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_rclIndex ........................................................... 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
Part 4: Supporting Routines

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
6.4.4 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() .................................................. 212
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 UpdateAllNoneTPM() ............................................... 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 GetV2BSize() ...................................................................... 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
8.4.5 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() ......................................................................... 326
    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 ObjectCleanupEvent() .......................................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 PCRBelongsToAuthGroup() ................................346
8.7.3.2 PCRBelongsToPolicyGroup() ...............................347
8.7.3.3 PCRBelongsToTCBGroup() ....................................347
8.7.3.4 PCRPolicyIsAvailable() ......................................348
8.7.3.5 PCRGetAuthValue() ...........................................348
8.7.3.6 PCRGetAuthPolicy() .........................................349
8.7.3.7 PCRStart() ......................................................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 PcrDtrm() .......................................................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 PCRsExtendAllowed() ........................................................... 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 PCRAssociate() ............................................................... 361
8.7.3.26 PCRSetsValue() ............................................................. 362
8.7.3.27 PCRDReadContext() ..................................................... 363
8.7.3.28 PCRCapGetAllocation() .............................................. 364
8.7.3.29 PCRCapGetSelectBit() ................................................. 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 PhysicalPresenceClear() ............................................... 370
  8.8.3.5 PhysicalPresenceCapGetCList() .................................... 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 SessionPcrValuesIsCurrent() ........................................ 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 SessionCapSetActiveNumber() ..................................... 387
  8.9.6.14 SessionCapSetActiveAvail() ........................................ 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 CommandCodeToCommandIndex() ....................... 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 IsHandlingResponse() ...................................... 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
<table>
<thead>
<tr>
<th>Section</th>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.1.1</td>
<td>BnValues.h</td>
<td>465</td>
</tr>
<tr>
<td>10.1.1.1</td>
