, h, i
56 TPM2B_BYTE_VALUE(80);
57 #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j)      a, b, c, d, e, f, g, h, i, j
58 #else
59 #define TO_ECC_192(a, b, c)      c, b, a
60 #define TO_ECC_224(a, b, c, d)      d, c, b, a
61 #define TO_ECC_256(a, b, c, d)      d, c, b, a
62 #define TO_ECC_384(a, b, c, d, e, f)      f, e, d, c, b, a
63 #define TO_ECC_528(a, b, c, d, e, f, g, h, i)      i, h, g, f, e, d, c, b, a
64 #define TO_ECC_640(a, b, c, d, e, f, g, h, i, j)      j, i, h, g, f, e, d, c, b, a
65 #endif // !USE_BN_ECC_DATA
66 #if ECC_NIST_P192
67 ECC_CONST(NIST_P192_p, 24, TO_ECC_192(
68     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
69     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
70     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
71 ECC_CONST(NIST_P192_a, 24, TO_ECC_192(
72     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
73     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
74     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
75 ECC_CONST(NIST_P192_b, 24, TO_ECC_192(
76     TO_ECC_64(0x64, 0x21, 0x05, 0x19, 0xE5, 0x9C, 0x80, 0xE7),
77     TO_ECC_64(0x0F, 0xA7, 0xE9, 0xAB, 0x72, 0x24, 0x30, 0x49),
78     TO_ECC_64(0xFE, 0xB8, 0xDE, 0xEC, 0xC1, 0x46, 0xB9, 0xB1)));
79 ECC_CONST(NIST_P192_gX, 24, TO_ECC_192(
80     TO_ECC_64(0x18, 0x8D, 0xA8, 0x0E, 0xB0, 0x30, 0x90, 0xF6),
81     TO_ECC_64(0x7C, 0xBF, 0x20, 0xEB, 0x43, 0xA1, 0x88, 0x00),
82     TO_ECC_64(0xF4, 0xFF, 0x0A, 0xFD, 0x82, 0xFF, 0x10, 0x12)));
83 ECC_CONST(NIST_P192_gY, 24, TO_ECC_192(
84     TO_ECC_64(0x07, 0x19, 0x2B, 0x95, 0xFF, 0xC8, 0xDA, 0x78),
85     TO_ECC_64(0x63, 0x10, 0x11, 0xED, 0x6B, 0x24, 0xCD, 0xD5),
86     TO_ECC_64(0x73, 0xF9, 0x77, 0xA1, 0x1E, 0x79, 0x48, 0x11)));
87 ECC_CONST(NIST_P192_n, 24, TO_ECC_192(
88     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
89     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x99, 0xDE, 0xF8, 0x36),
90     TO_ECC_64(0x14, 0x6B, 0xC9, 0xB1, 0xB4, 0xD2, 0x28, 0x31)));
91 #define NIST_P192_h      ECC_ONE
92 #define NIST_P192_gZ      ECC_ONE
93 #if USE_BN_ECC_DATA
94     const ECC_CURVE_DATA NIST_P192 = {
95         (bigNum)&NIST_P192_p, (bigNum)&NIST_P192_n, (bigNum)&NIST_P192_h,
96         (bigNum)&NIST_P192_a, (bigNum)&NIST_P192_b,
97         {(bigNum)&NIST_P192_gX, (bigNum)&NIST_P192_gY, (bigNum)&NIST_P192_gZ}};
98 #else
99     const ECC_CURVE_DATA NIST_P192 = {
100         &NIST_P192_p.b, &NIST_P192_n.b, &NIST_P192_h.b,
101         &NIST_P192_a.b, &NIST_P192_b.b,
102         {&NIST_P192_gX.b, &NIST_P192_gY.b, &NIST_P192_gZ.b}};
103 #endif // USE_BN_ECC_DATA
104 #endif // ECC_NIST_P192
105 #if ECC_NIST_P224
106 ECC_CONST(NIST_P224_p, 28, TO_ECC_224(
107     TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
108     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
109     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
110     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01)));
111 ECC_CONST(NIST_P224_a, 28, TO_ECC_224(
112     TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
113     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
114     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
115     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE)));
116 ECC_CONST(NIST_P224_b, 28, TO_ECC_224(
117     TO_ECC_32(0xB4, 0x05, 0x0A, 0x85),
118     TO_ECC_64(0x0C, 0x04, 0xB3, 0xAB, 0xF5, 0x41, 0x32, 0x56),

```

```

119     TO_ECC_64(0x50, 0x44, 0xB0, 0xB7, 0xD7, 0xBF, 0xD8, 0xBA),
120     TO_ECC_64(0x27, 0x0B, 0x39, 0x43, 0x23, 0x55, 0xFF, 0xB4));
121 ECC_CONST(NIST_P224_gX, 28, TO_ECC_224(
122     TO_ECC_32(0xB7, 0x0E, 0x0C, 0xBD),
123     TO_ECC_64(0x6B, 0xB4, 0xBF, 0x7F, 0x32, 0x13, 0x90, 0xB9),
124     TO_ECC_64(0x4A, 0x03, 0xC1, 0xD3, 0x56, 0xC2, 0x11, 0x22),
125     TO_ECC_64(0x34, 0x32, 0x80, 0xD6, 0x11, 0x5C, 0x1D, 0x21)));
126 ECC_CONST(NIST_P224_gY, 28, TO_ECC_224(
127     TO_ECC_32(0xBD, 0x37, 0x63, 0x88),
128     TO_ECC_64(0xB5, 0xF7, 0x23, 0xFB, 0x4C, 0x22, 0xDF, 0xE6),
129     TO_ECC_64(0xCD, 0x43, 0x75, 0xA0, 0x5A, 0x07, 0x47, 0x64),
130     TO_ECC_64(0x44, 0xD5, 0x81, 0x99, 0x85, 0x00, 0x7E, 0x34)));
131 ECC_CONST(NIST_P224_n, 28, TO_ECC_224(
132     TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
133     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
134     TO_ECC_64(0xFF, 0xFF, 0x16, 0xA2, 0xE0, 0xB8, 0xF0, 0x3E),
135     TO_ECC_64(0x13, 0xDD, 0x29, 0x45, 0x5C, 0x5C, 0x2A, 0x3D)));
136 #define NIST_P224_h      ECC_ONE
137 #define NIST_P224_gZ      ECC_ONE
138 #if USE_BN_ECC_DATA
139     const ECC_CURVE_DATA NIST_P224 = {
140         (bigNum)&NIST_P224_p, (bigNum)&NIST_P224_n, (bigNum)&NIST_P224_h,
141         (bigNum)&NIST_P224_a, (bigNum)&NIST_P224_b,
142         {(bigNum)&NIST_P224_gX, (bigNum)&NIST_P224_gY, (bigNum)&NIST_P224_gZ}};
143 #else
144     const ECC_CURVE_DATA NIST_P224 = {
145         &NIST_P224_p.b, &NIST_P224_n.b, &NIST_P224_h.b,
146         &NIST_P224_a.b, &NIST_P224_b.b,
147         {{&NIST_P224_gX.b, &NIST_P224_gY.b, &NIST_P224_gZ.b}};
148 #endif // USE_BN_ECC_DATA
149 #endif // ECC_NIST_P224
150 #if ECC_NIST_P256
151 ECC_CONST(NIST_P256_p, 32, TO_ECC_256(
152     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
153     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
154     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
155     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
156 ECC_CONST(NIST_P256_a, 32, TO_ECC_256(
157     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x01),
158     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
159     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF),
160     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
161 ECC_CONST(NIST_P256_b, 32, TO_ECC_256(
162     TO_ECC_64(0x5A, 0xC6, 0x35, 0xD8, 0xAA, 0x3A, 0x93, 0xE7),
163     TO_ECC_64(0xB3, 0xEB, 0xBD, 0x55, 0x76, 0x98, 0x86, 0xBC),
164     TO_ECC_64(0x65, 0x1D, 0x06, 0xB0, 0xCC, 0x53, 0xB0, 0xF6),
165     TO_ECC_64(0x3B, 0xCE, 0x3C, 0x3E, 0x27, 0xD2, 0x60, 0x4B)));
166 ECC_CONST(NIST_P256_gX, 32, TO_ECC_256(
167     TO_ECC_64(0x6B, 0x17, 0xD1, 0xF2, 0xE1, 0x2C, 0x42, 0x47),
168     TO_ECC_64(0xF8, 0xBC, 0xE6, 0xE5, 0x63, 0xA4, 0x40, 0xF2),
169     TO_ECC_64(0x77, 0x03, 0x7D, 0x81, 0x2D, 0xEB, 0x33, 0xA0),
170     TO_ECC_64(0xF4, 0xA1, 0x39, 0x45, 0xD8, 0x98, 0xC2, 0x96)));
171 ECC_CONST(NIST_P256_gY, 32, TO_ECC_256(
172     TO_ECC_64(0x4F, 0xE3, 0x42, 0xE2, 0xFE, 0x1A, 0x7F, 0x9B),
173     TO_ECC_64(0x8E, 0xE7, 0xEB, 0x4A, 0x7C, 0x0F, 0x9E, 0x16),
174     TO_ECC_64(0x2B, 0xCE, 0x33, 0x57, 0x6B, 0x31, 0x5E, 0xCE),
175     TO_ECC_64(0xCB, 0xB6, 0x40, 0x68, 0x37, 0xBF, 0x51, 0xF5)));
176 ECC_CONST(NIST_P256_n, 32, TO_ECC_256(
177     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
178     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
179     TO_ECC_64(0xBC, 0xE6, 0xFA, 0xAD, 0xA7, 0x17, 0x9E, 0x84),
180     TO_ECC_64(0xF3, 0xB9, 0xCA, 0xC2, 0xFC, 0x63, 0x25, 0x51)));
181 #define NIST_P256_h      ECC_ONE
182 #define NIST_P256_gZ      ECC_ONE
183 #if USE_BN_ECC_DATA
184     const ECC_CURVE_DATA NIST_P256 = {

```

```

185         (bigNum)&NIST_P256_p, (bigNum)&NIST_P256_n, (bigNum)&NIST_P256_h,
186         (bigNum)&NIST_P256_a, (bigNum)&NIST_P256_b,
187         {(bigNum)&NIST_P256_gX, (bigNum)&NIST_P256_gY, (bigNum)&NIST_P256_gZ});
188     #else
189         const ECC_CURVE_DATA NIST_P256 = {
190             &NIST_P256_p.b, &NIST_P256_n.b, &NIST_P256_h.b,
191             &NIST_P256_a.b, &NIST_P256_b.b,
192             {&NIST_P256_gX.b, &NIST_P256_gY.b, &NIST_P256_gZ.b}};
193     #endif // USE_BN_ECC_DATA
194     #endif // ECC_NIST_P256
195     #if ECC_NIST_P384
196     ECC_CONST(NIST_P384_p, 48, TO_ECC_384(
197         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
198         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
199         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
200         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
201         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
202         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFF)));
203     ECC_CONST(NIST_P384_a, 48, TO_ECC_384(
204         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
205         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
206         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
207         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFE),
208         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
209         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF, 0xFC)));
210     ECC_CONST(NIST_P384_b, 48, TO_ECC_384(
211         TO_ECC_64(0xB3, 0x31, 0x2F, 0xA7, 0xE2, 0x3E, 0xE7, 0xE4),
212         TO_ECC_64(0x98, 0x8E, 0x05, 0x6B, 0xE3, 0xF8, 0x2D, 0x19),
213         TO_ECC_64(0x18, 0x1D, 0x9C, 0x6E, 0xFE, 0x81, 0x41, 0x12),
214         TO_ECC_64(0x03, 0x14, 0x08, 0x8F, 0x50, 0x13, 0x87, 0x5A),
215         TO_ECC_64(0xC6, 0x56, 0x39, 0x8D, 0x8A, 0x2E, 0xD1, 0x9D),
216         TO_ECC_64(0x2A, 0x85, 0xC8, 0xED, 0xD3, 0xEC, 0x2A, 0xEF)));
217     ECC_CONST(NIST_P384_gX, 48, TO_ECC_384(
218         TO_ECC_64(0xAA, 0x87, 0xCA, 0x22, 0xBE, 0x8B, 0x05, 0x37),
219         TO_ECC_64(0x8E, 0xB1, 0xC7, 0x1E, 0xF3, 0x20, 0xAD, 0x74),
220         TO_ECC_64(0x6E, 0x1D, 0x3B, 0x62, 0x8B, 0xA7, 0x9B, 0x98),
221         TO_ECC_64(0x59, 0xF7, 0x41, 0xE0, 0x82, 0x54, 0x2A, 0x38),
222         TO_ECC_64(0x55, 0x02, 0xF2, 0x5D, 0xBF, 0x55, 0x29, 0x6C),
223         TO_ECC_64(0x3A, 0x54, 0x5E, 0x38, 0x72, 0x76, 0x0A, 0xB7)));
224     ECC_CONST(NIST_P384_gY, 48, TO_ECC_384(
225         TO_ECC_64(0x36, 0x17, 0xDE, 0x4A, 0x96, 0x26, 0x2C, 0x6F),
226         TO_ECC_64(0x5D, 0x9E, 0x98, 0xBF, 0x92, 0x92, 0xDC, 0x29),
227         TO_ECC_64(0xF8, 0xF4, 0x1D, 0xBD, 0x28, 0x9A, 0x14, 0x7C),
228         TO_ECC_64(0xE9, 0xDA, 0x31, 0x13, 0xB5, 0xF0, 0xB8, 0xC0),
229         TO_ECC_64(0x0A, 0x60, 0xB1, 0xCE, 0x1D, 0x7E, 0x81, 0x9D),
230         TO_ECC_64(0x7A, 0x43, 0x1D, 0x7C, 0x90, 0xEA, 0x0E, 0x5F)));
231     ECC_CONST(NIST_P384_n, 48, TO_ECC_384(
232         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
233         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
234         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
235         TO_ECC_64(0xC7, 0x63, 0x4D, 0x81, 0xF4, 0x37, 0x2D, 0xDF),
236         TO_ECC_64(0x58, 0x1A, 0x0D, 0xB2, 0x48, 0xB0, 0xA7, 0x7A),
237         TO_ECC_64(0xEC, 0xEC, 0x19, 0x6A, 0xCC, 0xC5, 0x29, 0x73)));
238     #define NIST_P384_h        ECC_ONE
239     #define NIST_P384_gZ      ECC_ONE
240     #if USE_BN_ECC_DATA
241         const ECC_CURVE_DATA NIST_P384 = {
242             (bigNum)&NIST_P384_p, (bigNum)&NIST_P384_n, (bigNum)&NIST_P384_h,
243             (bigNum)&NIST_P384_a, (bigNum)&NIST_P384_b,
244             {(bigNum)&NIST_P384_gX, (bigNum)&NIST_P384_gY, (bigNum)&NIST_P384_gZ}};
245     #else
246         const ECC_CURVE_DATA NIST_P384 = {
247             &NIST_P384_p.b, &NIST_P384_n.b, &NIST_P384_h.b,
248             &NIST_P384_a.b, &NIST_P384_b.b,
249             {&NIST_P384_gX.b, &NIST_P384_gY.b, &NIST_P384_gZ.b}};
250     #endif // USE_BN_ECC_DATA

```



```

251 #endif // ECC_NIST_P384
252 #if ECC_NIST_P521
253 ECC_CONST(NIST_P521_p, 66, TO_ECC_528(
254     TO_ECC_16(0x01, 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(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
258     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
259     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
260     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
261     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
262     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF));
263 ECC_CONST(NIST_P521_a, 66, TO_ECC_528(
264     TO_ECC_16(0x01, 0xFF),
265     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
266     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
267     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
268     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
269     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
270     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
271     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
272     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF));
273 ECC_CONST(NIST_P521_b, 66, TO_ECC_528(
274     TO_ECC_16(0x00, 0x51),
275     TO_ECC_64(0x95, 0x3E, 0xB9, 0x61, 0x8E, 0x1C, 0x9A, 0x1F),
276     TO_ECC_64(0x92, 0x9A, 0x21, 0xA0, 0xB6, 0x85, 0x40, 0xEE),
277     TO_ECC_64(0xA2, 0xDA, 0x72, 0x5B, 0x99, 0xB3, 0x15, 0xF3),
278     TO_ECC_64(0xB8, 0xB4, 0x89, 0x91, 0x8E, 0xF1, 0x09, 0xE1),
279     TO_ECC_64(0x56, 0x19, 0x39, 0x51, 0xEC, 0x7E, 0x93, 0x7B),
280     TO_ECC_64(0x16, 0x52, 0xC0, 0xBD, 0x3B, 0xB1, 0xBF, 0x07),
281     TO_ECC_64(0x35, 0x73, 0xDF, 0x88, 0x3D, 0x2C, 0x34, 0xF1),
282     TO_ECC_64(0xEF, 0x45, 0x1F, 0xD4, 0x6B, 0x50, 0x3F, 0x00));
283 ECC_CONST(NIST_P521_gX, 66, TO_ECC_528(
284     TO_ECC_16(0x00, 0xC6),
285     TO_ECC_64(0x85, 0x8E, 0x06, 0xB7, 0x04, 0x04, 0xE9, 0xCD),
286     TO_ECC_64(0x9E, 0x3E, 0xCB, 0x66, 0x23, 0x95, 0xB4, 0x42),
287     TO_ECC_64(0x9C, 0x64, 0x81, 0x39, 0x05, 0x3F, 0xB5, 0x21),
288     TO_ECC_64(0xF8, 0x28, 0xAF, 0x60, 0x6B, 0x4D, 0x3D, 0xBA),
289     TO_ECC_64(0xA1, 0x4B, 0x5E, 0x77, 0xEF, 0xE7, 0x59, 0x28),
290     TO_ECC_64(0xFE, 0x1D, 0xC1, 0x27, 0xA2, 0xFF, 0xA8, 0xDE),
291     TO_ECC_64(0x33, 0x48, 0xB3, 0xC1, 0x85, 0x6A, 0x42, 0x9B),
292     TO_ECC_64(0xF9, 0x7E, 0x7E, 0x31, 0xC2, 0xE5, 0xBD, 0x66));
293 ECC_CONST(NIST_P521_gY, 66, TO_ECC_528(
294     TO_ECC_16(0x01, 0x18),
295     TO_ECC_64(0x39, 0x29, 0x6A, 0x78, 0x9A, 0x3B, 0xC0, 0x04),
296     TO_ECC_64(0x5C, 0x8A, 0x5F, 0xB4, 0x2C, 0x7D, 0x1B, 0xD9),
297     TO_ECC_64(0x98, 0xF5, 0x44, 0x49, 0x57, 0x9B, 0x44, 0x68),
298     TO_ECC_64(0x17, 0xAF, 0xBD, 0x17, 0x27, 0x3E, 0x66, 0x2C),
299     TO_ECC_64(0x97, 0xEE, 0x72, 0x99, 0x5E, 0xF4, 0x26, 0x40),
300     TO_ECC_64(0xC5, 0x50, 0xB9, 0x01, 0x3F, 0xAD, 0x07, 0x61),
301     TO_ECC_64(0x35, 0x3C, 0x70, 0x86, 0xA2, 0x72, 0xC2, 0x40),
302     TO_ECC_64(0x88, 0xBE, 0x94, 0x76, 0x9F, 0xD1, 0x66, 0x50));
303 ECC_CONST(NIST_P521_n, 66, TO_ECC_528(
304     TO_ECC_16(0x01, 0xFF),
305     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
306     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
307     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
308     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
309     TO_ECC_64(0x51, 0x86, 0x87, 0x83, 0xBF, 0x2F, 0x96, 0x6B),
310     TO_ECC_64(0x7F, 0xCC, 0x01, 0x48, 0xF7, 0x09, 0xA5, 0xD0),
311     TO_ECC_64(0x3B, 0xB5, 0xC9, 0xB8, 0x89, 0x9C, 0x47, 0xAE),
312     TO_ECC_64(0xBB, 0x6F, 0xB7, 0x1E, 0x91, 0x38, 0x64, 0x09));
313 #define NIST_P521_h ECC_ONE
314 #define NIST_P521_gZ ECC_ONE
315 #if USE_BN_ECC_DATA
316     const ECC_CURVE_DATA NIST_P521 = {

```



```

317         (bigNum)&NIST_P521_p, (bigNum)&NIST_P521_n, (bigNum)&NIST_P521_h,
318         (bigNum)&NIST_P521_a, (bigNum)&NIST_P521_b,
319         {(bigNum)&NIST_P521_gX, (bigNum)&NIST_P521_gY, (bigNum)&NIST_P521_gZ});
320     #else
321         const ECC_CURVE_DATA NIST_P521 = {
322             &NIST_P521_p.b, &NIST_P521_n.b, &NIST_P521_h.b,
323             &NIST_P521_a.b, &NIST_P521_b.b,
324             {&NIST_P521_gX.b, &NIST_P521_gY.b, &NIST_P521_gZ.b}};
325     #endif // USE_BN_ECC_DATA
326     #endif // ECC_NIST_P521
327     #if ECC_BN_P256
328     ECC_CONST(BN_P256_p, 32, TO_ECC_256(
329         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
330         TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9F),
331         TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x98, 0x0A, 0x82),
332         TO_ECC_64(0xD3, 0x29, 0x2D, 0xDB, 0xAE, 0xD3, 0x30, 0x13)));
333     #define BN_P256_a          ECC_ZERO
334     ECC_CONST(BN_P256_b, 1, TO_ECC_8(3));
335     #define BN_P256_gX          ECC_ONE
336     ECC_CONST(BN_P256_gY, 1, TO_ECC_8(2));
337     ECC_CONST(BN_P256_n, 32, TO_ECC_256(
338         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC, 0xF0, 0xCD),
339         TO_ECC_64(0x46, 0xE5, 0xF2, 0x5E, 0xEE, 0x71, 0xA4, 0x9E),
340         TO_ECC_64(0x0C, 0xDC, 0x65, 0xFB, 0x12, 0x99, 0x92, 0x1A),
341         TO_ECC_64(0xF6, 0x2D, 0x53, 0x6C, 0xD1, 0x0B, 0x50, 0x0D)));
342     #define BN_P256_h          ECC_ONE
343     #define BN_P256_gZ          ECC_ONE
344     #if USE_BN_ECC_DATA
345         const ECC_CURVE_DATA BN_P256 = {
346             (bigNum)&BN_P256_p, (bigNum)&BN_P256_n, (bigNum)&BN_P256_h,
347             (bigNum)&BN_P256_a, (bigNum)&BN_P256_b,
348             {(bigNum)&BN_P256_gX, (bigNum)&BN_P256_gY, (bigNum)&BN_P256_gZ}};
349     #else
350         const ECC_CURVE_DATA BN_P256 = {
351             &BN_P256_p.b, &BN_P256_n.b, &BN_P256_h.b,
352             &BN_P256_a.b, &BN_P256_b.b,
353             {&BN_P256_gX.b, &BN_P256_gY.b, &BN_P256_gZ.b}};
354     #endif // USE_BN_ECC_DATA
355     #endif // ECC_BN_P256
356     #if ECC_BN_P638
357     ECC_CONST(BN_P638_p, 80, TO_ECC_640(
358         TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
359         TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
360         TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
361         TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
362         TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
363         TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
364         TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
365         TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
366         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
367         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x67)));
368     #define BN_P638_a          ECC_ZERO
369     ECC_CONST(BN_P638_b, 2, TO_ECC_16(0x01, 0x01));
370     ECC_CONST(BN_P638_gX, 80, TO_ECC_640(
371         TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
372         TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
373         TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
374         TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
375         TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
376         TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0xE5, 0x5B),
377         TO_ECC_64(0xFF, 0xFF, 0xF5, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
378         TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
379         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xEC, 0xE0),
380         TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x66)));
381     ECC_CONST(BN_P638_gY, 1, TO_ECC_8(0x10));
382     ECC_CONST(BN_P638_n, 80, TO_ECC_640(

```

```

383     TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
384     TO_ECC_64(0x7F, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
385     TO_ECC_64(0xFF, 0xFF, 0xF9, 0x42, 0xD0, 0x00, 0x16, 0x5E),
386     TO_ECC_64(0x3F, 0xFF, 0x94, 0x87, 0x00, 0x00, 0xD5, 0x2F),
387     TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
388     TO_ECC_64(0x60, 0x00, 0x86, 0x55, 0x00, 0x21, 0xE5, 0x55),
389     TO_ECC_64(0xFF, 0xFF, 0xF5, 0x4F, 0xFF, 0xF4, 0xEA, 0xC0),
390     TO_ECC_64(0x00, 0x00, 0x00, 0x49, 0x80, 0x01, 0x54, 0xD9),
391     TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xED, 0xA0),
392     TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x61));
393 #define BN_P638_h      ECC_ONE
394 #define BN_P638_gZ     ECC_ONE
395 #if USE_BN_ECC_DATA
396     const ECC_CURVE_DATA BN_P638 = {
397         (bigNum)&BN_P638_p, (bigNum)&BN_P638_n, (bigNum)&BN_P638_h,
398         (bigNum)&BN_P638_a, (bigNum)&BN_P638_b,
399         {(bigNum)&BN_P638_gX, (bigNum)&BN_P638_gY, (bigNum)&BN_P638_gZ}};
400 #else
401     const ECC_CURVE_DATA BN_P638 = {
402         &BN_P638_p.b, &BN_P638_n.b, &BN_P638_h.b,
403         &BN_P638_a.b, &BN_P638_b.b,
404         {&BN_P638_gX.b, &BN_P638_gY.b, &BN_P638_gZ.b}};
405 #endif // USE_BN_ECC_DATA
406 #endif // ECC_BN_P638
407 #if ECC_SM2_P256
408     ECC_CONST(SM2_P256_p, 32, TO_ECC_256(
409         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
410         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
411         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
412         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
413     ECC_CONST(SM2_P256_a, 32, TO_ECC_256(
414         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
415         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
416         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
417         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFC)));
418     ECC_CONST(SM2_P256_b, 32, TO_ECC_256(
419         TO_ECC_64(0x28, 0xE9, 0xFA, 0x9E, 0x9D, 0x9F, 0x5E, 0x34),
420         TO_ECC_64(0x4D, 0x5A, 0x9E, 0x4B, 0xCF, 0x65, 0x09, 0xA7),
421         TO_ECC_64(0xF3, 0x97, 0x89, 0xF5, 0x15, 0xAB, 0x8F, 0x92),
422         TO_ECC_64(0xDD, 0xBC, 0xBD, 0x41, 0x4D, 0x94, 0x0E, 0x93)));
423     ECC_CONST(SM2_P256_gX, 32, TO_ECC_256(
424         TO_ECC_64(0x32, 0xC4, 0xAE, 0x2C, 0x1F, 0x19, 0x81, 0x19),
425         TO_ECC_64(0x5F, 0x99, 0x04, 0x46, 0x6A, 0x39, 0xC9, 0x94),
426         TO_ECC_64(0x8F, 0xE3, 0x0B, 0xBF, 0xF2, 0x66, 0x0B, 0xE1),
427         TO_ECC_64(0x71, 0x5A, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7)));
428     ECC_CONST(SM2_P256_gY, 32, TO_ECC_256(
429         TO_ECC_64(0xBC, 0x37, 0x36, 0xA2, 0xF4, 0xF6, 0x77, 0x9C),
430         TO_ECC_64(0x59, 0xBD, 0xCE, 0xE3, 0x6B, 0x69, 0x21, 0x53),
431         TO_ECC_64(0xD0, 0xA9, 0x87, 0x7C, 0xC6, 0x2A, 0x47, 0x40),
432         TO_ECC_64(0x02, 0xDF, 0x32, 0xE5, 0x21, 0x39, 0xF0, 0xA0)));
433     ECC_CONST(SM2_P256_n, 32, TO_ECC_256(
434         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
435         TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
436         TO_ECC_64(0x72, 0x03, 0xDF, 0x6B, 0x21, 0xC6, 0x05, 0x2B),
437         TO_ECC_64(0x53, 0xBB, 0xF4, 0x09, 0x39, 0xD5, 0x41, 0x23)));
438 #define SM2_P256_h      ECC_ONE
439 #define SM2_P256_gZ     ECC_ONE
440 #if USE_BN_ECC_DATA
441     const ECC_CURVE_DATA SM2_P256 = {
442         (bigNum)&SM2_P256_p, (bigNum)&SM2_P256_n, (bigNum)&SM2_P256_h,
443         (bigNum)&SM2_P256_a, (bigNum)&SM2_P256_b,
444         {(bigNum)&SM2_P256_gX, (bigNum)&SM2_P256_gY, (bigNum)&SM2_P256_gZ}};
445 #else
446     const ECC_CURVE_DATA SM2_P256 = {
447         &SM2_P256_p.b, &SM2_P256_n.b, &SM2_P256_h.b,
448         &SM2_P256_a.b, &SM2_P256_b.b,

```

```

449         {&SM2_P256_gX.b, &SM2_P256_gY.b, &SM2_P256_gZ.b});
450 #endif // USE_BN_ECC_DATA
451 #endif // ECC_SM2_P256
452 #define comma
453 const ECC_CURVE eccCurves[] = {
454 #if ECC_NIST_P192
455     comma
456     {TPM_ECC_NIST_P192,
457      192,
458      {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
459      {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
460      &NIST_P192,
461      OID_ECC_NIST_P192
462      CURVE_NAME("NIST_P192")}
463 # undef comma
464 # define comma ,
465 #endif // ECC_NIST_P192
466 #if ECC_NIST_P224
467     comma
468     {TPM_ECC_NIST_P224,
469      224,
470      {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
471      {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
472      &NIST_P224,
473      OID_ECC_NIST_P224
474      CURVE_NAME("NIST_P224")}
475 # undef comma
476 # define comma ,
477 #endif // ECC_NIST_P224
478 #if ECC_NIST_P256
479     comma
480     {TPM_ECC_NIST_P256,
481      256,
482      {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
483      {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
484      &NIST_P256,
485      OID_ECC_NIST_P256
486      CURVE_NAME("NIST_P256")}
487 # undef comma
488 # define comma ,
489 #endif // ECC_NIST_P256
490 #if ECC_NIST_P384
491     comma
492     {TPM_ECC_NIST_P384,
493      384,
494      {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA384_VALUE}}},
495      {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
496      &NIST_P384,
497      OID_ECC_NIST_P384
498      CURVE_NAME("NIST_P384")}
499 # undef comma
500 # define comma ,
501 #endif // ECC_NIST_P384
502 #if ECC_NIST_P521
503     comma
504     {TPM_ECC_NIST_P521,
505      521,
506      {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA512_VALUE}}},
507      {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
508      &NIST_P521,
509      OID_ECC_NIST_P521
510      CURVE_NAME("NIST_P521")}
511 # undef comma
512 # define comma ,
513 #endif // ECC_NIST_P521
514 #if ECC_BN_P256

```

```

515     comma
516     {TPM_ECC_BN_P256,
517     256,
518     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
519     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
520     &BN_P256,
521     OID_ECC_BN_P256
522     CURVE_NAME("BN_P256")}
523 #   undef comma
524 #   define comma ,
525 #endif // ECC_BN_P256
526 #if ECC_BN_P638
527     comma
528     {TPM_ECC_BN_P638,
529     638,
530     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
531     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
532     &BN_P638,
533     OID_ECC_BN_P638
534     CURVE_NAME("BN_P638")}
535 #   undef comma
536 #   define comma ,
537 #endif // ECC_BN_P638
538 #if ECC_SM2_P256
539     comma
540     {TPM_ECC_SM2_P256,
541     256,
542     {ALG_KDF1_SP800_56A_VALUE, {{ALG_SM3_256_VALUE}}},
543     {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
544     &SM2_P256,
545     OID_ECC_SM2_P256
546     CURVE_NAME("SM2_P256")}
547 #   undef comma
548 #   define comma ,
549 #endif // ECC_SM2_P256
550 };
551 #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  #if ALG_TDES
3  #define DES_NUM_WEAK 64
4  const UINT64 DesWeakKeys[DES_NUM_WEAK] = {
5      0x0101010101010101ULL, 0xFEFEFEFEFEFEFEFEULL,
6      0xE0E0E0E0F1F1F1F1ULL, 0x1F1F1F1F0E0E0E0EULL,
7      0x011F011F010E010EULL, 0x1F011F010E010E01ULL,
8      0x01E001E001F101F1ULL, 0xE001E001F101F101ULL,
9      0x01FE01FE01FE01FEULL, 0xFE01FE01FE01FE01ULL,
10     0x1FE01FE00EF10EF1ULL, 0xE01FE01FF10EF10EULL,
11     0x1FFE1FFE0EFE0EFEULL, 0xFE1FFE1FFE0EFE0EULL,
12     0xE0FEE0FEF1FEF1FEULL, 0xFEE0FEE0FEF1FEF1ULL,
13     0x01011F1F01010E0EULL, 0x1F1F01010E0E0101ULL,
14     0xE0E01F1FF1F10E0EULL, 0x0101E0E00101F1F1ULL,
15     0x1F1FE0E00E0EF1F1ULL, 0xE0E0FEFEF1F1FEFEULL,
16     0x0101FEFE0101FEFEULL, 0x1F1FFEFE0E0EFEFEULL,
17     0xE0FE011FF1FE010EULL, 0x011F1F01010E0E01ULL,
18     0x1FE001FE0EF101FEULL, 0xE0FE1F01F1FE0E01ULL,
19     0x011FE0FE010EF1FEULL, 0x1FE0E01F0EF1F10EULL,
20     0xE0FEFEE0F1FEFEF1ULL, 0x011FFEE0010EFEF1ULL,
21     0x1FE0FE010EF1FE01ULL, 0xFE0101FEFE0101FEULL,
22     0x01E01FFE01F10EFEULL, 0x1FFE01E00EFE01F1ULL,
23     0xFE011FE0FE010EF1ULL, 0xFE01E01FFE01F10EULL,
24     0x1FFEE0010EFEF101ULL, 0xFE1F01E0FE0E01F1ULL,
25     0x01E0E00101F1F101ULL, 0x1FFFEF1F0EFEF0EULL,
26     0xFE1FE001FE0EF101ULL, 0x01E0FE1F01F1FE0EULL,
27     0xE00101E0F10101F1ULL, 0xFE1F1FFEFE0E0EFEULL,
28     0x01FE1FE001FE0EF1ULL, 0xE0011FFE1F1010EFEULL,
29     0xFEE0011FFE1F1010EULL, 0x01FEE01F01FEF10EULL,
30     0xE001FE1FF101FE0EULL, 0xFEE01F01FEF10E01ULL,
31     0x01FEFE0101FEFE01ULL, 0xE01F01FEF10E01FEULL,
32     0xFEE0E0FEFEF1F1FEULL, 0x1F01011F0E01010EULL,
33     0xE01F1FE0F10E0EF1ULL, 0xFEFE0101FEFE0101ULL,
34     0x1F01E0FE0E01F1FEULL, 0xE01FFE01F10EFE01ULL,
35     0xFEFE1F1FFEFE0E0EULL, 0x1F01FEE00E01FEF1ULL,
36     0xE0E00101F1F10101ULL, 0xFEFE0E0FEFEF1F1ULL};

```

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.

```

37  UINT64
38  CryptSetOddByteParity(
39      UINT64      k
40  )
41  {
42      #define PMASK 0x0101010101010101ULL
43      UINT64      out;
44      k |= PMASK;    // set the parity bit
45      out = k;
46      k ^= k >> 4;
47      k ^= k >> 2;

```

```

48     k ^= k >> 1;
49     k &= PMASK;      // odd parity extracted
50     out ^= k;        // out is now even parity because parity bit was already set
51     out ^= PMASK;    // out is now even parity
52     return out;
53 }

```

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

```

54 static BOOL
55 CryptDesIsWeakKey(
56     UINT64      k
57 )
58 {
59     int          i;
60 //
61     for(i = 0; i < DES_NUM_WEAK; i++)
62     {
63         if(k == DesWeakKeys[i])
64             return TRUE;
65     }
66     return FALSE;
67 }

```

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.

```

68 BOOL
69 CryptDesValidateKey(
70     TPM2B_SYM_KEY *desKey    // IN: key to validate
71 )
72 {
73     UINT64      k[3];
74     int          i;
75     int          keys = (desKey->t.size + 7) / 8;
76     BYTE         *pk = desKey->t.buffer;
77     BOOL         ok;
78 //
79     // Note: 'keys' is the number of keys, not the maximum index for 'k'
80     ok = ((keys == 2) || (keys == 3)) && ((desKey->t.size % 8) == 0);
81     for(i = 0; ok && i < keys; pk += 8, i++)
82     {
83         k[i] = CryptSetOddByteParity(BYTE_ARRAY_TO_UINT64(pk));
84         ok = !CryptDesIsWeakKey(k[i]);
85     }
86     ok = ok && k[0] != k[1];
87     if(keys == 3)
88         ok = ok && k[1] != k[2];
89     return ok;
90 }

```

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.

```

91  TPM_RC
92  CryptGenerateKeyDes (
93      TPMT_PUBLIC          *publicArea,          // IN/OUT: The public area template
94                          // for the new key.
95      TPMT_SENSITIVE       *sensitive,          // OUT: sensitive area
96      RAND_STATE           *rand                // IN: the "entropy" source for
97  )
98  {
99
100     // Assume that the publicArea key size has been validated and is a supported
101     // number of bits.
102     sensitive->sensitive.sym.t.size =
103         BITS_TO_BYTES(publicArea->parameters.symDetail.sym.keyBits.sym);
104     do
105     {
106         BYTE                *pK = sensitive->sensitive.sym.t.buffer;
107         int                 i = (sensitive->sensitive.sym.t.size + 7) / 8;
108     // Use the random number generator to generate the required number of bits
109         if(DRBG_Generate(rand, pK, sensitive->sensitive.sym.t.size) == 0)
110             return TPM_RC_NO_RESULT;
111         for(; i > 0; pK += 8, i--)
112         {
113             UINT64          k = BYTE_ARRAY_TO_UINT64(pK);
114             k = CryptSetOddByteParity(k);
115             UINT64_TO_BYTE_ARRAY(k, pK);
116         }
117     } while(!CryptDesValidateKey(&sensitive->sensitive.sym));
118     return TPM_RC_SUCCESS;
119 }
120 #endif

```


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  #if CC_ZGen_2Phase == YES
```

10.2.10.2 Functions

```
3  #if ALG_ECMQV
```

10.2.10.2.1 avf1()

This function does the associated value computation required by MQV key exchange. Process:

- Convert xQ to an integer xqi using the convention specified in Appendix C.3.
- Calculate $xqm = xqi \bmod 2^{\lceil f/2 \rceil}$ (where $f = \lceil \log_2(n) \rceil$).
- Calculate the associate value function $avf(Q) = xqm + 2^{\lceil f/2 \rceil}$. Always returns TRUE(1).

```
4  static BOOL
5  avf1(
6      bigNum          bnX,          // IN/OUT: the reduced value
7      bigNum          bnN          // IN: the order of the curve
8  )
9  {
10     // compute f = 2^(ceil(ceil(log2(n)) / 2))
11     int              f = (BnSizeInBits(bnN) + 1) / 2;
12     // x' = 2^f + (x mod 2^f)
13     BnMaskBits(bnX, f); // This is mod 2*2^f but it doesn't matter because
14                          // the next operation will SET the extra bit anyway
15     BnSetBit(bnX, f);
16     return TRUE;
17 }
```

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 Returns	Meaning
TPM_RC_NO_RESULT	the value for dsA does not give a valid point on the curve

```
18 static TPM_RC
19 C_2_2_MQV(
20     TPMS_ECC_POINT      *outZ,          // OUT: the computed point
21     TPM_ECC_CURVE       curveId,       // IN: the curve for the computations
22     TPM2B_ECC_PARAMETER *dsA,          // IN: static private TPM key
23     TPM2B_ECC_PARAMETER *deA,          // IN: ephemeral private TPM key
24     TPMS_ECC_POINT      *QsB,          // IN: static public party B key
25     TPMS_ECC_POINT      *QeB          // IN: ephemeral public party B key
```

```

26     )
27 {
28     CURVE_INITIALIZED(E, curveId);
29     const ECC_CURVE_DATA *C;
30     POINT(pQeA);
31     POINT_INITIALIZED(pQeB, QeB);
32     POINT_INITIALIZED(pQsB, QsB);
33     ECC_NUM(bnTa);
34     ECC_INITIALIZED(bnDeA, deA);
35     ECC_INITIALIZED(bnDsA, dsA);
36     ECC_NUM(bnN);
37     ECC_NUM(bnXeB);
38     TPM_RC retVal;
39 //
40 // Parameter checks
41 if(E == NULL)
42     ERROR_RETURN(TPM_RC_VALUE);
43 pAssert(outZ != NULL && pQeB != NULL && pQsB != NULL && deA != NULL
44         && dsA != NULL);
45 C = AccessCurveData(E);
46 // Process:
47 // 1. implicitSigA = (de,A + avf(Qe,A)ds,A ) mod n.
48 // 2. P = h(implicitSigA)(Qe,B + avf(Qe,B)Qs,B).
49 // 3. If P = O, output an error indicator.
50 // 4. Z=xP, where xP is the x-coordinate of P.
51
52 // Compute the public ephemeral key pQeA = [de,A]G
53 if((retVal = BnPointMult(pQeA, CurveGetG(C), bnDeA, NULL, NULL, E))
54     != TPM_RC_SUCCESS)
55     goto Exit;
56
57 // 1. implicitSigA = (de,A + avf(Qe,A)ds,A ) mod n.
58 // tA := (ds,A + de,A avf(Xe,A)) mod n (3)
59 // Compute 'tA' = ('deA' + 'dsA' avf('XeA')) mod n
60 // Ta = avf(XeA);
61 BnCopy(bnTa, pQeA->x);
62 avf1(bnTa, bnN);
63 // do Ta = ds,A * Ta mod n = dsA * avf(XeA) mod n
64 BnModMult(bnTa, bnDsA, bnTa, bnN);
65 // now Ta = deA + Ta mod n = deA + dsA * avf(XeA) mod n
66 BnAdd(bnTa, bnTa, bnDeA);
67 BnMod(bnTa, bnN);
68
69 // 2. P = h(implicitSigA)(Qe,B + avf(Qe,B)Qs,B).
70 // Put this in because almost every case of h is == 1 so skip the call when
71 // not necessary.
72 if(!BnEqualWord(CurveGetCofactor(C), 1))
73     // Cofactor is not 1 so compute Ta := Ta * h mod n
74     BnModMult(bnTa, bnTa, CurveGetCofactor(C), CurveGetOrder(C));
75
76 // Now that 'tA' is (h * 'tA' mod n)
77 // 'outZ' = (tA)(Qe,B + avf(Qe,B)Qs,B).
78
79 // first, compute XeB = avf(XeB)
80 avf1(bnXeB, bnN);
81
82 // QsB := [XeB]QsB
83 BnPointMult(pQsB, pQsB, bnXeB, NULL, NULL, E);
84 BnEccAdd(pQeB, pQeB, pQsB, E);
85
86 // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
87 // If the result is not the point at infinity, return QeB
88 BnPointMult(pQeB, pQeB, bnTa, NULL, NULL, E);
89 if(BnEqualZero(pQeB->z))
90     ERROR_RETURN(TPM_RC_NO_RESULT);
91 // Convert BIGNUM E to TPM2B E

```

```

92     BnPointTo2B(outZ, pQeB, E);
93
94 Exit:
95     CURVE_FREE(E);
96     return retVal;
97 }
98 #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).

```

99 static TPM_RC
100 C_2_2_ECDH(
101     TPMS_ECC_POINT      *outZs,           // OUT: Zs
102     TPMS_ECC_POINT      *outZe,           // OUT: Ze
103     TPM_ECC_CURVE        curveId,          // IN: the curve for the computations
104     TPM2B_ECC_PARAMETER  *dsA,            // IN: static private TPM key
105     TPM2B_ECC_PARAMETER  *deA,            // IN: ephemeral private TPM key
106     TPMS_ECC_POINT      *QsB,             // IN: static public party B key
107     TPMS_ECC_POINT      *QeB,             // IN: ephemeral public party B key
108 )
109 {
110     CURVE_INITIALIZED(E, curveId);
111     ECC_INITIALIZED(bnAs, dsA);
112     ECC_INITIALIZED(bnAe, deA);
113     POINT_INITIALIZED(ecBs, QsB);
114     POINT_INITIALIZED(ecBe, QeB);
115     POINT(ecZ);
116     TPM_RC      retVal;
117 //
118 // Parameter checks
119 if(E == NULL)
120     ERROR_RETURN(TPM_RC_CURVE);
121 pAssert(outZs != NULL && dsA != NULL && deA != NULL && QsB != NULL
122     && QeB != NULL);
123
124 // Do the point multiply for the Zs value ([dsA]QsB)
125 retVal = BnPointMult(ecZ, ecBs, bnAs, NULL, NULL, E);
126 if(retVal == TPM_RC_SUCCESS)
127 {
128     // Convert the Zs value.
129     BnPointTo2B(outZs, ecZ, E);
130     // Do the point multiply for the Ze value ([deA]QeB)
131     retVal = BnPointMult(ecZ, ecBe, bnAe, NULL, NULL, E);
132     if(retVal == TPM_RC_SUCCESS)
133         BnPointTo2B(outZe, ecZ, E);
134 }
135 Exit:
136     CURVE_FREE(E);
137     return retVal;
138 }

```

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 Returns	Meaning
TPM_RC_SCHEME	scheme is not defined

```

139 LIB_EXPORT TPM_RC
140 CryptEcc2PhaseKeyExchange(
141     TPMS_ECC_POINT *outZ1,           // OUT: a computed point
142     TPMS_ECC_POINT *outZ2,           // OUT: and optional second point
143     TPM_ECC_CURVE  curveId,          // IN: the curve for the computations
144     TPM_ALG_ID      scheme,           // IN: the key exchange scheme
145     TPM2B_ECC_PARAMETER *dsA,        // IN: static private TPM key
146     TPM2B_ECC_PARAMETER *deA,        // IN: ephemeral private TPM key
147     TPMS_ECC_POINT *QsB,             // IN: static public party B key
148     TPMS_ECC_POINT *QeB,             // IN: ephemeral public party B key
149 )
150 {
151     pAssert(outZ1 != NULL
152             && dsA != NULL && deA != NULL
153             && QsB != NULL && QeB != NULL);
154
155     // Initialize the output points so that they are empty until one of the
156     // functions decides otherwise
157     outZ1->x.b.size = 0;
158     outZ1->y.b.size = 0;
159     if(outZ2 != NULL)
160     {
161         outZ2->x.b.size = 0;
162         outZ2->y.b.size = 0;
163     }
164     switch(scheme)
165     {
166         case ALG_ECDH_VALUE:
167             return C_2_2_ECDH(outZ1, outZ2, curveId, dsA, deA, QsB, QeB);
168             break;
169     #if ALG_ECMQV
170         case ALG_ECMQV_VALUE:
171             return C_2_2_MQV(outZ1, curveId, dsA, deA, QsB, QeB);
172             break;
173     #endif
174     #if ALG_SM2
175         case ALG_SM2_VALUE:
176             return SM2KeyExchange(outZ1, curveId, dsA, deA, QsB, QeB);
177             break;
178     #endif
179         default:
180             return TPM_RC_SCHEME;
181     }
182 }
183 #if ALG_SM2

```

10.2.10.2.5 ComputeWForSM2()

Compute the value for w used by SM2

```

184 static UINT32
185 ComputeWForSM2(
186     bigCurve      E
187 )
188 {
189     // w := ceil(ceil(log2(n)) / 2) - 1
190     return (BnMsb(CurveGetOrder(AccessCurveData(E))) / 2 - 1);
191 }

```

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.

```

192 static bigNum
193 avfSm2(
194     bigNum          bn,          // IN/OUT: the reduced value
195     UINT32          w           // IN: the value of w
196 )
197 {
198     // a) set w := ceil(ceil(log2(n)) / 2) - 1
199     // b) set x' := 2^w + (x & (2^w - 1))
200     // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
201
202     BnMaskBits(bn, w); // as with avf1, this is too big by a factor of 2 but
203                       // it doesn't matter because we SET the extra bit
204                       // anyway
205     BnSetBit(bn, w);
206     return bn;
207 }

```

10.2.10.2.7 SM2KeyExchange()

This function performs the key exchange defined in SM2. The first step is to compute $tA = (dsA + deA \text{ avf}(XeA)) \bmod n$. 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 Returns	Meaning
TPM_RC_NO_RESULT	the value for dsA does not give a valid point on the curve

```

208 LIB_EXPORT TPM_RC
209 SM2KeyExchange(
210     TPMS_ECC_POINT    *outZ,          // OUT: the computed point
211     TPM_ECC_CURVE      curveId,       // IN: the curve for the computations
212     TPM2B_ECC_PARAMETER *dsAIn,       // IN: static private TPM key
213     TPM2B_ECC_PARAMETER *deAIn,       // IN: ephemeral private TPM key
214     TPMS_ECC_POINT    *QsBIn,        // IN: static public party B key
215     TPMS_ECC_POINT    *QeBIn,        // IN: ephemeral public party B key
216 )
217 {
218     CURVE_INITIALIZED(E, curveId);
219     const ECC_CURVE_DATA *C;
220     ECC_INITIALIZED(dsA, dsAIn);
221     ECC_INITIALIZED(deA, deAIn);
222     POINT_INITIALIZED(QsB, QsBIn);
223     POINT_INITIALIZED(QeB, QeBIn);
224     BN_WORD_INITIALIZED(One, 1);
225     POINT(QeA);
226     ECC_NUM(XeB);
227     POINT(Z);
228     ECC_NUM(Ta);
229     UINT32 w;
230     TPM_RC retVal = TPM_RC_NO_RESULT;
231     //
232     // Parameter checks
233     if(E == NULL)

```

```

234     ERROR_RETURN(TPM_RC_CURVE);
235     C = AccessCurveData(E);
236     pAssert(outZ != NULL && dsA != NULL && deA != NULL && QsB != NULL
237           && QeB != NULL);
238
239     // Compute the value for w
240     w = ComputeWForSM2(E);
241
242     // Compute the public ephemeral key pQeA = [de,A]G
243     if(!BnEccModMult(QeA, CurveGetG(C), deA, E))
244         goto Exit;
245
246     // tA := (ds,A + de,A avf(Xe,A)) mod n      (3)
247     // Compute 'tA' = ('dsA' + 'deA' avf('XeA')) mod n
248     // Ta = avf(XeA);
249     // do Ta = de,A * Ta = deA * avf(XeA)
250     BnMult(Ta, deA, avfSm2(QeA->x, w));
251     // now Ta = dsA + Ta = dsA + deA * avf(XeA)
252     BnAdd(Ta, dsA, Ta);
253     BnMod(Ta, CurveGetOrder(C));
254
255     // outZ = [h tA mod n] (Qs,B + [avf(Xe,B)](Qe,B)) (4)
256     // Put this in because almost every case of h is == 1 so skip the call when
257     // not necessary.
258     if(!BnEqualWord(CurveGetCofactor(C), 1))
259         // Cofactor is not 1 so compute Ta := Ta * h mod n
260         BnModMult(Ta, Ta, CurveGetCofactor(C), CurveGetOrder(C));
261     // Now that 'tA' is (h * 'tA' mod n)
262     // 'outZ' = ['tA'] (QsB + [avf(QeB.x)](QeB)).
263     BnCopy(XeB, QeB->x);
264     if(!BnEccModMult2(Z, QsB, One, QeB, avfSm2(XeB, w), E))
265         goto Exit;
266     // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
267     if(!BnEccModMult(Z, Z, Ta, E))
268         goto Exit;
269     // Convert BIGNUM E to TPM2B E
270     BnPointTo2B(outZ, Z, E);
271     retVal = TPM_RC_SUCCESS;
272 Exit:
273     CURVE_FREE(E);
274     return retVal;
275 }
276 #endif
277 #endif // CC_ZGen_2Phase

```

10.2.11 CryptEccMain.c

10.2.11.1 Includes and Defines

```
1  #include "Tpm.h"
2  #if      ALG_ECC
```

This version requires that the new format for ECC data be used

```
3  #if !USE_BN_ECC_DATA
4  #error "Need to SET USE_BN_ECC_DATA to YES in Implementaion.h"
5  #endif
```

10.2.11.2 Functions

```
6  #if SIMULATION
7  void
8  EccSimulationEnd(
9      void
10 )
11 {
12     #if SIMULATION
13     // put things to be printed at the end of the simulation here
14     #endif
15 }
16 #endif // SIMULATION
```

10.2.11.2.1 CryptEccInit()

This function is called at _TPM_Init()

```
17  BOOL
18  CryptEccInit(
19      void
20  )
21  {
22      return TRUE;
23  }
```

10.2.11.2.2 CryptEccStartup()

This function is called at TPM2_Startup().

```
24  BOOL
25  CryptEccStartup(
26      void
27  )
28  {
29      return TRUE;
30  }
```

10.2.11.2.3 ClearPoint2B(generic

Initialize the size values of a TPMS_ECC_POINT structure.