<td>Introduction</td>
<td>465</td>
</tr>
<tr>
<td>10.1.1.2</td>
<td>Defines</td>
<td>465</td>
</tr>
<tr>
<td>10.1.2</td>
<td>CryptEcc.h</td>
<td>470</td>
</tr>
<tr>
<td>10.1.2.1</td>
<td>Introduction</td>
<td>470</td>
</tr>
<tr>
<td>10.1.2.2</td>
<td>Structures</td>
<td>470</td>
</tr>
<tr>
<td>10.1.3</td>
<td>CryptHash.h</td>
<td>471</td>
</tr>
<tr>
<td>10.1.3.1</td>
<td>Introduction</td>
<td>471</td>
</tr>
<tr>
<td>10.1.3.2</td>
<td>Hash-related Structures</td>
<td>471</td>
</tr>
<tr>
<td>10.1.3.3</td>
<td>HMAC State Structures</td>
<td>474</td>
</tr>
<tr>
<td>10.1.4</td>
<td>CryptRand.h</td>
<td>476</td>
</tr>
<tr>
<td>10.1.4.1</td>
<td>Introduction</td>
<td>476</td>
</tr>
<tr>
<td>10.1.4.2</td>
<td>DRBG Structures and Defines</td>
<td>476</td>
</tr>
<tr>
<td>10.1.5</td>
<td>CryptRsa.h</td>
<td>479</td>
</tr>
<tr>
<td>10.1.6</td>
<td>CryptTest.h</td>
<td>480</td>
</tr>
<tr>
<td>10.1.7</td>
<td>HashTestData.h</td>
<td>481</td>
</tr>
<tr>
<td>10.1.8</td>
<td>KdfTestData.h</td>
<td>482</td>
</tr>
<tr>
<td>10.1.9</td>
<td>RsaTestData.h</td>
<td>483</td>
</tr>
<tr>
<td>10.1.10</td>
<td>SymmetricTestData.h</td>
<td>489</td>
</tr>
<tr>
<td>10.1.11</td>
<td>SymmetricTest.h</td>
<td>491</td>
</tr>
<tr>
<td>10.1.11.1</td>
<td>Introduction</td>
<td>491</td>
</tr>
<tr>
<td>10.1.11.2</td>
<td>Symmetric Test Structures</td>
<td>491</td>
</tr>
<tr>
<td>10.1.12</td>
<td>EccTestData.h</td>
<td>492</td>
</tr>
<tr>
<td>10.1.13</td>
<td>CryptSym.h</td>
<td>494</td>
</tr>
<tr>
<td>10.1.13.1</td>
<td>Introduction</td>
<td>494</td>
</tr>
<tr>
<td>10.1.13.2</td>
<td>Includes, Defines, and Typedefs</td>
<td>494</td>
</tr>
<tr>
<td>10.1.14</td>
<td>OIDs.h</td>
<td>496</td>
</tr>
<tr>
<td>10.1.15</td>
<td>TpmAsn1.h</td>
<td>500</td>
</tr>
<tr>
<td>10.1.15.1</td>
<td>Introduction</td>
<td>500</td>
</tr>
<tr>
<td>10.1.15.2</td>
<td>Includes</td>
<td>500</td>
</tr>
<tr>
<td>10.1.15.3</td>
<td>Defined Constants</td>
<td>500</td>
</tr>
<tr>
<td>10.1.15.3.1</td>
<td>ASN.1 Universal Types (Class 00b)</td>
<td>500</td>
</tr>
<tr>
<td>10.1.15.4</td>
<td>Macros</td>
<td>500</td>
</tr>
<tr>
<td>10.1.15.4.1</td>
<td>Unmarshaling Macros</td>
<td>500</td>
</tr>
<tr>
<td>10.1.15.4.2</td>
<td>Marshaling Macros</td>
<td>501</td>
</tr>
<tr>
<td>10.1.15.5</td>
<td>Structures</td>
<td>501</td>
</tr>
<tr>
<td>10.1.16</td>
<td>X509.h</td>
<td>502</td>
</tr>
<tr>
<td>10.1.16.1</td>
<td>Introduction</td>
<td>502</td>
</tr>
<tr>
<td>10.1.16.2</td>
<td>Includes</td>
<td>502</td>
</tr>
<tr>
<td>10.1.16.3</td>
<td>Defined Constants</td>
<td>502</td>
</tr>
<tr>
<td>10.1.16.3.1</td>
<td>X509 Application-specific types</td>
<td>502</td>
</tr>
<tr>
<td>10.1.16.4</td>
<td>Structures</td>
<td>502</td>
</tr>
<tr>
<td>10.1.16.5</td>
<td>Global X509 Constants</td>
<td>503</td>
</tr>
<tr>
<td>10.1.17</td>
<td>MinMax.h</td>
<td>504</td>
</tr>
<tr>
<td>10.1.18</td>
<td>TpmAlgorithmDefines.h</td>
<td>505</td>
</tr>
<tr>
<td>10.2</td>
<td>Source</td>
<td>511</td>
</tr>
</tbody>
</table>
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 MakeIV() .................................................................... 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
## Part 4: Supporting Routines

**Trusted Platform Module Library**

### 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 CryptCmacEnc() ......................................... 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 CryptXOROblfuscation() ................................ 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
10.2.7 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 CryptDeSIssWeakKey() ............................ 596
10.2.9.2.3 CryptDeSIssValidateKey() .......................... 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 CryptEcclInit() .................................... 604
10.2.11.2.2 CryptEcclStartup() .................................. 604
10.2.11.2.3 ClearPoint2B(generic) .............................. 604
10.2.11.2.4 CryptEccGetParametersByCurvelId() ............ 605
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 BnSignEccSm2()..............................................624
10.2.12.3.7 CryptEccSign().................................................625
10.2.12.3.8 BnValidateSignatureEcdsa().................................627
10.2.12.3.9 BnValidateSignatureEccSm2()..............................628
10.2.12.3.10 BnValidateSignatureEccSchnorr()..........................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()..........................................633

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
<table>
<thead>
<tr>
<th>Section</th>
<th>File</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2.13.7.1</td>
<td>CryptHmacStart()</td>
<td>642</td>
</tr>
<tr>
<td>10.2.13.7.2</td>
<td>CryptHmacEnd()</td>
<td>643</td>
</tr>
<tr>
<td>10.2.13.7.3</td>
<td>CryptHmacStart2B()</td>
<td>643</td>
</tr>
<tr>
<td>10.2.13.7.4</td>
<td>CryptHmacEnd2B()</td>
<td>644</td>
</tr>
<tr>
<td>10.2.13.8.1</td>
<td>CryptMGF1()</td>
<td>644</td>
</tr>
<tr>
<td>10.2.13.8.2</td>
<td>CryptKDFa()</td>
<td>645</td>
</tr>
<tr>
<td>10.2.13.8.3</td>
<td>CryptKDFe()</td>
<td>647</td>
</tr>
<tr>
<td>10.2.14</td>
<td>CryptPrime.c</td>
<td>649</td>
</tr>
<tr>
<td>10.2.14.1</td>
<td>Introduction</td>
<td>649</td>
</tr>
<tr>
<td>10.2.14.2</td>
<td>Functions</td>
<td>649</td>
</tr>
<tr>
<td>10.2.14.2.1</td>
<td>Root2()</td>
<td>649</td>
</tr>
<tr>
<td>10.2.14.2.2</td>
<td>IsPrimeInt()</td>
<td>649</td>
</tr>
<tr>
<td>10.2.14.2.3</td>
<td>BnIsProbablyPrime()</td>
<td>650</td>
</tr>
<tr>
<td>10.2.14.2.4</td>
<td>MillerRabinRounds()</td>
<td>650</td>
</tr>
<tr>
<td>10.2.14.2.5</td>
<td>MillerRabin()</td>
<td>651</td>
</tr>
<tr>
<td>10.2.14.2.6</td>
<td>RsacCheckPrime()</td>
<td>652</td>
</tr>
<tr>
<td>10.2.14.2.7</td>
<td>AdjustPrimeCandidate()</td>
<td>653</td>
</tr>
<tr>
<td>10.2.14.2.8</td>
<td>BnGeneratePrimeForRSA()</td>
<td>653</td>
</tr>
<tr>
<td>10.2.15</td>
<td>CryptPrimeSieve.c</td>
<td>655</td>
</tr>
<tr>
<td>10.2.15.1</td>
<td>Includes and defines</td>
<td>655</td>
</tr>
<tr>
<td>10.2.15.2</td>
<td>Functions</td>
<td>655</td>
</tr>
<tr>
<td>10.2.15.2.1</td>
<td>RsAdjustPrimeLimit()</td>
<td>655</td>
</tr>
<tr>
<td>10.2.15.2.2</td>
<td>RsaNExtPrime()</td>
<td>655</td>
</tr>
<tr>
<td>10.2.15.2.3</td>
<td>BitsInArry()</td>
<td>657</td>
</tr>
<tr>
<td>10.2.15.2.4</td>
<td>FindNthSetBit()</td>
<td>657</td>
</tr>
<tr>
<td>10.2.15.2.5</td>
<td>PrimeSieve()</td>
<td>658</td>
</tr>
<tr>
<td>10.2.15.2.6</td>
<td>SetFieldSize()</td>
<td>660</td>
</tr>
<tr>
<td>10.2.15.2.7</td>
<td>PrimeSelectWithSieve()</td>
<td>660</td>
</tr>
<tr>
<td>10.2.16</td>
<td>CryptRand.c</td>
<td>664</td>
</tr>
<tr>
<td>10.2.16.1</td>
<td>Introduction</td>
<td>664</td>
</tr>
<tr>
<td>10.2.16.2</td>
<td>Derivation Functions</td>
<td>664</td>
</tr>
<tr>
<td>10.2.16.2.1</td>
<td>Description</td>
<td>664</td>
</tr>
</tbody>
</table>
10.2.18 CryptSmac.c ............................................. 706

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
  10.2.17.4 Internal Functions ................................ 682
     10.2.17.4.1 RsaiInitializeExponent() ................... 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 RsaiPrivateKeyOp() .......................... 685
     10.2.17.4.7 RSAEP() ...................................... 685
     10.2.17.4.8 RSDAP() ...................................... 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 RSAES_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

10.2.16.3 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_Initiate() ................................ 680
  10.2.16.3.6 DRBG_Initialize() ................................ 681