```
31  void
32  ClearPoint2B(
33      TPMS_ECC_POINT    *p          // IN: the point
```



```

34     )
35 {
36     if(p != NULL)
37     {
38         p->x.t.size = 0;
39         p->y.t.size = 0;
40     }
41 }

```

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

```

42 LIB_EXPORT const ECC_CURVE *
43 CryptEccGetParametersByCurveId(
44     TPM_ECC_CURVE    curveId    // IN: the curveID
45 )
46 {
47     int    i;
48     for(i = 0; i < ECC_CURVE_COUNT; i++)
49     {
50         if(eccCurves[i].curveId == curveId)
51             return &eccCurves[i];
52     }
53     return NULL;
54 }

```

10.2.11.2.5 CryptEccGetKeySizeForCurve()

This function returns the key size in bits of the indicated curve.

```

55 LIB_EXPORT UINT16
56 CryptEccGetKeySizeForCurve(
57     TPM_ECC_CURVE    curveId    // IN: the curve
58 )
59 {
60     const ECC_CURVE *curve = CryptEccGetParametersByCurveId(curveId);
61     UINT16    keySizeInBits;
62     //
63     keySizeInBits = (curve != NULL) ? curve->keySizeBits : 0;
64     return keySizeInBits;
65 }

```

10.2.11.2.6 GetCurveData()

This function returns the a pointer for the parameter data associated with a curve.

```

66 const ECC_CURVE_DATA *
67 GetCurveData(
68     TPM_ECC_CURVE    curveId    // IN: the curveID
69 )
70 {
71     const ECC_CURVE    *curve = CryptEccGetParametersByCurveId(curveId);

```

```

72     return (curve != NULL) ? curve->curveData : NULL;
73 }

```

10.2.11.2.7 CryptEccGetOID()

```

74 const BYTE *
75 CryptEccGetOID(
76     TPM_ECC_CURVE      curveId
77 )
78 {
79     const ECC_CURVE      *curve = CryptEccGetParametersByCurveId(curveId);
80     return (curve != NULL) ? curve->OID : NULL;
81 }

```

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.

```

82 LIB_EXPORT TPM_ECC_CURVE
83 CryptEccGetCurveByIndex(
84     UINT16      i
85 )
86 {
87     if(i >= ECC_CURVE_COUNT)
88         return TPM_ECC_NONE;
89     return eccCurves[i].curveId;
90 }

```

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

```

91 LIB_EXPORT BOOL
92 CryptEccGetParameter(
93     TPM2B_ECC_PARAMETER *out,      // OUT: place to put parameter
94     char      p,                  // IN: the parameter selector
95     TPM_ECC_CURVE      curveId     // IN: the curve id
96 )
97 {
98     const ECC_CURVE_DATA *curve = GetCurveData(curveId);
99     bigConst      parameter = NULL;
100
101     if(curve != NULL)
102     {
103         switch(p)
104         {
105             case 'p':
106                 parameter = CurveGetPrime(curve);
107                 break;
108             case 'n':
109                 parameter = CurveGetOrder(curve);
110                 break;

```

```

111         case 'a':
112             parameter = CurveGet_a(curve);
113             break;
114         case 'b':
115             parameter = CurveGet_b(curve);
116             break;
117         case 'x':
118             parameter = CurveGetGx(curve);
119             break;
120         case 'y':
121             parameter = CurveGetGy(curve);
122             break;
123         case 'h':
124             parameter = CurveGetCofactor(curve);
125             break;
126         default:
127             FAIL(FATAL_ERROR_INTERNAL);
128             break;
129     }
130 }
131 // If not debugging and we get here with parameter still NULL, had better
132 // not try to convert so just return FALSE instead.
133 return (parameter != NULL) ? BnTo2B(parameter, &out->b, 0) : 0;
134 }

```

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

```

135 TPMI_YES_NO
136 CryptCapGetECCCurve(
137     TPM_ECC_CURVE    curveID,           // IN: the starting ECC curve
138     UINT32            maxCount,         // IN: count of returned curves
139     TPML_ECC_CURVE    *curveList        // OUT: ECC curve list
140 )
141 {
142     TPMI_YES_NO        more = NO;
143     UINT16              i;
144     UINT32              count = ECC_CURVE_COUNT;
145     TPM_ECC_CURVE       curve;
146
147     // Initialize output property list
148     curveList->count = 0;
149
150     // The maximum count of curves we may return is MAX_ECC_CURVES
151     if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;
152
153     // Scan the eccCurveValues array
154     for(i = 0; i < count; i++)
155     {
156         curve = CryptEccGetCurveByIndex(i);
157         // If curveID is less than the starting curveID, skip it
158         if(curve < curveID)
159             continue;
160         if(curveList->count < maxCount)
161         {
162             // If we have not filled up the return list, add more curves to
163             // it

```

```

164         curveList->eccCurves[curveList->count] = curve;
165         curveList->count++;
166     }
167     else
168     {
169         // If the return list is full but we still have curves
170         // available, report this and stop iterating
171         more = YES;
172         break;
173     }
174 }
175 return more;
176 }

```

10.2.11.2.11 CryptGetCurveSignScheme()

This function will return a pointer to the scheme of the curve.

```

177 const TPMT_ECC_SCHEME *
178 CryptGetCurveSignScheme(
179     TPM_ECC_CURVE    curveId        // IN: The curve selector
180 )
181 {
182     const ECC_CURVE    *curve = CryptEccGetParametersByCurveId(curveId);
183
184     if(curve != NULL)
185         return &(curve->sign);
186     else
187         return NULL;
188 }

```

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)	<i>r</i> value computed
FALSE(0)	no <i>r</i> value computed

```

189 BOOL
190 CryptGenerateR(
191     TPM2B_ECC_PARAMETER    *r,                // OUT: the generated random value
192     UINT16                  *c,                // IN/OUT: count value.
193     TPMI_ECC_CURVE          curveID,           // IN: the curve for the value
194     TPM2B_NAME               *name             // IN: optional name of a key to
195                                           // associate with 'r'
196 )
197 {
198     // This holds the marshaled g_commitCounter.
199     TPM2B_TYPE(8B, 8);
200     TPM2B_8B                cntr = {{8,{0}}};
201     UINT32                   iterations;
202     TPM2B_ECC_PARAMETER      n;
203     UINT64                   currentCount = gr.commitCounter;
204     UINT16                   t1;
205     //
206     if(!CryptEccGetParameter(&n, 'n', curveID))

```

```

207         return FALSE;
208
209     // If this is the commit phase, use the current value of the commit counter
210     if(c != NULL)
211     {
212         // if the array bit is not set, can't use the value.
213         if(!TEST_BIT((*c & COMMIT_INDEX_MASK), gr.commitArray))
214             return FALSE;
215
216         // If it is the sign phase, figure out what the counter value was
217         // when the commitment was made.
218         //
219         // When gr.commitArray has less than 64K bits, the extra
220         // bits of 'c' are used as a check to make sure that the
221         // signing operation is not using an out of range count value
222         t1 = (UINT16)currentCount;
223
224         // If the lower bits of c are greater or equal to the lower bits of t1
225         // then the upper bits of t1 must be one more than the upper bits
226         // of c
227         if((*c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
228             // Since the counter is behind, reduce the current count
229             currentCount = currentCount - (COMMIT_INDEX_MASK + 1);
230
231         t1 = (UINT16)currentCount;
232         if((t1 & ~COMMIT_INDEX_MASK) != (*c & ~COMMIT_INDEX_MASK))
233             return FALSE;
234         // set the counter to the value that was
235         // present when the commitment was made
236         currentCount = (currentCount & 0xffffffff0000) | *c;
237     }
238     // Marshal the count value to a TPM2B buffer for the KDF
239     cntr.t.size = sizeof(currentCount);
240     UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
241
242     // Now can do the KDF to create the random value for the signing operation
243     // During the creation process, we may generate an r that does not meet the
244     // requirements of the random value.
245     // want to generate a new r.
246     r->t.size = n.t.size;
247
248     for(iterations = 1; iterations < 1000000;)
249     {
250         int i;
251         CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, COMMIT_STRING,
252                 (TPM2B *)name, &cntr.b, n.t.size * 8, r->t.buffer,
253                 &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 Returns	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 Returns	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 Returns	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  #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.

```

4  static bigNum
5  EcdsaDigest(
6      bigNum          bnD,                // OUT: the adjusted digest
7      const TPM2B_DIGEST *digest,        // IN: digest to adjust
8      bigConst        max                 // IN: value that indicates the maximum
9                                          // number of bits in the results
10 )
11 {
12     int          bitsInMax = BnSizeInBits(max);
13     int          shift;
14     //
15     if(digest == NULL)
16         BnSetWord(bnD, 0);
17     else
18     {
19         BnFromBytes(bnD, digest->t.buffer,
20                     (NUMBYTES)MIN(digest->t.size, BITS_TO_BYTES(bitsInMax)));
21         shift = BnSizeInBits(bnD) - bitsInMax;
22         if(shift > 0)
23             BnShiftRight(bnD, bnD, shift);
24     }
25     return bnD;
26 }

```

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

- s is the signature
- k is a random value
- r is the value to sign
- d is the private EC key
- n is the order of the curve

Error Returns	Meaning
TPM_RC_NO_RESULT	the result of the operation was zero or $r \pmod{n}$ is zero

```

27  static TPM_RC
28  BnSchnorrSign(
29      bigNum          bnS,                // OUT: 's' component of the signature

```



```

30     bigConst      bnK,          // IN: a random value
31     bigNum        bnR,          // IN: the signature 'r' value
32     bigConst      bnD,          // IN: the private key
33     bigConst      bnN          // IN: the order of the curve
34 )
35 {
36     // Need a local temp value to store the intermediate computation because product
37     // size can be larger than will fit in bnS.
38     BN_VAR(bnT1, MAX_ECC_PARAMETER_BYTES * 2 * 8);
39     //
40     // Reduce bnR without changing the input value
41     BnDiv(NULL, bnT1, bnR, bnN);
42     if(BnEqualZero(bnT1))
43         return TPM_RC_NO_RESULT;
44     // compute s = (k + r * d) (mod n)
45     // r * d
46     BnMult(bnT1, bnT1, bnD);
47     // k * r * d
48     BnAdd(bnT1, bnT1, bnK);
49     // k + r * d (mod n)
50     BnDiv(NULL, bnS, bnT1, bnN);
51     return (BnEqualZero(bnS)) ? TPM_RC_NO_RESULT : TPM_RC_SUCCESS;
52 }

```

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.

```

53 TPM_RC
54 BnSignEcdsa(
55     bigNum      bnR,          // OUT: 'r' component of the signature
56     bigNum      bnS,          // OUT: 's' component of the signature
57     bigCurve     E,           // IN: the curve used in the signature
58                                     // process
59     bigNum      bnD,          // IN: private signing key
60     const TPM2B_DIGEST *digest, // IN: the digest to sign
61     RAND_STATE   *rand,       // IN: used in debug of signing
62 )
63 {
64     ECC_NUM(bnK);
65     ECC_NUM(bnIk);
66     BN_VAR(bnE, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE) * 8);
67     POINT(ecR);
68     bigConst      order = CurveGetOrder(AccessCurveData(E));
69     TPM_RC         retVal = TPM_RC_SUCCESS;
70     INT32          tries = 10;
71     BOOL           OK = FALSE;
72     //
73     pAssert(digest != NULL);
74     // The algorithm as described in "Suite B Implementer's Guide to FIPS
75     // 186-3(ECDSA)"
76     // 1. Use one of the routines in Appendix A.2 to generate (k, k^-1), a
77     //    per-message secret number and its inverse modulo n. Since n is prime,
78     //    the output will be invalid only if there is a failure in the RBG.
79     // 2. Compute the elliptic curve point R = [k]G = (xR, yR) using EC scalar
80     //    multiplication (see [Routines]), where G is the base point included in
81     //    the set of domain parameters.
82     // 3. Compute r = xR mod n. If r = 0, then return to Step 1. 1.
83     // 4. Use the selected hash function to compute H = Hash(M).
84     // 5. Convert the bit string H to an integer e as described in Appendix B.2.
85     // 6. Compute s = (k^-1 * (e + d * r)) mod q. If s = 0, return to Step 1.2.

```

```

86 // 7. Return (r, s).
87 // In the code below, q is n (that it, the order of the curve is p)
88
89 do // This implements the loop at step 6. If s is zero, start over.
90 {
91     for(; tries > 0; tries--)
92     {
93         // Step 1 and 2 -- generate an ephemeral key and the modular inverse
94         // of the private key.
95         if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
96             continue;
97         // x coordinate is mod p. Make it mod q
98         BnMod(ecR->x, order);
99         // Make sure that it is not zero;
100         if(BnEqualZero(ecR->x))
101             continue;
102         // write the modular reduced version of r as part of the signature
103         BnCopy(bnR, ecR->x);
104         // Make sure that a modular inverse exists and try again if not
105         OK = (BnModInverse(bnIk, bnK, order));
106         if(OK)
107             break;
108     }
109     if(!OK)
110         goto Exit;
111
112     EcdsaDigest(bnE, digest, order);
113
114     // now have inverse of K (bnIk), e (bnE), r (bnR), d (bnD) and
115     // CurveGetOrder(E)
116     // Compute s = k^-1 (e + r*d) (mod q)
117     // first do s = r*d mod q
118     BnModMult(bnS, bnR, bnD, order);
119     // s = e + s = e + r * d
120     BnAdd(bnS, bnE, bnS);
121     // s = k^-1 s (mod n) = k^-1 (e + r * d) (mod n)
122     BnModMult(bnS, bnIk, bnS, order);
123
124     // If S is zero, try again
125 } while(BnEqualZero(bnS));
126 Exit:
127     return retVal;
128 }
129 #if ALG_ECDA

```

10.2.12.3.2 BnSignEcdaa()

This function performs $s = r + T * d \bmod q$ where

- 'r' is a random, or pseudo-random value created in the commit phase
- nonceK* is a TPM-generated, random value $0 < \text{nonceK} < n$
- T* is mod *q* of **Hash**(*nonceK* || *digest*), and
- 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 EC Schnorr making for a lot of confusion.

Error Returns	Meaning
TPM_RC_SCHEME	unsupported hash algorithm
TPM_RC_NO_RESULT	cannot get values from random number generator

```

130 static TPM_RC
131 BnSignEcdaa(
132     TPM2B_ECC_PARAMETER *nonceK,           // OUT: 'nonce' component of the signature
133     bigNum bnS,                             // OUT: 's' component of the signature
134     bigCurve E,                             // IN: the curve used in signing
135     bigNum bnD,                             // IN: the private key
136     const TPM2B_DIGEST *digest,            // IN: the value to sign (mod 'q')
137     TPMT_ECC_SCHEME *scheme,               // IN: signing scheme (contains the
138                                         // commit count value).
139     OBJECT *eccKey,                        // IN: The signing key
140     RAND_STATE *rand                      // IN: a random number state
141 )
142 {
143     TPM_RC retVal;
144     TPM2B_ECC_PARAMETER r;
145     HASH_STATE state;
146     TPM2B_DIGEST T;
147     BN_MAX(bnT);
148     //
149     NOT_REFERENCED(rand);
150     if(!CryptGenerateR(&r, &scheme->details.ecdaa.count,
151                       eccKey->publicArea.parameters.eccDetail.curveID,
152                       &eccKey->name))
153         retVal = TPM_RC_VALUE;
154     else
155     {
156         // This allocation is here because 'r' doesn't have a value until
157         // CryptGenerateR() is done.
158         ECC_INITIALIZED(bnR, &r);
159         do
160         {
161             // generate nonceK such that 0 < nonceK < n
162             // use bnT as a temp.
163             if(!BnEccGetPrivate(bnT, AccessCurveData(E), rand))
164             {
165                 retVal = TPM_RC_NO_RESULT;
166                 break;
167             }
168             BnTo2B(bnT, &nonceK->b, 0);
169
170             T.t.size = CryptHashStart(&state, scheme->details.ecdaa.hashAlg);
171             if(T.t.size == 0)
172             {
173                 retVal = TPM_RC_SCHEME;
174             }
175             else
176             {
177                 CryptDigestUpdate2B(&state, &nonceK->b);
178                 CryptDigestUpdate2B(&state, &digest->b);
179                 CryptHashEnd2B(&state, &T.b);
180                 BnFrom2B(bnT, &T.b);
181                 // Watch out for the name collisions in this call!!
182                 retVal = BnSchnorrSign(bnS, bnR, bnT, bnD,
183                                       AccessCurveData(E)->order);
184             }
185         } while(retVal == TPM_RC_NO_RESULT);
186         // Because the rule is that internal state is not modified if the command
187         // fails, only end the commit if the command succeeds.
188         // NOTE that if the result of the Schnorr computation was zero

```

```

189         // it will probably not be worthwhile to run the same command again because
190         // the result will still be zero. This means that the Commit command will
191         // need to be run again to get a new commit value for the signature.
192         if(retVal == TPM_RC_SUCCESS)
193             CryptEndCommit(scheme->details.ecdaa.count);
194     }
195     return retVal;
196 }
197 #endif // ALG_ECDSA
198 #if ALG_EC Schnorr

```

10.2.12.3.3 SchnorrReduce()

Function to reduce a hash result if its 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*.

```

199 static void
200 SchnorrReduce(
201     TPM2B      *number,           // IN/OUT: Value to reduce
202     bigConst    reference        // IN: the reference value
203 )
204 {
205     UINT16      maxBytes = (UINT16)BITS_TO_BYTES(BnSizeInBits(reference));
206     if(number->size > maxBytes)
207         number->size = maxBytes;
208 }

```

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 Returns	Meaning
TPM_RC_NO_RESULT	failure in the Schnorr sign process
TPM_RC_SCHEME	<i>hashAlg</i> can't produce zero-length digest

```

209 static TPM_RC
210 BnSignEcSchnorr(
211     bigNum      bnR,           // OUT: 'r' component of the signature
212     bigNum      bnS,           // OUT: 's' component of the signature
213     bigCurve     E,           // IN: the curve used in signing
214     bigNum      bnD,           // IN: the signing key
215     const TPM2B_DIGEST *digest, // IN: the digest to sign
216     TPM_ALG_ID   hashAlg,      // IN: signing scheme (contains a hash)
217     RAND_STATE   *rand         // IN: non-NULL when testing
218 )
219 {
220     HASH_STATE   hashState;
221     UINT16       digestSize = CryptHashGetDigestSize(hashAlg);
222     TPM2B_TYPE(T, MAX(MAX_DIGEST_SIZE, MAX_ECC_KEY_BYTES));

```

```

223     TPM2B_T                T2b;
224     TPM2B                  *e = &T2b.b;
225     TPM_RC                  retVal = TPM_RC_NO_RESULT;
226     const ECC_CURVE_DATA    *C;
227     bigConst                 order;
228     bigConst                 prime;
229     ECC_NUM(bnK);
230     POINT(ecR);
231 //
232 // Parameter checks
233 if(E == NULL)
234     ERROR_RETURN(TPM_RC_VALUE);
235 C = AccessCurveData(E);
236 order = CurveGetOrder(C);
237 prime = CurveGetOrder(C);
238
239 // If the digest does not produce a hash, then null the signature and return
240 // a failure.
241 if(digestSize == 0)
242 {
243     BnSetWord(bnR, 0);
244     BnSetWord(bnS, 0);
245     ERROR_RETURN(TPM_RC_SCHEME);
246 }
247 do
248 {
249     // Generate a random key pair
250     if(!BnEccGenerateKeyPair(bnK, ecR, E, rand))
251         break;
252     // Convert R.x to a string
253     BnTo2B(ecR->x, e, (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(prime)));
254
255     // f) compute r = Hash(e || P) (mod n)
256     CryptHashStart(&hashState, hashAlg);
257     CryptDigestUpdate2B(&hashState, e);
258     CryptDigestUpdate2B(&hashState, &digest->b);
259     e->size = CryptHashEnd(&hashState, digestSize, e->buffer);
260     // Reduce the hash size if it is larger than the curve order
261     SchnorrReduce(e, order);
262     // Convert hash to number
263     BnFrom2B(bnR, e);
264     // Do the Schnorr computation
265     retVal = BnSchnorrSign(bnS, bnK, bnR, bnD, CurveGetOrder(C));
266 } while(retVal == TPM_RC_NO_RESULT);
267 Exit:
268     return retVal;
269 }
270 #endif // ALG_ECSCHNORR
271 #if ALG_SM2
272 #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

```

273 static BOOL
274 BnHexEqual(
275     bigNum          bn,          //IN: big number value

```

```

276     const char      *c           //IN: character string number
277   )
278 {
279     ECC_NUM(bnC) ;
280     BnFromHex(bnC, c) ;
281     return (BnUnsignedCmp(bn, bnC) == 0) ;
282 }
283 #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 Returns	Meaning
TPM_RC_VALUE	bad curve

```

284 static TPM_RC
285 BnSignEcSm2(
286     bigNum          bnR,           // OUT: 'r' component of the signature
287     bigNum          bnS,           // OUT: 's' component of the signature
288     bigCurve        E,             // IN: the curve used in signing
289     bigNum          bnD,           // IN: the private key
290     const TPM2B_DIGEST *digest,    // IN: the digest to sign
291     RAND_STATE      *rand          // IN: random number generator (mostly for
292                                     // debug)
293 )
294 {
295     BN_MAX_INITIALIZED(bnE, digest); // Don't know how big digest might be
296     ECC_NUM(bnN) ;
297     ECC_NUM(bnK) ;
298     ECC_NUM(bnT) ;                  // temp
299     POINT(Q1) ;
300     bigConst          order = (E != NULL)
301         ? CurveGetOrder(AccessCurveData(E)) : NULL;
302 //
303 #ifdef _SM2_SIGN_DEBUG
304     BnFromHex(bnE, "B524F552CD82B8B028476E005C377FB1"
305                 "9A87E6FC682D48BB5D42E3D9B9E7FE76");
306     BnFromHex(bnD, "128B2FA8BD433C6C068C8D803DFF7979"
307                 "2A519A55171B1B650C23661D15897263");
308 #endif
309     // A3: Use random number generator to generate random number 1 <= k <= n-1;
310     // NOTE: Ax: numbers are from the SM2 standard
311 loop:
312 {
313     // Get a random number 0 < k < n
314     BnGenerateRandomInRange(bnK, order, rand) ;
315 #ifdef _SM2_SIGN_DEBUG
316     BnFromHex(bnK, "6CB28D99385C175C94F94E934817663F"
317                 "C176D925DD72B727260DBAAE1FB2F96F");
318 #endif
319     // A4: Figure out the point of elliptic curve (x1, y1)=[k]G, and according
320     // to details specified in 4.2.7 in Part 1 of this document, transform the
321     // data type of x1 into an integer;
322     if(!BnEccModMult(Q1, NULL, bnK, E))
323         goto loop;
324     // A5: Figure out 'r' = ('e' + 'x1') mod 'n',
325     BnAdd(bnR, bnE, Q1->x) ;
326     BnMod(bnR, order) ;
327 #ifdef _SM2_SIGN_DEBUG
328     pAssert(BnHexEqual(bnR, "40F1EC59F793D9F49E09DCEF49130D41"

```

```

329                                     "94F79FB1EED2CAA55BACDB49C4E755D1")); ;
330 #endif
331     // if r=0 or r+k=n, return to A3;
332     if(BnEqualZero(bnR))
333         goto loop;
334     BnAdd(bnT, bnK, bnR);
335     if(BnUnsignedCmp(bnT, bnN) == 0)
336         goto loop;
337     // A6: Figure out s = ((1 + dA)^-1 (k - r dA) mod n,
338     // if s=0, return to A3;
339     // compute t = (1+dA)^-1
340     BnAddWord(bnT, bnD, 1);
341     BnModInverse(bnT, bnT, order);
342 #ifdef _SM2_SIGN_DEBUG
343     pAssert(BnHexEqual(bnT, "79BFCF3052C80DA7B939E0C6914A18CB"
344                         "B2D96D8555256E83122743A7D4F5F956")); ;
345 #endif
346     // compute s = t * (k - r * dA) mod n
347     BnModMult(bnS, bnR, bnD, order);
348     // k - r * dA mod n = k + n - ((r * dA) mod n)
349     BnSub(bnS, order, bnS);
350     BnAdd(bnS, bnK, bnS);
351     BnModMult(bnS, bnS, bnT, order);
352 #ifdef _SM2_SIGN_DEBUG
353     pAssert(BnHexEqual(bnS, "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
354                         "67A457872FB09EC56327A67EC7DEEBE7")); ;
355 #endif
356     if(BnEqualZero(bnS))
357         goto loop;
358 }
359 // A7: According to details specified in 4.2.1 in Part 1 of this document,
360 // transform the data type of r, s into bit strings, signature of message M
361 // is (r, s).
362 // This is handled by the common return code
363 #ifdef _SM2_SIGN_DEBUG
364     pAssert(BnHexEqual(bnR, "40F1EC59F793D9F49E09DCEF49130D41"
365                         "94F79FB1EED2CAA55BACDB49C4E755D1")); ;
366     pAssert(BnHexEqual(bnS, "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
367                         "67A457872FB09EC56327A67EC7DEEBE7")); ;
368 #endif
369     return TPM_RC_SUCCESS;
370 }
371 #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 Returns	Meaning
TPM_RC_SCHEME	<i>scheme</i> is not supported

```

372 LIB_EXPORT TPM_RC
373 CryptEccSign(
374     TPMT_SIGNATURE *signature,    // OUT: signature
375     OBJECT *signKey,             // IN: ECC key to sign the hash
376     const TPM2B_DIGEST *digest,  // IN: digest to sign

```



```

377     TPMT_ECC_SCHEME      *scheme,          // IN: signing scheme
378     RAND_STATE           *rand
379 )
380 {
381     CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
382     ECC_INITIALIZED(bnD, &signKey->sensitive.sensitive.ecc.b);
383     ECC_NUM(bnR);
384     ECC_NUM(bnS);
385     const ECC_CURVE_DATA *C;
386     TPM_RC               retVal = TPM_RC_SCHEME;
387     //
388     NOT_REFERENCED(scheme);
389     if(E == NULL)
390         ERROR_RETURN(TPM_RC_VALUE);
391     C = AccessCurveData(E);
392     signature->signature.ecdaa.signatureR.t.size
393         = sizeof(signature->signature.ecdaa.signatureR.t.buffer);
394     signature->signature.ecdaa.signatureS.t.size
395         = sizeof(signature->signature.ecdaa.signatureS.t.buffer);
396     TEST(signature->sigAlg);
397     switch(signature->sigAlg)
398     {
399         case ALG_ECDSA_VALUE:
400             retVal = BnSignEcdsa(bnR, bnS, E, bnD, digest, rand);
401             break;
402     #if ALG_ECDA
403         case ALG_ECDA_VALUE:
404             retVal = BnSignEcdaa(&signature->signature.ecdaa.signatureR, bnS, E,
405                                 bnD, digest, scheme, signKey, rand);
406             bnR = NULL;
407             break;
408     #endif
409     #if ALG_ECSCNORR
410         case ALG_ECSCNORR_VALUE:
411             retVal = BnSignEcSchnorr(bnR, bnS, E, bnD, digest,
412                                     signature->signature.ecschnorr.hash,
413                                     rand);
414             break;
415     #endif
416     #if ALG_SM2
417         case ALG_SM2_VALUE:
418             retVal = BnSignEcSm2(bnR, bnS, E, bnD, digest, rand);
419             break;
420     #endif
421     default:
422         break;
423     }
424     // If signature generation worked, convert the results.
425     if(retVal == TPM_RC_SUCCESS)
426     {
427         NUMBYTES      orderBytes =
428             (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(CurveGetOrder(C)));
429         if(bnR != NULL)
430             BnTo2B(bnR, &signature->signature.ecdaa.signatureR.b, orderBytes);
431         if(bnS != NULL)
432             BnTo2B(bnS, &signature->signature.ecdaa.signatureS.b, orderBytes);
433     }
434     Exit:
435     CURVE_FREE(E);
436     return retVal;
437 }
438 #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 Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```

439  TPM_RC
440  BnValidateSignatureEcdsa (
441      bigNum          bnR,          // IN: 'r' component of the signature
442      bigNum          bnS,          // IN: 's' component of the signature
443      bigCurve        E,           // IN: the curve used in the signature
444                               // process
445      bn_point_t      *ecQ,        // IN: the public point of the key
446      const TPM2B_DIGEST *digest   // IN: the digest that was signed
447  )
448  {
449      // Make sure that the allocation for the digest is big enough for a maximum
450      // digest
451      BN_VAR(bnE, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE) * 8);
452      POINT(ecR);
453      ECC_NUM(bnU1);
454      ECC_NUM(bnU2);
455      ECC_NUM(bnW);
456      bigConst          order = CurveGetOrder(AccessCurveData(E));
457      TPM_RC            retVal = TPM_RC_SIGNATURE;
458  //
459      // Get adjusted digest
460      EcdsaDigest(bnE, digest, order);
461      // 1. If r and s are not both integers in the interval [1, n - 1], output
462      // INVALID.
463      // bnR and bnS were validated by the caller
464      // 2. Use the selected hash function to compute H0 = Hash(M0).
465      // This is an input parameter
466      // 3. Convert the bit string H0 to an integer e as described in Appendix B.2.
467      // Done at entry
468      // 4. Compute w = (s')^-1 mod n, using the routine in Appendix B.1.
469      if(!BnModInverse(bnW, bnS, order))
470          goto Exit;
471      // 5. Compute u1 = (e' * w) mod n, and compute u2 = (r' * w) mod n.
472      BnModMult(bnU1, bnE, bnW, order);
473      BnModMult(bnU2, bnR, bnW, order);
474      // 6. Compute the elliptic curve point R = (xR, yR) = u1G+u2Q, using EC
475      // scalar multiplication and EC addition (see [Routines]). If R is equal to
476      // the point at infinity O, output INVALID.
477      if(BnPointMult(ecR, CurveGetG(AccessCurveData(E)), bnU1, ecQ, bnU2, E)
478         != TPM_RC_SUCCESS)
479          goto Exit;
480      // 7. Compute v = Rx mod n.
481      BnMod(ecR->x, order);
482      // 8. Compare v and r0. If v = r0, output VALID; otherwise, output INVALID
483      if(BnUnsignedCmp(ecR->x, bnR) != 0)
484          goto Exit;
485
486      retVal = TPM_RC_SUCCESS;
487  Exit:
488      return retVal;
489  }
490  #endif          // ALG_ECDSA
491  #if ALG_SM2

```

10.2.12.3.9 BnValidateSignatureEcSm2()

This function is used to validate an SM2 signature.

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```

492 static TPM_RC
493 BnValidateSignatureEcSm2(
494     bigNum      bnR,      // IN: 'r' component of the signature
495     bigNum      bnS,      // IN: 's' component of the signature
496     bigCurve     E,        // IN: the curve used in the signature
497                     // process
498     bigPoint     ecQ,      // IN: the public point of the key
499     const TPM2B_DIGEST *digest // IN: the digest that was signed
500 )
501 {
502     POINT(P);
503     ECC_NUM(bnRp);
504     ECC_NUM(bnT);
505     BN_MAX_INITIALIZED(bnE, digest);
506     BOOL OK;
507     bigConst order = CurveGetOrder(AccessCurveData(E));
508
509 #ifdef _SM2_SIGN_DEBUG
510     // Make sure that the input signature is the test signature
511     pAssert(BnHexEqual(bnR,
512         "40F1EC59F793D9F49E09DCEF49130D41"
513         "94F79FB1EED2CAA55BACDB49C4E755D1"));
514     pAssert(BnHexEqual(bnS,
515         "6FC6DAC32C5D5CF10C77DFB20F7C2EB6"
516         "67A457872FB09EC56327A67EC7DEEBE7"));
517 #endif
518     // b) compute t := (r + s) mod n
519     BnAdd(bnT, bnR, bnS);
520     BnMod(bnT, order);
521 #ifdef _SM2_SIGN_DEBUG
522     pAssert(BnHexEqual(bnT,
523         "2B75F07ED7ECE7CCC1C8986B991F441A"
524         "D324D6D619FE06DD63ED32E0C997C801"));
525 #endif
526     // c) verify that t > 0
527     OK = !BnEqualZero(bnT);
528     if(!OK)
529         // set T to a value that should allow rest of the computations to run
530         // without trouble
531         BnCopy(bnT, bnS);
532     // d) compute (x, y) := [s]G + [t]Q
533     OK = BnEccModMult2(P, NULL, bnS, ecQ, bnT, E);
534 #ifdef _SM2_SIGN_DEBUG
535     pAssert(OK && BnHexEqual(P->x,
536         "110FCDA57615705D5E7B9324AC4B856D"
537         "23E6D9188B2AE47759514657CE25D112"));
538 #endif
539     // e) compute r' := (e + x) mod n (the x coordinate is in bnT)
540     OK = OK && BnAdd(bnRp, bnE, P->x);
541     OK = OK && BnMod(bnRp, order);
542
543     // f) verify that r' = r
544     OK = OK && (BnUnsignedCmp(bnR, bnRp) == 0);
545
546     if(!OK)
547         return TPM_RC_SIGNATURE;
548     else

```

```

549         return TPM_RC_SUCCESS;
550     }
551 #endif // ALG_SM2
552 #if ALG_EC Schnorr

```

10.2.12.3.10 BnValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature.

Error Returns	Meaning
TPM_RC_SIGNATURE	signature not valid

```

553 static TPM_RC
554 BnValidateSignatureEcSchnorr(
555     bigNum      bnR,          // IN: 'r' component of the signature
556     bigNum      bnS,          // IN: 's' component of the signature
557     TPM_ALG_ID   hashAlg,     // IN: hash algorithm of the signature
558     bigCurve     E,           // IN: the curve used in the signature
559                                     // process
560     bigPoint     ecQ,         // IN: the public point of the key
561     const TPM2B_DIGEST *digest // IN: the digest that was signed
562 )
563 {
564     BN_MAX(bnRn);
565     POINT(ecE);
566     BN_MAX(bnEx);
567     const ECC_CURVE_DATA *C = AccessCurveData(E);
568     bigConst              order = CurveGetOrder(C);
569     UINT16                digestSize = CryptHashGetDigestSize(hashAlg);
570     HASH_STATE            hashState;
571     TPM2B_TYPE(BUFFER, MAX(MAX_ECC_PARAMETER_BYTES, MAX_DIGEST_SIZE));
572     TPM2B_BUFFER          Ex2 = {{sizeof(Ex2.t.buffer), { 0 }}};
573     BOOL                  OK;
574     //
575     // E = [s]G - [r]Q
576     BnMod(bnR, order);
577     // Make -r = n - r
578     BnSub(bnRn, order, bnR);
579     // E = [s]G + [-r]Q
580     OK = BnPointMult(ecE, CurveGetG(C), bnS, ecQ, bnRn, E) == TPM_RC_SUCCESS;
581     // // reduce the x portion of E mod q
582     // OK = OK && BnMod(ecE->x, order);
583     // Convert to byte string
584     OK = OK && BnTo2B(ecE->x, &Ex2.b,
585                     (NUMBYTES) (BITS_TO_BYTES(BnSizeInBits(order))));
586     if(OK)
587     {
588     // Ex = h(pE.x || digest)
589         CryptHashStart(&hashState, hashAlg);
590         CryptDigestUpdate(&hashState, Ex2.t.size, Ex2.t.buffer);
591         CryptDigestUpdate(&hashState, digest->t.size, digest->t.buffer);
592         Ex2.t.size = CryptHashEnd(&hashState, digestSize, Ex2.t.buffer);
593         SchnorrReduce(&Ex2.b, order);
594         BnFrom2B(bnEx, &Ex2.b);
595         // see if Ex matches R
596         OK = BnUnsignedCmp(bnEx, bnR) == 0;
597     }
598     return (OK) ? TPM_RC_SUCCESS : TPM_RC_SIGNATURE;
599 }
600 #endif // ALG_EC Schnorr

```

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 Returns	Meaning
TPM_RC_SIGNATURE	not a valid signature

```

601  LIB_EXPORT TPM_RC
602  CryptEccValidateSignature(
603      TPMT_SIGNATURE      *signature,      // IN: signature to be verified
604      OBJECT               *signKey,       // IN: ECC key signed the hash
605      const TPM2B_DIGEST   *digest        // IN: digest that was signed
606  )
607  {
608      CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
609      ECC_NUM(bnR);
610      ECC_NUM(bnS);
611      POINT_INITIALIZED(ecQ, &signKey->publicArea.unique.ecc);
612      bigConst          order;
613      TPM_RC             retVal;
614
615      if(E == NULL)
616          ERROR_RETURN(TPM_RC_VALUE);
617
618      order = CurveGetOrder(AccessCurveData(E));
619
620      // Make sure that the scheme is valid
621      switch(signature->sigAlg)
622      {
623          case ALG_ECDSA_VALUE:
624      #if ALG_ECDSCHNORR
625          case ALG_ECDSCHNORR_VALUE:
626      #endif
627      #if ALG_SM2
628          case ALG_SM2_VALUE:
629      #endif
630          break;
631          default:
632              ERROR_RETURN(TPM_RC_SCHEME);
633              break;
634      }
635      // Can convert r and s after determining that the scheme is an ECC scheme. If
636      // this conversion doesn't work, it means that the unmarshaling code for
637      // an ECC signature is broken.
638      BnFrom2B(bnR, &signature->signature.ecdsa.signatureR.b);
639      BnFrom2B(bnS, &signature->signature.ecdsa.signatureS.b);
640
641      // r and s have to be greater than 0 but less than the curve order
642      if(BnEqualZero(bnR) || BnEqualZero(bnS))
643          ERROR_RETURN(TPM_RC_SIGNATURE);
644      if((BnUnsignedCmp(bnS, order) >= 0)
645         || (BnUnsignedCmp(bnR, order) >= 0))
646          ERROR_RETURN(TPM_RC_SIGNATURE);
647
648      switch(signature->sigAlg)
649      {
650          case ALG_ECDSA_VALUE:
651              retVal = BnValidateSignatureEcDsa(bnR, bnS, E, ecQ, digest);
652              break;
653
654      #if ALG_ECDSCHNORR
655          case ALG_ECDSCHNORR_VALUE:
656              retVal = BnValidateSignatureEcSchnorr(bnR, bnS,

```

```

657                                     signature->signature.any.hashAlg,
658                                     E, ecQ, digest);
659                                     break;
660 #endif
661 #if ALG_SM2
662     case ALG_SM2_VALUE:
663         retVal = BnValidateSignatureEcSm2(bnR, bnS, E, ecQ, digest);
664         break;
665 #endif
666     default:
667         FAIL(FATAL_ERROR_INTERNAL);
668 }
669 Exit:
670     CURVE_FREE(E);
671     return retVal;
672 }

```

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 Returns	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

```

673 LIB_EXPORT TPM_RC
674 CryptEccCommitCompute(
675     TPMS_ECC_POINT *K,           // OUT: [d]B or [r]Q
676     TPMS_ECC_POINT *L,           // OUT: [r]B
677     TPMS_ECC_POINT *E,           // OUT: [r]M
678     TPM_ECC_CURVE   curveId,     // IN: the curve for the computations
679     TPMS_ECC_POINT *M,           // IN: M (optional)
680     TPMS_ECC_POINT *B,           // IN: B (optional)
681     TPM2B_ECC_PARAMETER *d,      // IN: d (optional)
682     TPM2B_ECC_PARAMETER *r,      // IN: the computed r value (required)
683 )
684 {
685     CURVE_INITIALIZED(curve, curveId); // Normally initialize E as the curve, but
686                                         // E means something else in this function
687     ECC_INITIALIZED(bnR, r);
688     TPM_RC retVal = TPM_RC_SUCCESS;
689 //
690 // Validate that the required parameters are provided.
691 // Note: E has to be provided if computing E := [r]Q or E := [r]M. Will do
692 // E := [r]Q if both M and B are NULL.
693 pAssert(r != NULL && E != NULL);
694
695 // Initialize the output points in case they are not computed
696 ClearPoint2B(K);
697 ClearPoint2B(L);
698 ClearPoint2B(E);
699
700 // Sizes of the r parameter may not be zero
701 pAssert(r->t.size > 0);
702
703 // If B is provided, compute K=[d]B and L=[r]B
704 if(B != NULL)

```

```

705     {
706         ECC_INITIALIZED(bnD, d);
707         POINT_INITIALIZED(pB, B);
708         POINT(pK);
709         POINT(pL);
710     //
711     pAssert(d != NULL && K != NULL && L != NULL);
712
713     if(!BnIsOnCurve(pB, AccessCurveData(curve)))
714         ERROR_RETURN(TPM_RC_VALUE);
715     // do the math for K = [d]B
716     if((retVal = BnPointMult(pK, pB, bnD, NULL, NULL, curve)) != TPM_RC_SUCCESS)
717         goto Exit;
718     // Convert BN K to TPM2B K
719     BnPointTo2B(K, pK, curve);
720     // compute L= [r]B after checking for cancel
721     if(_plat_IsCanceled())
722         ERROR_RETURN(TPM_RC_CANCELED);
723     // compute L = [r]B
724     if(!BnIsValidPrivateEcc(bnR, curve))
725         ERROR_RETURN(TPM_RC_VALUE);
726     if((retVal = BnPointMult(pL, pB, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
727         goto Exit;
728     // Convert BN L to TPM2B L
729     BnPointTo2B(L, pL, curve);
730 }
731 if((M != NULL) || (B == NULL))
732 {
733     POINT_INITIALIZED(pM, M);
734     POINT(pE);
735 //
736     // Make sure that a place was provided for the result
737     pAssert(E != NULL);
738
739     // if this is the third point multiply, check for cancel first
740     if((B != NULL) && _plat_IsCanceled())
741         ERROR_RETURN(TPM_RC_CANCELED);
742
743     // If M provided, then pM will not be NULL and will compute E = [r]M.
744     // However, if M was not provided, then pM will be NULL and E = [r]G
745     // will be computed
746     if((retVal = BnPointMult(pE, pM, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)
747         goto Exit;
748     // Convert E to 2B format
749     BnPointTo2B(E, pE, curve);
750 }
751 Exit:
752     CURVE_FREE(curve);
753     return retVal;
754 }
755 #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"
6  #define HASH_TABLE_SIZE      (HASH_COUNT + 1)
7  #if ALG_SHA1
8  HASH_DEF_TEMPLATE(SHA1, Sha1);
9  #endif
10 #if ALG_SHA256
11 HASH_DEF_TEMPLATE(SHA256, Sha256);
12 #endif
13 #if ALG_SHA384
14 HASH_DEF_TEMPLATE(SHA384, Sha384);
15 #endif
16 #if ALG_SHA512
17 HASH_DEF_TEMPLATE(SHA512, Sha512);
18 #endif
19 #if ALG_SM3_256
20 HASH_DEF_TEMPLATE(SM3_256, Sm3_256);
21 #endif
22 HASH_DEF_NULL_Def = {{0}};
23 PHASH_DEF HashDefArray[] = {
24 #if ALG_SHA1
25     &Sha1_Def,
26 #endif
27 #if ALG_SHA256
28     &Sha256_Def,
29 #endif
30 #if ALG_SHA384
31     &Sha384_Def,
32 #endif
33 #if ALG_SHA512
34     &Sha512_Def,
35 #endif
36 #if ALG_SM3_256
37     &Sm3_256_Def,
38 #endif
39     &NULL_Def
40 };

```

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.

```

41  BOOL
42  CryptHashInit(
43      void
44  )
45  {

```

```

46     LibHashInit();
47     return TRUE;
48 }

```

10.2.13.3.2 CryptHashStartup()

This function is called by TPM2_Startup() in case there is work to do at startup. Currently, this is a placeholder.

```

49  BOOL
50  CryptHashStartup(
51      void
52  )
53  {
54      int        i = sizeof(HashDefArray) / sizeof(PHASH_DEF) - 1;
55      return (i == HASH_COUNT);
56  }

```

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.

```

57  PHASH_DEF
58  CryptGetHashDef(
59      TPM_ALG_ID    hashAlg
60  )
61  {
62      INT16        i;
63      #define HASHES (sizeof(HashDefArray) / sizeof(PHASH_DEF))
64      for(i = 0; i < HASHES; i++)
65      {
66          PHASH_DEF p = HashDefArray[i];
67          if(p->hashAlg == hashAlg)
68              return p;
69      }
70      return &NULL_Def;
71  }

```

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

```

72  BOOL
73  CryptHashIsValidAlg(
74      TPM_ALG_ID    hashAlg,           // IN: the algorithm to check

```

```

75     BOOL                flag                // IN: TRUE if TPM_ALG_NULL is to be treated
76                                           //      as a valid hash
77     )
78 {
79     if(hashAlg == TPM_ALG_NULL)
80         return flag;
81     return CryptGetHashDef(hashAlg) != &NULL_Def;
82 }

```

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

```

83  LIB_EXPORT TPM_ALG_ID
84  CryptHashGetAlgByIndex(
85      UINT32                index            // IN: the index
86  )
87  {
88      TPM_ALG_ID            hashAlg;
89      if(index >= HASH_COUNT)
90          hashAlg = TPM_ALG_NULL;
91      else
92          hashAlg = HashDefArray[index]->hashAlg;
93      return hashAlg;
94  }

```

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

```

95  LIB_EXPORT UINT16
96  CryptHashGetDigestSize(
97      TPM_ALG_ID            hashAlg        // IN: hash algorithm to look up
98  )
99  {
100     return CryptGetHashDef(hashAlg)->digestSize;
101 }

```

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

```

102  LIB_EXPORT UINT16
103  CryptHashGetBlockSize(
104      TPM_ALG_ID      hashAlg      // IN: hash algorithm to look up
105  )
106  {
107      return CryptGetHashDef(hashAlg) ->blockSize;
108  }

```

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.

```

109  LIB_EXPORT const BYTE *
110  CryptHashGetOid(
111      TPM_ALG_ID      hashAlg
112  )
113  {
114      return CryptGetHashDef(hashAlg) ->OID;
115  }

```

10.2.13.4.8 CryptHashGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```

116  TPM_ALG_ID
117  CryptHashGetContextAlg(
118      PHASH_STATE      state      // IN: the context to check
119  )
120  {
121      return state->hashAlg;
122  }

```

10.2.13.5 State Import and Export

10.2.13.5.1 CryptHashCopyState

This function is used to clone a HASH_STATE.

```

123  LIB_EXPORT void
124  CryptHashCopyState(
125      HASH_STATE      *out,      // OUT: destination of the state
126      const HASH_STATE *in      // IN: source of the state
127  )
128  {
129      pAssert(out->type == in->type);
130      out->hashAlg = in->hashAlg;
131      out->def = in->def;
132      if(in->hashAlg != TPM_ALG_NULL)
133      {
134          HASH_STATE_COPY(out, in);
135      }
136      if(in->type == HASH_STATE_HMAC)
137      {

```

```

138     const HMAC_STATE    *hIn = (HMAC_STATE *)in;
139     HMAC_STATE          *hOut = (HMAC_STATE *)out;
140     hOut->hmacKey = hIn->hmacKey;
141 }
142 return;
143 }

```

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.

```

144 void
145 CryptHashExportState(
146     PCHASH_STATE          internalFmt,    // IN: the hash state formatted for use by
147                                     // library
148     PEXPORT_HASH_STATE    externalFmt    // OUT: the exported hash state
149 )
150 {
151     BYTE                  *outBuf = (BYTE *)externalFmt;
152 //
153     cAssert(sizeof(HASH_STATE) <= sizeof(EXPORT_HASH_STATE));
154     // the following #define is used to move data from an aligned internal data
155     // structure to a byte buffer (external format data).
156 #define CopyToOffset(value) \
157     memcpy(&outBuf[offsetof(HASH_STATE,value)], &internalFmt->value, \
158           sizeof(internalFmt->value))
159     // Copy the hashAlg
160     CopyToOffset(hashAlg);
161     CopyToOffset(type);
162 #ifndef HASH_STATE_SMAC
163     if(internalFmt->type == HASH_STATE_SMAC)
164     {
165         memcpy(outBuf, internalFmt, sizeof(HASH_STATE));
166         return;
167     }
168 #endif
169 #endif
170     if(internalFmt->type == HASH_STATE_HMAC)
171     {
172         HMAC_STATE          *from = (HMAC_STATE *)internalFmt;
173         memcpy(&outBuf[offsetof(HMAC_STATE, hmacKey)], &from->hmacKey,
174               sizeof(from->hmacKey));
175     }
176     if(internalFmt->hashAlg != TPM_ALG_NULL)
177         HASH_STATE_EXPORT(externalFmt, internalFmt);
178 }

```

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.

```

179 void
180 CryptHashImportState(
181     PHASH_STATE          internalFmt,    // OUT: the hash state formatted for use by
182                                     // the library
183     PCEXPORT_HASH_STATE  externalFmt    // IN: the exported hash state
184 )
185 {
186     BYTE                  *inBuf = (BYTE *)externalFmt;
187 //

```

```

188 #define CopyFromOffset(value) \
189     memcpy(&internalFmt->value, &inBuf[offsetof(HASH_STATE,value)], \
190           sizeof(internalFmt->value))
191
192 // Copy the hashAlg of the byte-aligned input structure to the structure-aligned
193 // internal structure.
194 CopyFromOffset(hashAlg);
195 CopyFromOffset(type);
196 if(internalFmt->hashAlg != TPM_ALG_NULL)
197 {
198 #ifdef HASH_STATE_SMAC
199     if(internalFmt->type == HASH_STATE_SMAC)
200     {
201         memcpy(internalFmt, inBuf, sizeof(HASH_STATE));
202         return;
203     }
204 #endif
205     internalFmt->def = CryptGetHashDef(internalFmt->hashAlg);
206     HASH_STATE_IMPORT(internalFmt, inBuf);
207     if(internalFmt->type == HASH_STATE_HMAC)
208     {
209         HMAC_STATE *to = (HMAC_STATE *)internalFmt;
210         memcpy(&to->hmacKey, &inBuf[offsetof(HMAC_STATE, hmacKey)],
211               sizeof(to->hmacKey));
212     }
213 }
214 }

```

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.

```

215 static UINT16
216 HashEnd(
217     PHASH_STATE    hashState,    // IN: the hash state
218     UINT32          dOutSize,     // IN: the size of receive buffer
219     PBYTE           dOut         // OUT: the receive buffer
220 )
221 {
222     BYTE            temp[MAX_DIGEST_SIZE];
223     if((hashState->hashAlg == TPM_ALG_NULL)
224        || (hashState->type != HASH_STATE_HASH))
225         dOutSize = 0;
226     if(dOutSize > 0)
227     {
228         hashState->def = CryptGetHashDef(hashState->hashAlg);
229         // Set the final size
230         dOutSize = MIN(dOutSize, hashState->def->digestSize);
231         // Complete into the temp buffer and then copy
232         HASH_END(hashState, temp);
233         // Don't want any other functions calling the HASH_END method
234         // directly.
235 #undef HASH_END
236         memcpy(dOut, &temp, dOutSize);
237     }
238     hashState->type = HASH_STATE_EMPTY;
239     return (UINT16)dOutSize;
240 }

```

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

```

241  LIB_EXPORT UINT16
242  CryptHashStart(
243      PHASH_STATE      hashState,      // OUT: the running hash state
244      TPM_ALG_ID      hashAlg         // IN: hash algorithm
245  )
246  {
247      UINT16          retVal;
248
249      TEST(hashAlg);
250
251      hashState->hashAlg = hashAlg;
252      if(hashAlg == TPM_ALG_NULL)
253      {
254          retVal = 0;
255      }
256      else
257      {
258          hashState->def = CryptGetHashDef(hashAlg);
259          HASH_START(hashState);
260          retVal = hashState->def->digestSize;
261      }
262      #undef HASH_START
263      hashState->type = HASH_STATE_HASH;
264      return retVal;
265  }

```

10.2.13.6.3 CryptDigestUpdate()

Add data to a hash or HMAC, SMAC stack.

```

266  void
267  CryptDigestUpdate(
268      PHASH_STATE      hashState,      // IN: the hash context information
269      UINT32          dataSize,        // IN: the size of data to be added
270      const BYTE       *data          // IN: data to be hashed
271  )
272  {
273      if(hashState->hashAlg != TPM_ALG_NULL)
274      {
275          if((hashState->type == HASH_STATE_HASH)
276             || (hashState->type == HASH_STATE_HMAC))
277              HASH_DATA(hashState, dataSize, (BYTE *)data);
278      #if SMAC_IMPLEMENTED
279          else if(hashState->type == HASH_STATE_SMAC)
280              (hashState->state.smac.smacMethods.data) (&hashState->state.smac.state,
281                                                         dataSize, data);

```



```

282 #endif // SMAC_IMPLEMENTED
283     else
284         FAIL(FATAL_ERROR_INTERNAL);
285     }
286     return;
287 }

```

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

```

288 LIB_EXPORT UINT16
289 CryptHashEnd(
290     PHASH_STATE    hashState,    // IN: the state of hash stack
291     UINT32          dOutSize,     // IN: size of digest buffer
292     BYTE            *dOut         // OUT: hash digest
293 )
294 {
295     pAssert(hashState->type == HASH_STATE_HASH);
296     return HashEnd(hashState, dOutSize, dOut);
297 }

```

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>

```

298 LIB_EXPORT UINT16
299 CryptHashBlock(
300     TPM_ALG_ID      hashAlg,      // IN: The hash algorithm
301     UINT32           dataSize,     // IN: size of buffer to hash
302     const BYTE       *data,        // IN: the buffer to hash
303     UINT32           dOutSize,     // IN: size of the digest buffer
304     BYTE            *dOut         // OUT: digest buffer
305 )
306 {
307     HASH_STATE       state;
308     CryptHashStart(&state, hashAlg);
309     CryptDigestUpdate(&state, dataSize, data);
310     return HashEnd(&state, dOutSize, dOut);
311 }

```

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.

```

312 LIB_EXPORT void
313 CryptDigestUpdate2B(
314     PHASH_STATE state,           // IN: the digest state
315     const TPM2B *bIn            // IN: 2B containing the data
316 )
317 {
318     // Only compute the digest if a pointer to the 2B is provided.
319     // In CryptDigestUpdate(), if size is zero or buffer is NULL, then no change
320     // to the digest occurs. This function should not provide a buffer if bIn is
321     // not provided.
322     pAssert(bIn != NULL);
323     CryptDigestUpdate(state, bIn->size, bIn->buffer);
324     return;
325 }

```

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>

```

326 LIB_EXPORT UINT16
327 CryptHashEnd2B(
328     PHASH_STATE state,           // IN: the hash state
329     P2B digest,                 // IN: the size of the buffer Out: requested
330                                // number of bytes
331 )
332 {
333     return CryptHashEnd(state, digest->size, digest->buffer);
334 }

```

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().

```

335 LIB_EXPORT void
336 CryptDigestUpdateInt(
337     void *state,                // IN: the state of hash stack
338     UINT32 intSize,             // IN: the size of 'intValue' in bytes
339     UINT64 intValue             // IN: integer value to be hashed
340 )
341 {
342     #if LITTLE_ENDIAN_TPM
343         intValue = REVERSE_ENDIAN_64(intValue);
344     #endif
345     CryptDigestUpdate(state, intSize, &((BYTE *)&intValue)[8 - intSize]);
346 }

```

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)

```

347 LIB_EXPORT UINT16
348 CryptHmacStart(
349     PHMAC_STATE    state,           // IN/OUT: the state buffer
350     TPM_ALG_ID     hashAlg,        // IN: the algorithm to use
351     UINT16          keySize,        // IN: the size of the HMAC key
352     const BYTE      *key            // IN: the HMAC key
353 )
354 {
355     PHASH_DEF        hashDef;
356     BYTE *           pb;
357     UINT32           i;
358     //
359     hashDef = CryptGetHashDef(hashAlg);
360     if(hashDef->digestSize != 0)
361     {
362         // If the HMAC key is larger than the hash block size, it has to be reduced
363         // to fit. The reduction is a digest of the hashKey.
364         if(keySize > hashDef->blockSize)
365         {
366             // if the key is too big, reduce it to a digest of itself
367             state->hmacKey.t.size = CryptHashBlock(hashAlg, keySize, key,
368                                                     hashDef->digestSize,
369                                                     state->hmacKey.t.buffer);
370         }
371         else
372         {
373             memcpy(state->hmacKey.t.buffer, key, keySize);
374             state->hmacKey.t.size = keySize;
375         }
376         // XOR the key with iPad (0x36)
377         pb = state->hmacKey.t.buffer;
378         for(i = state->hmacKey.t.size; i > 0; i--)
379             *pb++ ^= 0x36;
380
381         // if the keySize is smaller than a block, fill the rest with 0x36
382         for(i = hashDef->blockSize - state->hmacKey.t.size; i > 0; i--)
383             *pb++ = 0x36;
384
385         // Increase the oPadSize to a full block
386         state->hmacKey.t.size = hashDef->blockSize;
387
388         // Start a new hash with the HMAC key
389         // This will go in the caller's state structure and may be a sequence or not
390         CryptHashStart((PHASH_STATE)state, hashAlg);
391         CryptDigestUpdate((PHASH_STATE)state, state->hmacKey.t.size,
392                          state->hmacKey.t.buffer);
393         // XOR the key block with 0x5c ^ 0x36
394         for(pb = state->hmacKey.t.buffer, i = hashDef->blockSize; i > 0; i--)
395             *pb++ ^= (0x5c ^ 0x36);
396     }
397     // Set the hash algorithm

```

```

398     state->hashState.hashAlg = hashAlg;
399     // Set the hash state type
400     state->hashState.type = HASH_STATE_HMAC;
401
402     return hashDef->digestSize;
403 }

```

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)

```

404 LIB_EXPORT UINT16
405 CryptHmacEnd(
406     PHMAC_STATE    state,           // IN: the hash state buffer
407     UINT32          dOutSize,       // IN: size of digest buffer
408     BYTE            *dOut           // OUT: hash digest
409 )
410 {
411     BYTE            temp[MAX_DIGEST_SIZE];
412     PHASH_STATE     hState = (PHASH_STATE) &state->hashState;
413
414     #if SMAC_IMPLEMENTED
415     if(hState->type == HASH_STATE_SMAC)
416         return (state->hashState.state.smac.smacMethods.end)
417             (&state->hashState.state.smac.state,
418              dOutSize,
419              dOut);
420     #endif
421     pAssert(hState->type == HASH_STATE_HMAC);
422     hState->def = CryptGetHashDef(hState->hashAlg);
423     // Change the state type for completion processing
424     hState->type = HASH_STATE_HASH;
425     if(hState->hashAlg == TPM_ALG_NULL)
426         dOutSize = 0;
427     else
428     {
429
430         // Complete the current hash
431         HashEnd(hState, hState->def->digestSize, temp);
432         // Do another hash starting with the oPad
433         CryptHashStart(hState, hState->hashAlg);
434         CryptDigestUpdate(hState, state->hmacKey.t.size, state->hmacKey.t.buffer);
435         CryptDigestUpdate(hState, hState->def->digestSize, temp);
436     }
437     return HashEnd(hState, dOutSize, dOut);
438 }

```

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

```

439  LIB_EXPORT UINT16
440  CryptHmacStart2B(
441      PHMAC_STATE    hmacState,    // OUT: the state of HMAC stack. It will be used
442                          //      in HMAC update and completion
443      TPMI_ALG_HASH  hashAlg,      // IN: hash algorithm
444      P2B            key           // IN: HMAC key
445  )
446  {
447      return CryptHmacStart(hmacState, hashAlg, key->size, key->buffer);
448  }

```

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>

```

449  LIB_EXPORT UINT16
450  CryptHmacEnd2B(
451      PHMAC_STATE    hmacState,    // IN: the state of HMAC stack
452      P2B            digest        // OUT: HMAC
453  )
454  {
455      return CryptHmacEnd(hmacState, digest->size, digest->buffer);
456  }

```

10.2.13.8 Mask and Key Generation Functions

10.2.13.8.1 CryptMGF1()

This function performs MGF1 using the selected hash. MGF1 is $T(n) = T(n-1) \parallel H(\text{seed} \parallel \text{counter})$. 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>

```

457  LIB_EXPORT UINT16
458  CryptMGF1(
459      UINT32          mSize,        // IN: length of the mask to be produced
460      BYTE            *mask,        // OUT: buffer to receive the mask
461      TPMI_ALG_ID     hashAlg,      // IN: hash to use
462      UINT32          seedSize,     // IN: size of the seed
463      BYTE            *seed         // IN: seed size
464  )
465  {
466      HASH_STATE      hashState;
467      PHASH_DEF        hDef = CryptGetHashDef(hashAlg);
468      UINT32            remaining;

```

```

469     UINT32          counter = 0;
470     BYTE            swappedCounter[4];
471
472     // If there is no digest to compute return
473     if((hashAlg == TPM_ALG_NULL) || (mSize == 0))
474         return 0;
475
476     for(remaining = mSize; ; remaining -= hDef->digestSize)
477     {
478         // Because the system may be either Endian...
479         UINT32_TO_BYTE_ARRAY(counter, swappedCounter);
480
481         // Start the hash and include the seed and counter
482         CryptHashStart(&hashState, hashAlg);
483         CryptDigestUpdate(&hashState, seedSize, seed);
484         CryptDigestUpdate(&hashState, 4, swappedCounter);
485
486         // Handling the completion depends on how much space remains in the mask
487         // buffer. If it can hold the entire digest, put it there. If not
488         // put the digest in a temp buffer and only copy the amount that
489         // will fit into the mask buffer.
490         HashEnd(&hashState, remaining, mask);
491         if(remaining <= hDef->digestSize)
492             break;
493         mask = &mask[hDef->digestSize];
494         counter++;
495     }
496     return (UINT16)mSize;
497 }

```

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

```

498     LIB_EXPORT UINT16
499     CryptKDFa(
500         TPM_ALG_ID    hashAlg,          // IN: hash algorithm used in HMAC
501         const TPM2B    *key,             // IN: HMAC key
502         const TPM2B    *label,           // IN: a label for the KDF
503         const TPM2B    *contextU,        // IN: context U
504         const TPM2B    *contextV,        // IN: context V
505         UINT32         sizeInBits,        // IN: size of generated key in bits
506         BYTE           *keyStream,        // OUT: key buffer
507         UINT32         *counterInOut,     // IN/OUT: caller may provide the iteration
508                                         // counter for incremental operations to

```

```

509                                     //      avoid large intermediate buffers.
510     UINT16          blocks           // IN: If non-zero, this is the maximum number
511                                     //      of blocks to be returned, regardless
512                                     //      of sizeInBits
513 )
514 {
515     UINT32          counter = 0;      // counter value
516     INT16           bytes;            // number of bytes to produce
517     UINT16           generated;       // number of bytes generated
518     BYTE            *stream = keyStream;
519     HMAC_STATE       hState;
520     UINT16           digestSize = CryptHashGetDigestSize(hashAlg);
521
522     pAssert(key != NULL && keyStream != NULL);
523
524     TEST(TPM_ALG_KDF1_SP800_108);
525
526     if(digestSize == 0)
527         return 0;
528
529     if(counterInOut != NULL)
530         counter = *counterInOut;
531
532     // If the size of the request is larger than the numbers will handle,
533     // it is a fatal error.
534     pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);
535
536     // The number of bytes to be generated is the smaller of the sizeInBits bytes or
537     // the number of requested blocks. The number of blocks is the smaller of the
538     // number requested or the number allowed by sizeInBits. A partial block is
539     // a full block.
540     bytes = (blocks > 0) ? blocks * digestSize : (UINT16)BITS_TO_BYTES(sizeInBits);
541     generated = bytes;
542
543     // Generate required bytes
544     for(; bytes > 0; bytes -= digestSize)
545     {
546         counter++;
547         // Start HMAC
548         if(CryptHmacStart(&hState, hashAlg, key->size, key->buffer) == 0)
549             return 0;
550         // Adding counter
551         CryptDigestUpdateInt(&hState.hashState, 4, counter);
552
553         // Adding label
554         if(label != NULL)
555             HASH_DATA(&hState.hashState, label->size, (BYTE *)label->buffer);
556         // Add a null. SP108 is not very clear about when the 0 is needed but to
557         // make this like the previous version that did not add an 0x00 after
558         // a null-terminated string, this version will only add a null byte
559         // if the label parameter did not end in a null byte, or if no label
560         // is present.
561         if((label == NULL)
562            || (label->size == 0)
563            || (label->buffer[label->size - 1] != 0))
564             CryptDigestUpdateInt(&hState.hashState, 1, 0);
565         // Adding contextU
566         if(contextU != NULL)
567             HASH_DATA(&hState.hashState, contextU->size, contextU->buffer);
568         // Adding contextV
569         if(contextV != NULL)
570             HASH_DATA(&hState.hashState, contextV->size, contextV->buffer);
571         // Adding size in bits
572         CryptDigestUpdateInt(&hState.hashState, 4, sizeInBits);
573
574         // Complete and put the data in the buffer

```



```

575     CryptHmacEnd(&hState, bytes, stream);
576     stream = &stream[digestSize];
577 }
578 // Masking in the KDF is disabled. If the calling function wants something
579 // less than even number of bytes, then the caller should do the masking
580 // because there is no universal way to do it here
581 if(counterInOut != NULL)
582     *counterInOut = counter;
583 return generated;
584 }

```

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

```

585 LIB_EXPORT UINT16
586 CryptKDFe(
587     TPM_ALG_ID    hashAlg,           // IN: hash algorithm used in HMAC
588     TPM2B         *Z,                // IN: Z
589     const TPM2B    *label,           // IN: a label value for the KDF
590     TPM2B         *partyUInfo,       // IN: PartyUInfo
591     TPM2B         *partyVInfo,       // IN: PartyVInfo
592     UINT32        sizeInBits,        // IN: size of generated key in bits
593     BYTE          *keyStream         // OUT: key buffer
594 )
595 {
596     HASH_STATE     hashState;
597     PHASH_DEF      hashDef = CryptGetHashDef(hashAlg);
598
599     UINT32          counter = 0;      // counter value
600     UINT16          hLen;
601     BYTE            *stream = keyStream;
602     INT16           bytes;           // number of bytes to generate
603
604     pAssert(keyStream != NULL && Z != NULL && ((sizeInBits + 7) / 8) < INT16_MAX);
605     //
606     hLen = hashDef->digestSize;
607     bytes = (INT16)((sizeInBits + 7) / 8);
608     if(hashAlg == TPM_ALG_NULL || bytes == 0)
609         return 0;
610
611     // Generate required bytes
612     //The inner loop of that KDF uses:
613     // Hash[i] := H(counter | Z | OtherInfo) (5)
614     // Where:
615     // Hash[i]    the hash generated on the i-th iteration of the loop.
616     // H()        an approved hash function
617     // counter    a 32-bit counter that is initialized to 1 and incremented
618     //            on each iteration
619     // Z          the X coordinate of the product of a public ECC key and a
620     //            different private ECC key.
621     // OtherInfo  a collection of qualifying data for the KDF defined below.
622     // In this specification, OtherInfo will be constructed by:

```

```

623 //      OtherInfo := Use | PartyUInfo | PartyVInfo
624 for(; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
625 {
626     if(bytes < hLen)
627         hLen = bytes;
628     counter++;
629     // Do the hash
630     CryptHashStart(&hashState, hashAlg);
631     // Add counter
632     CryptDigestUpdateInt(&hashState, 4, counter);
633
634     // Add Z
635     if(Z != NULL)
636         CryptDigestUpdate2B(&hashState, Z);
637     // Add label
638     if(label != NULL)
639         CryptDigestUpdate2B(&hashState, label);
640     // Add a null. SP108 is not very clear about when the 0 is needed but to
641     // make this like the previous version that did not add an 0x00 after
642     // a null-terminated string, this version will only add a null byte
643     // if the label parameter did not end in a null byte, or if no label
644     // is present.
645     if((label == NULL)
646        || (label->size == 0)
647        || (label->buffer[label->size - 1] != 0))
648         CryptDigestUpdateInt(&hashState, 1, 0);
649     // Add PartyUInfo
650     if(partyUInfo != NULL)
651         CryptDigestUpdate2B(&hashState, partyUInfo);
652
653     // Add PartyVInfo
654     if(partyVInfo != NULL)
655         CryptDigestUpdate2B(&hashState, partyVInfo);
656
657     // Compute Hash. hLen was changed to be the smaller of bytes or hLen
658     // at the start of each iteration.
659     CryptHashEnd(&hashState, hLen, stream);
660 }
661
662 // Mask off bits if the required bits is not a multiple of byte size
663 if((sizeInBits % 8) != 0)
664     keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
665
666 return (UINT16)((sizeInBits + 7) / 8);
667 }

```

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  // #define CPRI_PRIME
4  // #include "PrimeTable.h"
5  #include "CryptPrimeSieve_fp.h"
6  extern const uint32_t s_LastPrimeInTable;
7  extern const uint32_t s_PrimeTableSize;
8  extern const uint32_t s_PrimesInTable;
9  extern const unsigned char s_PrimeTable[];
10 extern bigConst s_CompositeOfSmallPrimes;

```

10.2.14.2 Functions

10.2.14.2.1 Root2()

This finds ceil(sqrt(n)) to use as a stopping point for searching the prime table.

```

11 static uint32_t
12 Root2(
13     uint32_t n
14 )
15 {
16     int32_t last = (int32_t)(n >> 2);
17     int32_t next = (int32_t)(n >> 1);
18     int32_t diff;
19     int32_t stop = 10;
20     //
21     // get a starting point
22     for(; next != 0; last >= 1, next >= 2);
23     last++;
24     do
25     {
26         next = (last + (n / last)) >> 1;
27         diff = next - last;
28         last = next;
29         if(stop-- == 0)
30             FAIL(FATAL_ERROR_INTERNAL);
31     } while(diff < -1 || diff > 1);
32     if((n / next) > (unsigned)next)
33         next++;
34     pAssert(next != 0);
35     pAssert(((n / next) <= (unsigned)next) && (n / (next + 1) < (unsigned)next));
36     return next;
37 }

```

10.2.14.2.2 IsPrimeInt()

This will do a test of a word of up to 32-bits in size.

```

38 BOOL
39 IsPrimeInt(
40     uint32_t n
41 )
42 {

```

```

43     uint32_t      i;
44     uint32_t      stop;
45     if(n < 3 || ((n & 1) == 0))
46         return (n == 2);
47     if(n <= s_LastPrimeInTable)
48     {
49         n >>= 1;
50         return ((s_PrimeTable[n >> 3] >> (n & 7)) & 1);
51     }
52     // Need to search
53     stop = Root2(n) >> 1;
54     // starting at 1 is equivalent to staring at (1 << 1) + 1 = 3
55     for(i = 1; i < stop; i++)
56     {
57         if((s_PrimeTable[i >> 3] >> (i & 7)) & 1)
58             // see if this prime evenly divides the number
59             if((n % ((i << 1) + 1)) == 0)
60                 return FALSE;
61     }
62     return TRUE;
63 }

```

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.

```

64     BOOL
65     BnIsProbablyPrime(
66         bigNum      prime,           // IN:
67         RAND_STATE  *rand            // IN: the random state just
68                                     // in case Miller-Rabin is required
69     )
70 {
71     #if RADIX_BITS > 32
72     if(BnUnsignedCmpWord(prime, UINT32_MAX) <= 0)
73     #else
74     if(BnGetSize(prime) == 1)
75     #endif
76         return IsPrimeInt(prime->d[0]);
77
78     if(BnIsEven(prime))
79         return FALSE;
80     if(BnUnsignedCmpWord(prime, s_LastPrimeInTable) <= 0)
81     {
82         crypt_uword_t      temp = prime->d[0] >> 1;
83         return ((s_PrimeTable[temp >> 3] >> (temp & 7)) & 1);
84     }
85     {
86         BN_VAR(n, LARGEST_NUMBER_BITS);
87         BnGcd(n, prime, s_CompositeOfSmallPrimes);
88         if(!BnEqualWord(n, 1))
89             return FALSE;
90     }
91     return MillerRabin(prime, rand);
92 }

```

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.

93 UINT32

```

94  MillerRabinRounds(
95      UINT32      bits          // IN: Number of bits in the RSA prime
96  )
97  {
98      if(bits < 511) return 8;    // don't really expect this
99      if(bits < 1536) return 5;  // for 512 and 1K primes
100     return 4;                  // for 3K public modulus and greater
101 }

```

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

```

102  BOOL
103  MillerRabin(
104      bigNum      bnW,
105      RAND_STATE  *rand
106  )
107  {
108      BN_MAX(bnWm1);
109      BN_PRIME(bnM);
110      BN_PRIME(bnB);
111      BN_PRIME(bnZ);
112      BOOL      ret = FALSE;    // Assumed composite for easy exit
113      unsigned int a;
114      unsigned int j;
115      int          wLen;
116      int          i;
117      int          iterations = MillerRabinRounds(BnSizeInBits(bnW));
118  //
119  INSTRUMENT_INC(MillerRabinTrials[PrimeIndex]);
120
121      pAssert(bnW->size > 1);
122      // Let a be the largest integer such that 2^a divides w1.
123      BnSubWord(bnWm1, bnW, 1);
124      pAssert(bnWm1->size != 0);
125
126      // Since w is odd (w-1) is even so start at bit number 1 rather than 0
127      // Get the number of bits in bnWm1 so that it doesn't have to be recomputed
128      // on each iteration.
129      i = bnWm1->size * RADIX_BITS;
130      // Now find the largest power of 2 that divides w1
131      for(a = 1;
132          (a < (bnWm1->size * RADIX_BITS)) &&
133          (BnTestBit(bnWm1, a) == 0);
134          a++);
135      // 2. m = (w1) / 2^a
136      BnShiftRight(bnM, bnWm1, a);
137      // 3. wlen = len(w).
138      wLen = BnSizeInBits(bnW);
139      // 4. For i = 1 to iterations do
140      for(i = 0; i < iterations; i++)
141      {
142          // 4.1 Obtain a string b of wlen bits from an RBG.
143          // Ensure that 1 < b < w1.
144          // 4.2 If ((b <= 1) or (b >= w1)), then go to step 4.1.
145          while(BnGetRandomBits(bnB, wLen, rand) && ((BnUnsignedCmpWord(bnB, 1) <= 0)

```

```

146     || (BnUnsignedCmp(bnB, bnWm1) >= 0));
147     if(g_inFailureMode)
148         return FALSE;
149
150     // 4.3  $z = b^m \bmod w$ .
151     // if ModExp fails, then say this is not
152     // prime and bail out.
153     BnModExp(bnZ, bnB, bnM, bnW);
154
155     // 4.4 If ((z == 1) or (z = w == 1)), then go to step 4.7.
156     if((BnUnsignedCmpWord(bnZ, 1) == 0)
157        || (BnUnsignedCmp(bnZ, bnWm1) == 0))
158         goto step4point7;
159     // 4.5 For j = 1 to a - 1 do.
160     for(j = 1; j < a; j++)
161     {
162         // 4.5.1  $z = z^2 \bmod w$ .
163         BnModMult(bnZ, bnZ, bnZ, bnW);
164         // 4.5.2 If (z = w1), then go to step 4.7.
165         if(BnUnsignedCmp(bnZ, bnWm1) == 0)
166             goto step4point7;
167         // 4.5.3 If (z = 1), then go to step 4.6.
168         if(BnEqualWord(bnZ, 1))
169             goto step4point6;
170     }
171     // 4.6 Return COMPOSITE.
172 step4point6:
173     INSTRUMENT_INC(failedAtIteration[i]);
174     goto end;
175     // 4.7 Continue. Comment: Increment i for the do-loop in step 4.
176 step4point7:
177     continue;
178 }
179 // 5. Return PROBABLY PRIME
180 ret = TRUE;
181 end:
182 return ret;
183 }
184 #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.

```

185 TPM_RC
186 RsaCheckPrime(
187     bigNum          prime,
188     UINT32          exponent,
189     RAND_STATE      *rand
190 )
191 {
192     #if !RSA_KEY_SIEVE
193     TPM_RC          retVal = TPM_RC_SUCCESS;
194     UINT32          modE = BnModWord(prime, exponent);
195
196     NOT_REFERENCED(rand);
197
198     if(modE == 0)

```

```

199         // evenly divisible so add two keeping the number odd
200         BnAddWord(prime, prime, 2);
201         // want 0 != (p - 1) mod e
202         // which is 1 != p mod e
203         else if(modE == 1)
204             // subtract 2 keeping number odd and insuring that
205             // 0 != (p - 1) mod e
206             BnSubWord(prime, prime, 2);
207
208         if(BnIsProbablyPrime(prime, rand) == 0)
209             ERROR_RETURN(g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_VALUE);
210 Exit:
211     return retVal;
212 #else
213     return PrimeSelectWithSieve(prime, exponent, rand);
214 #endif
215 }

```

10.2.14.2.7 AdjustPrimeCandidate()

This function adjusts the candidate prime so that it is odd and $> \sqrt{2}/2$. 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.

The code maps the most significant crypt_uword_t in *prime* so that a 32-/64-bit value of 0 to 0xB5050...0 and a value of 0xff...f to 0xff...f. It also sets the LSb of *prime* to make sure that the number is odd.

This code has been fixed so that it will work with a RADIX_SIZE == 64.

The function also puts the number on a field boundary.

```

216 LIB_EXPORT void
217 RsaAdjustPrimeCandidate(
218     bigNum      prime
219 )
220 {
221     crypt_uword_t    msw = prime->d[prime->size - 1];
222     crypt_uword_t    adjusted;
223 #if RADIX_BITS == 64
224 #   define ADD_CONST ((crypt_uword_t)0xB505000000000000ULL)
225 #else
226 #   define ADD_CONST ((crypt_uword_t)0xB5050000UL)
227 #endif
228     // Multiplying 0xff...f by 0x4AFB gives 0xff...f - 0xB5050...0
229     adjusted = (crypt_uword_t)(msw >> 16) * (crypt_uword_t)0x4AFB;
230     adjusted += ((msw & 0xFFFF) * (crypt_uword_t)0x4AFB) >> 16;
231     adjusted += ADD_CONST;
232     prime->d[prime->size - 1] = adjusted;
233     // make sure the number is odd
234     prime->d[0] |= 1;
235 }

```

10.2.14.2.8 BnGeneratePrimeForRSA()

Function to generate a prime of the desired size with the proper attributes for an RSA prime.

```

236 TPM_RC
237 BnGeneratePrimeForRSA(
238     bigNum      prime,

```



```
239     UINT32      bits,  
240     UINT32      exponent,  
241     RAND_STATE  *rand  
242 )  
243 {  
244     BOOL          found = FALSE;  
245     //  
246     // Make sure that the prime is large enough  
247     pAssert(prime->allocated >= BITS_TO_CRYPT_WORDS(bits));  
248     // Only try to handle specific sizes of keys in order to save overhead  
249     pAssert((bits % 32) == 0);  
250  
251     prime->size = BITS_TO_CRYPT_WORDS(bits);  
252  
253     while(!found)  
254     {  
255         DRBG_Generate(rand, (BYTE *)prime->d, (UINT16)BITS_TO_BYTES(bits));  
256         if(g_inFailureMode)  
257             return TPM_RC_FAILURE;  
258         RsaAdjustPrimeCandidate(prime);  
259         found = RsaCheckPrime(prime, exponent, rand) == TPM_RC_SUCCESS;  
260     }  
261     return TPM_RC_SUCCESS;  
262 }  
263 #endif // ALG_RSA
```

10.2.15 CryptPrimeSieve.c

10.2.15.1 Includes and defines

```

1  #include "Tpm.h"
2  #if RSA_KEY_SIEVE
3  #include "CryptPrimeSieve_fp.h"

```

This determines the number of bits in the largest sieve field.

```

4  #define MAX_FIELD_SIZE 2048
5  extern const uint32_t    s_LastPrimeInTable;
6  extern const uint32_t    s_PrimeTableSize;
7  extern const uint32_t    s_PrimesInTable;
8  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.

```

9  const uint32_t    s_PrimeMarkersCount = 6;
10 const uint32_t    s_PrimeMarkers[] = {
11     8167, 17881, 28183, 38891, 49871, 60961 };
12 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.

```

13 LIB_EXPORT void
14 RsaAdjustPrimeLimit(
15     uint32_t    requestedPrimes
16 )
17 {
18     if(requestedPrimes == 0 || requestedPrimes > s_PrimesInTable)
19         requestedPrimes = s_PrimesInTable;
20     requestedPrimes = (requestedPrimes - 1) / 1024;
21     if(requestedPrimes < s_PrimeMarkersCount)
22         primeLimit = s_PrimeMarkers[requestedPrimes];
23     else
24         primeLimit = s_LastPrimeInTable;
25     primeLimit >>= 1;
26
27 }

```

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.

```

28 LIB_EXPORT uint32_t
29 RsaNextPrime(

```

```

30     uint32_t    lastPrime
31 )
32 {
33     if(lastPrime == 0)
34         return 0;
35     lastPrime >>= 1;
36     for(lastPrime += 1; lastPrime <= primeLimit; lastPrime++)
37     {
38         if(((s_PrimeTable[lastPrime >> 3] >> (lastPrime & 0x7)) & 1) == 1)
39             return ((lastPrime << 1) + 1);
40     }
41     return 0;
42 }

```

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.

```

43 const BYTE    seedValues[] = {
44     0x16, 0x29, 0xcb, 0xa4, 0x65, 0xda, 0x30, 0x6c,
45     0x99, 0x96, 0x4c, 0x53, 0xa2, 0x2d, 0x52, 0x96,
46     0x49, 0xcb, 0xb4, 0x61, 0xd8, 0x32, 0x2d, 0x99,
47     0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
48     0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
49     0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
50     0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
51     0x5a, 0xa6, 0x0d, 0xc3, 0x96, 0x69, 0xc9, 0x34,
52     0x25, 0xda, 0x22, 0x65, 0x99, 0xb4, 0x4c, 0x1b,
53     0x86, 0x2d, 0xd3, 0x92, 0x69, 0x4a, 0xb4, 0x45,
54     0xca, 0x32, 0x69, 0x99, 0x36, 0x0c, 0x5b, 0xa6,
55     0x25, 0xd3, 0x94, 0x68, 0x8b, 0x94, 0x65, 0xd2,
56     0x32, 0x6d, 0x18, 0xb6, 0x4c, 0x4b, 0xa6, 0x29,
57     0xd1};
58 #define USE_NIBBLE
59 #ifndef USE_NIBBLE
60 static const BYTE bitsInByte[256] = {
61     0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
62     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
63     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
64     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
65     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
66     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
67     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
68     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
69     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
70     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
71     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
72     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
73     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
74     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
75     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
76     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
77     0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
78     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
79     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
80     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
81     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
82     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
83     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
84     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
85     0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
86     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
87     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
88     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
89     0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
90     0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,

```

```

91      0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
92      0x05, 0x06, 0x06, 0x07, 0x06, 0x07, 0x07, 0x08
93  };
94  #define BitsInByte(x)    bitsInByte[(unsigned char)x]
95  #else
96  const BYTE bitsInNibble[16] = {
97      0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
98      0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04};
99  #define BitsInByte(x)    \
100      (bitsInNibble[(unsigned char)(x) & 0xf] \
101      + bitsInNibble[((unsigned char)(x) >> 4) & 0xf])
102  #endif

```

10.2.15.2.3 BitsInArray()

This function counts the number of bits set in an array of bytes.

```

103  static int
104  BitsInArray(
105      const unsigned char    *a,           // IN: A pointer to an array of bytes
106      unsigned int           aSize        // IN: the number of bytes to sum
107  )
108  {
109      int    j = 0;
110      for(; aSize; a++, aSize--)
111          j += BitsInByte(*a);
112      return j;
113  }

```

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

```

114  LIB_EXPORT int
115  FindNthSetBit(
116      const UINT16    aSize,           // IN: the size of the array to check
117      const BYTE      *a,             // IN: the array to check
118      const UINT32    n               // IN, the number of the SET bit
119  )
120  {
121      UINT16    i;
122      int       retValue;
123      UINT32    sum = 0;
124      BYTE      sel;
125
126      //find the bit
127      for(i = 0; (i < (int)aSize) && (sum < n); i++)
128          sum += BitsInByte(a[i]);
129      i--;
130      // The chosen bit is in the byte that was just accessed
131      // Compute the offset to the start of that byte
132      retValue = i * 8 - 1;
133      sel = a[i];
134      // Subtract the bits in the last byte added.
135      sum -= BitsInByte(sel);
136      // Now process the byte, one bit at a time.

```

```

137     for(; (sel != 0) && (sum != n); retValue++, sel = sel >> 1)
138         sum += (sel & 1) != 0;
139     return (sum == n) ? retValue : -1;
140 }
141 typedef struct
142 {
143     UINT16    prime;
144     UINT16    count;
145 } SIEVE_MARKS;
146 const SIEVE_MARKS sieveMarks[5] = {
147     {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.

```

148 LIB_EXPORT UINT32
149 PrimeSieve(
150     bigNum        bnN,        // IN/OUT: number to sieve
151     UINT32        fieldSize,  // IN: size of the field area in bytes
152     BYTE          *field      // IN: field
153 )
154 {
155     INT32          i;
156     UINT32         j;
157     UINT32         fieldBits = fieldSize * 8;
158     UINT32         r;
159     BYTE          *pField;
160     INT32          iter;
161     UINT32         adjust;
162     UINT32         mark = 0;
163     UINT32         count = sieveMarks[0].count;
164     UINT32         stop = sieveMarks[0].prime;
165     UINT32         composite;
166     UINT32         pList[8];
167     UINT32         next;
168
169     pAssert(field != NULL && bnN != NULL);
170
171     // If the remainder is odd, then subtracting the value will give an even number,
172     // but we want an odd number, so subtract the 105+rem. Otherwise, just subtract
173     // the even remainder.
174     adjust = BnModWord(bnN, 105);
175     if(adjust & 1)
176         adjust += 105;
177
178     // Adjust the input number so that it points to the first number in a
179     // aligned field.
180     BnSubWord(bnN, bnN, adjust);
181     // pAssert(BnModWord(bnN, 105) == 0);
182     pField = field;
183     for(i = fieldSize; i >= sizeof(seedValues);
184         pField += sizeof(seedValues), i -= sizeof(seedValues))
185     {

```

```

186     memcpy(pField, seedValues, sizeof(seedValues));
187 }
188 if(i != 0)
189     memcpy(pField, seedValues, i);
190
191 // Cycle through the primes, clearing bits
192 // Have already done 3, 5, and 7
193 iter = 7;
194
195 #define NEXT_PRIME(iter)    (iter = RsaNextPrime(iter))
196 // Get the next N primes where N is determined by the mark in the sieveMarks
197 while((composite = NEXT_PRIME(iter)) != 0)
198 {
199     next = 0;
200     i = count;
201     pList[i--] = composite;
202     for(; i > 0; i--)
203     {
204         next = NEXT_PRIME(iter);
205         pList[i] = next;
206         if(next != 0)
207             composite *= next;
208     }
209     // Get the remainder when dividing the base field address
210     // by the composite
211     composite = BnModWord(bnN, composite);
212     // 'composite' is divisible by the composite components. for each of the
213     // composite components, divide 'composite'. That remainder (r) is used to
214     // pick a starting point for clearing the array. The stride is equal to the
215     // composite component. Note, the field only contains odd numbers. If the
216     // field were expanded to contain all numbers, then half of the bits would
217     // have already been cleared. We can save the trouble of clearing them a
218     // second time by having a stride of 2*next. Or we can take all of the even
219     // numbers out of the field and use a stride of 'next'
220     for(i = count; i > 0; i--)
221     {
222         next = pList[i];
223         if(next == 0)
224             goto done;
225         r = composite % next;
226         // these computations deal with the fact that the field starts at some
227         // arbitrary offset within the number space. If the field were all numbers,
228         // then we would have gone through some number of bit clearings before we
229         // got to the start of this range. We don't know how many there were before,
230         // but we can tell from the remainder whether we are on an even or odd
231         // stride when we hit the beginning of the table. If we are on an odd stride
232         // (r & 1), we would start half a stride in (next - r)/2. If we are on an
233         // even stride, we need 1.5 strides (next + r/2) because the table only has
234         // odd numbers. If the remainder happens to be zero, then the start of the
235         // table is on stride so no adjustment is necessary.
236         if(r & 1)            j = (next - r) / 2;
237         else if(r == 0)      j = 0;
238         else                 j = next - r / 2;
239         for(; j < fieldBits; j += next)
240             ClearBit(j, field, fieldSize);
241     }
242     if(next >= stop)
243     {
244         mark++;
245         count = sieveMarks[mark].count;
246         stop = sieveMarks[mark].prime;
247     }
248 }
249 done:
250 INSTRUMENT_INC(totalFieldsSieved[PrimeIndex]);
251 i = BitsInArray(field, fieldSize);

```

```

252     INSTRUMENT_ADD(bitsInFieldAfterSieve[PrimeIndex], i);
253     INSTRUMENT_ADD(emptyFieldsSieved[PrimeIndex], (i == 0));
254     return i;
255 }
256 #ifndef SIEVE_DEBUG
257 static uint32_t fieldSize = 210;

```

10.2.15.2.6 SetFieldSize()

Function to set the field size used for prime generation. Used for tuning.

```

258 LIB_EXPORT uint32_t
259 SetFieldSize(
260     uint32_t      newFieldSize
261 )
262 {
263     if(newFieldSize == 0 || newFieldSize > MAX_FIELD_SIZE)
264         fieldSize = MAX_FIELD_SIZE;
265     else
266         fieldSize = newFieldSize;
267     return fieldSize;
268 }
269 #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 Returns	Meaning
TPM_RC_FAILURE	TPM in failure mode, probably due to entropy source
TPM_RC_SUCCESS	candidate is probably prime
TPM_RC_NO_RESULT	candidate is not prime and couldn't find an alternative in the field

```

270 LIB_EXPORT TPM_RC
271 PrimeSelectWithSieve(
272     bigNum      candidate,           // IN/OUT: The candidate to filter
273     UINT32      e,                  // IN: the exponent
274     RAND_STATE  *rand               // IN: the random number generator state
275 )
276 {
277     BYTE        field[MAX_FIELD_SIZE];
278     UINT32      first;
279     UINT32      ones;
280     INT32       chosen;
281     BN_PRIME(test);
282     UINT32      modE;
283     #ifndef SIEVE_DEBUG
284     UINT32      fieldSize = MAX_FIELD_SIZE;
285     #endif
286     UINT32      primeSize;
287     //
288     // Adjust the field size and prime table list to fit the size of the prime
289     // being tested. This is done to try to optimize the trade-off between the
290     // dividing done for sieving and the time for Miller-Rabin. When the size

```



```

291 // of the prime is large, the cost of Miller-Rabin is fairly high, as is the
292 // cost of the sieving. However, the time for Miller-Rabin goes up considerably
293 // faster than the cost of dividing by a number of primes.
294 primeSize = BnSizeInBits(candidate);
295
296 if(primeSize <= 512)
297 {
298     RsaAdjustPrimeLimit(1024); // Use just the first 1024 primes
299 }
300 else if(primeSize <= 1024)
301 {
302     RsaAdjustPrimeLimit(4096); // Use just the first 4K primes
303 }
304 else
305 {
306     RsaAdjustPrimeLimit(0); // Use all available
307 }
308
309 // Save the low-order word to use as a search generator and make sure that
310 // it has some interesting range to it
311 first = candidate->d[0] | 0x80000000;
312
313 // Sieve the field
314 ones = PrimeSieve(candidate, fieldSize, field);
315 pAssert(ones > 0 && ones < (fieldSize * 8));
316 for(; ones > 0; ones--)
317 {
318     // Decide which bit to look at and find its offset
319     chosen = FindNthSetBit((UINT16)fieldSize, field, ((first % ones) + 1));
320
321     if((chosen < 0) || (chosen >= (INT32)(fieldSize * 8)))
322         FAIL(FATAL_ERROR_INTERNAL);
323
324     // Set this as the trial prime
325     BnAddWord(test, candidate, (crypt_ushort_t)(chosen * 2));
326
327     // The exponent might not have been one of the tested primes so
328     // make sure that it isn't divisible and make sure that 0 != (p-1) mod e
329     // Note: This is the same as 1 != p mod e
330     modE = BnModWord(test, e);
331     if((modE != 0) && (modE != 1) && MillerRabin(test, rand))
332     {
333         BnCopy(candidate, test);
334         return TPM_RC_SUCCESS;
335     }
336     // Clear the bit just tested
337     ClearBit(chosen, field, fieldSize);
338 }
339 // Ran out of bits and couldn't find a prime in this field
340 INSTRUMENT_INC(noPrimeFields[PrimeIndex]);
341 return (g_inFailureMode ? TPM_RC_FAILURE : TPM_RC_NO_RESULT);
342 }
343 #if RSA_INSTRUMENT
344 static char a[256];
345 char *
346 PrintTuple(
347     UINT32 *i
348 )
349 {
350     sprintf(a, "{%d, %d, %d}", i[0], i[1], i[2]);
351     return a;
352 }
353
354 #define CLEAR_VALUE(x) memset(x, 0, sizeof(x))
355
356 void

```

```

357 RsaSimulationEnd(
358     void
359 )
360 {
361     int            i;
362     UINT32         averages[3];
363     UINT32         nonFirst = 0;
364     if((PrimeCounts[0] + PrimeCounts[1] + PrimeCounts[2]) != 0)
365     {
366         printf("Primes generated = %s\n", PrintTuple(PrimeCounts));
367         printf("Fields sieved = %s\n", PrintTuple(totalFieldsSieved));
368         printf("Fields with no primes = %s\n", PrintTuple(noPrimeFields));
369         printf("Primes checked with Miller-Rabin = %s\n",
370             PrintTuple(MillerRabinTrials));
371         for(i = 0; i < 3; i++)
372             averages[i] = (totalFieldsSieved[i]
373                 != 0 ? bitsInFieldAfterSieve[i] / totalFieldsSieved[i]
374                 : 0);
375         printf("Average candidates in field %s\n", PrintTuple(averages));
376         for(i = 1; i < (sizeof(failedAtIteration) / sizeof(failedAtIteration[0]));
377             i++)
378             nonFirst += failedAtIteration[i];
379         printf("Miller-Rabin failures not in first round = %d\n", nonFirst);
380
381     }
382     CLEAR_VALUE(PrimeCounts);
383     CLEAR_VALUE(totalFieldsSieved);
384     CLEAR_VALUE(noPrimeFields);
385     CLEAR_VALUE(MillerRabinTrials);
386     CLEAR_VALUE(bitsInFieldAfterSieve);
387 }
388
389 LIB_EXPORT void
390 GetSieveStats(
391     uint32_t      *trials,
392     uint32_t      *emptyFields,
393     uint32_t      *averageBits
394 )
395 {
396     uint32_t      totalBits;
397     uint32_t      fields;
398     *trials = MillerRabinTrials[0] + MillerRabinTrials[1] + MillerRabinTrials[2];
399     *emptyFields = noPrimeFields[0] + noPrimeFields[1] + noPrimeFields[2];
400     fields = totalFieldsSieved[0] + totalFieldsSieved[1]
401         + totalFieldsSieved[2];
402     totalBits = bitsInFieldAfterSieve[0] + bitsInFieldAfterSieve[1]
403         + bitsInFieldAfterSieve[2];
404     if(fields != 0)
405         *averageBits = totalBits / fields;
406     else
407         *averageBits = 0;
408     CLEAR_VALUE(PrimeCounts);
409     CLEAR_VALUE(totalFieldsSieved);
410     CLEAR_VALUE(noPrimeFields);
411     CLEAR_VALUE(MillerRabinTrials);
412     CLEAR_VALUE(bitsInFieldAfterSieve);
413 }
414 #endif
415
416 #endif // RSA_KEY_SIEVE
417
418 #if !RSA_INSTRUMENT
419 void
420 RsaSimulationEnd(
421     void

```

```
423     )  
424     {  
425     }  
426     #endif
```

DRAFT

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  const BYTE DRBG_NistTestVector_Entropy[] = {DRBG_TEST_INITIATE_ENTROPY};
4  const BYTE DRBG_NistTestVector_GeneratedInterm[] =
5      {DRBG_TEST_GENERATED_INTERM};
6  const BYTE DRBG_NistTestVector_EntropyReseed[] =
7      {DRBG_TEST_RESEED_ENTROPY};
8  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

```

9  #define      DF_COUNT (DRBG_KEY_SIZE_WORDS / DRBG_IV_SIZE_WORDS + 1)
10 #if DRBG_KEY_SIZE_BITS != 128 && DRBG_KEY_SIZE_BITS != 256
11 #  error "CryptRand.c only written for AES with 128- or 256-bit keys."
12 #endif
13 typedef struct
14 {
15     DRBG_KEY_SCHEDULE    keySchedule;
16     DRBG_IV              iv[DF_COUNT];
17     DRBG_IV              out1;
18     DRBG_IV              buf;
19     int                  contents;
20 } 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.

```

21 static void
22 DfCompute(
23     PDF_STATE          dfState
24 )
25 {
26     int                i;
27     int                iv;
28     crypt_uword_t      *pIv;
29     crypt_uword_t      temp[DRBG_IV_SIZE_WORDS] = {0};
30     //
31     for(iv = 0; iv < DF_COUNT; iv++)
32     {
33         pIv = (crypt_uword_t *)&dfState->iv[iv].words[0];
34         for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
35         {
36             temp[i] ^= pIv[i] ^ dfState->buf.words[i];
37         }
38         DRBG_ENCRYPT(&dfState->keySchedule, &temp, pIv);
39     }
40     for(i = 0; i < DRBG_IV_SIZE_WORDS; i++)
41         dfState->buf.words[i] = 0;
42     dfState->contents = 0;
43 }

```

10.2.16.2.4 DfStart()

This initializes the output blocks with an encrypted counter value and initializes the key schedule.

```

44 static void
45 DfStart(
46     PDF_STATE          dfState,
47     uint32_t           inputLength
48 )
49 {
50     BYTE               init[8];
51     int                i;
52     UINT32             drbgSeedSize = sizeof(DRBG_SEED);
53
54     const BYTE dfKey[DRBG_KEY_SIZE_BYTES] = {
55         0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
56         0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f

```

```

57     #if DRBG_KEY_SIZE_BYTES > 16
58         ,0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
59         0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
60     #endif
61 };
62 memset(dfState, 0, sizeof(DF_STATE));
63 DRBG_ENCRYPT_SETUP(&dfKey[0], DRBG_KEY_SIZE_BITS, &dfState->keySchedule);
64 // Create the first chaining values
65 for(i = 0; i < DF_COUNT; i++)
66     ((BYTE *)&dfState->iv[i])[3] = (BYTE)i;
67 DfCompute(dfState);
68 // initialize the first 64 bits of the IV in a way that doesn't depend
69 // on the size of the words used.
70 UINT32_TO_BYTE_ARRAY(inputLength, init);
71 UINT32_TO_BYTE_ARRAY(drbgSeedSize, &init[4]);
72 memcpy(&dfState->iv[0], init, 8);
73 dfState->contents = 4;
74 }

```

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().

```

75 static void
76 DfUpdate(
77     PDF_STATE      dfState,
78     int             size,
79     const BYTE      *data
80 )
81 {
82     while(size > 0)
83     {
84         int         toFill = DRBG_IV_SIZE_BYTES - dfState->contents;
85         if(size < toFill)
86             toFill = size;
87         // Copy as many bytes as there are or until the state buffer is full
88         memcpy(&dfState->buf.bytes[dfState->contents], data, toFill);
89         // Reduce the size left by the amount copied
90         size -= toFill;
91         // Advance the data pointer by the amount copied
92         data += toFill;
93         // increase the buffer contents count by the amount copied
94         dfState->contents += toFill;
95         pAssert(dfState->contents <= DRBG_IV_SIZE_BYTES);
96         // If we have a full buffer, do a computation pass.
97         if(dfState->contents == DRBG_IV_SIZE_BYTES)
98             DfCompute(dfState);
99     }
100 }

```

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.

```

101 static DRBG_SEED *
102 DfEnd(
103     PDF_STATE      dfState
104 )
105 {

```

```

106     // Since DfCompute is always called when a buffer is full, there is always
107     // space in the buffer for the terminator
108     dfState->buf.bytes[dfState->contents++] = 0x80;
109     // If the buffer is not full, pad with zeros
110     while(dfState->contents < DRBG_IV_SIZE_BYTES)
111         dfState->buf.bytes[dfState->contents++] = 0;
112     // Do a final state update
113     DfCompute(dfState);
114     return (DRBG_SEED *) &dfState->iv;
115 }

```

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();

```

116 static DRBG_SEED *
117 DfBuffer(
118     DRBG_SEED      *output,          // OUT: receives the result
119     int             size,             // IN: size of the buffer to add
120     BYTE            *buf,             // IN: address of the buffer
121 )
122 {
123     DF_STATE        dfState;
124     if(size == 0 || buf == NULL)
125         return NULL;
126     // Initialize the derivation function
127     DfStart(&dfState, size);
128     DfUpdate(&dfState, size, buf);
129     DfEnd(&dfState);
130     memcpy(output, &dfState.iv[0], sizeof(DRBG_SEED));
131     return output;
132 }

```

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

```

133 BOOL
134 DRBG_GetEntropy(
135     UINT32          requiredEntropy,  // IN: requested number of bytes of full
136                                     // entropy
137     BYTE            *entropy         // OUT: buffer to return collected entropy
138 )
139 {
140     #if !USE_DEBUG_RNG
141         UINT32      obtainedEntropy;
142         INT32       returnedEntropy;
143
144         // If in debug mode, always use the self-test values for initialization
145         if(IsSelfTest())
146         {
147             #endif

```



```

149     // If doing simulated DRBG, then check to see if the
150     // entropyFailure condition is being tested
151     if(!IsEntropyBad())
152     {
153         // In self-test, the caller should be asking for exactly the seed
154         // size of entropy.
155         pAssert(requiredEntropy == sizeof(DRBG_NistTestVector_Entropy));
156         memcpy(entropy, DRBG_NistTestVector_Entropy,
157             sizeof(DRBG_NistTestVector_Entropy));
158     }
159 #if !USE_DEBUG_RNG
160 }
161 else if(!IsEntropyBad())
162 {
163     // Collect entropy
164     // Note: In debug mode, the only "entropy" value ever returned
165     // is the value of the self-test vector.
166     for(returnedEntropy = 1, obtainedEntropy = 0;
167         obtainedEntropy < requiredEntropy && !IsEntropyBad();
168         obtainedEntropy += returnedEntropy)
169     {
170         returnedEntropy = _plat__GetEntropy(&entropy[obtainedEntropy],
171             requiredEntropy - obtainedEntropy);
172         if(returnedEntropy <= 0)
173             SetEntropyBad();
174     }
175 }
176 #endif
177 return !IsEntropyBad();
178 }

```

10.2.16.2.9 IncrementIv()

This function increments the IV value by 1. It is used by EncryptDRBG().

```

179 void
180 IncrementIv(
181     DRBG_IV *iv
182 )
183 {
184     BYTE *ivP = ((BYTE *)iv) + DRBG_IV_SIZE_BYTES;
185     while(--ivP >= (BYTE *)iv) && ((*ivP = ((*ivP + 1) & 0xFF)) == 0));
186 }

```

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.

```

187 static BOOL
188 EncryptDRBG(
189     BYTE *dOut,
190     UINT32 dOutBytes,
191     DRBG_KEY_SCHEDULE *keySchedule,
192     DRBG_IV *iv,
193     UINT32 *lastValue // Points to the last output value
194 )
195 {
196 #if FIPS_COMPLIANT
197     // For FIPS compliance, the DRBG has to do a continuous self-test to make sure that
198     // no two consecutive values are the same. This overhead is not incurred if the TPM
199     // is not required to be FIPS compliant
200     //

```

```

201     UINT32          temp[DRBG_IV_SIZE_BYTES / sizeof(UINT32)];
202     int             i;
203     BYTE            *p;
204
205     for(; dOutBytes > 0;)
206     {
207         // Increment the IV before each encryption (this is what makes this
208         // different from normal counter-mode encryption
209         IncrementIv(iv);
210         DRBG_ENCRYPT(keySchedule, iv, temp);
211         // Expect a 16 byte block
212         #if DRBG_IV_SIZE_BITS != 128
213         #error "Unsupported IV size in DRBG"
214         #endif
215         if((lastValue[0] == temp[0])
216             && (lastValue[1] == temp[1])
217             && (lastValue[2] == temp[2])
218             && (lastValue[3] == temp[3])
219             )
220         {
221             LOG_FAILURE(FATAL_ERROR_ENTROPY);
222             return FALSE;
223         }
224         lastValue[0] = temp[0];
225         lastValue[1] = temp[1];
226         lastValue[2] = temp[2];
227         lastValue[3] = temp[3];
228         i = MIN(dOutBytes, DRBG_IV_SIZE_BYTES);
229         dOutBytes -= i;
230         for(p = (BYTE *)temp; i > 0; i--)
231             *dOut++ = *p++;
232     }
233     #else // version without continuous self-test
234     NOT_REFERENCED(lastValue);
235     for(; dOutBytes >= DRBG_IV_SIZE_BYTES;
236         dOut = &dOut[DRBG_IV_SIZE_BYTES], dOutBytes -= DRBG_IV_SIZE_BYTES)
237     {
238         // Increment the IV
239         IncrementIv(iv);
240         DRBG_ENCRYPT(keySchedule, iv, dOut);
241     }
242     // If there is a partial, generate into a block-sized
243     // temp buffer and copy to the output.
244     if(dOutBytes != 0)
245     {
246         BYTE          temp[DRBG_IV_SIZE_BYTES];
247         // Increment the IV
248         IncrementIv(iv);
249         DRBG_ENCRYPT(keySchedule, iv, temp);
250         memcpy(dOut, temp, dOutBytes);
251     }
252     #endif
253     return TRUE;
254 }

```

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.

```

255 static BOOL
256 DRBG_Update(
257     DRBG_STATE          *drbgState,          // IN:OUT state to update
258     DRBG_KEY_SCHEDULE   *keySchedule,        // IN: the key schedule (optional)
259     DRBG_SEED           *providedData        // IN: additional data
260 )
261 {
262     UINT32              i;
263     BYTE                *temp = (BYTE *) &drbgState->seed;
264     DRBG_KEY            *key = pDRBG_KEY(&drbgState->seed);
265     DRBG_IV             *iv = pDRBG_IV(&drbgState->seed);
266     DRBG_KEY_SCHEDULE   localKeySchedule;
267     //
268     pAssert(drbgState->magic == DRBG_MAGIC);
269
270     // If an key schedule was not provided, make one
271     if(keySchedule == NULL)
272     {
273         if(DRBG_ENCRYPT_SETUP((BYTE *)key,
274             DRBG_KEY_SIZE_BITS, &localKeySchedule) != 0)
275         {
276             LOG_FAILURE(FATAL_ERROR_INTERNAL);
277             return FALSE;
278         }
279         keySchedule = &localKeySchedule;
280     }
281     // Encrypt the temp value
282
283     EncryptDRBG(temp, sizeof(DRBG_SEED), keySchedule, iv,
284         drbgState->lastValue);
285     if(providedData != NULL)
286     {
287         BYTE *pP = (BYTE *)providedData;
288         for(i = DRBG_SEED_SIZE_BYTES; i != 0; i--)
289             *temp++ ^= *pP++;
290     }
291     // Since temp points to the input key and IV, we are done and
292     // don't need to copy the resulting 'temp' to drbgState->seed
293     return TRUE;
294 }

```

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

```

295 BOOL
296 DRBG_Reseed(
297     DRBG_STATE          *drbgState,          // IN: the state to update
298     DRBG_SEED           *providedEntropy,    // IN: entropy
299     DRBG_SEED           *additionalData      // IN:
300 )
301 {
302     DRBG_SEED           seed;
303

```

```

304     pAssert((drbgState != NULL) && (drbgState->magic == DRBG_MAGIC));
305
306     if(providedEntropy == NULL)
307     {
308         providedEntropy = &seed;
309         if(!DRBG_GetEntropy(sizeof(DRBG_SEED), (BYTE *)providedEntropy))
310             return FALSE;
311     }
312     if(additionalData != NULL)
313     {
314         unsigned int        i;
315
316         // XOR the provided data into the provided entropy
317         for(i = 0; i < sizeof(DRBG_SEED); i++)
318             ((BYTE *)providedEntropy)[i] ^= ((BYTE *)additionalData)[i];
319     }
320     DRBG_Update(drbgState, NULL, providedEntropy);
321
322     drbgState->reseedCounter = 1;
323
324     return TRUE;
325 }

```

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

```

326  BOOL
327  DRBG_SelfTest(
328      void
329  )
330  {
331      BYTE        buf[sizeof(DRBG_NistTestVector_Generated)];
332      DRBG_SEED    seed;
333      UINT32       i;
334      BYTE        *p;
335      DRBG_STATE    testState;
336      //
337      pAssert(!IsSelfTest());
338
339      SetSelfTest();
340      SetDrbgTested();
341      // Do an instantiate
342      if(!DRBG_Instantiate(&testState, 0, NULL))
343          return FALSE;
344      #if DRBG_DEBUG_PRINT
345          dbgDumpMemBlock(pDRBG_KEY(&testState), DRBG_KEY_SIZE_BYTES,
346                          "Key after Instantiate");
347          dbgDumpMemBlock(pDRBG_IV(&testState), DRBG_IV_SIZE_BYTES,
348                          "Value after Instantiate");
349      #endif
350      if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
351          return FALSE;
352      #if DRBG_DEBUG_PRINT
353          dbgDumpMemBlock(pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
354                          "Key after 1st Generate");
355          dbgDumpMemBlock(pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
356                          "Value after 1st Generate");

```

```

357 #endif
358     if(memcmp(buf, DRBG_NistTestVector_GeneratedInterm, sizeof(buf)) != 0)
359         return FALSE;
360     memcpy(seed.bytes, DRBG_NistTestVector_EntropyReseed, sizeof(seed));
361     DRBG_Reseed(&testState, &seed, NULL);
362 #if DRBG_DEBUG_PRINT
363     dbgDumpMemBlock((BYTE *)pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
364                     "Key after 2nd Generate");
365     dbgDumpMemBlock((BYTE *)pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
366                     "Value after 2nd Generate");
367     dbgDumpMemBlock(buf, sizeof(buf), "2nd Generated");
368 #endif
369     if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
370         return FALSE;
371     if(memcmp(buf, DRBG_NistTestVector_Generated, sizeof(buf)) != 0)
372         return FALSE;
373     ClearSelfTest();
374
375     DRBG_Uninstantiate(&testState);
376     for(p = (BYTE *)&testState, i = 0; i < sizeof(DRBG_STATE); i++)
377     {
378         if(*p++)
379             return FALSE;
380     }
381     // Simulate hardware failure to make sure that we get an error when
382     // trying to instantiate
383     SetEntropyBad();
384     if(DRBG_Instantiate(&testState, 0, NULL))
385         return FALSE;
386     ClearEntropyBad();
387
388     return TRUE;
389 }

```

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. Other functions are only visible to programs in the LtcCryptoEngine().

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 Returns	Meaning
TPM_RC_NO_RESULT	failure of the entropy generator

```

390 LIB_EXPORT TPM_RC
391 CryptRandomStir(
392     UINT16        additionalDataSize,
393     BYTE          *additionalData
394 )
395 {
396 #if !USE_DEBUG_RNG
397     DRBG_SEED     tmpBuf;
398     DRBG_SEED     dfResult;
399 //
400 // All reseed with outside data starts with a buffer full of entropy
401 if(!DRBG_GetEntropy(sizeof(tmpBuf), (BYTE *)&tmpBuf))

```

```

402     return TPM_RC_NO_RESULT;
403
404     DRBG_Reseed(&drbgDefault, &tmpBuf,
405                DfBuffer(&dfResult, additionalDataSize, additionalData));
406     drbgDefault.reseedCounter = 1;
407
408     return TPM_RC_SUCCESS;
409
410 #else
411     // If doing debug, use the input data as the initial setting for the RNG state
412     // so that the test can be reset at any time.
413     // Note: If this is called with a data size of 0 or less, nothing happens. The
414     // presumption is that, in a debug environment, the caller will have specific
415     // values for initialization, so this check is just a simple way to prevent
416     // inadvertent programming errors from screwing things up. This doesn't use an
417     // pAssert() because the non-debug version of this function will accept these
418     // parameters as meaning that there is no additionalData and only hardware
419     // entropy is used.
420     if((additionalDataSize > 0) && (additionalData != NULL))
421     {
422         memset(drbgDefault.seed.bytes, 0, sizeof(drbgDefault.seed.bytes));
423         memcpy(drbgDefault.seed.bytes, additionalData,
424               MIN(additionalDataSize, sizeof(drbgDefault.seed.bytes)));
425     }
426     drbgDefault.reseedCounter = 1;
427
428     return TPM_RC_SUCCESS;
429 #endif
430 }

```

10.2.16.3.3 CryptRandomGenerate()

Generate a *randomSize* number of random bytes.

```

431 LIB_EXPORT UINT16
432 CryptRandomGenerate(
433     INT32      randomSize,
434     BYTE       *buffer
435 )
436 {
437     if(randomSize > UINT16_MAX)
438         randomSize = UINT16_MAX;
439     return DRBG_Generate((RAND_STATE *)&drbgDefault, buffer, (UINT16)randomSize);
440 }

```

10.2.16.3.3.1 DRBG_InstantiateSeededKdf()

This function is used to instantiate a KDF-based RNG. This is used for derivations. This function always returns TRUE.

```
441 LIB_EXPORT BOOL
442 DRBG_InstantiateSeededKdf(
443     KDF_STATE      *state,           // OUT: buffer to hold the state
444     TPM_ALG_ID      hashAlg,         // IN: hash algorithm
445     TPM_ALG_ID      kdf,             // IN: the KDF to use
446     TPM2B            *seed,          // IN: the seed to use
447     const TPM2B      *label,         // IN: a label for the generation process.
448     TPM2B            *context,       // IN: the context value
449     UINT32           limit           // IN: Maximum number of bits from the KDF
450 )
451 {
452     state->magic = KDF_MAGIC;
453     state->limit = limit;
454     state->seed = seed;
455     state->hash = hashAlg;
456     state->kdf = kdf;
457     state->label = label;
458     state->context = context;
459     state->digestSize = CryptHashGetDigestSize(hashAlg);
460     state->counter = 0;
461     state->residual.t.size = 0;
462     return TRUE;
463 }
```


10.2.16.3.3.2 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.

```
464 LIB_EXPORT void
465 DRBG_AdditionalData(
466     DRBG_STATE      *drbgState,      // IN:OUT state to update
467     TPM2B            *additionalData // IN: value to incorporate
468 )
469 {
470     DRBG_SEED        dfResult;
471     if(drbgState->magic == DRBG_MAGIC)
472     {
473         DfBuffer(&dfResult, additionalData->size, additionalData->buffer);
474         DRBG_Reseed(drbgState, &dfResult, NULL);
475     }
476 }
```

10.2.16.3.3.3 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. This function always returns TRUE.

```

477 LIB_EXPORT TPM_RC
478 DRBG_InstantiateSeeded(
479     DRBG_STATE      *drbgState,      // IN/OUT: buffer to hold the state
480     const TPM2B      *seed,          // IN: the seed to use
481     const TPM2B      *purpose,       // IN: a label for the generation process.
482     const TPM2B      *name,          // IN: name of the object
483     const TPM2B      *additional    // IN: additional data
484 )
485 {
486     DF_STATE          dfState;
487     int               totalInputSize;
488     // DRBG should have been tested, but...
489     if(!IsDrbgTested() && !DRBG_SelfTest())
490     {
491         LOG_FAILURE(FATAL_ERROR_SELF_TEST);
492         return TPM_RC_FAILURE;
493     }
494     // Initialize the DRBG state
495     memset(drbgState, 0, sizeof(DRBG_STATE));
496     drbgState->magic = DRBG_MAGIC;
497
498     // Size all of the values
499     totalInputSize = (seed != NULL) ? seed->size : 0;
500     totalInputSize += (purpose != NULL) ? purpose->size : 0;
501     totalInputSize += (name != NULL) ? name->size : 0;
502     totalInputSize += (additional != NULL) ? additional->size : 0;
503
504     // Initialize the derivation
505     DfStart(&dfState, totalInputSize);
506
507     // Run all the input strings through the derivation function
508     if(seed != NULL)
509         DfUpdate(&dfState, seed->size, seed->buffer);
510     if(purpose != NULL)
511         DfUpdate(&dfState, purpose->size, purpose->buffer);
512     if(name != NULL)
513         DfUpdate(&dfState, name->size, name->buffer);
514     if(additional != NULL)
515         DfUpdate(&dfState, additional->size, additional->buffer);
516
517     // Used the derivation function output as the "entropy" input. This is not
518     // how it is described in SP800-90A but this is the equivalent function
519     DRBG_Reseed((DRBG_STATE *)drbgState, DfEnd(&dfState), NULL);
520
521     return TPM_RC_SUCCESS;
522 }

```

10.2.16.3.3.4 CryptRandStartup()

This function is called when TPM_Startup is executed. This function always returns TRUE.

```
523 LIB_EXPORT BOOL
524 CryptRandStartup(
525     void
526 )
527 {
528     #if ! _DRBG_STATE_SAVE
529         // If not saved in NV, re-instantiate on each startup
530         DRBG_Instantiate(&drbgDefault, 0, NULL);
531     #else
532         // If the running state is saved in NV, NV has to be loaded before it can
533         // be updated
534         if(go.drbgState.magic == DRBG_MAGIC)
535             DRBG_Reseed(&go.drbgState, NULL, NULL);
536         else
537             DRBG_Instantiate(&go.drbgState, 0, NULL);
538     #endif
539     return TRUE;
540 }
```

10.2.16.3.3.5 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  }

```

10.2.16.3.4 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 TRUE without generating any bits or updating the reseed counter. This function returns 0 if a reseed is required. Otherwise, it returns the number of bytes produced which could be less than the number requested if the request is too large.

```

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
561      // If the caller used a KDF state, generate a sequence from the KDF not to
562      // exceed the limit.
563      if(state->kdf.magic == KDF_MAGIC)
564      {
565          KDF_STATE      *kdf = (KDF_STATE *) state;
566          UINT32          counter = (UINT32) kdf->counter;
567          INT32           bytesLeft = randomSize;
568
569          if(random == NULL)
570              return 0;
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 // Increment the reseed counter
675 drbgState->reseedCounter += 1;
676 }
677 else
678 {
679     LOG_FAILURE(FATAL_ERROR_INTERNAL);
680     return FALSE;
681 }
682 return randomSize;
683 }
684 }

```

10.2.16.3.5 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 }

```

10.2.16.3.6 DRBG_Uninstantiate()

This is Uninstantiate_function() from [SP 800-90A 9.4].

Error Returns	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  #if ALG_RSA
```

10.2.17.3 Obligatory Initialization Functions

10.2.17.3.1 CryptRsaInit()

Function called at _TPM_Init().

```
4  BOOL
5  CryptRsaInit(
6      void
7  )
8  {
9      return TRUE;
10 }
```

10.2.17.3.2 CryptRsaStartup()

Function called at TPM2_Startup()

```
11 BOOL
12 CryptRsaStartup(
13     void
14 )
15 {
16     return TRUE;
17 }
```

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.

```
18 static privateExponent *
19 RsaInitializeExponent(
20     privateExponent *Z
21 )
22 {
23     bigNum *bn = (bigNum *) &Z->P;
24     int i;
25     //
```

```

26     for(i = 0; i < 5; i++)
27     {
28         bn[i] = (bigNum) &Z->entries[i];
29         BnInit(bn[i], BYTES_TO_CRYPT_WORDS(sizeof(Z->entries[0].d)));
30     }
31     return Z;
32 }

```

10.2.17.4.2 MakePgreaterThanQ()

This function swaps the pointers for P and Q if Q happens to be larger than Q.

```

33 static void
34 MakePgreaterThanQ(
35     privateExponent    *Z
36 )
37 {
38     if(BnUnsignedCmp(Z->P, Z->Q) < 0)
39     {
40         bigNum          bnT = Z->P;
41         Z->P = Z->Q;
42         Z->Q = bnT;
43     }
44 }

```

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 , $qInv$ 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

```

45 static BOOL
46 PackExponent(
47     TPM2B_PRIVATE_KEY_RSA    *packed,
48     privateExponent          *Z
49 )
50 {
51     int                i;
52     UINT16             primeSize = (UINT16)BITS_TO_BYTES(BnMsb(Z->P));
53     UINT16             pS = primeSize;
54     //
55     pAssert((primeSize * 5) <= sizeof(packed->t.buffer));
56     packed->t.size = (primeSize * 5) + RSA_prime_flag;
57     for(i = 0; i < 5; i++)
58         if(!BnToBytes((bigNum) &Z->entries[i], &packed->t.buffer[primeSize * i], &pS))
59             return FALSE;
60     if(pS != primeSize)
61         return FALSE;
62     return TRUE;
63 }

```

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

```

64  static BOOL
65  UnpackExponent(
66      TPM2B_PRIVATE_KEY_RSA    *b,
67      privateExponent          *Z
68  )
69  {
70      UINT16                    primeSize = b->t.size & ~RSA_prime_flag;
71      int                       i;
72      bigNum                    *bn = &Z->P;
73  //
74      VERIFY(b->t.size & RSA_prime_flag);
75      RsaInitializeExponent(Z);
76      VERIFY((primeSize % 5) == 0);
77      primeSize /= 5;
78      for(i = 0; i < 5; i++)
79          VERIFY(BnFromBytes(bn[i], &b->t.buffer[primeSize * i], primeSize)
80              != NULL);
81      MakePgreaterThanQ(Z);
82      return TRUE;
83  Error:
84      return FALSE;
85  }

```

10.2.17.4.5 ComputePrivateExponent()

This function computes the private exponent from the primes.

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure

```

86  static BOOL
87  ComputePrivateExponent(
88      bigNum                    pubExp,          // IN: the public exponent
89      privateExponent          *Z               // IN/OUT: on input, has primes P and Q. On
90                                              // output, has P, Q, dP, dQ, and pInv
91  )
92  {
93      BOOL                      pOK;
94      BOOL                      qOK;
95      BN_PRIME(pT);
96  //
97      // make p the larger value so that m2 is always less than p
98      MakePgreaterThanQ(Z);
99
100     //dP = (1/e) mod (p-1)
101     pOK = BnSubWord(pT, Z->P, 1);
102     pOK = pOK && BnModInverse(Z->dP, pubExp, pT);
103     //dQ = (1/e) mod (q-1)
104     qOK = BnSubWord(pT, Z->Q, 1);
105     qOK = qOK && BnModInverse(Z->dQ, pubExp, pT);
106     // qInv = (1/q) mod p

```

```

107     if(pOK && qOK)
108         pOK = qOK = BnModInverse(Z->qInv, Z->Q, Z->P);
109     if(!pOK)
110         BnSetWord(Z->P, 0);
111     if(!qOK)
112         BnSetWord(Z->Q, 0);
113     return pOK && qOK;
114 }

```

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

```

115 static BOOL
116 RsaPrivateKeyOp(
117     bigNum          inOut, // IN/OUT: number to be exponentiated
118     privateExponent *Z
119 )
120 {
121     BN_RSA(M1);
122     BN_RSA(M2);
123     BN_RSA(M);
124     BN_RSA(H);
125     //
126     MakePgreaterThanQ(Z);
127     // m1 = cdP mod p
128     VERIFY(BnModExp(M1, inOut, Z->dP, Z->P));
129     // m2 = cdQ mod q
130     VERIFY(BnModExp(M2, inOut, Z->dQ, Z->Q));
131     // h = qInv * (m1 - m2) mod p = qInv * (m1 + P - m2) mod P because Q < P
132     // so m2 < P
133     VERIFY(BnSub(H, Z->P, M2));
134     VERIFY(BnAdd(H, H, M1));
135     VERIFY(BnModMult(H, H, Z->qInv, Z->P));
136     // m = m2 + h * q
137     VERIFY(BnMult(M, H, Z->Q));
138     VERIFY(BnAdd(inOut, M2, M));
139     return TRUE;
140 Error:
141     return FALSE;
142 }

```

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 Returns	Meaning
TPM_RC_VALUE	number to exponentiate is larger than the modulus

```

143 static TPM_RC
144 RSAEP(
145     TPM2B          *dInOut, // IN: size of the encrypted block and the size of
146                             // the encrypted value. It must be the size of

```

```

147                                     // the modulus.
148                                     // OUT: the encrypted data. Will receive the
149                                     // decrypted value
150     OBJECT      *key                // IN: the key to use
151 )
152 {
153     TPM2B_TYPE(4BYTES, 4);
154     TPM2B_4BYTES e2B;
155     UINT32      e = key->publicArea.parameters.rsaDetail.exponent;
156 //
157     if(e == 0)
158         e = RSA_DEFAULT_PUBLIC_EXPONENT;
159     UINT32_TO_BYTE_ARRAY(e, e2B.t.buffer);
160     e2B.t.size = 4;
161     return ModExpB(dInOut->size, dInOut->buffer, dInOut->size, dInOut->buffer,
162                   e2B.t.size, e2B.t.buffer, key->publicArea.unique.rsa.t.size,
163                   key->publicArea.unique.rsa.t.buffer);
164 }

```

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 Returns	Meaning
TPM_RC_SIZE	the value to decrypt is larger than the modulus

```

165 static TPM_RC
166 RSADP(
167     TPM2B      *inOut,          // IN/OUT: the value to encrypt
168     OBJECT      *key            // IN: the key
169 )
170 {
171     BN_RSA_INITIALIZED(bnM, inOut);
172     NEW_PRIVATE_EXPONENT(Z);
173     if(UndersignedCompareB(inOut->size, inOut->buffer,
174                           key->publicArea.unique.rsa.t.size,
175                           key->publicArea.unique.rsa.t.buffer) >= 0)
176         return TPM_RC_SIZE;
177     // private key operation requires that private exponent be loaded
178     // During self-test, this might not be the case so load it up if it hasn't
179     // already done
180     // been done
181     if((key->sensitive.sensitive.rsa.t.size & RSA_prime_flag) == 0)
182     {
183         if(CryptRsaLoadPrivateExponent(&key->publicArea, &key->sensitive)
184           != TPM_RC_SUCCESS)
185             return TPM_RC_BINDING;
186     }
187     VERIFY(UnpackExponent(&key->sensitive.sensitive.rsa, Z));
188     VERIFY(RsaPrivateKeyOp(bnM, Z));
189     VERIFY(BnTo2B(bnM, inOut, inOut->size));
190     return TPM_RC_SUCCESS;
191 Error:
192     return TPM_RC_FAILURE;
193 }

```

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 Returns	Meaning
TPM_RC_VALUE	<i>hashAlg</i> is not valid or message size is too large

```

194 static TPM_RC
195 OaepEncode(
196     TPM2B      *padded,          // OUT: the pad data
197     TPM_ALG_ID hashAlg,          // IN: algorithm to use for padding
198     const TPM2B *label,          // IN: null-terminated string (may be NULL)
199     TPM2B      *message,         // IN: the message being padded
200     RAND_STATE *rand             // IN: the random number generator to use
201 )
202 {
203     INT32      padLen;
204     INT32      dbSize;
205     INT32      i;
206     BYTE       mySeed[MAX_DIGEST_SIZE];
207     BYTE       *seed = mySeed;
208     INT32      hLen = CryptHashGetDigestSize(hashAlg);
209     BYTE       mask[MAX_RSA_KEY_BYTES];
210     BYTE       *pp;
211     BYTE       *pm;
212     TPM_RC     retVal = TPM_RC_SUCCESS;
213
214     pAssert(padded != NULL && message != NULL);
215
216     // A value of zero is not allowed because the KDF can't produce a result
217     // if the digest size is zero.
218     if(hLen <= 0)
219         return TPM_RC_VALUE;
220
221     // Basic size checks
222     // make sure digest isn't too big for key size
223     if(padded->size < (2 * hLen) + 2)
224         ERROR_RETURN(TPM_RC_HASH);
225
226     // and that message will fit messageSize <= k - 2hLen - 2
227     if(message->size > (padded->size - (2 * hLen) - 2))
228         ERROR_RETURN(TPM_RC_VALUE);
229
230     // Hash L even if it is null
231     // Offset into padded leaving room for masked seed and byte of zero
232     pp = &padded->buffer[hLen + 1];
233     if(CryptHashBlock(hashAlg, label->buffer, (BYTE *)label->buffer,
234         hLen, pp) != hLen)
235         ERROR_RETURN(TPM_RC_FAILURE);
236
237     // concatenate PS of k mLen 2hLen 2
238     padLen = padded->size - message->size - (2 * hLen) - 2;
239     MemorySet(&pp[hLen], 0, padLen);
240     pp[hLen + padLen] = 0x01;
241     padLen += 1;
242     memcpy(&pp[hLen + padLen], message->buffer, message->size);
243
244     // The total size of db = hLen + pad + mSize;
245     dbSize = hLen + padLen + message->size;
246
247     // If testing, then use the provided seed. Otherwise, use values
248     // from the RNG
249     CryptRandomGenerate(hLen, mySeed);

```

```

250     DRBG_Generate(rand, mySeed, (UINT16)hLen);
251     if(g_inFailureMode)
252         ERROR_RETURN(TPM_RC_FAILURE);
253     // mask = MGF1 (seed, nSize hLen 1)
254     CryptMGF1(dbSize, mask, hashAlg, hLen, seed);
255
256     // Create the masked db
257     pm = mask;
258     for(i = dbSize; i > 0; i--)
259         *pp++ ^= *pm++;
260     pp = &padded->buffer[hLen + 1];
261
262     // Run the masked data through MGF1
263     if(CryptMGF1(hLen, &padded->buffer[1], hashAlg, dbSize, pp) != (unsigned)hLen)
264         ERROR_RETURN(TPM_RC_VALUE);
265 // Now XOR the seed to create masked seed
266 pp = &padded->buffer[1];
267 pm = seed;
268 for(i = hLen; i > 0; i--)
269     *pp++ ^= *pm++;
270 // Set the first byte to zero
271 padded->buffer[0] = 0x00;
272 Exit:
273     return retVal;
274 }

```

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 Returns	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

```

275 static TPM_RC
276 OaepDecode(
277     TPM2B          *dataOut,          // OUT: the recovered data
278     TPM_ALG_ID     hashAlg,          // IN: algorithm to use for padding
279     const TPM2B    *label,           // IN: null-terminated string (may be NULL)
280     TPM2B          *padded           // IN: the padded data
281 )
282 {
283     UINT32          i;
284     BYTE            seedMask[MAX_DIGEST_SIZE];
285     UINT32          hLen = CryptHashGetDigestSize(hashAlg);
286
287     BYTE            mask[MAX_RSA_KEY_BYTES];
288     BYTE            *pp;
289     BYTE            *pm;
290     TPM_RC          retVal = TPM_RC_SUCCESS;
291
292     // Strange size (anything smaller can't be an OAEP padded block)
293     // Also check for no leading 0
294     if((padded->size < (unsigned)((2 * hLen) + 2)) || (padded->buffer[0] != 0))
295         ERROR_RETURN(TPM_RC_VALUE);
296 // Use the hash size to determine what to put through MGF1 in order
297 // to recover the seedMask
298     CryptMGF1(hLen, seedMask, hashAlg, padded->size - hLen - 1,
299         &padded->buffer[hLen + 1]);

```



```

300
301 // Recover the seed into seedMask
302 pAssert(hLen <= sizeof(seedMask));
303 pp = &padded->buffer[1];
304 pm = seedMask;
305 for(i = hLen; i > 0; i--)
306     *pm++ ^= *pp++;
307
308 // Use the seed to generate the data mask
309 CryptMGF1(padded->size - hLen - 1, mask, hashAlg, hLen, seedMask);
310
311 // Use the mask generated from seed to recover the padded data
312 pp = &padded->buffer[hLen + 1];
313 pm = mask;
314 for(i = (padded->size - hLen - 1); i > 0; i--)
315     *pm++ ^= *pp++;
316
317 // Make sure that the recovered data has the hash of the label
318 // Put trial value in the seed mask
319 if((CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
320                     hLen, seedMask)) != hLen)
321     FAIL(FATAL_ERROR_INTERNAL);
322 if(memcmp(seedMask, mask, hLen) != 0)
323     ERROR_RETURN(TPM_RC_VALUE);
324
325 // find the start of the data
326 pm = &mask[hLen];
327 for(i = (UINT32)padded->size - (2 * hLen) - 1; i > 0; i--)
328 {
329     if(*pm++ != 0)
330         break;
331 }
332 // If we ran out of data or didn't end with 0x01, then return an error
333 if(i == 0 || pm[-1] != 0x01)
334     ERROR_RETURN(TPM_RC_VALUE);
335
336 // pm should be pointing at the first part of the data
337 // and i is one greater than the number of bytes to move
338 i--;
339 if(i > dataOut->size)
340     // Special exit to preserve the size of the output buffer
341     return TPM_RC_VALUE;
342 memcpy(dataOut->buffer, pm, i);
343 dataOut->size = (UINT16)i;
344 Exit:
345 if(retVal != TPM_RC_SUCCESS)
346     dataOut->size = 0;
347 return retVal;
348 }

```

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 Returns	Meaning
TPM_RC_VALUE	message size is too large

```

349 static TPM_RC
350 RSAES_PKCS1v1_5Encode(
351     TPM2B      *padded,           // OUT: the pad data
352     TPM2B      *message,         // IN: the message being padded
353     RAND_STATE *rand
354 )

```

```

355 {
356     UINT32      ps = padded->size - message->size - 3;
357     //
358     if(message->size > padded->size - 11)
359         return TPM_RC_VALUE;
360     // move the message to the end of the buffer
361     memcpy(&padded->buffer[padded->size - message->size], message->buffer,
362           message->size);
363     // Set the first byte to 0x00 and the second to 0x02
364     padded->buffer[0] = 0;
365     padded->buffer[1] = 2;
366
367     // Fill with random bytes
368     DRBG_Generate(rand, &padded->buffer[2], (UINT16)ps);
369     if(g_inFailureMode)
370         return TPM_RC_FAILURE;
371
372     // Set the delimiter for the random field to 0
373     padded->buffer[2 + ps] = 0;
374
375     // Now, the only messy part. Make sure that all the 'ps' bytes are non-zero
376     // In this implementation, use the value of the current index
377     for(ps++; ps > 1; ps--)
378     {
379         if(padded->buffer[ps] == 0)
380             padded->buffer[ps] = 0x55; // In the < 0.5% of the cases that the
381                                         // random value is 0, just pick a value to
382                                         // put into the spot.
383     }
384     return TPM_RC_SUCCESS;
385 }

```

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 Returns	Meaning
TPM_RC_FAIL	decoding error or results would no fit into provided buffer

```

386 static TPM_RC
387 RSAES_Decode(
388     TPM2B      *message, // OUT: the recovered message
389     TPM2B      *coded    // IN: the encoded message
390 )
391 {
392     BOOL      fail = FALSE;
393     UINT16    pSize;
394
395     fail = (coded->size < 11);
396     fail = (coded->buffer[0] != 0x00) | fail;
397     fail = (coded->buffer[1] != 0x02) | fail;
398     for(pSize = 2; pSize < coded->size; pSize++)
399     {
400         if(coded->buffer[pSize] == 0)
401             break;
402     }
403     pSize++;
404
405     // Make sure that pSize has not gone over the end and that there are at least 8
406     // bytes of pad data.
407     fail = (pSize > coded->size) | fail;
408     fail = ((pSize - 2) < 8) | fail;
409     if((message->size < (UINT16)(coded->size - pSize)) || fail)

```

```

410     return TPM_RC_VALUE;
411     message->size = coded->size - pSize;
412     memcpy(message->buffer, &coded->buffer[pSize], coded->size - pSize);
413     return TPM_RC_SUCCESS;
414 }

```

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.

```

415 INT16
416 CryptRsaPssSaltSize(
417     INT16      hashSize,
418     INT16      outSize
419 )
420 {
421     INT16      saltSize;
422     //
423     // (Mask Length) = (outSize - hashSize - 1);
424     // Max saltSize is (Mask Length) - 1
425     saltSize = (outSize - hashSize - 1) - 1;
426     // Use the maximum salt size allowed by FIPS 186-4
427     if(saltSize > hashSize)
428         saltSize = hashSize;
429     else if(saltSize < 0)
430         saltSize = 0;
431     return saltSize;
432 }

```

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.

```

433 static TPM_RC
434 PssEncode(
435     TPM2B      *out,           // OUT: the encoded buffer
436     TPM_ALG_ID hashAlg,       // IN: hash algorithm for the encoding
437     TPM2B      *digest,       // IN: the digest
438     RAND_STATE *rand,         // IN: random number source
439 )
440 {
441     UINT32      hLen = CryptHashGetDigestSize(hashAlg);
442     BYTE        salt[MAX_RSA_KEY_BYTES - 1];
443     UINT16      saltSize;
444     BYTE        *ps = salt;
445     BYTE        *pOut;
446     UINT16      mLen;
447     HASH_STATE  hashState;
448
449     // These are fatal errors indicating bad TPM firmware
450     pAssert(out != NULL && hLen > 0 && digest != NULL);
451
452     // Get the size of the mask
453     mLen = (UINT16)(out->size - hLen - 1);
454
455     // Set the salt size
456     saltSize = CryptRsaPssSaltSize((INT16)hLen, (INT16)out->size);
457
458     //using eOut for scratch space

```

```

459     // Set the first 8 bytes to zero
460     pOut = out->buffer;
461     memset(pOut, 0, 8);
462
463     // Get set the salt
464     DRBG_Generate(rand, salt, saltSize);
465     if(g_inFailureMode)
466         return TPM_RC_FAILURE;
467
468     // Create the hash of the pad || input hash || salt
469     CryptHashStart(&hashState, hashAlg);
470     CryptDigestUpdate(&hashState, 8, pOut);
471     CryptDigestUpdate2B(&hashState, digest);
472     CryptDigestUpdate(&hashState, saltSize, salt);
473     CryptHashEnd(&hashState, hLen, &pOut[out->size - hLen - 1]);
474
475     // Create a mask
476     if(CryptMGF1(mLen, pOut, hashAlg, hLen, &pOut[mLen]) != mLen)
477         FAIL(FATAL_ERROR_INTERNAL);
478
479     // Since this implementation uses key sizes that are all even multiples of
480     // 8, just need to make sure that the most significant bit is CLEAR
481     *pOut &= 0x7f;
482
483     // Before we mess up the pOut value, set the last byte to 0xbc
484     pOut[out->size - 1] = 0xbc;
485
486     // XOR a byte of 0x01 at the position just before where the salt will be XOR'ed
487     pOut = &pOut[mLen - saltSize - 1];
488     *pOut++ ^= 0x01;
489
490     // XOR the salt data into the buffer
491     for(; saltSize > 0; saltSize--)
492         *pOut++ ^= *ps++;
493
494     // and we are done
495     return TPM_RC_SUCCESS;
496 }

```

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 Returns	Meaning
TPM_RC_SCHEME	<i>hashAlg</i> is not a supported hash algorithm
TPM_RC_VALUE	decode operation failed

```

497 static TPM_RC
498 PssDecode(
499     TPM_ALG_ID    hashAlg,           // IN: hash algorithm to use for the encoding
500     TPM2B         *dIn,              // IN: the digest to compare
501     TPM2B         *eIn,              // IN: the encoded data
502 )
503 {
504     UINT32        hLen = CryptHashGetDigestSize(hashAlg);
505     BYTE          mask[MAX_RSA_KEY_BYTES];

```

```

506     BYTE          *pm = mask;
507     BYTE          *pe;
508     BYTE          pad[8] = {0};
509     UINT32        i;
510     UINT32        mLen;
511     BYTE          fail;
512     TPM_RC        retVal = TPM_RC_SUCCESS;
513     HASH_STATE    hashState;
514
515     // These errors are indicative of failures due to programmer error
516     pAssert(dIn != NULL && eIn != NULL);
517     pe = eIn->buffer;
518
519     // check the hash scheme
520     if(hLen == 0)
521         ERROR_RETURN(TPM_RC_SCHEME);
522
523     // most significant bit must be zero
524     fail = pe[0] & 0x80;
525
526     // last byte must be 0xbc
527     fail |= pe[eIn->size - 1] ^ 0xbc;
528
529     // Use the hLen bytes at the end of the buffer to generate a mask
530     // Doesn't start at the end which is a flag byte
531     mLen = eIn->size - hLen - 1;
532     CryptMGF1(mLen, mask, hashAlg, hLen, &pe[mLen]);
533
534     // Clear the MSO of the mask to make it consistent with the encoding.
535     mask[0] &= 0x7F;
536
537     pAssert(mLen <= sizeof(mask));
538     // XOR the data into the mask to recover the salt. This sequence
539     // advances eIn so that it will end up pointing to the seed data
540     // which is the hash of the signature data
541     for(i = mLen; i > 0; i--)
542         *pm++ ^= *pe++;
543
544     // Find the first byte of 0x01 after a string of all 0x00
545     for(pm = mask, i = mLen; i > 0; i--)
546     {
547         if(*pm == 0x01)
548             break;
549         else
550             fail |= *pm++;
551     }
552     // i should not be zero
553     fail |= (i == 0);
554
555     // if we have failed, will continue using the entire mask as the salt value so
556     // that the timing attacks will not disclose anything (I don't think that this
557     // is a problem for TPM applications but, usually, we don't fail so this
558     // doesn't cost anything).
559     if(fail)
560     {
561         i = mLen;
562         pm = mask;
563     }
564     else
565     {
566         pm++;
567         i--;
568     }
569     // i contains the salt size and pm points to the salt. Going to use the input
570     // hash and the seed to recreate the hash in the lower portion of eIn.
571     CryptHashStart(&hashState, hashAlg);

```

```

572
573 // add the pad of 8 zeros
574 CryptDigestUpdate(&hashState, 8, pad);
575
576 // add the provided digest value
577 CryptDigestUpdate(&hashState, dIn->size, dIn->buffer);
578
579 // and the salt
580 CryptDigestUpdate(&hashState, i, pm);
581
582 // get the result
583 fail |= (CryptHashEnd(&hashState, hLen, mask) != hLen);
584
585 // Compare all bytes
586 for(pm = mask; hLen > 0; hLen--)
587     // don't use fail = because that could skip the increment and compare
588     // operations after the first failure and that gives away timing
589     // information.
590     fail |= *pm++ ^ *pe++;
591
592 retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
593 Exit:
594     return retVal;
595 }

```

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

```

596 INT16
597 MakeDerTag(
598     TPM_ALG_ID    hashAlg,
599     INT16         sizeofBuffer,
600     BYTE          *buffer
601 )
602 {
603     // 0x30, 0x31, // SEQUENCE (2 elements) 1st
604     // 0x30, 0x0D, // SEQUENCE (2 elements)
605     // 0x06, 0x09, // HASH OID
606     // 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01,
607     // 0x05, 0x00, // NULL
608     // 0x04, 0x20 // OCTET STRING
609     HASH_DEF *info = CryptGetHashDef(hashAlg);
610     INT16 oidSize;
611     // If no OID, can't do encode
612     VERIFY(info != NULL);
613     oidSize = 2 + (info->OID)[1];
614     // make sure this fits in the buffer
615     VERIFY(sizeofBuffer >= (oidSize + 8));
616     *buffer++ = 0x30; // 1st SEQUENCE
617     // Size of the 1st SEQUENCE is 6 bytes + size of the hash OID + size of the
618     // digest size
619     *buffer++ = (BYTE) (6 + oidSize + info->digestSize); //
620     *buffer++ = 0x30; // 2nd SEQUENCE
621     // size is 4 bytes of overhead plus the side of the OID
622     *buffer++ = (BYTE) (2 + oidSize);
623     MemoryCopy(buffer, info->OID, oidSize);
624     buffer += oidSize;

```

```

625     *buffer++ = 0x05;    // Add a NULL
626     *buffer++ = 0x00;
627
628     *buffer++ = 0x04;
629     *buffer++ = (BYTE) (info->digestSize);
630     return oidSize + 8;
631 Error:
632     return 0;
633
634 }

```

10.2.17.4.17 RSASSA_Encode()

Encode a message using PKCS1v1.5 method.

Error Returns	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>

```

635 static TPM_RC
636 RSASSA_Encode(
637     TPM2B          *pOut,          // IN:OUT on in, the size of the public key
638                                     // on out, the encoded area
639     TPM_ALG_ID      hashAlg,       // IN: hash algorithm for PKCS1v1_5
640     TPM2B          *hIn           // IN: digest value to encode
641 )
642 {
643     BYTE            DER[20];
644     BYTE            *der = DER;
645     INT32           derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
646     BYTE            *eOut;
647     INT32           fillSize;
648     TPM_RC          retVal = TPM_RC_SUCCESS;
649
650     // Can't use this scheme if the algorithm doesn't have a DER string defined.
651     if(derSize == 0)
652         ERROR_RETURN(TPM_RC_SCHEME);
653
654     // If the digest size of 'hashAlg' doesn't match the input digest size, then
655     // the DER will misidentify the digest so return an error
656     if(CryptHashGetDigestSize(hashAlg) != hIn->size)
657         ERROR_RETURN(TPM_RC_VALUE);
658     fillSize = pOut->size - derSize - hIn->size - 3;
659     eOut = pOut->buffer;
660
661     // Make sure that this combination will fit in the provided space
662     if(fillSize < 8)
663         ERROR_RETURN(TPM_RC_SIZE);
664
665     // Start filling
666     *eOut++ = 0; // initial byte of zero
667     *eOut++ = 1; // byte of 0x01
668     for(; fillSize > 0; fillSize--)
669         *eOut++ = 0xff; // bunch of 0xff
670     *eOut++ = 0; // another 0
671     for(; derSize > 0; derSize--)
672         *eOut++ = *der++; // copy the DER
673     der = hIn->buffer;
674     for(fillSize = hIn->size; fillSize > 0; fillSize--)
675         *eOut++ = *der++; // copy the hash
676 Exit:

```



```

677     return retVal;
678 }

```

10.2.17.4.18 RSASSA_Decode()

This function performs the RSASSA decoding of a signature.

Error Returns	Meaning
TPM_RC_VALUE	decode unsuccessful
TPM_RC_SCHEME	<i>hashAlg</i> is not supported

```

679 static TPM_RC
680 RSASSA_Decode(
681     TPM_ALG_ID    hashAlg,        // IN: hash algorithm to use for the encoding
682     TPM2B         *hIn,           // In: the digest to compare
683     TPM2B         *eIn,           // IN: the encoded data
684 )
685 {
686     BYTE          fail;
687     BYTE          DER[20];
688     BYTE          *der = DER;
689     INT32         derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
690     BYTE          *pe;
691     INT32         hashSize = CryptHashGetDigestSize(hashAlg);
692     INT32         fillSize;
693     TPM_RC        retVal;
694     BYTE          *digest;
695     UINT16        digestSize;
696
697     pAssert(hIn != NULL && eIn != NULL);
698     pe = eIn->buffer;
699
700     // Can't use this scheme if the algorithm doesn't have a DER string
701     // defined or if the provided hash isn't the right size
702     if(derSize == 0 || (unsigned)hashSize != hIn->size)
703         ERROR_RETURN(TPM_RC_SCHEME);
704
705     // Make sure that this combination will fit in the provided space
706     // Since no data movement takes place, can just walk through this
707     // and accept nearly random values. This can only be called from
708     // CryptValidateSignature() so eInSize is known to be in range.
709     fillSize = eIn->size - derSize - hashSize - 3;
710
711     // Start checking (fail will become non-zero if any of the bytes do not have
712     // the expected value.
713     fail = *pe++; // initial byte of zero
714     fail |= *pe++ ^ 1; // byte of 0x01
715     for(; fillSize > 0; fillSize--)
716         fail |= *pe++ ^ 0xff; // bunch of 0xff
717     fail |= *pe++; // another 0
718     for(; derSize > 0; derSize--)
719         fail |= *pe++ ^ *der++; // match the DER
720     digestSize = hIn->size;
721     digest = hIn->buffer;
722     for(; digestSize > 0; digestSize--)
723         fail |= *pe++ ^ *digest++; // match the hash
724     retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
725 Exit:
726     return retVal;
727 }

```

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.

```

728 TPMT_RSA_DECRYPT*
729 CryptRsaSelectScheme(
730     TPMI_DH_OBJECT      rsaHandle,      // IN: handle of an RSA key
731     TPMT_RSA_DECRYPT     *scheme        // IN: a sign or decrypt scheme
732 )
733 {
734     OBJECT              *rsaObject;
735     TPMT_ASYM_SCHEME    *keyScheme;
736     TPMT_RSA_DECRYPT     *retVal = NULL;
737
738     // Get sign object pointer
739     rsaObject = HandleToObject(rsaHandle);
740     keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;
741
742     // if the default scheme of the object is TPM_ALG_NULL, then select the
743     // input scheme
744     if(keyScheme->scheme == TPM_ALG_NULL)
745     {
746         retVal = scheme;
747     }
748     // if the object scheme is not TPM_ALG_NULL and the input scheme is
749     // TPM_ALG_NULL, then select the default scheme of the object.
750     else if(scheme->scheme == TPM_ALG_NULL)
751     {
752         // if input scheme is NULL
753         retVal = (TPMT_RSA_DECRYPT *)keyScheme;
754     }
755     // get here if both the object scheme and the input scheme are
756     // not TPM_ALG_NULL. Need to insure that they are the same.
757     // IMPLEMENTATION NOTE: This could cause problems if future versions have
758     // schemes that have more values than just a hash algorithm. A new function
759     // (IsSchemeSame()) might be needed then.
760     else if(keyScheme->scheme == scheme->scheme
761             && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
762     {
763         retVal = scheme;
764     }
765     // two different, incompatible schemes specified will return NULL
766     return retVal;
767 }
```

10.2.17.5.2 CryptRsaLoadPrivateExponent()

This function is called to generate the private exponent of an RSA key.

Error Returns	Meaning
TPM_RC_BINDING	public and private parts of <i>rsaKey</i> are not matched

```

768 TPM_RC
769 CryptRsaLoadPrivateExponent(
770     TPMT_PUBLIC          *publicArea,
771     TPMT_SENSITIVE       *sensitive
772 )
773 {
774     //
775     if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) == 0)
776     {
777         if((sensitive->sensitive.rsa.t.size * 2) == publicArea->unique.rsa.t.size)
778         {
779             NEW_PRIVATE_EXPONENT(Z);
780             BN_RSA_INITIALIZED(bnN, &publicArea->unique.rsa);
781             BN_RSA(bnQr);
782             BN_VAR(bnE, RADIX_BITS);
783
784             TEST(ALG_NULL_VALUE);
785
786             VERIFY((sensitive->sensitive.rsa.t.size * 2)
787                 == publicArea->unique.rsa.t.size);
788             // Initialize the exponent
789             BnSetWord(bnE, publicArea->parameters.rsaDetail.exponent);
790             if(BnEqualZero(bnE))
791                 BnSetWord(bnE, RSA_DEFAULT_PUBLIC_EXPONENT);
792             // Convert first prime to 2B
793             VERIFY(BnFrom2B(Z->P, &sensitive->sensitive.rsa.b) != NULL);
794
795             // Find the second prime by division. This uses 'bQ' rather than Z->Q
796             // because the division could make the quotient larger than a prime during
797             // some intermediate step.
798             VERIFY(BnDiv(Z->Q, bnQr, bnN, Z->P));
799             VERIFY(BnEqualZero(bnQr));
800             // Compute the private exponent and return it if found
801             VERIFY(ComputePrivateExponent(bnE, Z));
802             VERIFY(PackExponent(&sensitive->sensitive.rsa, Z));
803         }
804         else
805             VERIFY(((sensitive->sensitive.rsa.t.size / 5) * 2)
806                 == publicArea->unique.rsa.t.size);
807         sensitive->sensitive.rsa.t.size |= RSA_prime_flag;
808     }
809     return TPM_RC_SUCCESS;
810 Error:
811     return TPM_RC_BINDING;
812 }

```

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 Returns	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

```

813 LIB_EXPORT TPM_RC
814 CryptRsaEncrypt(
815     TPM2B_PUBLIC_KEY_RSA    *cOut,           // OUT: the encrypted data
816     TPM2B                    *dIn,           // IN: the data to encrypt
817     OBJECT                   *key,           // IN: the key used for encryption
818     TPMT_RSA_DECRYPT          *scheme,        // IN: the type of padding and hash
819                                     // if needed
820     const TPM2B              *label,         // IN: in case it is needed
821     RAND_STATE               *rand           // IN: random number generator
822                                     // state (mostly for testing)
823 )
824 {
825     TPM_RC                    retVal = TPM_RC_SUCCESS;
826     TPM2B_PUBLIC_KEY_RSA     dataIn;
827 //
828 // if the input and output buffers are the same, copy the input to a scratch
829 // buffer so that things don't get messed up.
830 if(dIn == &cOut->b)
831 {
832     MemoryCopy2B(&dataIn.b, dIn, sizeof(dataIn.t.buffer));
833     dIn = &dataIn.b;
834 }
835 // All encryption schemes return the same size of data
836 cOut->t.size = key->publicArea.unique.rsa.t.size;
837 TEST(scheme->scheme);
838
839 switch(scheme->scheme)
840 {
841     case ALG_NULL_VALUE: // 'raw' encryption
842     {
843         INT32 i;
844         INT32 dSize = dIn->size;
845         // dIn can have more bytes than cOut as long as the extra bytes
846         // are zero. Note: the more significant bytes of a number in a byte
847         // buffer are the bytes at the start of the array.
848         for(i = 0; (i < dSize) && (dIn->buffer[i] == 0); i++);
849         dSize -= i;
850         if(dSize > cOut->t.size)
851             ERROR_RETURN(TPM_RC_VALUE);
852         // Pad cOut with zeros if dIn is smaller
853         memset(cOut->t.buffer, 0, cOut->t.size - dSize);
854         // And copy the rest of the value
855         memcpy(&cOut->t.buffer[cOut->t.size - dSize], &dIn->buffer[i], dSize);
856
857         // If the size of dIn is the same as cOut dIn could be larger than
858         // the modulus. If it is, then RSAEP() will catch it.
859     }
860     break;
861     case ALG_RSAES_VALUE:
862         retVal = RSAES_PKCS1v1_5Encode(&cOut->b, dIn, rand);
863         break;
864     case ALG_OAEP_VALUE:
865         retVal = OaepEncode(&cOut->b, scheme->details.oaep.hashAlg, label, dIn,
866                             rand);

```

```

867         break;
868     default:
869         ERROR_RETURN(TPM_RC_SCHEME);
870         break;
871 }
872 // All the schemes that do padding will come here for the encryption step
873 // Check that the Encoding worked
874 if(retVal == TPM_RC_SUCCESS)
875     // Padding OK so do the encryption
876     retVal = RSAEP(&cOut->b, key);
877 Exit:
878     return retVal;
879 }

```

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 Returns	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

```

880 LIB_EXPORT TPM_RC
881 CryptRsaDecrypt(
882     TPM2B          *dOut,          // OUT: the decrypted data
883     TPM2B          *cIn,          // IN: the data to decrypt
884     OBJECT         *key,          // IN: the key to use for decryption
885     TPMT_RSA_DECRYPT *scheme,      // IN: the padding scheme
886     const TPM2B    *label,        // IN: in case it is needed for the scheme
887 )
888 {
889     TPM_RC retVal;
890
891     // Make sure that the necessary parameters are provided
892     pAssert(cIn != NULL && dOut != NULL && key != NULL);
893
894     // Size is checked to make sure that the encrypted value is the right size
895     if(cIn->size != key->publicArea.unique.rsa.t.size)
896         ERROR_RETURN(TPM_RC_SIZE);
897
898     TEST(scheme->scheme);
899
900     // For others that do padding, do the decryption in place and then
901     // go handle the decoding.
902     retVal = RSADP(cIn, key);
903     if(retVal == TPM_RC_SUCCESS)
904     {
905         // Remove padding
906         switch(scheme->scheme)
907         {
908             case ALG_NULL_VALUE:
909                 if(dOut->size < cIn->size)
910                     return TPM_RC_VALUE;
911                 MemoryCopy2B(dOut, cIn, dOut->size);
912                 break;
913             case ALG_RSAES_VALUE:
914                 retVal = RSAES_Decode(dOut, cIn);
915                 break;
916             case ALG_OAEP_VALUE:

```

```

917         retVal = OaepDecode(dOut, scheme->details.oaep.hashAlg, label, cIn);
918         break;
919     default:
920         retVal = TPM_RC_SCHEME;
921         break;
922     }
923 }
924 Exit:
925     return retVal;
926 }

```

10.2.17.5.5 CryptRsaSign()

This function is used to generate an RSA signature of the type indicated in *scheme*.

Error Returns	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)

```

927 LIB_EXPORT TPM_RC
928 CryptRsaSign(
929     TPMT_SIGNATURE *sigOut,
930     OBJECT *key,           // IN: key to use
931     TPM2B_DIGEST *hIn,    // IN: the digest to sign
932     RAND_STATE *rand,     // IN: the random number generator
933                         // to use (mostly for testing)
934 )
935 {
936     TPM_RC retVal = TPM_RC_SUCCESS;
937     UINT16 modSize;
938
939     // parameter checks
940     pAssert(sigOut != NULL && key != NULL && hIn != NULL);
941
942     modSize = key->publicArea.unique.rsa.t.size;
943
944     // for all non-null signatures, the size is the size of the key modulus
945     sigOut->signature.rsapss.sig.t.size = modSize;
946
947     TEST(sigOut->sigAlg);
948
949     switch(sigOut->sigAlg)
950     {
951     case ALG_NULL_VALUE:
952         sigOut->signature.rsapss.sig.t.size = 0;
953         return TPM_RC_SUCCESS;
954     case ALG_RSAPSS_VALUE:
955         retVal = PssEncode(&sigOut->signature.rsapss.sig.b,
956                           sigOut->signature.rsapss.hash, &hIn->b, rand);
957         break;
958     case ALG_RSASSA_VALUE:
959         retVal = RSASSA_Encode(&sigOut->signature.rsassa.sig.b,
960                               sigOut->signature.rsassa.hash, &hIn->b);
961         break;
962     default:
963         retVal = TPM_RC_SCHEME;
964     }
965     if(retVal == TPM_RC_SUCCESS)
966     {
967         // Do the encryption using the private key
968         retVal = RSADP(&sigOut->signature.rsapss.sig.b, key);
969     }

```

```

970     return retVal;
971 }

```

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 Returns	Meaning
TPM_RC_SIGNATURE	the signature does not check
TPM_RC_SCHEME	unsupported scheme or hash algorithm

```

972 LIB_EXPORT TPM_RC
973 CryptRsaValidateSignature(
974     TPMT_SIGNATURE *sig,           // IN: signature
975     OBJECT *key,                  // IN: public modulus
976     TPM2B_DIGEST *digest          // IN: The digest being validated
977 )
978 {
979     TPM_RC      retVal;
980     //
981     // Fatal programming errors
982     pAssert(key != NULL && sig != NULL && digest != NULL);
983     switch(sig->sigAlg)
984     {
985         case ALG_RSAPSS_VALUE:
986         case ALG_RSASSA_VALUE:
987             break;
988         default:
989             return TPM_RC_SCHEME;
990     }
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 ALG_RSAPSS_VALUE:
1005                 retVal = PssDecode(sig->signature.any.hashAlg, &digest->b,
1006                                     &sig->signature.rsassa.sig.b);
1007                 break;
1008             case ALG_RSASSA_VALUE:
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 Returns	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
1060     // check for supported key size.
1061     keySizeInBits = publicArea->parameters.rsaDetail.keyBits;
1062     if(((keySizeInBits % 1024) != 0)
1063        || (keySizeInBits > MAX_RSA_KEY_BITS) // this might be redundant, but...
1064        || (keySizeInBits == 0))
1065         ERROR_RETURN(TPM_RC_VALUE);
1066
1067     // Set the prime size for instrumentation purposes
1068     INSTRUMENT_SET(PrimeIndex, PRIME_INDEX(keySizeInBits / 2));
1069
1070     #if SIMULATION && USE_RSA_KEY_CACHE
1071         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(ALG_NULL_VALUE);
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
1098         INSTRUMENT_INC(PrimeCounts[PrimeIndex]);
1099
1100         // If this is the second prime, make sure that it differs from the
1101         // first prime by at least 2^100
1102         if(BnEqualZero(Z->Q))
1103         {
1104             // copy p to q and compute another prime in p
1105             BnCopy(Z->Q, Z->P);
1106             continue;
1107         }
1108         // Make sure that the difference is at least 100 bits. Need to do it this
1109         // way because the big numbers are only positive values
1110         if(BnUnsignedCmp(Z->P, Z->Q) < 0)
1111             BnSub(bnD, Z->Q, Z->P);
1112         else
1113             BnSub(bnD, Z->P, Z->Q);
1114         if(BnMsb(bnD) < 100)
1115             continue;
1116
1117         //Form the public modulus and set the unique value
1118         BnMult(bnN, Z->P, Z->Q);
1119         BnTo2B(bnN, &publicArea->unique.rsa.b,
1120             (NUMBYTES)BITS_TO_BYTES(keySizeInBits));
1121         // Make sure everything came out right. The MSb of the values must be one
1122         if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1123             || (publicArea->unique.rsa.t.size
1124                 != (NUMBYTES)BITS_TO_BYTES(keySizeInBits)))
1125             FAIL(FATAL_ERROR_INTERNAL);
1126
1127         // Make sure that we can form the private exponent values
1128         if(ComputePrivateExponent(bnPubExp, Z) != TRUE)
1129         {
1130             // If ComputePrivateExponent could not find an inverse for
1131             // Q, then copy P and recompute P. This might
1132             // cause both to be recomputed if P is also zero
1133             if(BnEqualZero(Z->Q))
1134                 BnCopy(Z->Q, Z->P);
1135             continue;
1136         }
1137
1138         // Pack the private exponent into the sensitive area

```

```
1139     PackExponent(&sensitive->sensitive.rsa, Z);
1140     // Make sure everything came out right. The MSb of the values must be one
1141     if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
1142        || ((sensitive->sensitive.rsa.t.buffer[0] & 0x80) == 0))
1143         FAIL(FATAL_ERROR_INTERNAL);
1144
1145     retVal = TPM_RC_SUCCESS;
1146     // Do a trial encryption decryption if this is a signing key
1147     if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
1148     {
1149         BN_RSA(temp1);
1150         BN_RSA(temp2);
1151         BnGenerateRandomInRange(temp1, bnN, rand);
1152
1153         // Encrypt with public exponent...
1154         BnModExp(temp2, temp1, bnPubExp, bnN);
1155         // ... then decrypt with private exponent
1156         RsaPrivateKeyOp(temp2, Z);
1157
1158         // If the starting and ending values are not the same,
1159         // start over )-;
1160         if(BnUnsignedCmp(temp2, temp1) != 0)
1161         {
1162             BnSetWord(Z->Q, 0);
1163             retVal = TPM_RC_NO_RESULT;
1164         }
1165     }
1166 }
1167 Exit:
1168     return retVal;
1169 }
1170 #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  #if SMAC_IMPLEMENTED
```

10.2.18.2.1 CryptSmacStart()

Function to start an SMAC.

```
4  UINT16
5  CryptSmacStart(
6      HASH_STATE          *state,
7      TPMU_PUBLIC_PARMS  *keyParameters,
8      TPM_ALG_ID          macAlg,      // IN: the type of MAC
9      TPM2B               *key
10 )
11 {
12     UINT16          retVal = 0;
13     //
14     // Make sure that the key size is correct. This should have been checked
15     // at key load, but...
16     if (BITS_TO_BYTES(keyParameters->symDetail.sym.keyBits.sym) == key->size)
17     {
18         switch (macAlg)
19         {
20 #if ALG_CMIC
21             case ALG_CMIC VALUE:
22                 retVal = CryptCmicStart(&state->state.smac, keyParameters,
23                                         macAlg, key);
24             break;
25 #endif
26             default:
27                 break;
28         }
29     }
30     state->type = (retVal != 0) ? HASH_STATE_SMAC : HASH_STATE_EMPTY;
31     return retVal;
32 }
```

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.

```
33  UINT16
34  CryptMacStart(
35      HMAC_STATE          *state,
36      TPMU_PUBLIC_PARMS  *keyParameters,
37      TPM_ALG_ID          macAlg,      // IN: the type of MAC
38      TPM2B               *key
```

```

39 )
40 {
41     MemorySet(state, 0, sizeof(HMAC_STATE));
42     if(CryptHashIsValidAlg(macAlg, FALSE))
43     {
44         return CryptHmacStart(state, macAlg, key->size, key->buffer);
45     }
46     else if(CryptSmacIsValidAlg(macAlg, FALSE))
47     {
48         return CryptSmacStart(&state->hashState, keyParameters, macAlg, key);
49     }
50     else
51         return 0;
52 }

```

10.2.18.2.3 CryptMacEnd()

Dispatch to the MAC end function using a size and buffer pointer.

```

53 UINT16
54 CryptMacEnd(
55     HMAC_STATE      *state,
56     UINT32           size,
57     BYTE             *buffer
58 )
59 {
60     UINT16           retVal = 0;
61     if(state->hashState.type == HASH_STATE_SMAC)
62         retVal = (state->hashState.state.smac.smacMethods.end)(
63             &state->hashState.state.smac.state, size, buffer);
64     else if(state->hashState.type == HASH_STATE_HMAC)
65         retVal = CryptHmacEnd(state, size, buffer);
66     state->hashState.type = HASH_STATE_EMPTY;
67     return retVal;
68 }

```

10.2.18.2.4 CryptMacEnd2B()

Dispatch to the MAC end function using a 2B.

```

69 UINT16
70 CryptMacEnd2B (
71     HMAC_STATE      *state,
72     TPM2B            *data
73 )
74 {
75     return CryptMacEnd(state, data->size, data->buffer);
76 }
77 #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  #include "CryptSym.h"
3  #define KEY_BLOCK_SIZES(ALG, alg)
4  static const INT16 alg##_KeyBlockSizes[] = {
5                                     ALG##_KEY_SIZES_BITS, -1, ALG##_BLOCK_SIZES };
6  #if ALG_AES
7      KEY_BLOCK_SIZES(AES, aes);
8  #endif // ALG_AES
9  #if ALG_SM4
10     KEY_BLOCK_SIZES(SM4, sm4);
11 #endif
12 #if ALG_CAMELLIA
13     KEY_BLOCK_SIZES(CAMELLIA, camellia);
14 #endif
15 #if ALG_TDES
16     KEY_BLOCK_SIZES(TDES, tdes);
17 #endif

```

10.2.19.3 Initialization and Data Access Functions

10.2.19.3.1 CryptSymInit()

This function is called to do _TPM_Init() processing

```

18  BOOL
19  CryptSymInit(
20      void
21  )
22  {
23      return TRUE;
24  }

```

10.2.19.3.2 CryptSymStartup()

This function is called to do TPM2_Startup() processing

```

25  BOOL
26  CryptSymStartup(
27      void
28  )
29  {
30      return TRUE;
31  }

```

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

```

32  LIB_EXPORT INT16
33  CryptGetSymmetricBlockSize(
34      TPM_ALG_ID      symmetricAlg,    // IN: the symmetric algorithm
35      UINT16          keySizeInBits    // IN: the key size
36  )
37  {
38      const INT16      *sizes;
39      INT16            i;
40      #define ALG_CASE(SYM, sym) case ALG_##SYM##_VALUE: sizes = sym##KeyBlockSizes; break
41      switch(symmetricAlg)
42      {
43          #if ALG_AES
44              ALG_CASE(AES, aes);
45          #endif
46          #if ALG_SM4
47              ALG_CASE(SM4, sm4);
48          #endif
49          #if ALG_CAMELLIA
50              ALG_CASE(CAMELLIA, camellia);
51          #endif
52          #if ALG_TDES
53              ALG_CASE(TDES, tdes);
54          #endif
55          default:
56              return 0;
57      }
58      // Find the index of the indicated keySizeInBits
59      for(i = 0; *sizes >= 0; i++, sizes++)
60      {
61          if(*sizes == keySizeInBits)
62              break;
63      }
64      // If sizes is pointing at the end of the list of key sizes, then the desired
65      // key size was not found so set the block size to zero.
66      if(*sizes++ < 0)
67          return 0;
68      // Advance until the end of the list is found
69      while(*sizes++ >= 0);
70      // sizes is pointing to the first entry in the list of block sizes. Use the
71      // ith index to find the block size for the corresponding key size.
72      return sizes[i];
73  }

```

10.2.19.4 Symmetric Encryption

This function performs symmetric encryption based on the mode.

Error Returns	Meaning
TPM_RC_SIZE	dSize is not a multiple of the block size for an algorithm that requires it
TPM_RC_FAILURE	Fatal error

```

74  LIB_EXPORT TPM_RC
75  CryptSymmetricEncrypt(
76      BYTE          *dOut,          // OUT:
77      TPM_ALG_ID     algorithm,      // IN: the symmetric algorithm
78      UINT16         keySizeInBits,  // IN: key size in bits
79      const BYTE     *key,          // IN: key buffer. The size of this buffer
80                                   // in bytes is (keySizeInBits + 7) / 8
81      TPM2B_IV       *ivInOut,      // IN/OUT: IV for decryption.
82      TPM_ALG_ID     mode,          // IN: Mode to use
83      INT32          dSize,         // IN: data size (may need to be a
84                                   // multiple of the blockSize)
85      const BYTE     *dIn           // IN: data buffer
86  )
87  {
88      BYTE          *pIv;
89      int           i;
90      BYTE          tmp[MAX_SYM_BLOCK_SIZE];
91      BYTE          *pT;
92      tpmCryptKeySchedule_t keySchedule;
93      INT16         blockSize;
94      TpmCryptSetSymKeyCall_t encrypt;
95      BYTE          *iv;
96      BYTE          defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
97  //
98      pAssert(dOut != NULL && key != NULL && dIn != NULL);
99      if(dSize == 0)
100         return TPM_RC_SUCCESS;
101
102      TEST(algorithm);
103      blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
104      if(blockSize == 0)
105         return TPM_RC_FAILURE;
106      // If the iv is provided, then it is expected to be block sized. In some cases,
107      // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
108      // with no knowledge of the actual block size. This function will set it.
109      if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
110      {
111         ivInOut->t.size = blockSize;
112         iv = ivInOut->t.buffer;
113      }
114      else
115         iv = defaultIv;
116      pIv = iv;
117
118      // Create encrypt key schedule and set the encryption function pointer.
119
120      SELECT(ENCRYPT);
121
122      switch(mode)
123      {
124  #if ALG_CTR
125      case ALG_CTR_VALUE:
126         for(; dSize > 0; dSize -= blockSize)
127         {
128             // Encrypt the current value of the IV(counter)
129             ENCRYPT(&keySchedule, iv, tmp);
130
131             //increment the counter (counter is big-endian so start at end)

```

```

132         for(i = blockSize - 1; i >= 0; i--)
133             if((iv[i] += 1) != 0)
134                 break;
135         // XOR the encrypted counter value with input and put into output
136         pT = tmp;
137         for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
138             *dOut++ = *dIn++ ^ *pT++;
139     }
140     break;
141 #endif
142 #if ALG_OFB
143     case ALG_OFB_VALUE:
144         // This is written so that dIn and dOut may be the same
145         for(; dSize > 0; dSize -= blockSize)
146         {
147             // Encrypt the current value of the "IV"
148             ENCRYPT(&keySchedule, iv, iv);
149
150             // XOR the encrypted IV into dIn to create the cipher text (dOut)
151             pIv = iv;
152             for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
153                 *dOut++ = (*pIv++ ^ *dIn++);
154         }
155         break;
156 #endif
157 #if ALG_CBC
158     case ALG_CBC_VALUE:
159         // For CBC the data size must be an even multiple of the
160         // cipher block size
161         if((dSize % blockSize) != 0)
162             return TPM_RC_SIZE;
163         // XOR the data block into the IV, encrypt the IV into the IV
164         // and then copy the IV to the output
165         for(; dSize > 0; dSize -= blockSize)
166         {
167             pIv = iv;
168             for(i = blockSize; i > 0; i--)
169                 *pIv++ ^= *dIn++;
170             ENCRYPT(&keySchedule, iv, iv);
171             pIv = iv;
172             for(i = blockSize; i > 0; i--)
173                 *dOut++ = *pIv++;
174         }
175         break;
176 #endif
177 // CFB is not optional
178 case ALG_CFB_VALUE:
179     // Encrypt the IV into the IV, XOR in the data, and copy to output
180     for(; dSize > 0; dSize -= blockSize)
181     {
182         // Encrypt the current value of the IV
183         ENCRYPT(&keySchedule, iv, iv);
184         pIv = iv;
185         for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)
186             // XOR the data into the IV to create the cipher text
187             // and put into the output
188             *dOut++ = *pIv++ ^= *dIn++;
189     }
190     // If the inner loop (i loop) was smaller than blockSize, then dSize
191     // would have been smaller than blockSize and it is now negative. If
192     // it is negative, then it indicates how many bytes are needed to pad
193     // out the IV for the next round.
194     for(; dSize < 0; dSize++)
195         *pIv++ = 0;
196     break;
197 #if ALG_ECB

```

```

198     case ALG_ECB_VALUE:
199         // For ECB the data size must be an even multiple of the
200         // cipher block size
201         if((dSize % blockSize) != 0)
202             return TPM_RC_SIZE;
203         // Encrypt the input block to the output block
204         for(; dSize > 0; dSize -= blockSize)
205         {
206             ENCRYPT(&keySchedule, dIn, dOut);
207             dIn = &dIn[blockSize];
208             dOut = &dOut[blockSize];
209         }
210         break;
211 #endif
212     default:
213         return TPM_RC_FAILURE;
214 }
215 return TPM_RC_SUCCESS;
216 }

```

10.2.19.4.1 CryptSymmetricDecrypt()

This function performs symmetric decryption based on the mode.

Error Returns	Meaning
TPM_RC_FAILURE	A fatal error
TPM_RCS_SIZE	<i>dSize</i> is not a multiple of the block size for an algorithm that requires it

```

217 LIB_EXPORT TPM_RC
218 CryptSymmetricDecrypt(
219     BYTE                *dOut,           // OUT: decrypted data
220     TPM_ALG_ID          algorithm,       // IN: the symmetric algorithm
221     UINT16              keySizeInBits,   // IN: key size in bits
222     const BYTE          *key,           // IN: key buffer. The size of this buffer
223                                     // in bytes is (keySizeInBits + 7) / 8
224     TPM2B_IV            *ivInOut,       // IN/OUT: IV for decryption.
225     TPM_ALG_ID          mode,           // IN: Mode to use
226     INT32               dSize,          // IN: data size (may need to be a
227                                     // multiple of the blockSize)
228     const BYTE          *dIn            // IN: data buffer
229 )
230 {
231     BYTE                *pIv;
232     int                 i;
233     BYTE                tmp[MAX_SYM_BLOCK_SIZE];
234     BYTE                *pT;
235     tpmCryptKeySchedule_t keySchedule;
236     INT16               blockSize;
237     BYTE                *iv;
238     TpmCryptSetSymKeyCall_t encrypt;
239     TpmCryptSetSymKeyCall_t decrypt;
240     BYTE                defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
241
242     // These are used but the compiler can't tell because they are initialized
243     // in case statements and it can't tell if they are always initialized
244     // when needed, so... Comment these out if the compiler can tell or doesn't
245     // care that these are initialized before use.
246     encrypt = NULL;
247     decrypt = NULL;
248
249     pAssert(dOut != NULL && key != NULL && dIn != NULL);

```

```

250     if(dSize == 0)
251         return TPM_RC_SUCCESS;
252
253     TEST(algorithm);
254     blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
255     if(blockSize == 0)
256         return TPM_RC_FAILURE;
257     // If the iv is provided, then it is expected to be block sized. In some cases,
258     // the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
259     // with no knowledge of the actual block size. This function will set it.
260     if((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
261     {
262         ivInOut->t.size = blockSize;
263         iv = ivInOut->t.buffer;
264     }
265     else
266         iv = defaultIv;
267
268     pIv = iv;
269     // Use the mode to select the key schedule to create. Encrypt always uses the
270     // encryption schedule. Depending on the mode, decryption might use either
271     // the decryption or encryption schedule.
272     switch(mode)
273     {
274 #if ALG_CBC || ALG_ECB
275         case ALG_CBC_VALUE: // decrypt = decrypt
276         case ALG_ECB_VALUE:
277             // For ECB and CBC, the data size must be an even multiple of the
278             // cipher block size
279             if((dSize % blockSize) != 0)
280                 return TPM_RC_SIZE;
281             SELECT(DECRYPT);
282             break;
283 #endif
284         default:
285             // For the remaining stream ciphers, use encryption to decrypt
286             SELECT(ENCRYPT);
287             break;
288     }
289     // Now do the mode-dependent decryption
290     switch(mode)
291     {
292 #if ALG_CBC
293         case ALG_CBC_VALUE:
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 ALG_CFB_VALUE:
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 ALG_CTR_VALUE:
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 ALG_ECB_VALUE:
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 ALG_OFB_VALUE:
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 Returns	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, 0xBFBA1AB25,
11 0x70FA3841, 0x51B3D076, 0xCC2359ED, 0xD9EE0769,
12 0x75E47AF0, 0xD45FF31E, 0x52CCE4F6, 0x04DBC891,
13 0x96658ED2, 0x1753EFE5, 0x3AE4A5A6, 0x8FD4A97F,
14 0x8B15E7EB, 0x0243C3E1, 0xE0F0C31D, 0x0000000B }
15 };
16 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().

```
17 const uint32_t s_LastPrimeInTable = 65537;
18 const uint32_t s_PrimeTableSize = 4097;
19 const uint32_t s_PrimesInTable = 6542;
20 const unsigned char s_PrimeTable[] = {
21 0x6e, 0xcb, 0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x81, 0x32, 0x4c, 0x4a, 0x86,
22 0x0d, 0x82, 0x96, 0x21, 0xc9, 0x34, 0x04, 0x5a, 0x20, 0x61, 0x89, 0xa4,
23 0x44, 0x11, 0x86, 0x29, 0xd1, 0x82, 0x28, 0x4a, 0x30, 0x40, 0x42, 0x32,
24 0x21, 0x99, 0x34, 0x08, 0x4b, 0x06, 0x25, 0x42, 0x84, 0x48, 0x8a, 0x14,
25 0x05, 0x42, 0x30, 0x6c, 0x08, 0xb4, 0x40, 0x0b, 0xa0, 0x08, 0x51, 0x12,
26 0x28, 0x89, 0x04, 0x65, 0x98, 0x30, 0x4c, 0x80, 0x96, 0x44, 0x12, 0x80,
27 0x21, 0x42, 0x12, 0x41, 0xc9, 0x04, 0x21, 0xc0, 0x32, 0x2d, 0x98, 0x00,
28 0x00, 0x49, 0x04, 0x08, 0x81, 0x96, 0x68, 0x82, 0xb0, 0x25, 0x08, 0x22,
29 0x48, 0x89, 0xa2, 0x40, 0x59, 0x26, 0x04, 0x90, 0x06, 0x40, 0x43, 0x30,
30 0x44, 0x92, 0x00, 0x69, 0x10, 0x82, 0x08, 0x08, 0xa4, 0x0d, 0x41, 0x12,
31 0x60, 0xc0, 0x00, 0x24, 0xd2, 0x22, 0x61, 0x08, 0x84, 0x04, 0x1b, 0x82,
32 0x01, 0xd3, 0x10, 0x01, 0x02, 0xa0, 0x44, 0xc0, 0x22, 0x60, 0x91, 0x14,
33 0x0c, 0x40, 0xa6, 0x04, 0xd2, 0x94, 0x20, 0x09, 0x94, 0x20, 0x52, 0x00,
34 0x08, 0x10, 0xa2, 0x4c, 0x00, 0x82, 0x01, 0x51, 0x10, 0x08, 0x8b, 0xa4,
35 0x25, 0x9a, 0x30, 0x44, 0x81, 0x10, 0x4c, 0x03, 0x02, 0x25, 0x52, 0x80,
36 0x08, 0x49, 0x84, 0x20, 0x50, 0x32, 0x00, 0x18, 0xa2, 0x40, 0x11, 0x24,
37 0x28, 0x01, 0x84, 0x01, 0x01, 0xa0, 0x41, 0x0a, 0x12, 0x45, 0x00, 0x36,
38 0x08, 0x00, 0x26, 0x29, 0x83, 0x82, 0x61, 0xc0, 0x80, 0x04, 0x10, 0x10,
39 0x6d, 0x00, 0x22, 0x48, 0x58, 0x26, 0x0c, 0xc2, 0x10, 0x48, 0x89, 0x24,
40 0x20, 0x58, 0x20, 0x45, 0x88, 0x24, 0x00, 0x19, 0x02, 0x25, 0xc0, 0x10,
41 0x68, 0x08, 0x14, 0x01, 0xca, 0x32, 0x28, 0x80, 0x00, 0x04, 0x4b, 0x26,
42 0x00, 0x13, 0x90, 0x60, 0x82, 0x80, 0x25, 0xd0, 0x00, 0x01, 0x10, 0x32,
43 0x0c, 0x43, 0x86, 0x21, 0x11, 0x00, 0x08, 0x43, 0x24, 0x04, 0x48, 0x10,
44 0x0c, 0x90, 0x92, 0x00, 0x43, 0x20, 0x2d, 0x00, 0x06, 0x09, 0x88, 0x24,
45 0x40, 0xc0, 0x32, 0x09, 0x09, 0x82, 0x00, 0x53, 0x80, 0x08, 0x80, 0x96,
46 0x41, 0x81, 0x00, 0x40, 0x48, 0x10, 0x48, 0x08, 0x96, 0x48, 0x58, 0x20,
47 0x29, 0xc3, 0x80, 0x20, 0x02, 0x94, 0x60, 0x92, 0x00, 0x20, 0x81, 0x22,
48 0x44, 0x10, 0xa0, 0x05, 0x40, 0x90, 0x01, 0x49, 0x20, 0x04, 0x0a, 0x00,
49 0x24, 0x89, 0x34, 0x48, 0x13, 0x80, 0x2c, 0xc0, 0x82, 0x29, 0x00, 0x24,
50 0x45, 0x08, 0x00, 0x08, 0x98, 0x36, 0x04, 0x52, 0x84, 0x04, 0xd0, 0x04,
51 0x00, 0x8a, 0x90, 0x44, 0x82, 0x32, 0x65, 0x18, 0x90, 0x00, 0x0a, 0x02,
```


52 0x01, 0x40, 0x02, 0x28, 0x40, 0xa4, 0x04, 0x92, 0x30, 0x04, 0x11, 0x86,
53 0x08, 0x42, 0x00, 0x2c, 0x52, 0x04, 0x08, 0xc9, 0x84, 0x60, 0x48, 0x12,
54 0x09, 0x99, 0x24, 0x44, 0x00, 0x24, 0x00, 0x03, 0x14, 0x21, 0x00, 0x10,
55 0x01, 0x1a, 0x32, 0x05, 0x88, 0x20, 0x40, 0x06, 0x09, 0xc3, 0x84,
56 0x40, 0x01, 0x30, 0x60, 0x18, 0x02, 0x68, 0x11, 0x90, 0x0c, 0x02, 0xa2,
57 0x04, 0x00, 0x86, 0x29, 0x89, 0x14, 0x24, 0x82, 0x02, 0x41, 0x08, 0x80,
58 0x04, 0x19, 0x80, 0x08, 0x10, 0x12, 0x68, 0x42, 0xa4, 0x04, 0x00, 0x02,
59 0x61, 0x10, 0x06, 0x0c, 0x10, 0x00, 0x01, 0x12, 0x10, 0x20, 0x03, 0x94,
60 0x21, 0x42, 0x12, 0x65, 0x18, 0x94, 0x0c, 0x0a, 0x04, 0x28, 0x01, 0x14,
61 0x29, 0x0a, 0xa4, 0x40, 0xd0, 0x00, 0x40, 0x01, 0x90, 0x04, 0x41, 0x12,
62 0x2d, 0x40, 0x82, 0x48, 0xc1, 0x20, 0x00, 0x10, 0x30, 0x01, 0x08, 0x24,
63 0x04, 0x59, 0x84, 0x24, 0x00, 0x02, 0x29, 0x82, 0x00, 0x61, 0x58, 0x02,
64 0x48, 0x81, 0x16, 0x48, 0x10, 0x00, 0x21, 0x11, 0x06, 0x00, 0xca, 0xa0,
65 0x40, 0x02, 0x00, 0x04, 0x91, 0xb0, 0x00, 0x42, 0x04, 0x0c, 0x81, 0x06,
66 0x09, 0x48, 0x14, 0x25, 0x92, 0x20, 0x25, 0x11, 0xa0, 0x00, 0x0a, 0x86,
67 0x0c, 0xc1, 0x02, 0x48, 0x00, 0x20, 0x45, 0x08, 0x32, 0x00, 0x98, 0x06,
68 0x04, 0x13, 0x22, 0x00, 0x82, 0x04, 0x48, 0x81, 0x14, 0x44, 0x82, 0x12,
69 0x24, 0x18, 0x10, 0x40, 0x43, 0x80, 0x28, 0xd0, 0x04, 0x20, 0x81, 0x24,
70 0x64, 0xd8, 0x00, 0x2c, 0x09, 0x12, 0x08, 0x41, 0xa2, 0x00, 0x00, 0x02,
71 0x41, 0xca, 0x20, 0x41, 0xc0, 0x10, 0x01, 0x18, 0xa4, 0x04, 0x18, 0xa4,
72 0x20, 0x12, 0x94, 0x20, 0x83, 0xa0, 0x40, 0x02, 0x32, 0x44, 0x80, 0x04,
73 0x00, 0x18, 0x00, 0x0c, 0x40, 0x86, 0x60, 0x8a, 0x00, 0x64, 0x88, 0x12,
74 0x05, 0x01, 0x82, 0x00, 0x4a, 0xa2, 0x01, 0xc1, 0x10, 0x61, 0x09, 0x04,
75 0x01, 0x88, 0x00, 0x60, 0x01, 0xb4, 0x40, 0x08, 0x06, 0x01, 0x03, 0x80,
76 0x08, 0x40, 0x94, 0x04, 0x8a, 0x20, 0x29, 0x80, 0x02, 0x0c, 0x52, 0x02,
77 0x01, 0x42, 0x84, 0x00, 0x80, 0x84, 0x64, 0x02, 0x32, 0x48, 0x00, 0x30,
78 0x44, 0x40, 0x22, 0x21, 0x00, 0x02, 0x08, 0xc3, 0xa0, 0x04, 0xd0, 0x20,
79 0x40, 0x18, 0x16, 0x40, 0x40, 0x00, 0x28, 0x52, 0x90, 0x08, 0x82, 0x14,
80 0x01, 0x18, 0x10, 0x08, 0x09, 0x82, 0x40, 0x0a, 0xa0, 0x20, 0x93, 0x80,
81 0x08, 0xc0, 0x00, 0x20, 0x52, 0x00, 0x05, 0x01, 0x10, 0x40, 0x11, 0x06,
82 0x0c, 0x82, 0x00, 0x00, 0x4b, 0x90, 0x44, 0x9a, 0x00, 0x28, 0x80, 0x90,
83 0x04, 0x4a, 0x06, 0x09, 0x43, 0x02, 0x28, 0x00, 0x34, 0x01, 0x18, 0x00,
84 0x65, 0x09, 0x80, 0x44, 0x03, 0x00, 0x24, 0x02, 0x82, 0x61, 0x48, 0x14,
85 0x41, 0x00, 0x12, 0x28, 0x00, 0x34, 0x08, 0x51, 0x04, 0x05, 0x12, 0x90,
86 0x28, 0x89, 0x84, 0x60, 0x12, 0x10, 0x49, 0x10, 0x26, 0x40, 0x49, 0x82,
87 0x00, 0x91, 0x10, 0x01, 0x0a, 0x24, 0x40, 0x88, 0x10, 0x4c, 0x10, 0x04,
88 0x00, 0x50, 0xa2, 0x2c, 0x40, 0x90, 0x48, 0x0a, 0xb0, 0x01, 0x50, 0x12,
89 0x08, 0x00, 0xa4, 0x04, 0x09, 0xa0, 0x28, 0x92, 0x02, 0x00, 0x43, 0x10,
90 0x21, 0x02, 0x20, 0x41, 0x81, 0x32, 0x00, 0x08, 0x04, 0x0c, 0x52, 0x00,
91 0x21, 0x49, 0x84, 0x20, 0x10, 0x02, 0x01, 0x81, 0x10, 0x48, 0x40, 0x22,
92 0x01, 0x01, 0x84, 0x69, 0xc1, 0x30, 0x01, 0xc8, 0x02, 0x44, 0x88, 0x00,
93 0x0c, 0x01, 0x02, 0x2d, 0xc0, 0x12, 0x61, 0x00, 0xa0, 0x00, 0xc0, 0x30,
94 0x40, 0x01, 0x12, 0x08, 0x0b, 0x20, 0x00, 0x80, 0x94, 0x40, 0x01, 0x84,
95 0x40, 0x00, 0x32, 0x00, 0x10, 0x84, 0x00, 0x0b, 0x24, 0x00, 0x01, 0x06,
96 0x29, 0x8a, 0x84, 0x41, 0x80, 0x10, 0x08, 0x08, 0x94, 0x4c, 0x03, 0x80,
97 0x01, 0x40, 0x96, 0x40, 0x41, 0x20, 0x20, 0x50, 0x22, 0x25, 0x89, 0xa2,
98 0x40, 0x40, 0xa4, 0x20, 0x02, 0x86, 0x28, 0x01, 0x20, 0x21, 0x4a, 0x10,
99 0x08, 0x00, 0x14, 0x08, 0x40, 0x04, 0x25, 0x42, 0x02, 0x21, 0x43, 0x10,
100 0x04, 0x92, 0x00, 0x21, 0x11, 0xa0, 0x4c, 0x18, 0x22, 0x09, 0x03, 0x84,
101 0x41, 0x89, 0x10, 0x04, 0x82, 0x22, 0x24, 0x01, 0x14, 0x08, 0x08, 0x84,
102 0x08, 0xc1, 0x00, 0x09, 0x42, 0xb0, 0x41, 0x8a, 0x02, 0x00, 0x80, 0x36,
103 0x04, 0x49, 0xa0, 0x24, 0x91, 0x00, 0x00, 0x02, 0x94, 0x41, 0x92, 0x02,
104 0x01, 0x08, 0x06, 0x08, 0x09, 0x00, 0x01, 0xd0, 0x16, 0x28, 0x89, 0x80,
105 0x60, 0x00, 0x00, 0x68, 0x01, 0x90, 0x0c, 0x50, 0x20, 0x01, 0x40, 0x80,
106 0x40, 0x42, 0x30, 0x41, 0x00, 0x20, 0x25, 0x81, 0x06, 0x40, 0x49, 0x00,
107 0x08, 0x01, 0x12, 0x49, 0x00, 0xa0, 0x20, 0x18, 0x30, 0x05, 0x01, 0xa6,
108 0x00, 0x10, 0x24, 0x28, 0x00, 0x02, 0x20, 0xc8, 0x20, 0x00, 0x88, 0x12,
109 0x0c, 0x90, 0x92, 0x00, 0x02, 0x26, 0x01, 0x42, 0x16, 0x49, 0x00, 0x04,
110 0x24, 0x42, 0x02, 0x01, 0x88, 0x80, 0x0c, 0x1a, 0x80, 0x08, 0x10, 0x00,
111 0x60, 0x02, 0x94, 0x44, 0x88, 0x00, 0x69, 0x11, 0x30, 0x08, 0x12, 0xa0,
112 0x24, 0x13, 0x84, 0x00, 0x82, 0x00, 0x65, 0xc0, 0x10, 0x28, 0x00, 0x30,
113 0x04, 0x03, 0x20, 0x01, 0x11, 0x06, 0x01, 0xc8, 0x80, 0x00, 0xc2, 0x20,
114 0x08, 0x10, 0x82, 0x0c, 0x13, 0x02, 0x0c, 0x52, 0x06, 0x40, 0x00, 0xb0,
115 0x61, 0x40, 0x10, 0x01, 0x98, 0x86, 0x04, 0x10, 0x84, 0x08, 0x92, 0x14,
116 0x60, 0x41, 0x80, 0x41, 0x1a, 0x10, 0x04, 0x81, 0x22, 0x40, 0x41, 0x20,
117 0x29, 0x52, 0x00, 0x41, 0x08, 0x34, 0x60, 0x10, 0x00, 0x28, 0x01, 0x10,

118 0x40, 0x00, 0x84, 0x08, 0x42, 0x90, 0x20, 0x48, 0x04, 0x04, 0x52, 0x02,
119 0x00, 0x08, 0x20, 0x04, 0x00, 0x82, 0x0d, 0x00, 0x82, 0x40, 0x02, 0x10,
120 0x05, 0x48, 0x20, 0x40, 0x99, 0x00, 0x00, 0x01, 0x06, 0x24, 0xc0, 0x00,
121 0x68, 0x82, 0x04, 0x21, 0x12, 0x10, 0x44, 0x08, 0x04, 0x00, 0x40, 0xa6,
122 0x20, 0xd0, 0x16, 0x09, 0xc9, 0x24, 0x41, 0x02, 0x20, 0x0c, 0x09, 0x92,
123 0x40, 0x12, 0x00, 0x00, 0x40, 0x00, 0x09, 0x43, 0x84, 0x20, 0x98, 0x02,
124 0x01, 0x11, 0x24, 0x00, 0x43, 0x24, 0x00, 0x03, 0x90, 0x08, 0x41, 0x30,
125 0x24, 0x58, 0x20, 0x4c, 0x80, 0x82, 0x08, 0x10, 0x24, 0x25, 0x81, 0x06,
126 0x41, 0x09, 0x10, 0x20, 0x18, 0x10, 0x44, 0x80, 0x10, 0x00, 0x4a, 0x24,
127 0x0d, 0x01, 0x94, 0x28, 0x80, 0x30, 0x00, 0xc0, 0x02, 0x60, 0x10, 0x84,
128 0x0c, 0x02, 0x00, 0x09, 0x02, 0x82, 0x01, 0x08, 0x10, 0x04, 0xc2, 0x20,
129 0x68, 0x09, 0x06, 0x04, 0x18, 0x00, 0x00, 0x11, 0x90, 0x08, 0x0b, 0x10,
130 0x21, 0x82, 0x02, 0x0c, 0x10, 0xb6, 0x08, 0x00, 0x26, 0x00, 0x41, 0x02,
131 0x01, 0x4a, 0x24, 0x21, 0x1a, 0x20, 0x24, 0x80, 0x00, 0x44, 0x02, 0x00,
132 0x2d, 0x40, 0x02, 0x00, 0x8b, 0x94, 0x20, 0x10, 0x00, 0x20, 0x90, 0xa6,
133 0x40, 0x13, 0x00, 0x2c, 0x11, 0x86, 0x61, 0x01, 0x80, 0x41, 0x10, 0x02,
134 0x04, 0x81, 0x30, 0x48, 0x48, 0x20, 0x28, 0x50, 0x80, 0x21, 0x8a, 0x10,
135 0x04, 0x08, 0x10, 0x09, 0x10, 0x10, 0x48, 0x42, 0xa0, 0x0c, 0x82, 0x92,
136 0x60, 0xc0, 0x20, 0x05, 0xd2, 0x20, 0x40, 0x01, 0x00, 0x04, 0x08, 0x82,
137 0x2d, 0x82, 0x02, 0x00, 0x48, 0x80, 0x41, 0x48, 0x10, 0x00, 0x91, 0x04,
138 0x04, 0x03, 0x84, 0x00, 0xc2, 0x04, 0x68, 0x00, 0x00, 0x64, 0xc0, 0x22,
139 0x40, 0x08, 0x32, 0x44, 0x09, 0x86, 0x00, 0x91, 0x02, 0x28, 0x01, 0x00,
140 0x64, 0x48, 0x00, 0x24, 0x10, 0x90, 0x00, 0x43, 0x00, 0x21, 0x52, 0x86,
141 0x41, 0x8b, 0x90, 0x20, 0x40, 0x20, 0x08, 0x88, 0x04, 0x44, 0x13, 0x20,
142 0x00, 0x02, 0x84, 0x60, 0x81, 0x90, 0x24, 0x40, 0x30, 0x00, 0x08, 0x10,
143 0x08, 0x08, 0x02, 0x01, 0x10, 0x04, 0x20, 0x43, 0xb4, 0x40, 0x90, 0x12,
144 0x68, 0x01, 0x80, 0x4c, 0x18, 0x00, 0x08, 0xc0, 0x12, 0x49, 0x40, 0x10,
145 0x24, 0x1a, 0x00, 0x41, 0x89, 0x24, 0x4c, 0x10, 0x00, 0x04, 0x52, 0x10,
146 0x09, 0x4a, 0x20, 0x41, 0x48, 0x22, 0x69, 0x11, 0x14, 0x08, 0x10, 0x06,
147 0x24, 0x80, 0x84, 0x28, 0x00, 0x10, 0x00, 0x40, 0x10, 0x01, 0x08, 0x26,
148 0x08, 0x48, 0x06, 0x28, 0x00, 0x14, 0x01, 0x42, 0x84, 0x04, 0x0a, 0x20,
149 0x00, 0x01, 0x82, 0x08, 0x00, 0x82, 0x24, 0x12, 0x04, 0x40, 0x40, 0xa0,
150 0x40, 0x90, 0x10, 0x04, 0x90, 0x22, 0x40, 0x10, 0x20, 0x2c, 0x80, 0x10,
151 0x28, 0x43, 0x00, 0x04, 0x58, 0x00, 0x01, 0x81, 0x10, 0x48, 0x09, 0x20,
152 0x21, 0x83, 0x04, 0x00, 0x42, 0xa4, 0x44, 0x00, 0x00, 0x6c, 0x10, 0xa0,
153 0x44, 0x48, 0x80, 0x00, 0x83, 0x80, 0x48, 0xc9, 0x00, 0x00, 0x00, 0x02,
154 0x05, 0x10, 0xb0, 0x04, 0x13, 0x04, 0x29, 0x10, 0x92, 0x40, 0x08, 0x04,
155 0x44, 0x82, 0x22, 0x00, 0x19, 0x20, 0x00, 0x19, 0x20, 0x01, 0x81, 0x90,
156 0x60, 0x8a, 0x00, 0x41, 0xc0, 0x02, 0x45, 0x10, 0x04, 0x00, 0x02, 0xa2,
157 0x09, 0x40, 0x10, 0x21, 0x49, 0x20, 0x01, 0x42, 0x30, 0x2c, 0x00, 0x14,
158 0x44, 0x01, 0x22, 0x04, 0x02, 0x92, 0x08, 0x89, 0x04, 0x21, 0x80, 0x10,
159 0x05, 0x01, 0x20, 0x40, 0x41, 0x80, 0x04, 0x00, 0x12, 0x09, 0x40, 0xb0,
160 0x64, 0x58, 0x32, 0x01, 0x08, 0x90, 0x00, 0x41, 0x04, 0x09, 0xc1, 0x80,
161 0x61, 0x08, 0x90, 0x00, 0x9a, 0x00, 0x24, 0x01, 0x12, 0x08, 0x02, 0x26,
162 0x05, 0x82, 0x06, 0x08, 0x08, 0x00, 0x20, 0x48, 0x20, 0x00, 0x18, 0x24,
163 0x48, 0x03, 0x02, 0x00, 0x11, 0x00, 0x09, 0x00, 0x84, 0x01, 0x4a, 0x10,
164 0x01, 0x98, 0x00, 0x04, 0x18, 0x86, 0x00, 0xc0, 0x00, 0x20, 0x81, 0x80,
165 0x04, 0x10, 0x30, 0x05, 0x00, 0xb4, 0x0c, 0x4a, 0x82, 0x29, 0x91, 0x02,
166 0x28, 0x00, 0x20, 0x44, 0xc0, 0x00, 0x2c, 0x91, 0x80, 0x40, 0x01, 0xa2,
167 0x00, 0x12, 0x04, 0x09, 0xc3, 0x20, 0x00, 0x08, 0x02, 0x0c, 0x10, 0x22,
168 0x04, 0x00, 0x00, 0x2c, 0x11, 0x86, 0x00, 0xc0, 0x00, 0x00, 0x12, 0x32,
169 0x40, 0x89, 0x80, 0x40, 0x40, 0x02, 0x05, 0x50, 0x86, 0x60, 0x82, 0xa4,
170 0x60, 0x0a, 0x12, 0x4d, 0x80, 0x90, 0x08, 0x12, 0x80, 0x09, 0x02, 0x14,
171 0x48, 0x01, 0x24, 0x20, 0x8a, 0x00, 0x44, 0x90, 0x04, 0x04, 0x01, 0x02,
172 0x00, 0xd1, 0x12, 0x00, 0x0a, 0x04, 0x40, 0x00, 0x32, 0x21, 0x81, 0x24,
173 0x08, 0x19, 0x84, 0x20, 0x02, 0x04, 0x08, 0x89, 0x80, 0x24, 0x02, 0x02,
174 0x68, 0x18, 0x82, 0x44, 0x42, 0x00, 0x21, 0x40, 0x00, 0x28, 0x01, 0x80,
175 0x45, 0x82, 0x20, 0x40, 0x11, 0x80, 0x0c, 0x02, 0x00, 0x24, 0x40, 0x90,
176 0x01, 0x40, 0x20, 0x20, 0x50, 0x20, 0x28, 0x19, 0x00, 0x40, 0x09, 0x20,
177 0x08, 0x80, 0x04, 0x60, 0x40, 0x80, 0x20, 0x08, 0x30, 0x49, 0x09, 0x34,
178 0x00, 0x11, 0x24, 0x24, 0x82, 0x00, 0x41, 0xc2, 0x00, 0x04, 0x92, 0x02,
179 0x24, 0x80, 0x00, 0x0c, 0x02, 0xa0, 0x00, 0x01, 0x06, 0x60, 0x41, 0x04,
180 0x21, 0xd0, 0x00, 0x01, 0x00, 0x48, 0x12, 0x84, 0x04, 0x91, 0x12,
181 0x08, 0x00, 0x24, 0x44, 0x00, 0x12, 0x41, 0x18, 0x26, 0x0c, 0x41, 0x80,
182 0x00, 0x52, 0x04, 0x20, 0x09, 0x00, 0x24, 0x90, 0x20, 0x48, 0x18, 0x02,
183 0x00, 0x03, 0xa2, 0x09, 0xd0, 0x14, 0x00, 0x8a, 0x84, 0x25, 0x4a, 0x00,

184 0x20, 0x98, 0x14, 0x40, 0x00, 0xa2, 0x05, 0x00, 0x00, 0x00, 0x40, 0x14,
185 0x01, 0x58, 0x20, 0x2c, 0x80, 0x84, 0x00, 0x09, 0x20, 0x20, 0x91, 0x02,
186 0x08, 0x02, 0xb0, 0x41, 0x08, 0x30, 0x00, 0x09, 0x10, 0x00, 0x18, 0x02,
187 0x21, 0x02, 0x02, 0x00, 0x00, 0x24, 0x44, 0x08, 0x12, 0x60, 0x00, 0xb2,
188 0x44, 0x12, 0x02, 0x0c, 0xc0, 0x80, 0x40, 0xc8, 0x20, 0x04, 0x50, 0x20,
189 0x05, 0x00, 0xb0, 0x04, 0x0b, 0x04, 0x29, 0x53, 0x00, 0x61, 0x48, 0x30,
190 0x00, 0x82, 0x20, 0x29, 0x00, 0x16, 0x00, 0x53, 0x22, 0x20, 0x43, 0x10,
191 0x48, 0x00, 0x80, 0x04, 0xd2, 0x00, 0x40, 0x00, 0xa2, 0x44, 0x03, 0x80,
192 0x29, 0x00, 0x04, 0x08, 0xc0, 0x04, 0x64, 0x40, 0x30, 0x28, 0x09, 0x84,
193 0x44, 0x50, 0x80, 0x21, 0x02, 0x92, 0x00, 0xc0, 0x10, 0x60, 0x88, 0x22,
194 0x08, 0x80, 0x00, 0x00, 0x18, 0x84, 0x04, 0x83, 0x96, 0x00, 0x81, 0x20,
195 0x05, 0x02, 0x00, 0x45, 0x88, 0x84, 0x00, 0x51, 0x20, 0x20, 0x51, 0x86,
196 0x41, 0x4b, 0x94, 0x00, 0x80, 0x00, 0x08, 0x11, 0x20, 0x4c, 0x58, 0x80,
197 0x04, 0x03, 0x06, 0x20, 0x89, 0x00, 0x05, 0x08, 0x22, 0x05, 0x90, 0x00,
198 0x40, 0x00, 0x82, 0x09, 0x50, 0x00, 0x00, 0x00, 0xa0, 0x41, 0xc2, 0x20,
199 0x08, 0x00, 0x16, 0x08, 0x40, 0x26, 0x21, 0xd0, 0x90, 0x08, 0x81, 0x90,
200 0x41, 0x00, 0x02, 0x44, 0x08, 0x10, 0x0c, 0x0a, 0x86, 0x09, 0x90, 0x04,
201 0x00, 0xc8, 0xa0, 0x04, 0x08, 0x30, 0x20, 0x89, 0x84, 0x00, 0x11, 0x22,
202 0x2c, 0x40, 0x00, 0x08, 0x02, 0xb0, 0x01, 0x48, 0x02, 0x01, 0x09, 0x20,
203 0x04, 0x03, 0x04, 0x00, 0x80, 0x02, 0x60, 0x42, 0x30, 0x21, 0x4a, 0x10,
204 0x44, 0x09, 0x02, 0x00, 0x01, 0x24, 0x00, 0x12, 0x82, 0x21, 0x80, 0xa4,
205 0x20, 0x10, 0x02, 0x04, 0x91, 0xa0, 0x40, 0x18, 0x04, 0x00, 0x02, 0x06,
206 0x69, 0x09, 0x00, 0x05, 0x58, 0x02, 0x01, 0x00, 0x00, 0x48, 0x00, 0x00,
207 0x00, 0x03, 0x92, 0x20, 0x00, 0x34, 0x01, 0xc8, 0x20, 0x48, 0x08, 0x30,
208 0x08, 0x42, 0x80, 0x20, 0x91, 0x90, 0x68, 0x01, 0x04, 0x40, 0x12, 0x02,
209 0x61, 0x00, 0x12, 0x08, 0x01, 0xa0, 0x00, 0x11, 0x04, 0x21, 0x48, 0x04,
210 0x24, 0x92, 0x00, 0x0c, 0x01, 0x84, 0x04, 0x00, 0x00, 0x01, 0x12, 0x96,
211 0x40, 0x01, 0xa0, 0x41, 0x88, 0x22, 0x28, 0x88, 0x00, 0x44, 0x42, 0x80,
212 0x24, 0x12, 0x14, 0x01, 0x42, 0x90, 0x60, 0x1a, 0x10, 0x04, 0x81, 0x10,
213 0x48, 0x08, 0x06, 0x29, 0x83, 0x02, 0x40, 0x02, 0x24, 0x64, 0x80, 0x10,
214 0x05, 0x80, 0x10, 0x40, 0x02, 0x02, 0x08, 0x42, 0x84, 0x01, 0x09, 0x20,
215 0x04, 0x50, 0x00, 0x60, 0x11, 0x30, 0x40, 0x13, 0x02, 0x04, 0x81, 0x00,
216 0x09, 0x08, 0x20, 0x45, 0x4a, 0x10, 0x61, 0x90, 0x26, 0x0c, 0x08, 0x02,
217 0x21, 0x91, 0x00, 0x60, 0x02, 0x04, 0x00, 0x02, 0x00, 0x0c, 0x08, 0x06,
218 0x08, 0x48, 0x84, 0x08, 0x11, 0x02, 0x00, 0x80, 0xa4, 0x00, 0x5a, 0x20,
219 0x00, 0x88, 0x04, 0x04, 0x02, 0x00, 0x09, 0x00, 0x14, 0x08, 0x49, 0x14,
220 0x20, 0xc8, 0x00, 0x04, 0x91, 0xa0, 0x40, 0x59, 0x80, 0x00, 0x12, 0x10,
221 0x00, 0x80, 0x80, 0x65, 0x00, 0x00, 0x04, 0x00, 0x80, 0x40, 0x19, 0x00,
222 0x21, 0x03, 0x84, 0x60, 0xc0, 0x04, 0x24, 0x1a, 0x12, 0x61, 0x80, 0x80,
223 0x08, 0x02, 0x04, 0x09, 0x42, 0x12, 0x20, 0x08, 0x34, 0x04, 0x90, 0x20,
224 0x01, 0x01, 0xa0, 0x00, 0x0b, 0x00, 0x08, 0x91, 0x92, 0x40, 0x02, 0x34,
225 0x40, 0x88, 0x10, 0x61, 0x19, 0x02, 0x00, 0x40, 0x04, 0x25, 0xc0, 0x80,
226 0x68, 0x08, 0x04, 0x21, 0x80, 0x22, 0x04, 0x00, 0xa0, 0x0c, 0x01, 0x84,
227 0x20, 0x41, 0x00, 0x08, 0x8a, 0x00, 0x20, 0x8a, 0x00, 0x48, 0x88, 0x04,
228 0x04, 0x11, 0x82, 0x08, 0x40, 0x86, 0x09, 0x49, 0xa4, 0x40, 0x00, 0x10,
229 0x01, 0x01, 0xa2, 0x04, 0x50, 0x80, 0x0c, 0x80, 0x00, 0x48, 0x82, 0xa0,
230 0x01, 0x18, 0x12, 0x41, 0x01, 0x04, 0x48, 0x41, 0x00, 0x24, 0x01, 0x00,
231 0x00, 0x88, 0x14, 0x00, 0x02, 0x00, 0x68, 0x01, 0x20, 0x08, 0x4a, 0x22,
232 0x08, 0x83, 0x80, 0x00, 0x89, 0x04, 0x01, 0xc2, 0x00, 0x00, 0x00, 0x34,
233 0x04, 0x00, 0x82, 0x28, 0x02, 0x02, 0x41, 0x4a, 0x90, 0x05, 0x82, 0x02,
234 0x09, 0x80, 0x24, 0x04, 0x41, 0x00, 0x01, 0x92, 0x80, 0x28, 0x01, 0x14,
235 0x00, 0x50, 0x20, 0x4c, 0x10, 0xb0, 0x04, 0x43, 0xa4, 0x21, 0x90, 0x04,
236 0x01, 0x02, 0x00, 0x44, 0x48, 0x00, 0x64, 0x08, 0x06, 0x00, 0x42, 0x20,
237 0x08, 0x02, 0x92, 0x01, 0x4a, 0x00, 0x20, 0x50, 0x32, 0x25, 0x90, 0x22,
238 0x04, 0x09, 0x00, 0x08, 0x11, 0x80, 0x21, 0x01, 0x10, 0x05, 0x00, 0x32,
239 0x08, 0x88, 0x94, 0x08, 0x08, 0x24, 0xd, 0xc1, 0x80, 0x40, 0xb, 0x20,
240 0x40, 0x18, 0x12, 0x04, 0x00, 0x22, 0x40, 0x10, 0x26, 0x05, 0xc1, 0x82,
241 0x00, 0x01, 0x30, 0x24, 0x02, 0x22, 0x41, 0x08, 0x24, 0x48, 0x1a, 0x00,
242 0x25, 0xd2, 0x12, 0x28, 0x42, 0x00, 0x04, 0x40, 0x30, 0x41, 0x00, 0x02,
243 0x00, 0x13, 0x20, 0x24, 0xd1, 0x84, 0x08, 0x89, 0x80, 0x04, 0x52, 0x00,
244 0x44, 0x18, 0xa4, 0x00, 0x00, 0x06, 0x20, 0x91, 0x10, 0x09, 0x42, 0x20,
245 0x24, 0x40, 0x30, 0x28, 0x00, 0x84, 0x40, 0x40, 0x80, 0x08, 0x10, 0x04,
246 0x09, 0x08, 0x04, 0x40, 0x08, 0x22, 0x00, 0x19, 0x02, 0x00, 0x00, 0x80,
247 0x2c, 0x02, 0x02, 0x21, 0x01, 0x90, 0x20, 0x40, 0x00, 0x0c, 0x00, 0x34,
248 0x48, 0x58, 0x20, 0x01, 0x43, 0x04, 0x20, 0x80, 0x14, 0x00, 0x90, 0x00,
249 0x6d, 0x11, 0x00, 0x00, 0x40, 0x20, 0x00, 0x03, 0x10, 0x40, 0x88, 0x30,

250 0x05, 0x4a, 0x00, 0x65, 0x10, 0x24, 0x08, 0x18, 0x84, 0x28, 0x03, 0x80,
251 0x20, 0x42, 0xb0, 0x40, 0x00, 0x10, 0x69, 0x19, 0x04, 0x00, 0x00, 0x80,
252 0x04, 0xc2, 0x04, 0x00, 0x01, 0x00, 0x05, 0x00, 0x22, 0x25, 0x08, 0x96,
253 0x04, 0x02, 0x22, 0x00, 0xd0, 0x10, 0x29, 0x01, 0xa0, 0x60, 0x08, 0x10,
254 0x04, 0x01, 0x16, 0x44, 0x10, 0x02, 0x28, 0x02, 0x82, 0x48, 0x40, 0x84,
255 0x20, 0x90, 0x22, 0x28, 0x80, 0x04, 0x00, 0x40, 0x04, 0x24, 0x00, 0x80,
256 0x29, 0x03, 0x10, 0x60, 0x48, 0x00, 0x00, 0x81, 0xa0, 0x00, 0x51, 0x20,
257 0x0c, 0xd1, 0x00, 0x01, 0x41, 0x20, 0x04, 0x92, 0x00, 0x00, 0x10, 0x92,
258 0x00, 0x42, 0x04, 0x05, 0x01, 0x86, 0x40, 0x80, 0x10, 0x20, 0x52, 0x20,
259 0x21, 0x00, 0x10, 0x48, 0x0a, 0x02, 0x00, 0xd0, 0x12, 0x41, 0x48, 0x80,
260 0x04, 0x00, 0x00, 0x48, 0x09, 0x22, 0x04, 0x00, 0x24, 0x00, 0x43, 0x10,
261 0x60, 0x0a, 0x00, 0x44, 0x12, 0x20, 0x2c, 0x08, 0x20, 0x44, 0x00, 0x84,
262 0x09, 0x40, 0x06, 0x08, 0xc1, 0x00, 0x40, 0x80, 0x20, 0x00, 0x98, 0x12,
263 0x48, 0x10, 0xa2, 0x20, 0x00, 0x84, 0x48, 0xc0, 0x10, 0x20, 0x90, 0x12,
264 0x08, 0x98, 0x82, 0x00, 0x0a, 0xa0, 0x04, 0x03, 0x00, 0x28, 0xc3, 0x00,
265 0x44, 0x42, 0x10, 0x04, 0x08, 0x04, 0x40, 0x00, 0x00, 0x05, 0x10, 0x00,
266 0x21, 0x03, 0x80, 0x04, 0x88, 0x12, 0x69, 0x10, 0x00, 0x04, 0x08, 0x04,
267 0x04, 0x02, 0x84, 0x48, 0x49, 0x04, 0x20, 0x18, 0x02, 0x64, 0x80, 0x30,
268 0x08, 0x01, 0x02, 0x00, 0x52, 0x12, 0x49, 0x08, 0x20, 0x41, 0x88, 0x10,
269 0x48, 0x08, 0x34, 0x00, 0x01, 0x86, 0x05, 0xd0, 0x00, 0x00, 0x83, 0x84,
270 0x21, 0x40, 0x02, 0x41, 0x10, 0x80, 0x48, 0x40, 0xa2, 0x20, 0x51, 0x00,
271 0x00, 0x49, 0x00, 0x01, 0x90, 0x20, 0x40, 0x18, 0x02, 0x40, 0x02, 0x22,
272 0x05, 0x40, 0x80, 0x08, 0x82, 0x10, 0x20, 0x18, 0x00, 0x05, 0x01, 0x82,
273 0x40, 0x58, 0x00, 0x04, 0x81, 0x90, 0x29, 0x01, 0xa0, 0x64, 0x00, 0x22,
274 0x40, 0x01, 0xa2, 0x00, 0x18, 0x04, 0xd0, 0x00, 0x00, 0x60, 0x80, 0x94,
275 0x60, 0x82, 0x10, 0xd0, 0x80, 0x30, 0xc0, 0x12, 0x20, 0x00, 0x00, 0x12,
276 0x40, 0xc0, 0x20, 0x21, 0x58, 0x02, 0x41, 0x10, 0x80, 0x44, 0x03, 0x02,
277 0x04, 0x13, 0x90, 0x29, 0x08, 0x00, 0x44, 0xc0, 0x00, 0x21, 0x00, 0x26,
278 0x00, 0x1a, 0x80, 0x01, 0x13, 0x14, 0x20, 0x0a, 0x14, 0x20, 0x00, 0x32,
279 0x61, 0x08, 0x00, 0x40, 0x42, 0x20, 0x09, 0x80, 0x06, 0x01, 0x81, 0x80,
280 0x60, 0x42, 0x00, 0x68, 0x90, 0x82, 0x08, 0x42, 0x80, 0x04, 0x02, 0x80,
281 0x09, 0x0b, 0x04, 0x00, 0x98, 0x00, 0xc0, 0x81, 0x06, 0x44, 0x48, 0x84,
282 0x28, 0x03, 0x92, 0x00, 0x01, 0x80, 0x40, 0x0a, 0x00, 0xc0, 0x81, 0x02,
283 0x08, 0x51, 0x04, 0x28, 0x90, 0x02, 0x20, 0x09, 0x10, 0x60, 0x00, 0x00,
284 0x09, 0x81, 0xa0, 0xc0, 0x00, 0xa4, 0x09, 0x00, 0x02, 0x28, 0x80, 0x20,
285 0x00, 0x02, 0x02, 0x04, 0x81, 0x14, 0x04, 0x00, 0x04, 0x09, 0x11, 0x12,
286 0x60, 0x40, 0x20, 0x01, 0x48, 0x30, 0x40, 0x11, 0x00, 0x08, 0x0a, 0x86,
287 0x00, 0x00, 0x04, 0x60, 0x81, 0x04, 0x01, 0xd0, 0x02, 0x41, 0x18, 0x90,
288 0x00, 0x0a, 0x20, 0x00, 0xc1, 0x06, 0x01, 0x08, 0x80, 0x64, 0xca, 0x10,
289 0x04, 0x99, 0x80, 0x48, 0x01, 0x82, 0x20, 0x50, 0x90, 0x48, 0x80, 0x84,
290 0x20, 0x90, 0x22, 0x00, 0x19, 0x00, 0x04, 0x18, 0x20, 0x24, 0x10, 0x86,
291 0x40, 0xc2, 0x00, 0x24, 0x12, 0x10, 0x44, 0x00, 0x16, 0x08, 0x10, 0x24,
292 0x00, 0x12, 0x06, 0x01, 0x08, 0x90, 0x00, 0x12, 0x02, 0x4d, 0x10, 0x80,
293 0x40, 0x50, 0x22, 0x00, 0x43, 0x10, 0x01, 0x00, 0x30, 0x21, 0x0a, 0x00,
294 0x00, 0x01, 0x14, 0x00, 0x10, 0x84, 0x04, 0xc1, 0x10, 0x29, 0x0a, 0x00,
295 0x01, 0x8a, 0x00, 0x20, 0x01, 0x12, 0xc0, 0x49, 0x20, 0x04, 0x81, 0x00,
296 0x48, 0x01, 0x04, 0x60, 0x80, 0x12, 0xc0, 0x08, 0x10, 0x48, 0x4a, 0x04,
297 0x28, 0x10, 0x00, 0x28, 0x40, 0x84, 0x45, 0x50, 0x10, 0x60, 0x10, 0x06,
298 0x44, 0x01, 0x80, 0x09, 0x00, 0x86, 0x01, 0x42, 0xa0, 0x00, 0x90, 0x00,
299 0x05, 0x90, 0x22, 0x40, 0x41, 0x00, 0x08, 0x80, 0x02, 0x08, 0xc0, 0x00,
300 0x01, 0x58, 0x30, 0x49, 0x09, 0x14, 0x00, 0x41, 0x02, 0xc0, 0x02, 0x80,
301 0x40, 0x89, 0x00, 0x24, 0x08, 0x10, 0x05, 0x90, 0x32, 0x40, 0x0a, 0x82,
302 0x08, 0x00, 0x12, 0x61, 0x00, 0x04, 0x21, 0x00, 0x22, 0x04, 0x10, 0x24,
303 0x08, 0x0a, 0x04, 0x01, 0x10, 0x00, 0x20, 0x40, 0x84, 0x04, 0x88, 0x22,
304 0x20, 0x90, 0x12, 0x00, 0x53, 0x06, 0x24, 0x01, 0x04, 0x40, 0x0b, 0x14,
305 0x60, 0x82, 0x02, 0xd0, 0x10, 0x90, 0xc0, 0x08, 0x20, 0x09, 0x00, 0x14,
306 0x09, 0x80, 0x80, 0x24, 0x82, 0x00, 0x40, 0x01, 0x02, 0x44, 0x01, 0x20,
307 0xc0, 0x40, 0x84, 0x40, 0x0a, 0x10, 0x41, 0x00, 0x30, 0x05, 0x09, 0x80,
308 0x44, 0x08, 0x20, 0x20, 0x02, 0x00, 0x49, 0x43, 0x20, 0x21, 0x00, 0x20,
309 0x00, 0x01, 0xb6, 0x08, 0x40, 0x04, 0x08, 0x02, 0x80, 0x01, 0x41, 0x80,
310 0x40, 0x08, 0x10, 0x24, 0x00, 0x20, 0x04, 0x12, 0x86, 0x09, 0xc0, 0x12,
311 0x21, 0x81, 0x14, 0x04, 0x00, 0x02, 0x20, 0x89, 0xb4, 0x44, 0x12, 0x80,
312 0x00, 0xd1, 0x00, 0x69, 0x40, 0x80, 0x00, 0x42, 0x12, 0x00, 0x18, 0x04,
313 0x00, 0x49, 0x06, 0x21, 0x02, 0x04, 0x28, 0x02, 0x84, 0x01, 0xc0, 0x10,
314 0x68, 0x00, 0x20, 0x08, 0x40, 0x00, 0x08, 0x91, 0x10, 0x01, 0x81, 0x24,
315 0x04, 0xd2, 0x10, 0x4c, 0x88, 0x86, 0x00, 0x10, 0x80, 0xc0, 0x02, 0x14,

```

316    0x00, 0x8a, 0x90, 0x40, 0x18, 0x20, 0x21, 0x80, 0xa4, 0x00, 0x58, 0x24,
317    0x20, 0x10, 0x10, 0x60, 0xc1, 0x30, 0x41, 0x48, 0x02, 0x48, 0x09, 0x00,
318    0x40, 0x09, 0x02, 0x05, 0x11, 0x82, 0x20, 0x4a, 0x20, 0x24, 0x18, 0x02,
319    0x0c, 0x10, 0x22, 0x0c, 0x0a, 0x04, 0x00, 0x03, 0x06, 0x48, 0x48, 0x04,
320    0x04, 0x02, 0x00, 0x21, 0x80, 0x84, 0x00, 0x18, 0x00, 0x0c, 0x02, 0x12,
321    0x01, 0x00, 0x14, 0x05, 0x82, 0x10, 0x41, 0x89, 0x12, 0x08, 0x40, 0xa4,
322    0x21, 0x01, 0x84, 0x48, 0x02, 0x10, 0x60, 0x40, 0x02, 0x28, 0x00, 0x14,
323    0x08, 0x40, 0xa0, 0x20, 0x51, 0x12, 0x00, 0xc2, 0x00, 0x01, 0x1a, 0x30,
324    0x40, 0x89, 0x12, 0x4c, 0x02, 0x80, 0x00, 0x00, 0x14, 0x01, 0x01, 0xa0,
325    0x21, 0x18, 0x22, 0x21, 0x18, 0x06, 0x40, 0x01, 0x80, 0x00, 0x90, 0x04,
326    0x48, 0x02, 0x30, 0x04, 0x08, 0x00, 0x05, 0x88, 0x24, 0x08, 0x48, 0x04,
327    0x24, 0x02, 0x06, 0x00, 0x80, 0x00, 0x00, 0x00, 0x10, 0x65, 0x11, 0x90,
328    0x00, 0x0a, 0x82, 0x04, 0xc3, 0x04, 0x60, 0x48, 0x24, 0x04, 0x92, 0x02,
329    0x44, 0x88, 0x80, 0x40, 0x18, 0x06, 0x29, 0x80, 0x10, 0x01, 0x00, 0x00,
330    0x44, 0xc8, 0x10, 0x21, 0x89, 0x30, 0x00, 0x4b, 0xa0, 0x01, 0x10, 0x14,
331    0x00, 0x02, 0x94, 0x40, 0x00, 0x20, 0x65, 0x00, 0xa2, 0x0c, 0x40, 0x22,
332    0x20, 0x81, 0x12, 0x20, 0x82, 0x04, 0x01, 0x10, 0x00, 0x08, 0x88, 0x00,
333    0x00, 0x11, 0x80, 0x04, 0x42, 0x80, 0x40, 0x41, 0x14, 0x00, 0x40, 0x32,
334    0x2c, 0x80, 0x24, 0x04, 0x19, 0x00, 0x00, 0x91, 0x00, 0x20, 0x83, 0x00,
335    0x05, 0x40, 0x20, 0x09, 0x01, 0x84, 0x40, 0x40, 0x20, 0x20, 0x11, 0x00,
336    0x40, 0x41, 0x90, 0x20, 0x00, 0x00, 0x40, 0x90, 0x92, 0x48, 0x18, 0x06,
337    0x08, 0x81, 0x80, 0x48, 0x01, 0x34, 0x24, 0x10, 0x20, 0x04, 0x00, 0x20,
338    0x04, 0x18, 0x06, 0x2d, 0x90, 0x10, 0x01, 0x00, 0x90, 0x00, 0x0a, 0x22,
339    0x01, 0x00, 0x22, 0x00, 0x11, 0x84, 0x01, 0x01, 0x00, 0x20, 0x88, 0x00,
340    0x44, 0x00, 0x22, 0x01, 0x00, 0xa6, 0x40, 0x02, 0x06, 0x20, 0x11, 0x00,
341    0x01, 0xc8, 0xa0, 0x04, 0x8a, 0x00, 0x28, 0x19, 0x80, 0x00, 0x52, 0xa0,
342    0x24, 0x12, 0x12, 0x09, 0x08, 0x24, 0x01, 0x48, 0x00, 0x04, 0x00, 0x24,
343    0x40, 0x02, 0x84, 0x08, 0x00, 0x04, 0x48, 0x40, 0x90, 0x60, 0x0a, 0x22,
344    0x01, 0x88, 0x14, 0x08, 0x01, 0x02, 0x08, 0xd3, 0x00, 0x20, 0xc0, 0x90,
345    0x24, 0x10, 0x00, 0x00, 0x01, 0xb0, 0x08, 0x0a, 0xa0, 0x00, 0x80, 0x00,
346    0x01, 0x09, 0x00, 0x20, 0x52, 0x02, 0x25, 0x00, 0x24, 0x04, 0x02, 0x84,
347    0x24, 0x10, 0x92, 0x40, 0x02, 0xa0, 0x40, 0x00, 0x22, 0x08, 0x11, 0x04,
348    0x08, 0x01, 0x22, 0x00, 0x42, 0x14, 0x00, 0x09, 0x90, 0x21, 0x00, 0x30,
349    0x6c, 0x00, 0x00, 0x0c, 0x00, 0x22, 0x09, 0x90, 0x10, 0x28, 0x40, 0x00,
350    0x20, 0xc0, 0x20, 0x00, 0x90, 0x00, 0x40, 0x01, 0x82, 0x05, 0x12, 0x12,
351    0x09, 0xc1, 0x04, 0x61, 0x80, 0x02, 0x28, 0x81, 0x24, 0x00, 0x49, 0x04,
352    0x08, 0x10, 0x86, 0x29, 0x41, 0x80, 0x21, 0x0a, 0x30, 0x49, 0x88, 0x90,
353    0x00, 0x41, 0x04, 0x29, 0x81, 0x80, 0x41, 0x09, 0x00, 0x40, 0x12, 0x10,
354    0x40, 0x00, 0x10, 0x40, 0x48, 0x02, 0x05, 0x80, 0x02, 0x21, 0x40, 0x20,
355    0x00, 0x58, 0x20, 0x60, 0x00, 0x90, 0x48, 0x00, 0x80, 0x28, 0xc0, 0x80,
356    0x48, 0x00, 0x00, 0x44, 0x80, 0x02, 0x00, 0x09, 0x06, 0x00, 0x12, 0x02,
357    0x01, 0x00, 0x10, 0x08, 0x83, 0x10, 0x45, 0x12, 0x00, 0x2c, 0x08, 0x04,
358    0x44, 0x00, 0x20, 0x20, 0xc0, 0x10, 0x20, 0x01, 0x00, 0x05, 0xc8, 0x20,
359    0x04, 0x98, 0x10, 0x08, 0x10, 0x00, 0x24, 0x02, 0x16, 0x40, 0x88, 0x00,
360    0x61, 0x88, 0x12, 0x24, 0x80, 0xa6, 0x00, 0x42, 0x00, 0x08, 0x10, 0x06,
361    0x48, 0x40, 0xa0, 0x00, 0x50, 0x20, 0x04, 0x81, 0xa4, 0x40, 0x18, 0x00,
362    0x08, 0x10, 0x80, 0x01, 0x01};
363    #if RSA_KEY_SIEVE && SIMULATION && RSA_INSTRUMENT
364    UINT32 PrimeIndex = 0;
365    UINT32 failedAtIteration[10] = {0};
366    UINT32 PrimeCounts[3] = {0};
367    UINT32 MillerRabinTrials[3] = {0};
368    UINT32 totalFieldsSieved[3] = {0};
369    UINT32 bitsInFieldAfterSieve[3] = {0};
370    UINT32 emptyFieldsSieved[3] = {0};
371    UINT32 noPrimeFields[3] = {0};
372    UINT32 primesChecked[3] = {0};
373    UINT16 lastSievePrime = 0;
374    #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  #if USE_RSA_KEY_CACHE
3  #include <stdio.h>
4  #include "Platform_fp.h"
5  #include "RsaKeyCache_fp.h"
6  #if CRT_FORMAT_RSA == YES
7  #define CACHE_FILE_NAME "RsaKeyCacheCrt.data"
8  #else
9  #define CACHE_FILE_NAME "RsaKeyCacheNoCrt.data"
10 #endif
11 typedef struct _RSA_KEY_CACHE_
12 {
13     TPM2B_PUBLIC_KEY_RSA    publicModulus;
14     TPM2B_PRIVATE_KEY_RSA   privateExponent;
15 } RSA_KEY_CACHE;
```

Determine the number of RSA key sizes for the cache

```

16 TPMI_RSA_KEY_BITS    SupportedRsaKeySizes[] = {
17 #if RSA_1024
18     1024,
19 #endif
20 #if RSA_2048
21     2048,
22 #endif
23 #if RSA_3072
24     3072,
25 #endif
26 #if RSA_4096
27     4096,
28 #endif
29     0
30 };
31 #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

```
32  RSA_KEY_CACHE          s_rsaKeyCache[RSA_KEY_CACHE_ENTRIES];
```

Indicates if the key cache is loaded. It can be loaded and enabled or disabled.

```
33  BOOL                   s_keyCacheLoaded = 0;
```

Indicates if the key cache is enabled

```
34  int                    s_rsaKeyCacheEnabled = FALSE;
```

10.2.21.2.1 RsaKeyCacheControl()

Used to enable and disable the RSA key cache.

```
35  LIB_EXPORT void
36  RsaKeyCacheControl(
37      int                state
38  )
39  {
40      s_rsaKeyCacheEnabled = state;
41  }
```

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

```
42  static BOOL
43  InitializeKeyCache(
44      TPMT_PUBLIC          *publicArea,
45      TPMT_SENSITIVE       *sensitive,
46      RAND_STATE           *rand          // IN: if not NULL, the deterministic
47                                          // RNG state
48  )
49  {
50      int                index;
51      TPM_KEY_BITS       keySave = publicArea->parameters.rsaDetail.keyBits;
52      BOOL               OK = TRUE;
53      //
54      s_rsaKeyCacheEnabled = FALSE;
55      for(index = 0; OK && index < RSA_KEY_CACHE_ENTRIES; index++)
56      {
57          publicArea->parameters.rsaDetail.keyBits
58              = SupportedRsaKeySizes[index];
59          OK = (CryptRsaGenerateKey(publicArea, sensitive, rand) == TPM_RC_SUCCESS);
60          if(OK)
61          {
62              s_rsaKeyCache[index].publicModulus = publicArea->unique.rsa;
63              s_rsaKeyCache[index].privateExponent = sensitive->sensitive.rsa;
64          }
65      }
66      publicArea->parameters.rsaDetail.keyBits = keySave;
67      s_keyCacheLoaded = OK;
68      #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
```



```

69     if(OK)
70     {
71         FILE                *cacheFile;
72         const char          *fn = CACHE_FILE_NAME;
73
74         #if defined _MSC_VER
75             if(fopen_s(&cacheFile, fn, "w+b") != 0)
76             #else
77                 cacheFile = fopen(fn, "w+b");
78                 if(NULL == cacheFile)
79             #endif
80             {
81                 printf("Can't open %s for write.\n", fn);
82             }
83             else
84             {
85                 fseek(cacheFile, 0, SEEK_SET);
86                 if(fwrite(s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
87                     != sizeof(s_rsaKeyCache))
88                 {
89                     printf("Error writing cache to %s.", fn);
90                 }
91             }
92             if(cacheFile)
93                 fclose(cacheFile);
94         }
95     #endif
96     return s_keyCacheLoaded;
97 }

```

10.2.21.2.3 KeyCacheLoaded()

Checks that key cache is loaded.

Return Value	Meaning
TRUE(1)	cache loaded
FALSE(0)	cache not loaded

```

98 static BOOL
99 KeyCacheLoaded(
100     TPMT_PUBLIC      *publicArea,
101     TPMT_SENSITIVE   *sensitive,
102     RAND_STATE        *rand
103     // IN: if not NULL, the deterministic
104     //      RNG state
105 )
106 {
107     #if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
108         if(!s_keyCacheLoaded)
109         {
110             FILE                *cacheFile;
111             const char *        fn = CACHE_FILE_NAME;
112             #if defined _MSC_VER && 1
113                 if(fopen_s(&cacheFile, fn, "r+b") == 0)
114             #else
115                 cacheFile = fopen(fn, "r+b");
116                 if(NULL != cacheFile)
117             #endif
118             {
119                 fseek(cacheFile, 0L, SEEK_END);
120                 if(ftell(cacheFile) == sizeof(s_rsaKeyCache))
121                 {
122                     fseek(cacheFile, 0L, SEEK_SET);

```

```

122         s_keyCacheLoaded = (
123             fread(&s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
124             == sizeof(s_rsaKeyCache));
125     }
126     fclose(cacheFile);
127 }
128 }
129 #endif
130 if(!s_keyCacheLoaded)
131     s_rsaKeyCacheEnabled = InitializeKeyCache(publicArea, sensitive, rand);
132 return s_keyCacheLoaded;
133 }

```

10.2.21.2.4 GetCachedRsaKey()

Return Value	Meaning
TRUE(1)	key loaded
FALSE(0)	key not loaded

```

134 BOOL
135 GetCachedRsaKey(
136     TPMT_PUBLIC      *publicArea,
137     TPMT_SENSITIVE   *sensitive,
138     RAND_STATE        *rand
139     // IN: if not NULL, the deterministic
140     //      RNG state
141 )
142 {
143     int keyBits = publicArea->parameters.rsaDetail.keyBits;
144     int index;
145     //
146     if(KeyCacheLoaded(publicArea, sensitive, rand))
147     {
148         for(index = 0; index < RSA_KEY_CACHE_ENTRIES; index++)
149         {
150             if((s_rsaKeyCache[index].publicModulus.t.size * 8) == keyBits)
151             {
152                 publicArea->unique.rsa = s_rsaKeyCache[index].publicModulus;
153                 sensitive->sensitive.rsa = s_rsaKeyCache[index].privateExponent;
154                 return TRUE;
155             }
156         }
157         return FALSE;
158     }
159     return s_keyCacheLoaded;
160 }
161 #endif // defined SIMULATION && defined USE_RSA_KEY_CACHE

```

10.2.22 Ticket.c

10.2.22.1 Introduction

This clause contains the functions used for ticket computations.

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_GENERATED  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_GENERATED_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 // Get proper proof
69 proof = HierarchyGetProof(hierarchy);
70
71 // Fill in ticket fields
72 ticket->tag = type;
73 ticket->hierarchy = hierarchy;
74
75 // Start HMAC with hierarchy proof as the HMAC key
76 ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
77                                         &proof->b);
78 // TPM_ST_AUTH_SECRET or TPM_ST_AUTH_SIGNED,
79 CryptDigestUpdateInt(&hmacState, sizeof(UINT16), ticket->tag);
80 // cpHash
81 CryptDigestUpdate2B(&hmacState.hashState, &cpHashA->b);
82 // policyRef
83 CryptDigestUpdate2B(&hmacState.hashState, &policyRef->b);
84 // keyName
85 CryptDigestUpdate2B(&hmacState.hashState, &entityName->b);
86 // timeout
87 CryptDigestUpdateInt(&hmacState, sizeof(timeout), timeout);
88 if(timeout != 0)
89 {
90     // epoch

```

```

91     CryptDigestUpdateInt(&hmacState.hashState, sizeof(CLOCK_NONCE),
92                         g_timeEpoch);
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. 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      //
95
96      VERIFY((length = ASN1NextTag(ctx)) >= 1);
97      VERIFY(ctx->tag == ASN1_BITSTRING);
98      // Get the shift value for the bit field (how many bits to loop off of the end)
99      shift = NEXT_OCTET(ctx);
100     length--;
101     // the shift count has to make sense
102     VERIFY((shift < 8) && ((length > 0) || (shift == 0)));
103     // if there are any bytes left
104     for(; length > 0; length--)
105     {
106         if(length > 1)
107         {
108             // for all but the last octet, just shift and add the new octet
109             VERIFY((value & 0xFF000000) == 0); // can't loose significant bits
110             value = (value << 8) + NEXT_OCTET(ctx);
111         }
112         else
113         {
114             // for the last octet, just shift the accumulated value enough to
115             // accept the significant bits in the last octet and shift the last
116             // octet down
117             VERIFY(((value & (0xFF000000 << (8 - shift))) == 0);
118             value = (value << (8 - shift)) + (NEXT_OCTET(ctx) >> shift);
119         }
120     }
121     *val = value;
122     return TRUE;
123 Error:
124     ctx->size = -1;
125     return FALSE;
126 }
```

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.

```

127 void
128 ASN1InitializeMarshalContext(
129     ASN1MarshalContext *ctx,
130     INT16 length,
131     BYTE *buffer
132 )
133 {
134     ctx->buffer = buffer;
135     if(buffer)
136         ctx->offset = length;
137     else
138         ctx->offset = INT16_MAX;
139     ctx->end = ctx->offset;
140     ctx->depth = -1;
141 }

```

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.

```

142 void
143 ASN1StartMarshalContext(
144     ASN1MarshalContext *ctx
145 )
146 {
147     pAssert((ctx->depth + 1) < MAX_DEPTH);
148     ctx->depth++;

```

```

149     ctx->ends[ctx->depth] = ctx->end;
150     ctx->end = ctx->offset;
151 }

```

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

```

152  INT16
153  ASN1EndMarshalContext(
154      ASN1MarshalContext    *ctx
155  )
156  {
157      INT16                length;
158      pAssert(ctx->depth >= 0);
159      length = ctx->end - ctx->offset;
160      ctx->end = ctx->ends[ctx->depth--];
161      if((ctx->depth == -1) && (ctx->buffer))
162      {
163          MemoryCopy(ctx->buffer, ctx->buffer + ctx->offset, ctx->end - ctx->offset);
164      }
165      return length;
166  }

```

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

```

167  UINT16
168  ASN1EndEncapsulation(
169      ASN1MarshalContext    *ctx,
170      BYTE                  tag
171  )
172  {
173      // only add a leading zero for an encapsulated BIT STRING
174      if (tag == ASN1_BITSTRING)
175          ASN1PushByte(ctx, 0);
176      ASN1PushTagAndLength(ctx, tag, ctx->end - ctx->offset);
177      return ASN1EndMarshalContext(ctx);
178  }

```

10.2.23.3.6 ASN1PushByte()

```

179  BOOL
180  ASN1PushByte(
181      ASN1MarshalContext    *ctx,
182      BYTE                  b

```

```

183 )
184 {
185     if(ctx->offset > 0)
186     {
187         ctx->offset -= 1;
188         if(ctx->buffer)
189             ctx->buffer[ctx->offset] = b;
190         return TRUE;
191     }
192     ctx->offset = -1;
193     return FALSE;
194 }

```

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

```

195 INT16
196 ASN1PushBytes(
197     ASN1MarshalContext    *ctx,
198     INT16                  count,
199     const BYTE             *buffer
200 )
201 {
202     // make sure that count is not negative which would mess up the math; and that
203     // if there is a count, there is a buffer
204     VERIFY((count >= 0) && ((buffer != NULL) || (count == 0)));
205     // back up the offset to determine where the new octets will get pushed
206     ctx->offset -= count;
207     // can't go negative
208     VERIFY(ctx->offset >= 0);
209     // if there are buffers, move the data, otherwise, assume that this is just a
210     // test.
211     if(count && buffer && ctx->buffer)
212         MemoryCopy(&ctx->buffer[ctx->offset], buffer, count);
213     return count;
214 Error:
215     ctx->offset = -1;
216     return 0;
217 }

```

10.2.23.3.8 ASN1PushNull()

Return Value	Meaning
> 0	count bytes
== 0	failure unless count was zero

```

218 INT16
219 ASN1PushNull(
220     ASN1MarshalContext    *ctx
221 )
222 {
223     ASN1PushByte(ctx, 0);
224     ASN1PushByte(ctx, ASN1_NULL);
225     return (ctx->offset >= 0) ? 2 : 0;

```

226 }

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

```

227  INT16
228  ASN1PushLength(
229      ASN1MarshalContext      *ctx,
230      INT16                    len
231  )
232  {
233      UINT16                    start = ctx->offset;
234      VERIFY(len >= 0);
235      if(len <= 127)
236          ASN1PushByte(ctx, (BYTE)len);
237      else
238      {
239          ASN1PushByte(ctx, (BYTE)(len & 0xFF));
240          len >>= 8;
241          if(len == 0)
242              ASN1PushByte(ctx, 0x81);
243          else
244          {
245              ASN1PushByte(ctx, (BYTE)(len));
246              ASN1PushByte(ctx, 0x82);
247          }
248      }
249      goto Exit;
250  Error:
251      ctx->offset = -1;
252  Exit:
253      return (ctx->offset > 0) ? start - ctx->offset : 0;
254  }
```

10.2.23.3.10 ASN1PushTagAndLength()

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

255  INT16
256  ASN1PushTagAndLength(
257      ASN1MarshalContext      *ctx,
258      BYTE                    tag,
259      INT16                    length
260  )
261  {
262      INT16                    bytes;
263      bytes = ASN1PushLength(ctx, length);
264      bytes += (INT16)ASN1PushByte(ctx, tag);
265      return (ctx->offset < 0) ? 0 : bytes;
266  }
```

10.2.23.3.11 ASN1PushTaggedOctetString()

This function will push a random octet string.

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

267  INT16
268  ASN1PushTaggedOctetString(
269      ASN1MarshalContext    *ctx,
270      INT16                  size,
271      const BYTE             *string,
272      BYTE                   tag
273  )
274  {
275      ASN1PushBytes(ctx, size, string);
276      // PushTagAndLength just tells how many octets it added so the total size of this
277      // element is the sum of those octets and input size.
278      size += ASN1PushTagAndLength(ctx, tag, size);
279      return size;
280  }
```

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

```

281  INT16
282  ASN1PushUINT(
283      ASN1MarshalContext    *ctx,
284      UINT32                 integer
285  )
286  {
287      BYTE                  marshaled[4];
288      UINT32_TO_BYTE_ARRAY(integer, marshaled);
289      return ASN1PushInteger(ctx, 4, marshaled);
290  }
```

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

```

291  INT16
292  ASN1PushInteger(
293      ASN1MarshalContext    *ctx,          // IN/OUT: buffer context
294      INT16                  iLen,         // IN: octets of the integer
```



```

295     BYTE                *integer        // IN: big-endian integer
296 )
297 {
298     // no leading 0's
299     while((*integer == 0) && (--iLen > 0))
300         integer++;
301     // Move the bytes to the buffer
302     ASN1PushBytes(ctx, iLen, integer);
303     // if needed, add a leading byte of 0 to make the number positive
304     if(*integer & 0x80)
305         iLen += (INT16)ASN1PushByte(ctx, 0);
306     // PushTagAndLength just tells how many octets it added so the total size of this
307     // element is the sum of those octets and the adjusted input size.
308     iLen += ASN1PushTagAndLength(ctx, ASN1_INTEGER, iLen);
309     return iLen;
310 }

```

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

```

311     INT16
312     ASN1PushOID(
313         ASN1MarshalContext    *ctx,
314         const BYTE             *OID
315     )
316     {
317         if((*OID == ASN1_OBJECT_IDENTIFIER) && ((OID[1] & 0x80) == 0))
318         {
319             return ASN1PushBytes(ctx, OID[1] + 2, OID);
320         }
321         ctx->offset = -1;
322         return 0;
323     }

```

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_spt_fp.h"
6  #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

```

7  INT16
8  X509PushPoint(
9      ASN1MarshalContext    *ctx,
10     TPMS_ECC_POINT        *p
11 )
12 {
13     // Push a bit string containing the public key. For now, push the x, and y
14     // coordinates of the public point, bottom up
15     ASN1StartMarshalContext(ctx); // BIT STRING
16     {
17         ASN1PushBytes(ctx, p->y.t.size, p->y.t.buffer);
18         ASN1PushBytes(ctx, p->x.t.size, p->x.t.buffer);
19         ASN1PushByte(ctx, 0x04);
20     }
21     return ASN1EndEncapsulation(ctx, ASN1_BITSTRING); // Ends BIT STRING
22 }

```

10.2.24.2.2 X509AddSigningAlgorithmECC()

This creates the signing algorithm data.

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

23  INT16
24  X509AddSigningAlgorithmECC(
25      OBJECT                *signKey,
26      TPMT_SIG_SCHEME       *scheme,
27      ASN1MarshalContext    *ctx
28  )
29  {
30      PHASH_DEF              hashDef = CryptGetHashDef(scheme->details.any.hashAlg);
31      //
32      NOT_REFERENCED(signKey);

```

```

33     // If the desired hashAlg definition wasn't found...
34     if(hashDef->hashAlg != scheme->details.any.hashAlg)
35         return 0;
36
37     switch(scheme->scheme)
38     {
39         case ALG_ECDSA_VALUE:
40             // Make sure that we have an OID for this hash and ECC
41             if((hashDef->ECDSA)[0] != ASN1_OBJECT_IDENTIFIER)
42                 break;
43             // if this is just an implementation check, indicate that this
44             // combination is supported
45             if(!ctx)
46                 return 1;
47             ASN1StartMarshalContext(ctx);
48             ASN1PushOID(ctx, hashDef->ECDSA);
49             return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
50         default:
51             break;
52     }
53     return 0;
54 }

```

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

```

55     INT16
56     X509AddPublicECC(
57         OBJECT *object,
58         ASN1MarshalContext *ctx
59     )
60     {
61         const BYTE *curveOid =
62             CryptEccGetOID(object->publicArea.parameters.eccDetail.curveID);
63         if((curveOid == NULL) || (*curveOid != ASN1_OBJECT_IDENTIFIER))
64             return 0;
65         //
66         //
67         // SEQUENCE (2 elem) 1st
68         // SEQUENCE (2 elem) 2nd
69         // OBJECT IDENTIFIER 1.2.840.10045.2.1 ecPublicKey (ANSI X9.62 public key type)
70         // OBJECT IDENTIFIER 1.2.840.10045.3.1.7 prime256v1 (ANSI X9.62 named curve)
71         // BIT STRING (520 bit) 000001001010000111010101010111001001101101000100000010...
72         //
73         // If this is a check to see if the key can be encoded, it can.
74         // Need to mark the end sequence
75         if(ctx == NULL)
76             return 1;
77         ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
78         {
79             X509PushPoint(ctx, &object->publicArea.unique.ecc); // BIT STRING
80             ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 2nd
81             {
82                 ASN1PushOID(ctx, curveOid); // curve dependent
83                 ASN1PushOID(ctx, OID_ECC_PUBLIC); // (1.2.840.10045.2.1)
84             }

```

```
85     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 2nd
86 }
87 return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 1st
88 }
```

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_spt_fp.h"
5  #include "CryptHash_fp.h"
6  #include "CryptRsa_fp.h"

```

10.2.25.2 Functions

```

7  #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

```

8  INT16
9  X509AddSigningAlgorithmRSA(
10     OBJECT          *signKey,
11     TPMT_SIG_SCHEME *scheme,
12     ASN1MarshalContext *ctx
13 )
14 {
15     TPM_ALG_ID      hashAlg = scheme->details.any.hashAlg;
16     PHASH_DEF       hashDef = CryptGetHashDef(hashAlg);
17     //
18     NOT_REFERENCED(signKey);
19     // return failure if hash isn't implemented
20     if(hashDef->hashAlg != hashAlg)
21         return 0;
22     switch(scheme->scheme)
23     {
24     case ALG_RSASSA_VALUE:
25     {
26         // if the hash is implemented but there is no PKCS1 OID defined
27         // then this is not a valid signing combination.
28         if(hashDef->PKCS1[0] != ASN1_OBJECT_IDENTIFIER)
29             break;
30         if(ctx == NULL)
31             return 1;
32         ASN1StartMarshalContext(ctx);
33         ASN1PushOID(ctx, hashDef->PKCS1);
34         return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
35     }
36     case ALG_RSAPSS_VALUE:
37         // leave if this is just an implementation check
38         if(ctx == NULL)
39             return 1;
40         // In the case of SHA1, everything is default and RFC4055 says that
41         // implementations that do signature generation MUST omit the parameter
42         // when defaults are used. )-:
43         if(hashDef->hashAlg == ALG_SHA1_VALUE)
44         {

```

```

45         return X509PushAlgorithmIdentifierSequence(ctx, OID_RSAPSS);
46     }
47     else
48     {
49         // Going to build something that looks like:
50         // SEQUENCE (2 elem)
51         //   OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
52         // SEQUENCE (3 elem)
53         //   [0] (1 elem)
54         //     SEQUENCE (2 elem)
55         //       OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
56         //       NULL
57         //   [1] (1 elem)
58         //     SEQUENCE (2 elem)
59         //       OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
60         //       SEQUENCE (2 elem)
61         //         OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
62         //         NULL
63         //   [2] (1 elem) salt length
64         //     INTEGER 32
65
66         // The indentation is just to keep track of where we are in the
67         // structure
68         ASN1StartMarshalContext(ctx); // SEQUENCE (2 elements)
69         {
70             ASN1StartMarshalContext(ctx); // SEQUENCE (3 elements)
71             {
72                 // [2] (1 elem) salt length
73                 //   INTEGER 32
74                 ASN1StartMarshalContext(ctx);
75                 {
76                     INT16 saltSize =
77                         CryptRsaPssSaltSize((INT16)hashDef->digestSize,
78                         (INT16)signKey->publicArea.unique.rsa.t.size);
79                     ASN1PushUINT(ctx, saltSize);
80                 }
81                 ASN1EndEncapsulation(ctx, ASN1_APPLICATION_SPECIFIC + 2);
82
83                 // Add the mask generation algorithm
84                 // [1] (1 elem)
85                 // SEQUENCE (2 elem) 1st
86                 //   OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF
87                 // SEQUENCE (2 elem) 2nd
88                 //   OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256
89                 //   NULL
90                 ASN1StartMarshalContext(ctx); // mask context [1] (1 elem)
91                 {
92                     ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
93                     // Handle the 2nd Sequence (sequence (object, null))
94                     {
95                         X509PushAlgorithmIdentifierSequence(ctx,
96                         hashDef->OID);
97                         // add the pkcs1-MGF OID
98                         ASN1PushOID(ctx, OID_MGF1);
99                     }
100                     // End outer sequence
101                     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
102                 }
103                 // End the [1]
104                 ASN1EndEncapsulation(ctx, ASN1_APPLICATION_SPECIFIC + 1);
105
106                 // Add the hash algorithm
107                 // [0] (1 elem)
108                 // SEQUENCE (2 elem) (done by
109                 //   X509PushAlgorithmIdentifierSequence)
110                 //   OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256 (NIST)

```

```

111         // NULL
112         ASN1StartMarshalContext(ctx); // [0] (1 elem)
113         {
114             X509PushAlgorithmIdentifierSequence(ctx, hashDef->OID);
115         }
116         ASN1EndEncapsulation(ctx, (ASN1_APPLICATION_SPECIFIC + 0));
117     }
118     // SEQUENCE (3 elements) end
119     ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
120
121     // RSA PSS OID
122     // OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)
123     ASN1PushOID(ctx, OID_RSAPSS);
124 }
125 // End Sequence (2 elements)
126 return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
127 }
128 default:
129     break;
130 }
131 return 0;
132 }

```

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

```

133 INT16
134 X509AddPublicRSA(
135     OBJECT *object,
136     ASN1MarshalContext *ctx
137 )
138 {
139     UINT32 exp = object->publicArea.parameters.rsaDetail.exponent;
140     //
141     /*
142     SEQUENCE (2 elem) 1st
143     SEQUENCE (2 elem) 2nd
144     OBJECT IDENTIFIER 1.2.840.113549.1.1.1 rsaEncryption (PKCS #1)
145     NULL
146     BIT STRING (1 elem)
147     SEQUENCE (2 elem) 3rd
148     INTEGER (2048 bit) 2197304513741227955725834199357401
149     INTEGER 65537
150     */
151     // If this is a check to see if the key can be encoded, it can.
152     // Need to mark the end sequence
153     if(ctx == NULL)
154         return 1;
155     ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
156     ASN1StartMarshalContext(ctx); // BIT STRING
157     ASN1StartMarshalContext(ctx); // SEQUENCE *(2 elem) 3rd
158
159     // Get public exponent in big-endian byte order.
160     if(exp == 0)
161         exp = RSA_DEFAULT_PUBLIC_EXPONENT;
162 }

```



```
163 // Push a 4 byte integer. This might get reduced if there are leading zeros or
164 // extended if the high order byte is negative.
165 ASN1PushUINT(ctx, exp);
166 // Push the public key as an integer
167 ASN1PushInteger(ctx, object->publicArea.unique.rsa.t.size,
168                 object->publicArea.unique.rsa.t.buffer);
169 // Embed this in a SEQUENCE tag and length in for the key, exponent sequence
170 ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // SEQUENCE (3rd)
171
172 // Embed this in a BIT STRING
173 ASN1EndEncapsulation(ctx, ASN1_BITSTRING);
174
175 // Now add the formatted SEQUENCE for the RSA public key OID. This is a
176 // fully constructed value so it doesn't need to have a context started
177 X509PushAlgorithmIdentifierSequence(ctx, OID_PKCS1_PUB);
178
179 return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
180 }
181 #endif // ALG_RSA
```

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 X509FindExtensionOID()

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      else if (ctx != ctxIn)
31          *ctx = *ctxIn;
32      for (; ctx->size > ctx->offset; ctx->offset += length)
33      {
34          VERIFY((length = ASN1NextTag(ctx)) >= 0);
35          // If this is not a constructed sequence, then it doesn't belong
36          // in the extensions.
37          VERIFY(ctx->tag == ASN1_CONSTRUCTED_SEQUENCE);
38          // Make sure that this entry could hold the OID
39          if (length >= OID_SIZE(OID))
40          {
41              // See if this is a match for the provided object identifier.
42              if (MemoryEqual(OID, &(ctx->buffer[ctx->offset]), OID_SIZE(OID)))
43              {

```

```

44         // Return with ' ctx' set to point to the start of the OID with the
size
45         // set to be the size of the SEQUENCE
46         ctx->buffer += ctx->offset;
47         ctx->offset = 0;
48         ctx->size = length;
49         return TRUE;
50     }
51 }
52 }
53 VERIFY(ctx->offset == ctx->size);
54 return FALSE;
55 Error:
56     ctxIn->size = -1;
57     ctx->size = -1;
58     return FALSE;
59 }

```

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

```

60  UINT32
61  X509GetExtensionBits(
62      ASN1UnmarshalContext      *ctx,
63      UINT32                     *value
64  )
65  {
66      INT16      length;
67      //
68      while (((length = ASN1NextTag(ctx)) > 0) && (ctx->size > ctx->offset))
69      {
70          // Since this is an extension, the extension value will be in an OCTET STRING
71          if (ctx->tag == ASN1_OCTET_STRING)
72          {
73              return ASN1GetBitStringValue(ctx, value);
74          }
75          ctx->offset += length;
76      }
77      ctx->size = -1;
78      return FALSE;
79  }

```

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 Returns	Meaning
TPM_RCS_ATTRIBUTES	the attributes of object are not consistent with the extension setting
TPM_RC_VALUE	problem parsing the extensions

```

80  TPM_RC
81  X509ProcessExtensions(

```

```

82     OBJECT                *object,           // IN: The object with the attributes to
83                               //             check
84     stringRef             *extension         // IN: The start and length of the extensions
85 )
86 {
87     ASN1UnmarshalContext    ctx;
88     ASN1UnmarshalContext    extensionCtx;
89     INT16                   length;
90     UINT32                  value;
91 //
92     if(!ASN1UnmarshalContextInitialize(&ctx, extension->len, extension->buf)
93         || ((length = ASN1NextTag(&ctx)) < 0)
94         || (ctx.tag != X509_EXTENSIONS))
95         return TPM_RCS_VALUE;
96     if( ((length = ASN1NextTag(&ctx)) < 0)
97         || (ctx.tag != (ASN1_CONSTRUCTED_SEQUENCE)))
98         return TPM_RCS_VALUE;
99
100    // Get the extension for the TPMA_OBJECT if there is one
101    if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_TCG_TPMA_OBJECT) &&
102        X509GetExtensionBits(&extensionCtx, &value))
103    {
104        // If an keyAttributes extension was found, it must be exactly the same as the
105        // attributes of the object.
106        // This cast will work because we know that a TPMA_OBJECT is in a UINT32.
107        // Set RUNTIME_SIZE_CHECKS to YES to force a check to verify this assumption
108        // during debug. Doing this is lot easier than having to revisit the code
109        // any time a new attribute is added.
110        // NOTE: MemoryEqual() is used to avoid type-punned pointer warning/error.
111        if(!MemoryEqual(&value, &object->publicArea.objectAttributes, sizeof(value)))
112            return TPM_RCS_ATTRIBUTES;
113    }
114    // Make sure the failure to find the value wasn't because of a fatal error
115    else if(extensionCtx.size < 0)
116        return TPM_RCS_VALUE;
117
118    // Get the keyUsage extension. This one is required
119    if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_KEY_USAGE_EXTENSION) &&
120        X509GetExtensionBits(&extensionCtx, &value))
121    {
122        x509KeyUsageUnion    keyUsage;
123        TPMA_OBJECT          attributes = object->publicArea.objectAttributes;
124        //
125        keyUsage.integer = value;
126        // For KeyUsage:
127        // the 'sign' attribute is SET if Key Usage includes signing
128        if(((keyUsageSign.integer & keyUsage.integer) != 0)
129            && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
130            // and the 'decrypt' attribute is Set if Key Usage includes decryption
131            uses
132            || ((keyUsageDecrypt.integer & keyUsage.integer) != 0)
133            && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
134            // Check that 'fixedTPM' is SET if Key Usage is non-repudiation
135            || (IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE, nonrepudiation)
136                && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
137            || (IS_ATTRIBUTE(keyUsage.x509, TPMA_X509_KEY_USAGE, keyAgreement)
138                && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
139            )
140            return TPM_RCS_ATTRIBUTES;
141    }
142    else
143        // The KeyUsage extension is required
144        return TPM_RCS_VALUE;
145    return TPM_RC_SUCCESS;
146 }

```

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

```

147  INT16
148  X509AddSigningAlgorithm(
149      ASN1MarshalContext  *ctx,
150      OBJECT               *signKey,
151      TPMT_SIG_SCHEME      *scheme
152  )
153  {
154      switch(signKey->publicArea.type)
155      {
156      #if ALG_RSA
157          case ALG_RSA_VALUE:
158              return X509AddSigningAlgorithmRSA(signKey, scheme, ctx);
159      #endif // ALG_RSA
160      #if ALG_ECC
161          case ALG_ECC_VALUE:
162              return X509AddSigningAlgorithmECC(signKey, scheme, ctx);
163      #endif // ALG_ECC
164      #if ALG_SM2
165          case ALG_SM2:
166              return X509AddSigningAlgorithmSM2(signKey, scheme, ctx);
167      #endif // ALG_SM2
168          default:
169              break;
170      }
171      return 0;
172  }

```

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

```

173  INT16
174  X509AddPublicKey(
175      ASN1MarshalContext  *ctx,
176      OBJECT               *object
177  )
178  {
179      switch(object->publicArea.type)
180      {
181      #if ALG_RSA
182          case ALG_RSA_VALUE:
183              return X509AddPublicKeyRSA(object, ctx);
184      #endif
185      #if ALG_ECC

```

```

186         case ALG_ECC_VALUE:
187             return X509AddPublicECC(object, ctx);
188     #endif
189     #if ALG_SM2
190         case ALG_SM2_VALUE:
191             break;
192     #endif
193     default:
194         break;
195     }
196     return FALSE;
197 }

```

10.2.26.3.3 X509PushAlgorithmIdentifierSequence()

Return Value	Meaning
> 0	number of bytes added
== 0	failure

```

198 INT16
199 X509PushAlgorithmIdentifierSequence(
200     ASN1MarshalContext *ctx,
201     const BYTE *OID
202 )
203 {
204     ASN1StartMarshalContext(ctx); // hash algorithm
205     ASN1PushNull(ctx);
206     ASN1PushOID(ctx, OID);
207     return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
208 }

```

10.2.27 AC_spt.c

10.2.27.1 Includes

```

1  #include "Tpm.h"
2  #include "AC_spt_fp.h"
3  #if 1 // This is the simulated AC data.
4  typedef struct {
5      TPMI_RH_AC          ac;
6      TPML_AC_CAPABILITIES *acData;
7  } acCapabilities;
8  TPML_AC_CAPABILITIES acData0001 = {1,
9      {{TPM_AT_PV1, 0x01234567}}};
10  acCapabilities ac[1] = {0x0001, &acData0001};
11  #define NUM_AC (sizeof(ac) / sizeof(acCapabilities))
12  #endif // 1 The simulated AC data

```

10.2.27.1.1 AcToCapabilities()

This function returns a pointer to a list of AC capabilities.

```

14  TPML_AC_CAPABILITIES *
15  AcToCapabilities(
16      TPMI_RH_AC    component    // IN: component
17  )
18  {
19      UINT32        index;
20      //
21      for(index = 0; index < NUM_AC; index++)
22      {
23          if(ac[index].ac == component)
24              return ac[index].acData;
25      }
26      return NULL;
27  }

```

10.2.27.1.2 AcIsAccessible()

Function to determine if an AC handle references an actual AC

Return Value	Meaning
--------------	---------

```

28  BOOL
29  AcIsAccessible(
30      TPM_HANDLE    acHandle
31  )
32  {
33      // In this implementation, the AC exists if there are some capabilities to go
34      // with the handle
35      return AcToCapabilities(acHandle) != NULL;
36  }

```

10.2.27.1.3 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

```

37  TPMI_YES_NO
38  AcCapabilitiesGet(
39      TPMI_RH_AC          component,      // IN: the component
40      TPM_AT              type,          // IN: start capability type
41      TPML_AC_CAPABILITIES *capabilityList // OUT: list of handle
42  )
43  {
44      TPMI_YES_NO          more = NO;
45      UINT32               i;
46      TPML_AC_CAPABILITIES *capabilities = AcToCapabilities(component);
47
48      pAssert(HandleGetType(component) == TPM_HT_AC);
49
50      // Initialize output handle list
51      capabilityList->count = 0;
52
53      if(capabilities != NULL)
54      {
55          // Find the first capability less than or equal to type
56          for(i = 0; i < capabilities->count; i++)
57          {
58              if(capabilities->acCapabilities[i].tag >= type)
59              {
60                  // copy the capabilities until we run out or fill the list
61                  for(; (capabilityList->count < MAX_AC_CAPABILITIES)
62                      && (i < capabilities->count); i++)
63                  {
64                      capabilityList->acCapabilities[capabilityList->count]
65                          = capabilities->acCapabilities[i];
66                      capabilityList->count++;
67                  }
68                  more = i < capabilities->count;
69              }
70          }
71      }
72      return more;
73  }

```

10.2.27.1.4 AcSendObject()

Stub to handle sending of an AC object

Error Returns	Meaning
---------------	---------

```

74  TPM_RC
75  AcSendObject(
76      TPM_HANDLE          acHandle,      // IN: Handle of AC receiving object
77      OBJECT              *object,      // IN: object structure to send
78      TPMS_AC_OUTPUT      *acDataOut    // OUT: results of operation
79  )
80  {
81      NOT_REFERENCED(object);
82      NOT_REFERENCED(acHandle);
83      acDataOut->tag = TPM_AT_ERROR; // indicate that the response contains an
84                                   // error code
85      acDataOut->data = TPM_AE_NONE; // but there is no error.
86  }

```

```
87     return TPM_RC_SUCCESS;  
88 }
```

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_
```

Table 2:4 - 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
```

Table 0:1 - 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
```

Table 0:4 - 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 NO
53 #endif

```

Table 0:7 - Defines for Implementation Values

```

54 #ifndef FIELD_UPGRADE_IMPLEMENTED
55 #define FIELD_UPGRADE_IMPLEMENTED NO
56 #endif
57 #ifndef RADIX_BITS
58 #define RADIX_BITS 32
59 #endif
60 #ifndef HASH_ALIGNMENT
61 #define HASH_ALIGNMENT 4
62 #endif
63 #ifndef SYMMETRIC_ALIGNMENT
64 #define SYMMETRIC_ALIGNMENT 4
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 BSIZE
76 #define BSIZE UINT16
77 #endif
78 #ifndef IMPLEMENTATION_PCR
79 #define IMPLEMENTATION_PCR 24
80 #endif
81 #ifndef PCR_SELECT_MAX
82 #define PCR_SELECT_MAX ((IMPLEMENTATION_PCR+7)/8)
83 #endif
84 #ifndef PLATFORM_PCR
85 #define PLATFORM_PCR 24
86 #endif
87 #ifndef PCR_SELECT_MIN
88 #define PCR_SELECT_MIN ((PLATFORM_PCR+7)/8)
89 #endif
90 #ifndef DRTM_PCR
91 #define DRTM_PCR 17

```

```

92  #endif
93  #ifndef HCRTM_PCR
94  #define HCRTM_PCR 0
95  #endif
96  #ifndef NUM_LOCALITIES
97  #define NUM_LOCALITIES 5
98  #endif
99  #ifndef MAX_HANDLE_NUM
100 #define MAX_HANDLE_NUM 3
101 #endif
102 #ifndef MAX_ACTIVE_SESSIONS
103 #define MAX_ACTIVE_SESSIONS 64
104 #endif
105 #ifndef CONTEXT_SLOT
106 #define CONTEXT_SLOT UINT16
107 #endif
108 #ifndef CONTEXT_COUNTER
109 #define CONTEXT_COUNTER UINT64
110 #endif
111 #ifndef MAX_LOADED_SESSIONS
112 #define MAX_LOADED_SESSIONS 3
113 #endif
114 #ifndef MAX_SESSION_NUM
115 #define MAX_SESSION_NUM 3
116 #endif
117 #ifndef MAX_LOADED_OBJECTS
118 #define MAX_LOADED_OBJECTS 3
119 #endif
120 #ifndef MIN_EVICT_OBJECTS
121 #define MIN_EVICT_OBJECTS 2
122 #endif
123 #ifndef NUM_POLICY_PCR_GROUP
124 #define NUM_POLICY_PCR_GROUP 1
125 #endif
126 #ifndef NUM_AUTHVALUE_PCR_GROUP
127 #define NUM_AUTHVALUE_PCR_GROUP 1
128 #endif
129 #ifndef MAX_CONTEXT_SIZE
130 #define MAX_CONTEXT_SIZE 2474
131 #endif
132 #ifndef MAX_DIGEST_BUFFER
133 #define MAX_DIGEST_BUFFER 1024
134 #endif
135 #ifndef MAX_NV_INDEX_SIZE
136 #define MAX_NV_INDEX_SIZE 2048
137 #endif
138 #ifndef MAX_NV_BUFFER_SIZE
139 #define MAX_NV_BUFFER_SIZE 1024
140 #endif
141 #ifndef MAX_CAP_BUFFER
142 #define MAX_CAP_BUFFER 1024
143 #endif
144 #ifndef NV_MEMORY_SIZE
145 #define NV_MEMORY_SIZE 16384
146 #endif
147 #ifndef MIN_COUNTER_INDICES
148 #define MIN_COUNTER_INDICES 8
149 #endif
150 #ifndef NUM_STATIC_PCR
151 #define NUM_STATIC_PCR 16
152 #endif
153 #ifndef MAX_ALG_LIST_SIZE
154 #define MAX_ALG_LIST_SIZE 64
155 #endif
156 #ifndef PRIMARY_SEED_SIZE
157 #define PRIMARY_SEED_SIZE 32

```

```

158 #endif
159 #ifndef CONTEXT_ENCRYPT_ALGORITHM
160 #define CONTEXT_ENCRYPT_ALGORITHM AES
161 #endif
162 #ifndef NV_CLOCK_UPDATE_INTERVAL
163 #define NV_CLOCK_UPDATE_INTERVAL 12
164 #endif
165 #ifndef NUM_POLICY_PCR
166 #define NUM_POLICY_PCR 1
167 #endif
168 #ifndef MAX_COMMAND_SIZE
169 #define MAX_COMMAND_SIZE 4096
170 #endif
171 #ifndef MAX_RESPONSE_SIZE
172 #define MAX_RESPONSE_SIZE 4096
173 #endif
174 #ifndef ORDERLY_BITS
175 #define ORDERLY_BITS 8
176 #endif
177 #ifndef MAX_SYM_DATA
178 #define MAX_SYM_DATA 128
179 #endif
180 #ifndef MAX_RNG_ENTROPY_SIZE
181 #define MAX_RNG_ENTROPY_SIZE 64
182 #endif
183 #ifndef RAM_INDEX_SPACE
184 #define RAM_INDEX_SPACE 512
185 #endif
186 #ifndef RSA_DEFAULT_PUBLIC_EXPONENT
187 #define RSA_DEFAULT_PUBLIC_EXPONENT 0x00010001
188 #endif
189 #ifndef ENABLE_PCR_NO_INCREMENT
190 #define ENABLE_PCR_NO_INCREMENT YES
191 #endif
192 #ifndef CRT_FORMAT_RSA
193 #define CRT_FORMAT_RSA YES
194 #endif
195 #ifndef VENDOR_COMMAND_COUNT
196 #define VENDOR_COMMAND_COUNT 0
197 #endif
198 #ifndef MAX_VENDOR_BUFFER_SIZE
199 #define MAX_VENDOR_BUFFER_SIZE 1024
200 #endif
201 #ifndef TPM_MAX_DERIVATION_BITS
202 #define TPM_MAX_DERIVATION_BITS 8192
203 #endif
204 #ifndef RSA_MAX_PRIME
205 #define RSA_MAX_PRIME (MAX_RSA_KEY_BYTES/2)
206 #endif
207 #ifndef RSA_PRIVATE_SIZE
208 #define RSA_PRIVATE_SIZE (RSA_MAX_PRIME*5)
209 #endif
210 #ifndef SIZE_OF_X509_SERIAL_NUMBER
211 #define SIZE_OF_X509_SERIAL_NUMBER 20
212 #endif
213 #ifndef PRIVATE_VENDOR_SPECIFIC_BYTES
214 #define PRIVATE_VENDOR_SPECIFIC_BYTES RSA_PRIVATE_SIZE
215 #endif

```

Table 0:2 - Defines for Implemented Algorithms

```

216 #ifndef ALG_AES
217 #define ALG_AES ALG_YES
218 #endif
219 #ifndef ALG_CAMELLIA

```

```

220 #define ALG_CAMELLIA          ALG_NO      /* Not specified by vendor */
221 #endif
222 #ifndef ALG_CBC
223 #define ALG_CBC                ALG_YES
224 #endif
225 #ifndef ALG_CFB
226 #define ALG_CFB                ALG_YES
227 #endif
228 #ifndef ALG_CMAC
229 #define ALG_CMAC                ALG_YES
230 #endif
231 #ifndef ALG_CTR
232 #define ALG_CTR                ALG_YES
233 #endif
234 #ifndef ALG_ECB
235 #define ALG_ECB                ALG_YES
236 #endif
237 #ifndef ALG_ECC
238 #define ALG_ECC                ALG_YES
239 #endif
240 #ifndef ALG_ECDSA
241 #define ALG_ECDSA              (ALG_YES && ALG_ECC)
242 #endif
243 #ifndef ALG_ECDH
244 #define ALG_ECDH              (ALG_YES && ALG_ECC)
245 #endif
246 #ifndef ALG_ECDSA
247 #define ALG_ECDSA              (ALG_YES && ALG_ECC)
248 #endif
249 #ifndef ALG_ECMQV
250 #define ALG_ECMQV              (ALG_NO && ALG_ECC)
251 #endif
252 #ifndef ALG_ECSCHNORR
253 #define ALG_ECSCHNORR         (ALG_YES && ALG_ECC)
254 #endif
255 #ifndef ALG_HMAC
256 #define ALG_HMAC                ALG_YES
257 #endif
258 #ifndef ALG_KDF1_SP800_108
259 #define ALG_KDF1_SP800_108     ALG_YES
260 #endif
261 #ifndef ALG_KDF1_SP800_56A
262 #define ALG_KDF1_SP800_56A     (ALG_YES && ALG_ECC)
263 #endif
264 #ifndef ALG_KDF2
265 #define ALG_KDF2                ALG_NO
266 #endif
267 #ifndef ALG_KEYEDHASH
268 #define ALG_KEYEDHASH          ALG_YES
269 #endif
270 #ifndef ALG_MGF1
271 #define ALG_MGF1                ALG_YES
272 #endif
273 #ifndef ALG_OAEP
274 #define ALG_OAEP                (ALG_YES && ALG_RSA)
275 #endif
276 #ifndef ALG_OFB
277 #define ALG_OFB                ALG_YES
278 #endif
279 #ifndef ALG_RSA
280 #define ALG_RSA                ALG_YES
281 #endif
282 #ifndef ALG_RSAES
283 #define ALG_RSAES              (ALG_YES && ALG_RSA)
284 #endif
285 #ifndef ALG_RSAPSS

```



```

286 #define ALG_RSAPSS (ALG_YES && ALG_RSA)
287 #endif
288 #ifndef ALG_RSASSA
289 #define ALG_RSASSA (ALG_YES && ALG_RSA)
290 #endif
291 #ifndef ALG_SHA
292 #define ALG_SHA ALG_NO /* Not specified by vendor */
293 #endif
294 #ifndef ALG_SHA1
295 #define ALG_SHA1 ALG_YES
296 #endif
297 #ifndef ALG_SHA256
298 #define ALG_SHA256 ALG_YES
299 #endif
300 #ifndef ALG_SHA384
301 #define ALG_SHA384 ALG_YES
302 #endif
303 #ifndef ALG_SHA3_256
304 #define ALG_SHA3_256 ALG_NO /* Not specified by vendor */
305 #endif
306 #ifndef ALG_SHA3_384
307 #define ALG_SHA3_384 ALG_NO /* Not specified by vendor */
308 #endif
309 #ifndef ALG_SHA3_512
310 #define ALG_SHA3_512 ALG_NO /* Not specified by vendor */
311 #endif
312 #ifndef ALG_SHA512
313 #define ALG_SHA512 ALG_NO
314 #endif
315 #ifndef ALG_SM2
316 #define ALG_SM2 (ALG_NO && ALG_ECC)
317 #endif
318 #ifndef ALG_SM3_256
319 #define ALG_SM3_256 ALG_NO
320 #endif
321 #ifndef ALG_SM4
322 #define ALG_SM4 ALG_NO
323 #endif
324 #ifndef ALG_SYMCIPHER
325 #define ALG_SYMCIPHER ALG_YES
326 #endif
327 #ifndef ALG_TDES
328 #define ALG_TDES ALG_NO
329 #endif
330 #ifndef ALG_XOR
331 #define ALG_XOR ALG_YES
332 #endif

```

Table 1:00 - Defines for RSA Asymmetric Cipher Algorithm Constants

```

333 #ifndef RSA_1024
334 #define RSA_1024 (ALG_RSA & YES)
335 #endif
336 #ifndef RSA_2048
337 #define RSA_2048 (ALG_RSA & YES)
338 #endif
339 #ifndef RSA_3072
340 #define RSA_3072 (ALG_RSA & NO)
341 #endif
342 #ifndef RSA_4096
343 #define RSA_4096 (ALG_RSA & NO)
344 #endif

```

Table 1:17 - Defines for AES Symmetric Cipher Algorithm Constants

```

345 #ifndef AES_128
346 #define AES_128 (ALG_AES & YES)
347 #endif
348 #ifndef AES_192
349 #define AES_192 (ALG_AES & NO)
350 #endif
351 #ifndef AES_256
352 #define AES_256 (ALG_AES & YES)
353 #endif

```

Table 1:18 - Defines for SM4 Symmetric Cipher Algorithm Constants

```

354 #ifndef SM4_128
355 #define SM4_128 (ALG_SM4 & YES)
356 #endif

```

Table 1:19 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

```

357 #ifndef CAMELLIA_128
358 #define CAMELLIA_128 (ALG_CAMELLIA & YES)
359 #endif
360 #ifndef CAMELLIA_192
361 #define CAMELLIA_192 (ALG_CAMELLIA & NO)
362 #endif
363 #ifndef CAMELLIA_256
364 #define CAMELLIA_256 (ALG_CAMELLIA & NO)
365 #endif

```

Table 1:17 - Defines for TDES Symmetric Cipher Algorithm Constants

```

366 #ifndef TDES_128
367 #define TDES_128 (ALG_TDES & YES)
368 #endif
369 #ifndef TDES_192
370 #define TDES_192 (ALG_TDES & YES)
371 #endif

```

Table 0:5 - Defines for Implemented Commands

```

372 #ifndef CC_AC_GetCapability
373 #define CC_AC_GetCapability CC_YES
374 #endif
375 #ifndef CC_AC_Send
376 #define CC_AC_Send CC_YES
377 #endif
378 #ifndef CC_ActivateCredential
379 #define CC_ActivateCredential CC_YES
380 #endif
381 #ifndef CC_Certify
382 #define CC_Certify CC_YES
383 #endif
384 #ifndef CC_CertifyCreation
385 #define CC_CertifyCreation CC_YES
386 #endif
387 #ifndef CC_CertifyX509
388 #define CC_CertifyX509 CC_YES
389 #endif
390 #ifndef CC_ChangeEPS
391 #define CC_ChangeEPS CC_YES
392 #endif
393 #ifndef CC_ChangePPS
394 #define CC_ChangePPS CC_YES
395 #endif
396 #ifndef CC_Clear

```

```

397 #define CC_Clear CC_YES
398 #endif
399 #ifndef CC_ClearControl
400 #define CC_ClearControl CC_YES
401 #endif
402 #ifndef CC_ClockRateAdjust
403 #define CC_ClockRateAdjust CC_YES
404 #endif
405 #ifndef CC_ClockSet
406 #define CC_ClockSet CC_YES
407 #endif
408 #ifndef CC_Commit
409 #define CC_Commit (CC_YES && ALG_ECC)
410 #endif
411 #ifndef CC_ContextLoad
412 #define CC_ContextLoad CC_YES
413 #endif
414 #ifndef CC_ContextSave
415 #define CC_ContextSave CC_YES
416 #endif
417 #ifndef CC_Create
418 #define CC_Create CC_YES
419 #endif
420 #ifndef CC_CreateLoaded
421 #define CC_CreateLoaded CC_YES
422 #endif
423 #ifndef CC_CreatePrimary
424 #define CC_CreatePrimary CC_YES
425 #endif
426 #ifndef CC_DictionaryAttackLockReset
427 #define CC_DictionaryAttackLockReset CC_YES
428 #endif
429 #ifndef CC_DictionaryAttackParameters
430 #define CC_DictionaryAttackParameters CC_YES
431 #endif
432 #ifndef CC_Duplicate
433 #define CC_Duplicate CC_YES
434 #endif
435 #ifndef CC_ECC_Parameters
436 #define CC_ECC_Parameters (CC_YES && ALG_ECC)
437 #endif
438 #ifndef CC_ECDH_KeyGen
439 #define CC_ECDH_KeyGen (CC_YES && ALG_ECC)
440 #endif
441 #ifndef CC_ECDH_ZGen
442 #define CC_ECDH_ZGen (CC_YES && ALG_ECC)
443 #endif
444 #ifndef CC_EC_Ephemeral
445 #define CC_EC_Ephemeral (CC_YES && ALG_ECC)
446 #endif
447 #ifndef CC_EncryptDecrypt
448 #define CC_EncryptDecrypt CC_YES
449 #endif
450 #ifndef CC_EncryptDecrypt2
451 #define CC_EncryptDecrypt2 CC_YES
452 #endif
453 #ifndef CC_EventSequenceComplete
454 #define CC_EventSequenceComplete CC_YES
455 #endif
456 #ifndef CC_EvictControl
457 #define CC_EvictControl CC_YES
458 #endif
459 #ifndef CC_FieldUpgradeData
460 #define CC_FieldUpgradeData CC_NO
461 #endif
462 #ifndef CC_FieldUpgradeStart

```

```

463 #define CC_FieldUpgradeStart          CC_NO
464 #endif
465 #ifndef CC_FirmwareRead
466 #define CC_FirmwareRead                CC_NO
467 #endif
468 #ifndef CC_FlushContext
469 #define CC_FlushContext                CC_YES
470 #endif
471 #ifndef CC_GetCapability
472 #define CC_GetCapability                CC_YES
473 #endif
474 #ifndef CC_GetCommandAuditDigest
475 #define CC_GetCommandAuditDigest       CC_YES
476 #endif
477 #ifndef CC_GetRandom
478 #define CC_GetRandom                  CC_YES
479 #endif
480 #ifndef CC_GetSessionAuditDigest
481 #define CC_GetSessionAuditDigest       CC_YES
482 #endif
483 #ifndef CC_GetTestResult
484 #define CC_GetTestResult                CC_YES
485 #endif
486 #ifndef CC_GetTime
487 #define CC_GetTime                     CC_YES
488 #endif
489 #ifndef CC_HMAC
490 #define CC_HMAC                        (CC_YES && !ALG_CMAC)
491 #endif
492 #ifndef CC_HMAC_Start
493 #define CC_HMAC_Start                  (CC_YES && !ALG_CMAC)
494 #endif
495 #ifndef CC_Hash
496 #define CC_Hash                        CC_YES
497 #endif
498 #ifndef CC_HashSequenceStart
499 #define CC_HashSequenceStart           CC_YES
500 #endif
501 #ifndef CC_HierarchyChangeAuth
502 #define CC_HierarchyChangeAuth         CC_YES
503 #endif
504 #ifndef CC_HierarchyControl
505 #define CC_HierarchyControl            CC_YES
506 #endif
507 #ifndef CC_Import
508 #define CC_Import                      CC_YES
509 #endif
510 #ifndef CC_IncrementalSelfTest
511 #define CC_IncrementalSelfTest         CC_YES
512 #endif
513 #ifndef CC_Load
514 #define CC_Load                        CC_YES
515 #endif
516 #ifndef CC_LoadExternal
517 #define CC_LoadExternal                CC_YES
518 #endif
519 #ifndef CC_MAC
520 #define CC_MAC                         (CC_YES && ALG_CMAC)
521 #endif
522 #ifndef CC_MAC_Start
523 #define CC_MAC_Start                   (CC_YES && ALG_CMAC)
524 #endif
525 #ifndef CC_MakeCredential
526 #define CC_MakeCredential               CC_YES
527 #endif
528 #ifndef CC_NV_Certify

```

```
529 #define CC_NV_Certify CC_YES
530 #endif
531 #ifndef CC_NV_ChangeAuth
532 #define CC_NV_ChangeAuth CC_YES
533 #endif
534 #ifndef CC_NV_DefineSpace
535 #define CC_NV_DefineSpace CC_YES
536 #endif
537 #ifndef CC_NV_Extend
538 #define CC_NV_Extend CC_YES
539 #endif
540 #ifndef CC_NV_GlobalWriteLock
541 #define CC_NV_GlobalWriteLock CC_YES
542 #endif
543 #ifndef CC_NV_Increment
544 #define CC_NV_Increment CC_YES
545 #endif
546 #ifndef CC_NV_Read
547 #define CC_NV_Read CC_YES
548 #endif
549 #ifndef CC_NV_ReadLock
550 #define CC_NV_ReadLock CC_YES
551 #endif
552 #ifndef CC_NV_ReadPublic
553 #define CC_NV_ReadPublic CC_YES
554 #endif
555 #ifndef CC_NV_SetBits
556 #define CC_NV_SetBits CC_YES
557 #endif
558 #ifndef CC_NV_UndefineSpace
559 #define CC_NV_UndefineSpace CC_YES
560 #endif
561 #ifndef CC_NV_UndefineSpaceSpecial
562 #define CC_NV_UndefineSpaceSpecial CC_YES
563 #endif
564 #ifndef CC_NV_Write
565 #define CC_NV_Write CC_YES
566 #endif
567 #ifndef CC_NV_WriteLock
568 #define CC_NV_WriteLock CC_YES
569 #endif
570 #ifndef CC_ObjectChangeAuth
571 #define CC_ObjectChangeAuth CC_YES
572 #endif
573 #ifndef CC_PCR_Allocate
574 #define CC_PCR_Allocate CC_YES
575 #endif
576 #ifndef CC_PCR_Event
577 #define CC_PCR_Event CC_YES
578 #endif
579 #ifndef CC_PCR_Extend
580 #define CC_PCR_Extend CC_YES
581 #endif
582 #ifndef CC_PCR_Read
583 #define CC_PCR_Read CC_YES
584 #endif
585 #ifndef CC_PCR_Reset
586 #define CC_PCR_Reset CC_YES
587 #endif
588 #ifndef CC_PCR_SetAuthPolicy
589 #define CC_PCR_SetAuthPolicy CC_YES
590 #endif
591 #ifndef CC_PCR_SetAuthValue
592 #define CC_PCR_SetAuthValue CC_YES
593 #endif
594 #ifndef CC_PP_Commands
```

```
595 #define CC_PP_Commands CC_YES
596 #endif
597 #ifndef CC_PolicyAuthValue
598 #define CC_PolicyAuthValue CC_YES
599 #endif
600 #ifndef CC_PolicyAuthorize
601 #define CC_PolicyAuthorize CC_YES
602 #endif
603 #ifndef CC_PolicyAuthorizeNV
604 #define CC_PolicyAuthorizeNV CC_YES
605 #endif
606 #ifndef CC_PolicyCommandCode
607 #define CC_PolicyCommandCode CC_YES
608 #endif
609 #ifndef CC_PolicyCounterTimer
610 #define CC_PolicyCounterTimer CC_YES
611 #endif
612 #ifndef CC_PolicyCpHash
613 #define CC_PolicyCpHash CC_YES
614 #endif
615 #ifndef CC_PolicyDuplicationSelect
616 #define CC_PolicyDuplicationSelect CC_YES
617 #endif
618 #ifndef CC_PolicyGetDigest
619 #define CC_PolicyGetDigest CC_YES
620 #endif
621 #ifndef CC_PolicyLocality
622 #define CC_PolicyLocality CC_YES
623 #endif
624 #ifndef CC_PolicyNV
625 #define CC_PolicyNV CC_YES
626 #endif
627 #ifndef CC_PolicyNameHash
628 #define CC_PolicyNameHash CC_YES
629 #endif
630 #ifndef CC_PolicyNvWritten
631 #define CC_PolicyNvWritten CC_YES
632 #endif
633 #ifndef CC_PolicyOR
634 #define CC_PolicyOR CC_YES
635 #endif
636 #ifndef CC_PolicyPCR
637 #define CC_PolicyPCR CC_YES
638 #endif
639 #ifndef CC_PolicyPassword
640 #define CC_PolicyPassword CC_YES
641 #endif
642 #ifndef CC_PolicyPhysicalPresence
643 #define CC_PolicyPhysicalPresence CC_YES
644 #endif
645 #ifndef CC_PolicyRestart
646 #define CC_PolicyRestart CC_YES
647 #endif
648 #ifndef CC_PolicySecret
649 #define CC_PolicySecret CC_YES
650 #endif
651 #ifndef CC_PolicySigned
652 #define CC_PolicySigned CC_YES
653 #endif
654 #ifndef CC_PolicyTemplate
655 #define CC_PolicyTemplate CC_YES
656 #endif
657 #ifndef CC_PolicyTicket
658 #define CC_PolicyTicket CC_YES
659 #endif
660 #ifndef CC_Policy_AC_SendSelect
```

```
661 #define CC_Policy_AC_SendSelect CC_YES
662 #endif
663 #ifndef CC_Quote
664 #define CC_Quote CC_YES
665 #endif
666 #ifndef CC_RSA_Decrypt
667 #define CC_RSA_Decrypt (CC_YES && ALG_RSA)
668 #endif
669 #ifndef CC_RSA_Encrypt
670 #define CC_RSA_Encrypt (CC_YES && ALG_RSA)
671 #endif
672 #ifndef CC_ReadClock
673 #define CC_ReadClock CC_YES
674 #endif
675 #ifndef CC_ReadPublic
676 #define CC_ReadPublic CC_YES
677 #endif
678 #ifndef CC_Rewrap
679 #define CC_Rewrap CC_YES
680 #endif
681 #ifndef CC_SelfTest
682 #define CC_SelfTest CC_YES
683 #endif
684 #ifndef CC_SequenceComplete
685 #define CC_SequenceComplete CC_YES
686 #endif
687 #ifndef CC_SequenceUpdate
688 #define CC_SequenceUpdate CC_YES
689 #endif
690 #ifndef CC_SetAlgorithmSet
691 #define CC_SetAlgorithmSet CC_YES
692 #endif
693 #ifndef CC_SetCommandCodeAuditStatus
694 #define CC_SetCommandCodeAuditStatus CC_YES
695 #endif
696 #ifndef CC_SetPrimaryPolicy
697 #define CC_SetPrimaryPolicy CC_YES
698 #endif
699 #ifndef CC_Shutdown
700 #define CC_Shutdown CC_YES
701 #endif
702 #ifndef CC_Sign
703 #define CC_Sign CC_YES
704 #endif
705 #ifndef CC_StartAuthSession
706 #define CC_StartAuthSession CC_YES
707 #endif
708 #ifndef CC_Startup
709 #define CC_Startup CC_YES
710 #endif
711 #ifndef CC_StirRandom
712 #define CC_StirRandom CC_YES
713 #endif
714 #ifndef CC_TestParms
715 #define CC_TestParms CC_YES
716 #endif
717 #ifndef CC_Unseal
718 #define CC_Unseal CC_YES
719 #endif
720 #ifndef CC_Vendor_TCG_Test
721 #define CC_Vendor_TCG_Test CC_YES
722 #endif
723 #ifndef CC_VerifySignature
724 #define CC_VerifySignature CC_YES
725 #endif
726 #ifndef CC_ZGen_2Phase
```



```
727 #define CC_ZGen_2Phase (CC_YES && ALG_ECC)
728 #endif
729 #endif // _TPM_PROFILE_H_
```

DRAFT

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  #define HASH_LIB_OSSL
4  #include <openssl/evp.h>
5  #include <openssl/sha.h>
6  #include <openssl/ssl_typ.h>
```

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.

```
7  #define tpmHashStateSHA1_t      SHA_CTX
8  #define tpmHashStateSHA256_t   SHA256_CTX
9  #define tpmHashStateSHA384_t    SHA512_CTX
10 #define tpmHashStateSHA512_t    SHA512_CTX
11 #if ALG_SM3_256
12 #  error "The version of OpenSSL used by this code does not support SM3"
13 #endif
```

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.

```
14 #ifdef _CRYPT_HASH_C_
15 typedef BYTE      *PBYTE;
16 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

```
17 #define HASH_START_METHOD_DEF void (HASH_START_METHOD) (PANY_HASH_STATE state)
18 #define HASH_START(hashState) \
```

```
19         ((hashState)->def->method.start) (&(hashState)->state);
```

Add data to the hash

```
20 #define HASH_DATA_METHOD_DEF \
21     void (HASH_DATA_METHOD) (PANY_HASH_STATE state, \
22                             PCBYTE buffer, \
23                             size_t size) \
24 #define HASH_DATA(hashState, dInSize, dIn) \
25     ((hashState)->def->method.data) (&(hashState)->state, dIn, dInSize)
```

Finalize the hash and get the digest

```
26 #define HASH_END_METHOD_DEF \
27     void (HASH_END_METHOD) (BYTE *buffer, PANY_HASH_STATE state) \
28 #define HASH_END(hashState, buffer) \
29     ((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.

```
30 #define HASH_STATE_COPY_METHOD_DEF \
31     void (HASH_STATE_COPY_METHOD) (PANY_HASH_STATE to, \
32                                     PCANY_HASH_STATE from, \
33                                     size_t size) \
34 #define HASH_STATE_COPY(hashStateOut, hashStateIn) \
35     ((hashStateIn)->def->method.copy) (&(hashStateOut)->state, \
36                                       &(hashStateIn)->state, \
37                                       (hashStateIn)->def->contextSize)
```

Copy (with reformatting when necessary) an internal hash structure to an external blob

```
38 #define HASH_STATE_EXPORT_METHOD_DEF \
39     void (HASH_STATE_EXPORT_METHOD) (BYTE *to, \
40                                       PCANY_HASH_STATE from, \
41                                       size_t size) \
42 #define HASH_STATE_EXPORT(to, hashStateFrom) \
43     ((hashStateFrom)->def->method.copyOut) \
44     (&((BYTE *) (to)) [offsetof(HASH_STATE, state)]), \
45     &(hashStateFrom)->state, \
46     (hashStateFrom)->def->contextSize)
```

Copy from an external blob to an internal format (with reformatting when necessary)

```
47 #define HASH_STATE_IMPORT_METHOD_DEF \
48     void (HASH_STATE_IMPORT_METHOD) (PANY_HASH_STATE to, \
49                                       const BYTE *from, \
50                                       size_t size) \
51 #define HASH_STATE_IMPORT(hashStateTo, from) \
52     ((hashStateTo)->def->method.copyIn) \
53     (&(hashStateTo)->state, \
54     &((const BYTE *) (from)) [offsetof(HASH_STATE, state)]), \
55     (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.

```
56 #define tpmHashStart_SHA1      SHA1_Init    // external name of the
57                                     // initialization method
58 #define tpmHashData_SHA1      SHA1_Update
59 #define tpmHashEnd_SHA1       SHA1_Final
```

```
60 #define tpmHashStateCopy_SHA1      memcpy
61 #define tpmHashStateExport_SHA1     memcpy
62 #define tpmHashStateImport_SHA1     memcpy
63 #define tpmHashStart_SHA256        SHA256_Init
64 #define tpmHashData_SHA256         SHA256_Update
65 #define tpmHashEnd_SHA256          SHA256_Final
66 #define tpmHashStateCopy_SHA256    memcpy
67 #define tpmHashStateExport_SHA256  memcpy
68 #define tpmHashStateImport_SHA256  memcpy
69 #define tpmHashStart_SHA384        SHA384_Init
70 #define tpmHashData_SHA384         SHA384_Update
71 #define tpmHashEnd_SHA384          SHA384_Final
72 #define tpmHashStateCopy_SHA384    memcpy
73 #define tpmHashStateExport_SHA384  memcpy
74 #define tpmHashStateImport_SHA384  memcpy
75 #define tpmHashStart_SHA512        SHA512_Init
76 #define tpmHashData_SHA512         SHA512_Update
77 #define tpmHashEnd_SHA512          SHA512_Final
78 #define tpmHashStateCopy_SHA512    memcpy
79 #define tpmHashStateExport_SHA512  memcpy
80 #define tpmHashStateImport_SHA512  memcpy
81 #endif // _CRYPT_HASH_C_
82 #define LibHashInit()
```

This definition would change if there were something to report

```
83 #define HashLibSimulationEnd()
84 #endif // HASH_LIB_DEFINED
```

B.2.2.2. TpmToOsslMath.h

B.2.2.2.1. Introduction

This file contains the structure definitions used for ECC in the LibTopCrypt() 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  #define MATH_LIB_OSSL
4  #include <openssl/evp.h>
5  #include <openssl/ec.h>
6  #if OPENSSL_VERSION_NUMBER >= 0x10100000L
7  #include <openssl/bn_lcl.h>
8  #endif
9  #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

```

10 #if defined THIRTY_TWO_BIT && (RADIX_BITS != 32) \
11    || ((defined SIXTY_FOUR_BIT_LONG || defined SIXTY_FOUR_BIT) \
12        && (RADIX_BITS != 64))
13 # error "Ossl library is using different radix"
14 #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.

```

15 #define BIG_VAR(name, bits) \
16     BN_VAR(name##Bn, (bits)); \
17     BIGNUM \
18     BIGNUM \
19     *name = BigInitialized(& ##name, \
20                           BnInit(name##Bn, \

```

Allocate a BIGNUM and initialize with the values in a *bigNum* initializer

```

21 #define BIG_INITIALIZED(name, initializer) \
22     BIGNUM \
23     BIGNUM \
24 typedef struct \
25 { \
26     const ECC_CURVE_DATA *C; // the TPM curve values \
27     EC_GROUP *G; // group parameters \
28     BN_CTX *CTX; // the context for the math (this might not be \
29                  // the context in which the curve was created>; \
30 } OSSL_CURVE_DATA; \
31 typedef OSSL_CURVE_DATA *bigCurve; \
32 #define AccessCurveData(E) ((E)->C) \
33 #include "TpmToOsslSupport_fp.h"

```

Start and end a context within which the OpenSSL memory management works

```

34 #define OSSL_ENTER() BN_CTX *CTX = OsslContextEnter()
35 #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.

```
36 #define CURVE_INITIALIZED(name, initializer) \
37     OSSL_CURVE_DATA _##name; \
38     bigCurve name = BnCurveInitialize(&_##name, initializer)
39 #define CURVE_FREE(name) BnCurveFree(name)
```

Start and end a local stack frame within the context of the curve frame

```
40 #define ECC_ENTER() BN_CTX *CTX = OsslPushContext(E->CTX)
41 #define ECC_LEAVE() OsslPopContext(CTX)
42 #define BN_NEW() BnNewVariable(CTX)
```

This definition would change if there were something to report

```
43 #define MathLibSimulationEnd()
44 #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.

```

1  #ifndef SYM_LIB_DEFINED
2  #define SYM_LIB_DEFINED
3  #define SYM_LIB_OSSL
4  #include <openssl/aes.h>
5  #include <openssl/des.h>
6  #include <openssl/bn.h>
7  #include <openssl/ssl_typ.h>

```

B.2.2.3.2. Links to the OpenSSL AES code

```

8  #if ALG_SM4
9  #error "SM4 is not available"
10 #endif
11 #if ALG_CAMELLIA
12 #error "Camellia is not available"
13 #endif

```

Define the order of parameters to the library functions that do block encryption and decryption.

```

14 typedef void(*TpmCryptSetSymKeyCall_t)(
15     const BYTE *in,
16     BYTE *out,
17     void *keySchedule
18 );

```

The Crypt functions that call the block encryption function use the parameters in the order:

- keySchedule*
- in buffer
- out buffer Since open SSL uses the order in *encryptCall_t* above, need to swizzle the values to the order required by the library.

```

19 #define SWIZZLE(keySchedule, in, out) \
20     ((const BYTE *) (in), (BYTE *) (out), (void *) (keySchedule))

```

Macros to set up the encryption/decryption key schedules

AES:

```

21 #define TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule) \
22     AES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *) (schedule))
23 #define TpmCryptSetDecryptKeyAES(key, keySizeInBits, schedule) \
24     AES_set_decrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *) (schedule))

```

TDES:

```

25 #define TpmCryptSetEncryptKeyTDES(key, keySizeInBits, schedule) \
26     TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *) (schedule))
27 #define TpmCryptSetDecryptKeyTDES(key, keySizeInBits, schedule) \
28     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

When using these calls, to call the AES block encryption code, the caller should use: TpmCryptEncryptAES(SWIZZLE(keySchedule, in, out));

```
29 #define TpmCryptEncryptAES      AES_encrypt
30 #define TpmCryptDecryptAES      AES_decrypt
31 #define tpmKeyScheduleAES      AES_KEY
32 #define TpmCryptEncryptTDES    TDES_encrypt
33 #define TpmCryptDecryptTDES    TDES_decrypt
34 #define tpmKeyScheduleTDES     DES_key_schedule
35 typedef union tpmCryptKeySchedule_t tpmCryptKeySchedule_t;
36 #if ALG_TDES
37 #include "TpmToOsslDesSupport_fp.h"
38 #endif
```

This definition would change if there were something to report

```
39 #define SymLibSimulationEnd()
40 #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  #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.

```
3  void
4  TDES_set_encrypt_key(
5      const BYTE                *key,
6      UINT16                    keySizeInBits,
7      tpmKeyScheduleTDES       *keySchedule
8  )
9  {
10     DES_set_key_unchecked((const DES_cblock *)key, &keySchedule[0]);
11     DES_set_key_unchecked((const DES_cblock *)&key[8], &keySchedule[1]);
12     // If is two-key, copy the schedule for K1 into K3, otherwise, compute the
13     // the schedule for K3
14     if(keySizeInBits == 128)
15         keySchedule[2] = keySchedule[0];
16     else
17         DES_set_key_unchecked((const DES_cblock *)&key[16],
18                               &keySchedule[2]);
19 }
```

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.

```
20 void TDES_encrypt(
21     const BYTE                *in,
22     BYTE                      *out,
23     tpmKeyScheduleTDES       *ks
24 )
25 {
26     DES_ecb3_encrypt((const DES_cblock *)in, (DES_cblock *)out,
27                     &ks[0], &ks[1], &ks[2],
28                     DES_ENCRYPT);
29 }
```

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.

```
30 void TDES_decrypt(  
31     const BYTE      *in,  
32     BYTE            *out,  
33     tpmKeyScheduleTDES *ks  
34 )  
35 {  
36     DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,  
37                     &ks[0], &ks[1], &ks[2],  
38                     DES_DECRYPT);  
39 }  
40 #endif // SYM_LIB_OSSL
```

B.2.3.2. TpmToOsslMath.c

B.2.3.2.1. Introduction

This file contains the math functions that are not implemented in the BnMath() library (yet). These math functions will call the OpenSSL library to execute the operations. There is a difference between the internal format and the OpenSSL format. To call the OpenSSL function, a BIGNUM structure is created for each passed variable. The sizes in the bignum_t are copied and the *d* pointer in the BIGNUM is set to point to the *d* parameter of the bignum_t. On return, SetSizeOsslToTpm() is used for each returned variable to make sure that the pointers are not changed. The size of the returned BIGGNUM is copied to bignum_t.

B.2.3.2.2. 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 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.2.3.2.3. Includes and Defines

```
1  #include "Tpm.h"
2  #ifdef MATH_LIB_OSSL
3  #include "TpmToOsslMath_fp.h"
```

B.2.3.2.4. Functions

B.2.3.2.4.1. 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

```
4  BOOL
5  OsslToTpmBn (
6      bigNum          bn,
7      BIGNUM          *osslBn
8  )
9  {
10     VERIFY (osslBn != NULL);
```

```

11     // If the bn is NULL, it means that an output value pointer was NULL meaning that
12     // the results is simply to be discarded.
13     if(bn != NULL)
14     {
15         int            i;
16         //
17         VERIFY((unsigned)osslBn->top <= BnGetAllocated(bn));
18         for(i = 0; i < ossslBn->top; i++)
19             bn->d[i] = ossslBn->d[i];
20         BnSetTop(bn, ossslBn->top);
21     }
22     return TRUE;
23 Error:
24     return FALSE;
25 }

```

B.2.3.2.4.2. 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().

```

26 BIGNUM *
27 BigInitialized(
28     BIGNUM            *toInit,
29     bigConst          initializer
30 )
31 {
32     if(initializer == NULL)
33         FAIL(FATAL_ERROR_PARAMETER);
34     if(toInit == NULL || initializer == NULL)
35         return NULL;
36     toInit->d = (BN_ULONG *) &initializer->d[0];
37     toInit->dmax = initializer->allocated;
38     toInit->top = initializer->size;
39     toInit->neg = 0;
40     toInit->flags = 0;
41     return toInit;
42 }
43 #ifndef OSSL_DEBUG
44 # define BIGNUM_PRINT(label, bn, eol)
45 # define DEBUG_PRINT(x)
46 #else
47 # define DEBUG_PRINT(x)    printf("%s", x)
48 # define BIGNUM_PRINT(label, bn, eol) BIGNUM_print((label), (bn), (eol))

```

B.2.3.2.4.3. BIGNUM_print()

```

49 static void
50 BIGNUM_print(
51     const char        *label,
52     const BIGNUM      *a,
53     BOOL              eol
54 )
55 {
56     BN_ULONG          *d;
57     int                i;
58     int                notZero = FALSE;
59
60     if(label != NULL)
61         printf("%s", label);
62     if(a == NULL)
63     {

```

```

64     printf("NULL");
65     goto done;
66 }
67 if (a->neg)
68     printf("-");
69 for(i = a->top, d = &a->d[i - 1]; i > 0; i--)
70 {
71     int j;
72     BN_ULONG l = *d--;
73     for(j = BN_BITS2 - 8; j >= 0; j -= 8)
74     {
75         BYTE b = (BYTE)((l >> j) & 0xFF);
76         notZero = notZero || (b != 0);
77         if(notZero)
78             printf("%02x", b);
79     }
80     if(!notZero)
81         printf("0");
82 }
83 done:
84 if(eol)
85     printf("\n");
86 return;
87 }
88 #endif

```

B.2.3.2.4.4. 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.

```

89 static BIGNUM *
90 BnNewVariable(
91     BN_CTX *CTX
92 )
93 {
94     BIGNUM *new;
95     //
96     // This check is intended to protect against calling this function without
97     // having initialized the CTX.
98     if((CTX == NULL) || ((new = BN_CTX_get(CTX)) == NULL))
99         FAIL(FATAL_ERROR_ALLOCATION);
100     return new;
101 }
102 #if LIBRARY_COMPATIBILITY_CHECK

```

B.2.3.2.4.5. MathLibraryCompatibilityCheck()

```

103 void
104 MathLibraryCompatibilityCheck(
105     void
106 )
107 {
108     OSSL_ENTER();
109     BIGNUM *osslTemp = BnNewVariable(CTX);
110     crypt_ushort_t i;
111     BYTE test[] = {0x1F, 0x1E, 0x1D, 0x1C, 0x1B, 0x1A, 0x19, 0x18,
112                    0x17, 0x16, 0x15, 0x14, 0x13, 0x12, 0x11, 0x10,
113                    0x0F, 0x0E, 0x0D, 0x0C, 0x0B, 0x0A, 0x09, 0x08,
114                    0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00};
115     BN_VAR(tpmTemp, sizeof(test) * 8); // allocate some space for a test value
116     //
117     // Convert the test data to a bigNum

```



```

118     BnFromBytes(tpmTemp, test, sizeof(test));
119     // Convert the test data to an OpenSSL BIGNUM
120     BN_bin2bn(test, sizeof(test), osslTemp);
121     // Make sure the values are consistent
122     VERIFY(osslTemp->top == (int)tpmTemp->size);
123     for(i = 0; i < tpmTemp->size; i++)
124         VERIFY(osslTemp->d[i] == tpmTemp->d[i]);
125     OSSL_LEAVE();
126     return;
127 Error:
128     FAIL(FATAL_ERROR_MATHLIBRARY);
129 }
130 #endif

```

B.2.3.2.4.6. 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

```

131 LIB_EXPORT BOOL
132 BnModMult(
133     bigNum          result,
134     bigConst        op1,
135     bigConst        op2,
136     bigConst        modulus
137 )
138 {
139     OSSL_ENTER();
140     BOOL          OK = TRUE;
141     BIGNUM        *bnResult = BN_NEW();
142     BIGNUM        *bnTemp = BN_NEW();
143     BIG_INITIALIZED(bnOp1, op1);
144     BIG_INITIALIZED(bnOp2, op2);
145     BIG_INITIALIZED(bnMod, modulus);
146     //
147     VERIFY(BN_mul(bnTemp, bnOp1, bnOp2, CTX));
148     VERIFY(BN_div(NULL, bnResult, bnTemp, bnMod, CTX));
149     VERIFY(OsslToTpmBn(result, bnResult));
150     goto Exit;
151 Error:
152     OK = FALSE;
153 Exit:
154     OSSL_LEAVE();
155     return OK;
156 }

```

B.2.3.2.4.7. BnMult()

Multiplies two numbers

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

157 LIB_EXPORT BOOL

```

```

158 BnMult(
159     bigNum          result,
160     bigConst        multiplicand,
161     bigConst        multiplier
162 )
163 {
164     OSSL_ENTER();
165     BIGNUM          *bnTemp = BN_NEW();
166     BOOL            OK = TRUE;
167     BIG_INITIALIZED(bnA, multiplicand);
168     BIG_INITIALIZED(bnB, multiplier);
169     //
170     VERIFY(BN_mul(bnTemp, bnA, bnB, CTX));
171     VERIFY(OsslToTpmBn(result, bnTemp));
172     goto Exit;
173 Error:
174     OK = FALSE;
175 Exit:
176     OSSL_LEAVE();
177     return OK;
178 }

```

B.2.3.2.4.8. 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

```

179 LIB_EXPORT BOOL
180 BnDiv(
181     bigNum          quotient,
182     bigNum          remainder,
183     bigConst        dividend,
184     bigConst        divisor
185 )
186 {
187     OSSL_ENTER();
188     BIGNUM          *bnQ = BN_NEW();
189     BIGNUM          *bnR = BN_NEW();
190     BOOL            OK = TRUE;
191     BIG_INITIALIZED(bnDend, dividend);
192     BIG_INITIALIZED(bnSor, divisor);
193     //
194     if(BnEqualZero(divisor))
195         FAIL(FATAL_ERROR_DIVIDE_ZERO);
196     VERIFY(BN_div(bnQ, bnR, bnDend, bnSor, CTX));
197     VERIFY(OsslToTpmBn(quotient, bnQ));
198     VERIFY(OsslToTpmBn(remainder, bnR));
199     DEBUG_PRINT("In BnDiv:\n");
200     BIGNUM_PRINT("    bnDividend: ", bnDend, TRUE);
201     BIGNUM_PRINT("    bnDivisor: ", bnSor, TRUE);
202     BIGNUM_PRINT("    bnQuotient: ", bnQ, TRUE);
203     BIGNUM_PRINT("    bnRemainder: ", bnR, TRUE);
204     goto Exit;
205 Error:
206     OK = FALSE;
207 Exit:
208     OSSL_LEAVE();
209     return OK;
210 }

```

211 **#if** ALG_RSA

B.2.3.2.4.9. BnGcd()

Get the greatest common divisor of two numbers

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

212 LIB_EXPORT BOOL
213 BnGcd(
214     bigNum      gcd,           // OUT: the common divisor
215     bigConst    number1,      // IN:
216     bigConst    number2      // IN:
217 )
218 {
219     OSSL_ENTER();
220     BIGNUM        *bnGcd = BN_NEW();
221     BOOL          OK = TRUE;
222     BIG_INITIALIZED(bn1, number1);
223     BIG_INITIALIZED(bn2, number2);
224     //
225     VERIFY(BN_gcd(bnGcd, bn1, bn2, CTX));
226     VERIFY(OsslToTpmBn(gcd, bnGcd));
227     goto Exit;
228 Error:
229     OK = FALSE;
230 Exit:
231     OSSL_LEAVE();
232     return OK;
233 }
```

B.2.3.2.4.10. 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

```

234 LIB_EXPORT BOOL
235 BnModExp(
236     bigNum      result,        // OUT: the result
237     bigConst    number,        // IN: number to exponentiate
238     bigConst    exponent,      // IN:
239     bigConst    modulus        // IN:
240 )
241 {
242     OSSL_ENTER();
243     BIGNUM        *bnResult = BN_NEW();
244     BOOL          OK = TRUE;
245     BIG_INITIALIZED(bnN, number);
246     BIG_INITIALIZED(bnE, exponent);
247     BIG_INITIALIZED(bnM, modulus);
248     //
249     VERIFY(BN_mod_exp(bnResult, bnN, bnE, bnM, CTX));
250     VERIFY(OsslToTpmBn(result, bnResult));
```

```

251     goto Exit;
252 Error:
253     OK = FALSE;
254 Exit:
255     OSSL_LEAVE();
256     return OK;
257 }

```

B.2.3.2.4.11. BnModInverse()

Modular multiplicative inverse

Return Value	Meaning
TRUE(1)	success
FALSE(0)	failure in operation

```

258 LIB_EXPORT BOOL
259 BnModInverse(
260     bigNum          result,
261     bigConst        number,
262     bigConst        modulus
263 )
264 {
265     OSSL_ENTER();
266     BIGNUM          *bnResult = BN_NEW();
267     BOOL            OK = TRUE;
268     BIG_INITIALIZED(bnN, number);
269     BIG_INITIALIZED(bnM, modulus);
270 //
271     VERIFY(BN_mod_inverse(bnResult, bnN, bnM, CTX) != NULL);
272     VERIFY(OsslToTpmBn(result, bnResult));
273     goto Exit;
274 Error:
275     OK = FALSE;
276 Exit:
277     OSSL_LEAVE();
278     return OK;
279 }
280 #endif // ALG_RSA
281 #if ALG_ECC

```

B.2.3.2.4.12. 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

```

282 static BOOL
283 PointFromOssl(
284     bigPoint        pOut,      // OUT: resulting point
285     EC_POINT        *pIn,      // IN: the point to return
286     bigCurve        E          // IN: the curve
287 )
288 {
289     BIGNUM          *x = NULL;
290     BIGNUM          *y = NULL;
291     BOOL            OK;

```

```

292     BN_CTX_start(E->CTX);
293     //
294     x = BN_CTX_get(E->CTX);
295     y = BN_CTX_get(E->CTX);
296
297     if(y == NULL)
298         FAIL(FATAL_ERROR_ALLOCATION);
299     // If this returns false, then the point is at infinity
300     OK = EC_POINT_get_affine_coordinates_GFp(E->G, pIn, x, y, E->CTX);
301     if(OK)
302     {
303         OsslToTpmBn(pOut->x, x);
304         OsslToTpmBn(pOut->y, y);
305         BnSetWord(pOut->z, 1);
306     }
307     else
308         BnSetWord(pOut->z, 0);
309     BN_CTX_end(E->CTX);
310     return OK;
311 }

```

B.2.3.2.4.13. EcPointInitialized()

Allocate and initialize a point.

```

312 static EC_POINT *
313 EcPointInitialized(
314     pointConst      initializer,
315     bigCurve         E
316 )
317 {
318     EC_POINT         *P = NULL;
319
320     if(initializer != NULL)
321     {
322         BIG_INITIALIZED(bnX, initializer->x);
323         BIG_INITIALIZED(bnY, initializer->y);
324         P = EC_POINT_new(E->G);
325         if(E == NULL)
326             FAIL(FATAL_ERROR_ALLOCATION);
327         if(!EC_POINT_set_affine_coordinates_GFp(E->G, P, bnX, bnY, E->CTX))
328             P = NULL;
329     }
330     return P;
331 }

```

B.2.3.2.4.14. 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 E

```

332 LIB_EXPORT bigCurve
333 BnCurveInitialize(
334     bigCurve         E,           // IN: curve structure to initialize
335     TPM_ECC_CURVE    curveId     // IN: curve identifier

```

```

336 )
337 {
338     const ECC_CURVE_DATA    *C = GetCurveData(curveId);
339     if(C == NULL)
340         E = NULL;
341     if(E != NULL)
342     {
343         // This creates the OpenSSL memory context that stays in effect as long as the
344         // curve (E) is defined.
345         OSSL_ENTER(); // if the allocation fails, the TPM fails
346         EC_POINT          *P = NULL;
347         BIG_INITIALIZED(bnP, C->prime);
348         BIG_INITIALIZED(bnA, C->a);
349         BIG_INITIALIZED(bnB, C->b);
350         BIG_INITIALIZED(bnX, C->base.x);
351         BIG_INITIALIZED(bnY, C->base.y);
352         BIG_INITIALIZED(bnN, C->order);
353         BIG_INITIALIZED(bnH, C->h);
354         //
355         E->C = C;
356         E->CTX = CTX;
357
358         // initialize EC group, associate a generator point and initialize the point
359         // from the parameter data
360         // Create a group structure
361         E->G = EC_GROUP_new_curve_Gfp(bnP, bnA, bnB, CTX);
362         VERIFY(E->G != NULL);
363
364         // Allocate a point in the group that will be used in setting the
365         // generator. This is not needed after the generator is set.
366         P = EC_POINT_new(E->G);
367         VERIFY(P != NULL);
368
369         // Need to use this in case Montgomery method is being used
370         VERIFY(EC_POINT_set_affine_coordinates_Gfp(E->G, P, bnX, bnY, CTX));
371         // Now set the generator
372         VERIFY(EC_GROUP_set_generator(E->G, P, bnN, bnH));
373
374         EC_POINT_free(P);
375         goto Exit;
376     Error:
377         EC_POINT_free(P);
378         BnCurveFree(E);
379         E = NULL;
380     }
381     Exit:
382     return E;
383 }

```

B.2.3.2.4.15. BnCurveFree()

This function will free the allocated components of the curve and end the frame in which the curve data exists

```

384 LIB_EXPORT void
385 BnCurveFree(
386     bigCurve          E
387 )
388 {
389     if(E)
390     {
391         EC_GROUP_free(E->G);
392         OsslContextLeave(E->CTX);
393     }

```

394 }

B.2.3.2.4.16. 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

```

395  LIB_EXPORT BOOL
396  BnEccModMult(
397      bigPoint          R,          // OUT: computed point
398      pointConst        S,          // IN: point to multiply by 'd' (optional)
399      bigConst          d,          // IN: scalar for [d]S
400      bigCurve          E
401  )
402  {
403      EC_POINT          *pR = EC_POINT_new(E->G);
404      EC_POINT          *pS = EcPointInitialized(S, E);
405      BIG_INITIALIZED(bnD, d);
406
407      if(S == NULL)
408          EC_POINT_mul(E->G, pR, bnD, NULL, NULL, E->CTX);
409      else
410          EC_POINT_mul(E->G, pR, NULL, pS, bnD, E->CTX);
411      PointFromOssl(R, pR, E);
412      EC_POINT_free(pR);
413      EC_POINT_free(pS);
414      return !BnEqualZero(R->z);
415  }

```

B.2.3.2.4.17. 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

```

416  LIB_EXPORT BOOL
417  BnEccModMult2(
418      bigPoint          R,          // OUT: computed point
419      pointConst        S,          // IN: optional point
420      bigConst          d,          // IN: scalar for [d]S or [d]G
421      pointConst        Q,          // IN: second point
422      bigConst          u,          // IN: second scalar
423      bigCurve          E          // IN: curve
424  )
425  {
426      EC_POINT          *pR = EC_POINT_new(E->G);
427      EC_POINT          *pS = EcPointInitialized(S, E);
428      BIG_INITIALIZED(bnD, d);
429      EC_POINT          *pQ = EcPointInitialized(Q, E);
430      BIG_INITIALIZED(bnU, u);
431
432      if(S == NULL || S == (pointConst)&(AccessCurveData(E)->base))
433          EC_POINT_mul(E->G, pR, bnD, pQ, bnU, E->CTX);
434      else

```



```

435     {
436         const EC_POINT      *points[2];
437         const BIGNUM         *scalars[2];
438         points[0] = pS;
439         points[1] = pQ;
440         scalars[0] = bnD;
441         scalars[1] = bnU;
442         EC_POINTs_mul(E->G, pR, NULL, 2, points, scalars, E->CTX);
443     }
444     PointFromOssl(R, pR, E);
445     EC_POINT_free(pR);
446     EC_POINT_free(pS);
447     EC_POINT_free(pQ);
448     return !BnEqualZero(R->z);
449 }

```

B.2.3.2.5. 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

```

450 LIB_EXPORT BOOL
451 BnEccAdd(
452     bigPoint      R,           // OUT: computed point
453     pointConst    S,           // IN: point to multiply by 'd'
454     pointConst    Q,           // IN: second point
455     bigCurve      E,           // IN: curve
456 )
457 {
458     EC_POINT      *pR = EC_POINT_new(E->G);
459     EC_POINT      *pS = EcPointInitialized(S, E);
460     EC_POINT      *pQ = EcPointInitialized(Q, E);
461     //
462     EC_POINT_add(E->G, pR, pS, pQ, E->CTX);
463
464     PointFromOssl(R, pR, E);
465     EC_POINT_free(pR);
466     EC_POINT_free(pS);
467     EC_POINT_free(pQ);
468     return !BnEqualZero(R->z);
469 }
470 #endif // ALG_ECC
471 #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  #ifndef MATH_LIB_OSSL
```

Used to pass the pointers to the correct sub-keys

```
3  typedef const BYTE *desKeyPointers[3];
```

B.2.3.3.2.1. SupportLibInit()

This does any initialization required by the support library.

```
4  LIB_EXPORT int
5  SupportLibInit(
6      void
7  )
8  {
9      #if LIBRARY_COMPATIBILITY_CHECK
10         MathLibraryCompatibilityCheck();
11     #endif
12     return TRUE;
13 }
```

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.

```
14 BN_CTX *
15 OsslContextEnter(
16     void
17 )
18 {
19     BN_CTX          *CTX = BN_CTX_new();
20     //
21     return OsslPushContext(CTX);
22 }
```

B.2.3.3.2.3. OsslContextLeave()

This is the companion function to OsslContextEnter().

```
23 void
24 OsslContextLeave(
25     BN_CTX          *CTX
26 )
27 {
28     OsslPopContext(CTX);
29     BN_CTX_free(CTX);
30 }
```

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())

```
31 BN_CTX *
32 OsslPushContext(
33     BN_CTX      *CTX
34 )
35 {
36     if (CTX == NULL)
37         FAIL(FATAL_ERROR_ALLOCATION);
38     BN_CTX_start(CTX);
39     return CTX;
40 }
```

B.2.3.3.2.5. OsslPopContext()

This is the companion function to OsslPushContext().

```
41 void
42 OsslPopContext(
43     BN_CTX      *CTX
44 )
45 {
46     // BN_CTX_end can't be called with NULL. It will blow up.
47     if (CTX != NULL)
48         BN_CTX_end(CTX);
49 }
50 #endif // MATH_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 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 BOOL
121  _plat__TimerWasReset(
122      void
123  )
124  {
125      BOOL      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 BOOL
130  _plat__TimerWasStopped(
131      void
132  )
133  {
134      BOOL      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
171 if(s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
172     s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
173 if(s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
174     s_adjustRate = CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT;
175
176 return;
177 }
```

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  #ifndef _MSC_VER
7  #include <process.h>
8  #else
9  #include <unistd.h>
10 #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 a 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."

```

11 extern uint32_t      lastEntropy;

```

C.4.2. Functions

C.4.2.1. rand32()

Local function to get a 32-bit random number

```

12 static uint32_t
13 rand32(
14     void
15 )
16 {
17     uint32_t    rndNum = rand();
18     #if RAND_MAX < UINT16_MAX
19         // If the maximum value of the random number is a 15-bit number, then shift it up
20         // 15 bits, get 15 more bits, shift that up 2 and then XOR in another value to get
21         // a full 32 bits.
22         rndNum = (rndNum << 15) ^ rand();
23         rndNum = (rndNum << 2) ^ rand();
24     #elif RAND_MAX == UINT16_MAX
25         // If the maximum size is 16-bits, shift it and add another 16 bits
26         rndNum = (rndNum << 16) ^ rand();
27     #elif RAND_MAX < UINT32_MAX
28         // If 31 bits, then shift 1 and include another random value to get the extra bit
29         rndNum = (rndNum << 1) ^ rand();
30     #endif
31     return rndNum;
32 }

```

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)

```

33  LIB_EXPORT int32_t
34  _plat_GetEntropy(
35      unsigned char      *entropy,          // output buffer
36      uint32_t           amount             // amount requested
37  )
38  {
39      uint32_t           rndNum;
40      int32_t           ret;
41      //
42      if (amount == 0)
43      {
44          // Seed the platform entropy source if the entropy source is software. There
45          // is no reason to put a guard macro (#if or #ifdef) around this code because
46          // this code would not be here if someone was changing it for a system with
47          // actual hardware.
48          //
49          // NOTE 1: The following command does not provide proper cryptographic
50          // entropy. Its primary purpose to make sure that different instances of the
51          // simulator, possibly started by a script on the same machine, are seeded
52          // differently. Vendors of the actual TPMs need to ensure availability of
53          // proper entropy using their platform-specific means.
54          //
55          // NOTE 2: In debug builds by default the reference implementation will seed
56          // its RNG deterministically (without using any platform provided randomness).
57          // See the USE_DEBUG_RNG macro and DRBG_GetEntropy() function.
58      #ifdef MSC_VER
59          srand((unsigned)_plat_RealTime() ^ _getpid());
60      #else
61          srand((unsigned)_plat_RealTime() ^ getpid());
62      #endif
63          lastEntropy = rand32();
64          ret = 0;
65      }
66      else
67      {
68          rndNum = rand32();
69          if (rndNum == lastEntropy)
70          {
71              ret = -1;
72          }
73          else
74          {
75              lastEntropy = rndNum;
76              // Each process will have its random number generator initialized
77              // according to the process id and the initialization time. This is not a
78              // lot of entropy so, to add a bit more, XOR the current time value into
79              // the returned entropy value.
80              // NOTE: the reason for including the time here rather than have it in
81              // in the value assigned to lastEntropy is that rand() could be broken and
82              // using the time would in the lastEntropy value would hide this.
83              rndNum ^= (uint32_t)_plat_RealTime();
84
85              // Only provide entropy 32 bits at a time to test the ability
86              // of the caller to deal with partial results.
87              ret = MIN(amount, sizeof(rndNum));
88              memcpy(entropy, &rndNum, ret);
89          }
90      }
91      return ret;

```

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  FILE      *s_NvFile = NULL;
8  #endif

```

C.6.3. Functions

C.6.3.1. NvFileOpen()

This function opens the file used to hold the NV image.

```

9  #if FILE_BACKED_NV

```

Return Value	Meaning
>= 0	success
-1	error

```

10 static int
11 NvFileOpen(
12     const char    *mode
13 )
14 {
15     // Try to open an exist NVChip file for read/write
16     # if defined _MSC_VER && 1
17     if(fopen_s(&s_NvFile, "NVChip", mode) != 0)
18         s_NvFile = NULL;
19     # else
20     s_NvFile = fopen("NVChip", mode);
21     # endif
22     return (s_NvFile == NULL) ? -1 : 0;
23 }

```

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

```

24 static int

```

```

25  NvFileCommit(
26      void
27  )
28  {
29      int          OK;
30      // If NV file is not available, return failure
31      if(s_NvFile == NULL)
32          return 1;
33      // Write RAM data to NV
34      fseek(s_NvFile, 0, SEEK_SET);
35      OK = (NV_MEMORY_SIZE == fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NvFile));
36      OK = OK && (0 == fflush(s_NvFile));
37      assert(OK);
38      return OK;
39  }

```

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.

```

40  static long
41  NvFileSize(
42      int          leaveAt
43  )
44  {
45      long          fileSize;
46      long          filePos = ftell(s_NvFile);
47      //
48      assert(NULL != s_NvFile);
49
50      fseek(s_NvFile, 0, SEEK_END);
51      fileSize = ftell(s_NvFile);
52      switch(leaveAt)
53      {
54          case SEEK_SET:
55              filePos = 0;
56          case SEEK_CUR:
57              fseek(s_NvFile, filePos, SEEK_SET);
58              break;
59          case SEEK_END:
60              break;
61          default:
62              assert(FALSE);
63              break;
64      }
65      return fileSize;
66  }
67  #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

```

68  LIB_EXPORT void
69  _plat__NvErrors(
70      int          recoverable,
71      int          unrecoverable
72  )
73  {

```

```

74     s_NV_unrecoverable = unrecoverable;
75     s_NV_recoverable = recoverable;
76 }

```

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

```

77  LIB_EXPORT int
78  _plat__NVEnable(
79      void                *platParameter // IN: platform specific parameters
80  )
81  {
82      NOT_REFERENCED(platParameter); // to keep compiler quiet
83      //
84      // Start assuming everything is OK
85      s_NV_unrecoverable = FALSE;
86      s_NV_recoverable = FALSE;
87      #if FILE_BACKED_NV
88      if(s_NvFile != NULL)
89          return 0;
90      // Initialize all the bytes in the ram copy of the NV
91      _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
92      // If the file exists
93      if(NvFileOpen("r+b") >= 0)
94      {
95          long    fileSize = NvFileSize(SEEK_SET); // get the file size and leave the
96                                                    // file pointer at the start
97          //
98          // If the size is right, read the data
99          if(NV_MEMORY_SIZE == fileSize)
100              fread(s_NV, 1, NV_MEMORY_SIZE, s_NvFile);
101          else
102              NvFileCommit(); // for any other size, initialize it
103      }
104      // If NVChip file does not exist, try to create it for read/write.
105      else if(NvFileOpen("w+b") >= 0)
106          NvFileCommit(); // Initialize the file
107      assert(NULL != s_NvFile); // Just in case we are broken for some reason.
108      #endif
109      // NV contents have been initialized and the error checks have been performed. For
110      // simulation purposes, use the signaling interface to indicate if an error is
111      // to be simulated and the type of the error.
112      if(s_NV_unrecoverable)
113          return -1;
114      return s_NV_recoverable;
115  }

```

C.6.3.6. _plat__NVDisable()

Disable NV memory

```

117 LIB_EXPORT void
118 _plat__NVDisable(
119     void
120 )
121 {
122     #if FILE_BACKED_NV
123         if(NULL != s_NvFile)
124             fclose(s_NvFile);    // Close NV file
125         s_NvFile = NULL;        // Set file handle to NULL
126     #endif
127     return;
128 }

```

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

```

129 LIB_EXPORT int
130 _plat__IsNvAvailable(
131     void
132 )
133 {
134     int retVal = 0;
135     // NV is not available if the TPM is in failure mode
136     if(!s_NvIsAvailable)
137         retVal = 1;
138     #if FILE_BACKED_NV
139     else
140         retVal = (s_NvFile == NULL);
141     #endif
142     return retVal;
143 }

```

C.6.3.8. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```

144 LIB_EXPORT void
145 _plat__NvMemoryRead(
146     unsigned int    startOffset,    // IN: read start
147     unsigned int    size,          // IN: size of bytes to read
148     void            *data          // OUT: data buffer
149 )
150 {
151     assert(startOffset + size <= NV_MEMORY_SIZE);
152     memcpy(data, &s_NV[startOffset], size);    // Copy data from RAM
153     return;
154 }

```

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

```

155  LIB_EXPORT int
156  _plat__NvIsDifferent(
157      unsigned int    startOffset,    // IN: read start
158      unsigned int    size,           // IN: size of bytes to read
159      void            *data           // IN: data buffer
160  )
161  {
162      return (memcmp(&s_NV[startOffset], data, size) != 0);
163  }

```

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.

```

164  LIB_EXPORT BOOL
165  _plat__NvMemoryWrite(
166      unsigned int    startOffset,    // IN: write start
167      unsigned int    size,           // IN: size of bytes to write
168      void            *data           // OUT: data buffer
169  )
170  {
171      if(startOffset + size <= NV_MEMORY_SIZE)
172      {
173          memcpy(&s_NV[startOffset], data, size);    // Copy the data to the NV image
174          return TRUE;
175      }
176      return FALSE;
177  }

```

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.

```

178  LIB_EXPORT void
179  _plat__NvMemoryClear(
180      unsigned int    start,           // IN: clear start
181      unsigned int    size             // IN: number of bytes to clear
182  )
183  {
184      assert(start + size <= NV_MEMORY_SIZE);
185      // In this implementation, assume that the erase value for NV is all 1s
186      memset(&s_NV[start], 0xff, size);
187  }

```

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

```

188 LIB_EXPORT void
189 _plat__NvMemoryMove(
190     unsigned int    sourceOffset, // IN: source offset
191     unsigned int    destOffset,   // IN: destination offset
192     unsigned int    size          // IN: size of data being moved
193 )
194 {
195     assert(sourceOffset + size <= NV_MEMORY_SIZE);
196     assert(destOffset + size <= NV_MEMORY_SIZE);
197     memmove(&s_NV[destOffset], &s_NV[sourceOffset], size); // Move data in RAM
198     return;
199 }

```

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

```

200 LIB_EXPORT int
201 _plat__NvCommit(
202     void
203 )
204 {
205     #if FILE_BACKED_NV
206     return (NvFileCommit() ? 0 : 1);
207     #else
208     return 0;
209     #endif
210 }

```

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

```

211 LIB_EXPORT void
212 _plat__SetNvAvail(
213     void
214 )
215 {
216     s_NvIsAvailable = TRUE;
217     return;
218 }

```

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

```
219  LIB_EXPORT void
220  _plat_ClearNvAvail(
221      void
222  )
223  {
224      s_NvIsAvailable = FALSE;
225      return;
226  }
```

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     BOOL      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();
50
51     return;
52 }
```

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_
```

From Cancel.c Cancel flag. It is initialized as FALSE, which indicate the command is not being canceled

```
3  extern int      s_isCanceled;
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
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 BOOL          s_timerReset;
```

This value indicates that the timer was stopped. It causes a clock discontinuity.

```
19 extern BOOL          s_timerStopped;
```

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.

```
20 #define      CLOCK_NOMINAL          30000
```

A 1% change in rate is 300 counts

```
21 #define      CLOCK_ADJUST_COARSE    300
```

A 0.1% change in rate is 30 counts

```
22  #define      CLOCK_ADJUST_MEDIUM      30
```

A minimum change in rate is 1 count

```
23  #define      CLOCK_ADJUST_FINE        1
```

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16.7%)

```
24  #define      CLOCK_ADJUST_LIMIT       5000
```

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

```
25  extern uint64_t      s_initClock;
```

This variable records the timer adjustment factor.

```
26  extern unsigned int      s_adjustRate;
```

For LocalityPlat.c Locality of current command

```
27  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.

```
28  #if (!defined VTPM) || ((VTPM != NO) && (VTPM != YES))
29  #   undef VTPM
30  #   define      VTPM          YES          // Default: Either YES or NO
31  #endif
```

For a simulation, use a file to back up the NV

```
32  #if (!defined FILE_BACKED_NV) || ((FILE_BACKED_NV != NO) && (FILE_BACKED_NV != YES))
33  #   undef FILE_BACKED_NV
34  #   define      FILE_BACKED_NV      (VTPM && YES)      // Default: Either YES or NO
35  #endif
36  #if SIMULATION
37  #   undef FILE_BACKED_NV
38  #   define      FILE_BACKED_NV      YES
39  #endif // SIMULATION
40  extern unsigned char      s_NV[NV_MEMORY_SIZE];
41  extern BOOL               s_NvIsAvailable;
42  extern BOOL               s_NV_unrecoverable;
43  extern BOOL               s_NV_recoverable;
```

For PPPlat.c Physical presence. It is initialized to FALSE

```
44  extern BOOL      s_physicalPresence;
```

From Power

```
45  extern BOOL      s_powerLost;
```

For Entropy.c

```
46  extern uint32_t      lastEntropy;
47  #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  #include      "Platform.h"

    From Cancel.c

2  BOOL          s_isCanceled;

    From Clock.c

3  unsigned int    s_adjustRate;
4  BOOL           s_timerReset;
5  BOOL           s_timerStopped;
6  #ifndef HARDWARE_CLOCK
7  clock64_t      s_realTimePrevious;
8  clock64_t      s_tpmTime;
9  clock64_t      s_lastSystemTime;
10 clock64_t      s_lastReportedTime;
11 #endif

    From LocalityPlat.c

12 unsigned char  s_locality;

    From Power.c

13 BOOL          s_powerLost;

    From Entropy.c This values is used to determine if the entropy generator is broken. If two consecutive
    values are the same, then the entropy generator is considered to be broken.

14 uint32_t      lastEntropy;

    For NVMem.c

15 unsigned char  s_NV[NV_MEMORY_SIZE];
16 BOOL           s_NvIsAvailable;
17 BOOL           s_NV_unrecoverable;
18 BOOL           s_NV_recoverable;

    From PPPlat.c

19 BOOL  s_physicalPresence;
```

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  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.

```
5  LIB_EXPORT void
6  _plat__RunCommand(
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     setjmp(s_jumpBuffer);
14     ExecuteCommand(requestSize, request, responseSize, response);
15 }
```

C.11.3.2. `_plat__Fail()`

This is the platform depended failure exit for the TPM.

```
16 LIB_EXPORT NORETURN void
17 _plat__Fail(
18     void
19 )
20 {
21     longjmp(&s_jumpBuffer[0], 1);
22 }
```

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  const char notReallyUnique[] =
3  "This is not really a unique value. A real unique value should"
4  " 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.

```
5  LIB_EXPORT uint32_t
6  _plat__GetUnique(
7      uint32_t      which,          // authorities (0) or details
8      uint32_t      bSize,         // size of the buffer
9      unsigned char *b,            // output buffer
10 )
11 {
12     const char      *from = notReallyUnique;
13     uint32_t        retVal = 0;
14
15     if(which == 0) // the authorities value
16     {
17         for(retVal = 0;
18             *from != 0 && retVal < bSize;
19             retVal++)
20         {
21             *b++ = *from++;
22         }
23     }
24     else
25     {
26         #define uSize sizeof(notReallyUnique)
27         b = &b[((bSize < uSize) ? bSize : uSize) - 1];
28         for(retVal = 0;
29             *from != 0 && retVal < bSize;
30             retVal++)
31         {
32             *b-- = *from++;
33         }
34     }
35     return retVal;
36 }
```

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  FILE        *fDebug = NULL;
4  unsigned char *fn = "DebugFile.txt";
5  static FILE *
6  fileOpen(
7      unsigned char    *fn,
8      const char        *mode
9  )
10 {
11     FILE        *f;
12     #if defined _MSC_VER
13     if(fopen_s(&f, fn, mode) != 0)
14         f = NULL;
15     #else
16     f = fopen(fn, "w");
17     #endif
18     return f;
19 }

```

C.13.2.1. DebugFileOpen()

This function opens the file used to hold the debug data.

Return Value	Meaning
0	success
!= 0	error

```

20 int
21 DebugFileOpen(
22     void
23 )
24 {
25     unsigned char    timeString[100];
26     time_t            t = time(NULL);
27     //
28     // Get current date and time.
29     ctime_s(timeString, sizeof(timeString), &t);
30     // Try to open the debug file
31     fDebug = fileOpen(fn, "w");
32     if(fDebug)
33     {
34         fprintf(fDebug, "%s\n", timeString);
35         fclose(fDebug);
36         return 0;
37     }
38     return -1;
39 }
40 void
41 DebugFileClose(

```



```
42     void
43 }
44 {
45     if(fDebug)
46         fclose(fDebug);
47 }
48 void
49 DebugDumpBuffer(
50     int             size,
51     unsigned char   *buf,
52     unsigned char   *identifier
53 )
54 {
55     int             i;
56 //
57     FILE *f = fopen(fn, "a");
58     if(!f)
59         return;
60     if(identifier)
61         fprintf(fDebug, "%s\n", identifier);
62     if(buf)
63     {
64         for(i = 0; i < size; i++)
65         {
66             if((i % 16) == 0) && (i))
67                 fprintf(fDebug, "\n");
68             fprintf(fDebug, " %02X", buf[i]);
69         }
70         if((size % 16) != 0)
71             fprintf(fDebug, "\n");
72     }
73     fclose(f);
74 }
```

C.14 Platform.h

```
1  #ifndef    _PLATFORM_H_
2  #define    _PLATFORM_H_
3  #include "TpmBuildSwitches.h"
4  #include "BaseTypes.h"
5  #include "TPMB.h"
6  #include "MinMax.h"
7  #include "TpmProfile.h"
8  #include "PlatformData.h"
9  #include "Platform_fp.h"
10 #endif // _PLATFORM_H_
```

DRAFT

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 BYTE 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 BufferSize, BYTE[BufferSize] Buffer}
10 #define TPM_SIGNAL_HASH_END          7
11 #define TPM_SEND_COMMAND              8
12 // {BYTE Locality, UINT32 InBufferSize, BYTE[InBufferSize] InBuffer} ->
13 // {UINT32 OutBufferSize, BYTE[OutBufferSize] OutBuffer}
14 #define TPM_SIGNAL_CANCEL_ON          9
15 #define TPM_SIGNAL_CANCEL_OFF        10
16 #define TPM_SIGNAL_NV_ON             11
17 #define TPM_SIGNAL_NV_OFF            12
18 #define TPM_SIGNAL_KEY_CACHE_ON      13
19 #define TPM_SIGNAL_KEY_CACHE_OFF     14
20 #define TPM_REMOTE_HANDSHAKE         15
21 #define TPM_SET_ALTERNATIVE_RESULT   16
22 #define TPM_SIGNAL_RESET              17
23 #define TPM_SIGNAL_RESTART           18
24 #define TPM_SESSION_END              20
25 #define TPM_STOP                     21
26 #define TPM_GET_COMMAND_RESPONSE_SIZES 25
27 #define TPM_TEST_FAILURE_MODE        30
```

D.2.4. Enumerations and Structures

```
28 enum TpmEndPointInfo
29 {
30     tpmPlatformAvailable = 0x01,
31     tpmUsesTbs = 0x02,
32     tpmInRawMode = 0x04,
33     tpmSupportsPP = 0x08
34 };
35
```

```
36 // Existing RPC interface type definitions retained so that the implementation
37 // can be re-used
38 typedef struct in_buffer
39 {
40     unsigned long BufferSize;
41     unsigned char *Buffer;
42 } _IN_BUFFER;
43
44 typedef unsigned char *_OUTPUT_BUFFER;
45
46 typedef struct out_buffer
47 {
48     uint32_t      BufferSize;
49     _OUTPUT_BUFFER Buffer;
50 } _OUT_BUFFER;
51
52 #ifndef WIN32
53 typedef unsigned long      DWORD;
54 typedef void               *LPVOID;
55 #undef WINAPI
56 #endif
57
58 #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  #ifdef _MSC_VER
4  #include <windows.h>
5  #include <winsock.h>
6  #else
7  typedef int SOCKET;
8  #endif
9  #include <string.h>
10 #include <stdlib.h>
11 #include <stdint.h>
12 // #include "BaseTypes.h"
13 #include "TpmTcpProtocol.h"
14 #include "Manufacture_fp.h"
15 #include "Simulator_fp.h"

```

To access key cache control in TPM

```

16 void RsaKeyCacheControl(int state);
17 #ifndef __IGNORE_STATE__
18 static uint32_t ServerVersion = 1;
19 #define MAX_BUFFER 1048576
20 char InputBuffer[MAX_BUFFER]; //The input data buffer for the simulator.
21 char OutputBuffer[MAX_BUFFER]; //The output data buffer for the simulator.
22 struct
23 {
24     uint32_t largestCommandSize;
25     uint32_t largestCommand;
26     uint32_t largestResponseSize;
27     uint32_t largestResponse;
28 } CommandResponseSizes = {0};
29 #endif // __IGNORE_STATE__

```

D.3.3. Functions

D.3.3.1. CreateSocket()

This function creates a socket listening on *PortNumber*.

```

30 static int
31 CreateSocket(
32     int                PortNumber,
33     SOCKET             *listenSocket
34 )
35 {
36     WSADATA            wsaData;
37     struct sockaddr_in MyAddress;
38     int res;
39     //
40     // Initialize Winsock
41     res = WSASStartup(MAKEWORD(2, 2), &wsaData);
42     if(res != 0)

```

```

43     {
44         printf("WSAStartup failed with error: %d\n", res);
45         return -1;
46     }
47     // create listening socket
48     *listenSocket = socket(PF_INET, SOCK_STREAM, 0);
49     if(INVALID_SOCKET == *listenSocket)
50     {
51         printf("Cannot create server listen socket. Error is 0x%x\n",
52             WSAGetLastError());
53         return -1;
54     }
55     // bind the listening socket to the specified port
56     ZeroMemory(&MyAddress, sizeof(MyAddress));
57     MyAddress.sin_port = htons((short)PortNumber);
58     MyAddress.sin_family = AF_INET;
59
60     res = bind(*listenSocket, (struct sockaddr*) &MyAddress, sizeof(MyAddress));
61     if(res == SOCKET_ERROR)
62     {
63         printf("Bind error. Error is 0x%x\n", WSAGetLastError());
64         return -1;
65     }
66     // listen/wait for server connections
67     res = listen(*listenSocket, 3);
68     if(res == SOCKET_ERROR)
69     {
70         printf("Listen error. Error is 0x%x\n", WSAGetLastError());
71         return -1;
72     }
73     return 0;
74 }

```

D.3.3.2. PlatformServer()

This function processes incoming platform requests.

```

75 BOOL
76 PlatformServer(
77     SOCKET s
78 )
79 {
80     BOOL OK = TRUE;
81     uint32_t Command;
82     //
83     for(;;)
84     {
85         OK = ReadBytes(s, (char*)&Command, 4);
86         // client disconnected (or other error). We stop processing this client
87         // and return to our caller who can stop the server or listen for another
88         // connection.
89         if(!OK) return TRUE;
90         Command = ntohl(Command);
91         switch(Command)
92         {
93             case TPM_SIGNAL_POWER_ON:
94                 _rpc_Signal_PowerOn(FALSE);
95                 break;
96             case TPM_SIGNAL_POWER_OFF:
97                 _rpc_Signal_PowerOff();
98                 break;
99             case TPM_SIGNAL_RESET:
100                 _rpc_Signal_PowerOn(TRUE);
101                 break;

```

```

102     case TPM_SIGNAL_RESTART:
103         _rpc_Signal_Restart();
104         break;
105     case TPM_SIGNAL_PHYS_PRE_ON:
106         _rpc_Signal_PhysicalPresenceOn();
107         break;
108     case TPM_SIGNAL_PHYS_PRE_OFF:
109         _rpc_Signal_PhysicalPresenceOff();
110         break;
111     case TPM_SIGNAL_CANCEL_ON:
112         _rpc_Signal_CancelOn();
113         break;
114     case TPM_SIGNAL_CANCEL_OFF:
115         _rpc_Signal_CancelOff();
116         break;
117     case TPM_SIGNAL_NV_ON:
118         _rpc_Signal_NvOn();
119         break;
120     case TPM_SIGNAL_NV_OFF:
121         _rpc_Signal_NvOff();
122         break;
123     case TPM_SIGNAL_KEY_CACHE_ON:
124         _rpc_RsaKeyCacheControl(TRUE);
125         break;
126     case TPM_SIGNAL_KEY_CACHE_OFF:
127         _rpc_RsaKeyCacheControl(FALSE);
128         break;
129     case TPM_SESSION_END:
130         // Client signaled end-of-session
131         TpmEndSimulation();
132         return TRUE;
133     case TPM_STOP:
134         // Client requested the simulator to exit
135         return FALSE;
136     case TPM_TEST_FAILURE_MODE:
137         _rpc_ForceFailureMode();
138         break;
139     case TPM_GET_COMMAND_RESPONSE_SIZES:
140         OK = WriteVarBytes(s, (char *)&CommandResponseSizes,
141                             sizeof(CommandResponseSizes));
142         memset(&CommandResponseSizes, 0, sizeof(CommandResponseSizes));
143         if(!OK)
144             return TRUE;
145         break;
146     default:
147         printf("Unrecognized platform interface command %d\n",
148               (int)Command);
149         WriteUINT32(s, 1);
150         return TRUE;
151     }
152     WriteUINT32(s, 0);
153 }
154 return FALSE;
155 }

```

D.3.3.3. PlatformSvcRoutine()

This function is called to set up the socket interfaces to listen for commands.

```

156 DWORD WINAPI
157 PlatformSvcRoutine(
158     LPVOID      port
159 )
160 {

```



```

161     int                PortNumber = (int)(INT_PTR)port;
162     SOCKET             listenSocket, serverSocket;
163     struct             sockaddr_in HerAddress;
164     int                res;
165     int                length;
166     BOOL               continueServing;
167 //
168     res = CreateSocket(PortNumber, &listenSocket);
169     if(res != 0)
170     {
171         printf("Create platform service socket fail\n");
172         return res;
173     }
174     // Loop accepting connections one-by-one until we are killed or asked to stop
175     // Note the platform service is single-threaded so we don't listen for a new
176     // connection until the prior connection drops.
177     do
178     {
179         printf("Platform server listening on port %d\n", PortNumber);
180
181         // blocking accept
182         length = sizeof(HerAddress);
183         serverSocket = accept(listenSocket,
184                               (struct sockaddr*) &HerAddress,
185                               &length);
186         if(serverSocket == INVALID_SOCKET)
187         {
188             printf("Accept error. Error is 0x%x\n", WSAGetLastError());
189             return -1;
190         }
191         printf("Client accepted\n");
192
193         // normal behavior on client disconnection is to wait for a new client
194         // to connect
195         continueServing = PlatformServer(serverSocket);
196         closesocket(serverSocket);
197     } while(continueServing);
198
199     return 0;
200 }

```

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.

```

201 int
202 PlatformSignalService(
203     int                PortNumber
204 )
205 {
206     HANDLE             hPlatformSvc;
207     int                ThreadId;
208     int                port = PortNumber;
209 //
210     // Create service thread for platform signals
211     hPlatformSvc = CreateThread(NULL, 0,
212                                (LPTHREAD_START_ROUTINE) PlatformSvcRoutine,
213                                (LPVOID)(INT_PTR)port, 0, (LPDWORD)&ThreadId);
214     if(hPlatformSvc == NULL)
215     {
216         printf("Thread Creation failed\n");
217         return -1;
218     }

```

```

219     return 0;
220 }

```

D.3.3.5. RegularCommandService()

This function services regular commands.

```

221 int
222 RegularCommandService(
223     int          PortNumber
224 )
225 {
226     SOCKET        listenSocket;
227     SOCKET        serverSocket;
228     struct        sockaddr_in HerAddress;
229     int           res, length;
230     BOOL          continueServing;
231 //
232     res = CreateSocket(PortNumber, &listenSocket);
233     if(res != 0)
234     {
235         printf("Create platform service socket fail\n");
236         return res;
237     }
238     // Loop accepting connections one-by-one until we are killed or asked to stop
239     // Note the TPM command service is single-threaded so we don't listen for
240     // a new connection until the prior connection drops.
241     do
242     {
243         printf("TPM command server listening on port %d\n", PortNumber);
244
245         // blocking accept
246         length = sizeof(HerAddress);
247         serverSocket = accept(listenSocket,
248                               (struct sockaddr*) &HerAddress,
249                               &length);
250         if(serverSocket == SOCKET_ERROR)
251         {
252             printf("Accept error. Error is 0x%x\n", WSAGetLastError());
253             return -1;
254         }
255         printf("Client accepted\n");
256
257         // normal behavior on client disconnection is to wait for a new client
258         // to connect
259         continueServing = TpmServer(serverSocket);
260         closesocket(serverSocket);
261     } while(continueServing);
262     return 0;
263 }

```

D.3.3.6. 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.

```

264 int
265 StartTcpServer(
266     int          PortNumber
267 )
268 {
269     int          res;

```

```

270 //
271 // Start Platform Signal Processing Service
272 res = PlatformSignalService(PortNumber + 1);
273 if(res != 0)
274 {
275     printf("PlatformSignalService failed\n");
276     return res;
277 }
278 // Start Regular/DRTM TPM command service
279 res = RegularCommandService(PortNumber);
280 if(res != 0)
281 {
282     printf("RegularCommandService failed\n");
283     return res;
284 }
285 return 0;
286 }

```

D.3.3.7. ReadBytes()

This function reads the indicated number of bytes (*NumBytes*) into buffer from the indicated socket.

```

287 BOOL
288 ReadBytes(
289     SOCKET          s,
290     char            *buffer,
291     int             NumBytes
292 )
293 {
294     int             res;
295     int             numGot = 0;
296 //
297 while(numGot < NumBytes)
298 {
299     res = recv(s, buffer + numGot, NumBytes - numGot, 0);
300     if(res == -1)
301     {
302         printf("Receive error. Error is 0x%x\n", WSAGetLastError());
303         return FALSE;
304     }
305     if(res == 0)
306     {
307         return FALSE;
308     }
309     numGot += res;
310 }
311 return TRUE;
312 }

```

D.3.3.8. WriteBytes()

This function will send the indicated number of bytes (*NumBytes*) to the indicated socket

```

313 BOOL
314 WriteBytes(
315     SOCKET          s,
316     char            *buffer,
317     int             NumBytes
318 )
319 {
320     int             res;
321     int             numSent = 0;
322 //

```

```

323     while(numSent < NumBytes)
324     {
325         res = send(s, buffer + numSent, NumBytes - numSent, 0);
326         if(res == -1)
327         {
328             if(WSAGetLastError() == 0x2745)
329             {
330                 printf("Client disconnected\n");
331             }
332             else
333             {
334                 printf("Send error. Error is 0x%x\n", WSAGetLastError());
335             }
336             return FALSE;
337         }
338         numSent += res;
339     }
340     return TRUE;
341 }

```

D.3.3.9. WriteUINT32()

Send 4 bytes containing htonl(1)

```

342  BOOL
343  WriteUINT32(
344      SOCKET          s,
345      uint32_t         val
346  )
347  {
348      UINT32 netVal = htonl(val);
349      //
350      return WriteBytes(s, (char*)&netVal, 4);
351  }

```

D.3.3.10. ReadVarBytes()

Get a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```

352  BOOL
353  ReadVarBytes(
354      SOCKET          s,
355      char             *buffer,
356      uint32_t         *BytesReceived,
357      int              MaxLen
358  )
359  {
360      int              length;
361      BOOL             res;
362      //
363      res = ReadBytes(s, (char*)&length, 4);
364      if(!res) return res;
365      length = ntohl(length);
366      *BytesReceived = length;
367      if(length > MaxLen)
368      {
369          printf("Buffer too big. Client says %d\n", length);
370          return FALSE;
371      }
372      if(length == 0) return TRUE;
373      res = ReadBytes(s, buffer, length);
374      if(!res) return res;

```

```

375     return TRUE;
376 }

```

D.3.3.11. WriteVarBytes()

Send a UINT32-length-prepended binary array. Note that the 4-byte length is in network byte order (big-endian).

```

377 BOOL
378 WriteVarBytes(
379     SOCKET          s,
380     char            *buffer,
381     int              BytesToSend
382 )
383 {
384     uint32_t          netLength = htonl(BytesToSend);
385     BOOL res;
386     //
387     res = WriteBytes(s, (char*)&netLength, 4);
388     if(!res)
389         return res;
390     res = WriteBytes(s, buffer, BytesToSend);
391     if(!res)
392         return res;
393     return TRUE;
394 }

```

D.3.3.12. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```

395 BOOL
396 TpmServer(
397     SOCKET          s
398 )
399 {
400     uint32_t          length;
401     uint32_t          Command;
402     BYTE              locality;
403     BOOL              OK;
404     int               result;
405     int               clientVersion;
406     _IN_BUFFER        InBuffer;
407     _OUT_BUFFER       OutBuffer;
408     //
409     for(;;)
410     {
411         OK = ReadBytes(s, (char*)&Command, 4);
412         // client disconnected (or other error). We stop processing this client
413         // and return to our caller who can stop the server or listen for another
414         // connection.
415         if(!OK)
416             return TRUE;
417         Command = ntohl(Command);
418         switch(Command)
419         {
420             case TPM_SIGNAL_HASH_START:
421                 _rpc_Signal_Hash_Start();
422                 break;
423             case TPM_SIGNAL_HASH_END:
424                 _rpc_Signal_HashEnd();
425                 break;
426             case TPM_SIGNAL_HASH_DATA:

```

```

427         OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
428         if(!OK) return TRUE;
429         InBuffer.Buffer = (BYTE*)InputBuffer;
430         InBuffer.BufferSize = length;
431         _rpc_Signal_Hash_Data(InBuffer);
432         break;
433     case TPM_SEND_COMMAND:
434         OK = ReadBytes(s, (char*)&locality, 1);
435         if(!OK)
436             return TRUE;
437         OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
438         if(!OK)
439             return TRUE;
440         InBuffer.Buffer = (BYTE*)InputBuffer;
441         InBuffer.BufferSize = length;
442         OutBuffer.BufferSize = MAX_BUFFER;
443         OutBuffer.Buffer = (_OUTPUT_BUFFER)OutputBuffer;
444         // record the number of bytes in the command if it is the largest
445         // we have seen so far.
446         if(InBuffer.BufferSize > CommandResponseSizes.largestCommandSize)
447         {
448             CommandResponseSizes.largestCommandSize = InBuffer.BufferSize;
449             memcpy(&CommandResponseSizes.largestCommand,
450                 &InputBuffer[6], sizeof(UINT32));
451         }
452         _rpc_Send_Command(locality, InBuffer, &OutBuffer);
453         // record the number of bytes in the response if it is the largest
454         // we have seen so far.
455         if(OutBuffer.BufferSize > CommandResponseSizes.largestResponseSize)
456         {
457             CommandResponseSizes.largestResponseSize
458                 = OutBuffer.BufferSize;
459             memcpy(&CommandResponseSizes.largestResponse,
460                 &OutputBuffer[6], sizeof(UINT32));
461         }
462         OK = WriteVarBytes(s,
463             (char*)OutBuffer.Buffer,
464             OutBuffer.BufferSize);
465         if(!OK)
466             return TRUE;
467         break;
468     case TPM_REMOTE_HANDSHAKE:
469         OK = ReadBytes(s, (char*)&clientVersion, 4);
470         if(!OK)
471             return TRUE;
472         if(clientVersion == 0)
473         {
474             printf("Unsupported client version (0).\n");
475             return TRUE;
476         }
477         OK &= WriteUINT32(s, ServerVersion);
478         OK &= WriteUINT32(s, tpmInRawMode
479             | tpmPlatformAvailable | tpmSupportsPP);
480         break;
481     case TPM_SET_ALTERNATIVE_RESULT:
482         OK = ReadBytes(s, (char*)&result, 4);
483         if(!OK)
484             return TRUE;
485         // Alternative result is not applicable to the simulator.
486         break;
487     case TPM_SESSION_END:
488         // Client signaled end-of-session
489         return TRUE;
490     case TPM_STOP:
491         // Client requested the simulator to exit
492         return FALSE;

```

```
493         default:
494             printf("Unrecognized TPM interface command %d\n", (int)Command);
495             return TRUE;
496     }
497     OK = WriteUINT32(s, 0);
498     if(!OK)
499         return TRUE;
500 }
501 return FALSE;
502 }
```

DRAFT

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 <stdlib.h>
2  #include <stdio.h>
3  #include <setjmp.h>
4  #include "TpmBuildSwitches.h"
5  #include <windows.h>
6  #include <winsock.h>
7  #include "Platform_fp.h"
8  #include "ExecCommand_fp.h"
9  #include "Manufacture_fp.h"
10 #include "_TPM_Init_fp.h"
11 #include "_TPM_Hash_Start_fp.h"
12 #include "_TPM_Hash_Data_fp.h"
13 #include "_TPM_Hash_End_fp.h"
14 #include "TpmFail_fp.h"
15 #include "TpmTcpProtocol.h"
16 #include "Simulator_fp.h"
17 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.

```

18 void
19 _rpc_Signal_PowerOn(
20     BOOL      isReset
21 )
22 {
23     // if power is on and this is not a call to do TPM reset then return
24     if(s_isPowerOn && !isReset)
25         return;
26
27     // If this is a reset but power is not on, then return
28     if(isReset && !s_isPowerOn)
29         return;
30
31     // Unless this is just a reset, pass power on signal to platform
32     if(!isReset)
33         _plat_Signal_PowerOn();
34
35     // Power on and reset both lead to _TPM_Init()
36     _plat_Signal_Reset();
37
38     // Set state as power on
39     s_isPowerOn = TRUE;
40 }

```


D.4.3.2. Signal_Restart()

This function processes the clock restart indication. All it does is call the platform function.

```

41 void
42 __rpc__Signal_Restart(
43     void
44 )
45 {
46     __plat__TimerRestart();
47 }

```

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.

```

48 void
49 __rpc__Signal_PowerOff(
50     void
51 )
52 {
53     if(!s_isPowerOn) return;
54
55     // Pass power off signal to platform
56     __plat__Signal_PowerOff();
57
58     s_isPowerOn = FALSE;
59
60     return;
61 }

```

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.

```

62 void
63 __rpc__ForceFailureMode(
64     void
65 )
66 {
67     SetForceFailureMode();
68 }

```

D.4.3.5. __rpc__Signal_PhysicalPresenceOn()

This function is called to simulate activation of the physical presence `pin`.

```

69 void
70 __rpc__Signal_PhysicalPresenceOn(
71     void
72 )
73 {
74     // If TPM is power off, reject this signal
75     if(!s_isPowerOn) return;
76
77     // Pass physical presence on to platform
78     __plat__Signal_PhysicalPresenceOn();
79 }

```

```

80     return;
81 }

```

D.4.3.6. _rpc__Signal_PhysicalPresenceOff()

This function is called to simulate deactivation of the physical presence **pin**.

```

82 void
83 _rpc__Signal_PhysicalPresenceOff(
84     void
85 )
86 {
87     // If TPM is power off, reject this signal
88     if(!s_isPowerOn) return;
89
90     // Pass physical presence off to platform
91     _plat__Signal_PhysicalPresenceOff();
92
93     return;
94 }

```

D.4.3.7. _rpc__Signal_Hash_Start()

This function is called to simulate a _TPM_Hash_Start() event. It will call

```

95 void
96 _rpc__Signal_Hash_Start(
97     void
98 )
99 {
100     // If TPM is power off, reject this signal
101     if(!s_isPowerOn) return;
102
103     // Pass _TPM_Hash_Start signal to TPM
104     _TPM_Hash_Start();
105     return;
106 }

```

D.4.3.8. _rpc__Signal_Hash_Data()

This function is called to simulate a _TPM_Hash_Data() event.

```

107 void
108 _rpc__Signal_Hash_Data(
109     IN_BUFFER    input
110 )
111 {
112     // If TPM is power off, reject this signal
113     if(!s_isPowerOn) return;
114
115     // Pass _TPM_Hash_Data signal to TPM
116     _TPM_Hash_Data(input.BufferSize, input.Buffer);
117     return;
118 }

```

D.4.3.9. _rpc__Signal_HashEnd()

This function is called to simulate a _TPM_Hash_End() event.

```

119 void

```

```

120 _rpc__Signal_HashEnd(
121     void
122 )
123 {
124     // If TPM is power off, reject this signal
125     if(!s_isPowerOn) return;
126
127     // Pass _TPM_HashEnd signal to TPM
128     _TPM_Hash_End();
129     return;
130 }

```

D.4.3.10. **_rpc__Send_Command()**

This is the interface to the TPM code.

```

131 void
132 _rpc__Send_Command(
133     unsigned char    locality,
134     _IN_BUFFER      request,
135     _OUT_BUFFER     *response
136 )
137 {
138     // If TPM is power off, reject any commands.
139     if(!s_isPowerOn)
140     {
141         response->BufferSize = 0;
142         return;
143     }
144     // Set the locality of the command so that it doesn't change during the command
145     _plat__LocalitySet(locality);
146     // Do implementation-specific command dispatch
147     _plat__RunCommand(request.BufferSize, request.Buffer,
148         &response->BufferSize, &response->Buffer);
149     return;
150 }

```

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.

```

151 void
152 _rpc__Signal_CancelOn(
153     void
154 )
155 {
156     // If TPM is power off, reject this signal
157     if(!s_isPowerOn) return;
158
159     // Set the platform canceling flag.
160     _plat__SetCancel();
161
162     return;
163 }

```

D.4.3.12. **_rpc__Signal_CancelOff()**

This function is used to turn off the indication to cancel a command in process.

```

164 void
165 __rpc__Signal_CancelOff(
166     void
167 )
168 {
169     // If TPM is power off, reject this signal
170     if(!s_isPowerOn) return;
171
172     // Set the platform canceling flag.
173     _plat__ClearCancel();
174
175     return;
176 }

```

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.

```

177 void
178 __rpc__Signal_NvOn(
179     void
180 )
181 {
182     // If TPM is power off, reject this signal
183     if(!s_isPowerOn) return;
184
185     _plat__SetNvAvail();
186     return;
187 }

```

D.4.3.14. __rpc__Signal_NvOff()

This function is used to set the indication that NV memory is no longer available.

```

188 void
189 __rpc__Signal_NvOff(
190     void
191 )
192 {
193     // If TPM is power off, reject this signal
194     if(!s_isPowerOn) return;
195
196     _plat__ClearNvAvail();
197     return;
198 }
199 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.

```

200 void
201 __rpc__RsaKeyCacheControl(
202     int state
203 )
204 {
205     #if USE_RSA_KEY_CACHE
206         RsaKeyCacheControl(state);
207     #else
208         NOT_REFERENCED(state);

```

```
209 #endif
210 }
```

D.4.3.16. `_rpc__Shutdown()`

This function is used to stop the TPM simulator.

```
211 void
212 _rpc__Shutdown(
213     void
214 )
215 {
216     RPC_STATUS status;
217
218     // Stop TPM
219     TPM_TearDown();
220
221     status = RpcMgmtStopServerListening(NULL);
222     if(status != RPC_S_OK)
223     {
224         printf("RpcMgmtStopServerListening returned: 0x%x\n", status);
225         exit(status);
226     }
227
228     status = RpcServerUnregisterIf(NULL, NULL, FALSE);
229     if(status != RPC_S_OK)
230     {
231         printf("RpcServerUnregisterIf returned 0x%x\n", status);
232         exit(status);
233     }
234     return;
235 }
```

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  #include <stdlib.h>
3  #include <stdio.h>
4  #include <stdint.h>
5  #include <ctype.h>
6  #include <string.h>
7  #include <windows.h>
8  #include <winsock.h>
9  #include "TpmTcpProtocol.h"
10 #include "Manufacture_fp.h"
11 #include "Platform_fp.h"
12 #include "Simulator_fp.h"
13 #define PURPOSE \
14 "TPM Reference Simulator.\nCopyright Microsoft Corp.\n"
15 #define DEFAULT_TPM_PORT 2321
16 void* MainPointer;
```

D.5.3. Functions

D.5.3.1. Usage()

This function prints the proper calling sequence for the simulator.

```

17 static void
18 Usage(
19     char                *pszProgramName
20 )
21 {
22     fprintf(stderr, "%s", PURPOSE);
23     fprintf(stderr, "Usage:\n");
24     fprintf(stderr, "%s          - Starts the TPM server listening on port %d\n",
25             pszProgramName, DEFAULT_TPM_PORT);
26     fprintf(stderr,
27             "%s PortNum - Starts the TPM server listening on port PortNum\n",
28             pszProgramName);
29     fprintf(stderr, "%s ?          - This message\n", pszProgramName);
30     exit(1);
31 }
```

D.5.3.2. main()

This is the main entry point for the simulator. It registers the interface and starts listening for clients

```

32 int
33 main(
34     int                argc,
35     char                *argv[]
36 )
37 {
38     int portNum = DEFAULT_TPM_PORT;
39     if(argc > 2)
```

```
40     {
41         Usage(argv[0]);
42     }
43
44     if(argc == 2)
45     {
46         if(strcmp(argv[1], "?") == 0)
47         {
48             Usage(argv[0]);
49         }
50         portNum = atoi(argv[1]);
51         if(portNum <= 0 || portNum > 65535)
52         {
53             Usage(argv[0]);
54         }
55     }
56     _plat__NVEnable(NULL);
57
58     if(TPM_Manufacture(1) != 0)
59     {
60         exit(1);
61     }
62     // Coverage test - repeated manufacturing attempt
63     if(TPM_Manufacture(0) != 1)
64     {
65         exit(2);
66     }
67     // Coverage test - re-manufacturing
68     TPM_TearDown();
69     if(TPM_Manufacture(1) != 0)
70     {
71         exit(3);
72     }
73     // Disable NV memory
74     _plat__NVDisable();
75
76     StartTcpServer(portNum);
77     return EXIT_SUCCESS;
78 }
```