10.2.18 CryptSmac.c ............................................. 706

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

Part 4: Supporting Routines

Trusted Platform Module Library
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 ASN1InitializeMarshalContext() ................ 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
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>10.2.23.9</td>
<td>ASN1PushLength()</td>
</tr>
<tr>
<td>10.2.23.10</td>
<td>ASN1PushTagAndLength()</td>
</tr>
<tr>
<td>10.2.23.11</td>
<td>ASN1PushTaggedOctetString()</td>
</tr>
<tr>
<td>10.2.23.12</td>
<td>ASN1PushUINT()</td>
</tr>
<tr>
<td>10.2.23.13</td>
<td>ASN1PushInteger</td>
</tr>
<tr>
<td>10.2.23.14</td>
<td>ASN1PushOID()</td>
</tr>
<tr>
<td>10.2.24</td>
<td>X509_ECC.c</td>
</tr>
<tr>
<td>10.2.25</td>
<td>X509_RSA.c</td>
</tr>
<tr>
<td>10.2.26</td>
<td>X509_spt.c</td>
</tr>
<tr>
<td>10.2.27</td>
<td>AC_spt.c</td>
</tr>
</tbody>
</table>

Annex A (informative) Implementation Dependent

A.1 Introduction

A.2 TpmProfile.h

Annex B (informative) Library-Specific

B.1 Introduction

B.2 OpenSSL-Specific Files

B.2.2 Header Files

B.2.2.1 TpmToOsslHash.h

B.2.2.1.1 Introduction

B.2.2.1.2 Links to the OpenSSL HASH code

B.2.2.2 TpmToOsslMath.h

B.2.2.2.1 Introduction

B.2.2.2.2 Macros and Defines
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_PhasicPresenceAsserted() ......................................... 813
    C.10.3.2. plat_Signal_PhasicPresenceOn() ....................................... 813
    C.10.3.3. plat_Signal_PhasicPresenceOff() ....................................... 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 TPMCmds.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
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:

```c
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:

```c
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

```c
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:

```c
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 into 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:

```c
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:

```c
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 extracts 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.

```c
#include CC_Create // Command must be enabled
#endif _Create_FP_H_
#define _Create_FP_H_

Input structure definition

typedef struct {
    TPMI_DH_OBJECT          parentHandle;
    TPM2B_SENSITIVE_CREATE  inSensitive;
    TPM2B_PUBLIC            inPublic;
    TPM2B_DATA              outsideInfo;
    TPML_PCR_SELECTION      creationPCR;
} Create_In;

Output structure definition

typedef struct {
    TPM2B_PRIVATE           outPrivate;
    TPM2B_PUBLIC            outPublic;
    TPM2B_CREATION_DATA     creationData;
```

Part 4: Supporting Routines
Trusted Platform Module Library
4.5 Portability

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

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

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

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

a) allocate all structures as packed to a byte boundary (this may not be possible if the processor does not handle unaligned accesses); or

b) modify the code that manipulates bit fields that are not defined as being the alignment size of the system.

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

5.1 Introduction

The files in this 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

```c
#ifndef BASE_TYPES_H
#define BASE_TYPES_H

NULL definition

#ifndef NULL
#define NULL (0)
#endif

typedef uint8_t             UINT8;
typedef uint8_t             BYTE;
typedef int8_t              INT8;
typedef int                BOOL;
typedef uint16_t            UINT16;
typedef int16_t             INT16;
typedef uint32_t            UINT32;
typedef int32_t             INT32;
typedef uint64_t            UINT64;
typedef int64_t             INT64;
#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().

```c
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_TPMS_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

```c
#ifndef _COMMAND_CODE_ATTRIBUTES_
#include "CommandAttributes.h"
#endif

#define PAD_LIST 0

#define PAD_LIST 1
#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.

```c
const TPMA_CC s_ccAttr [] = {
  #if (PAD_LIST || CC_NV.UndefineSpaceSpecial)
    TPMA_CC_INITIALIZER(0x011f, 0, 1, 0, 2, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_EvictControl)
    TPMA_CC_INITIALIZER(0x0120, 0, 1, 0, 2, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_HierarchyControl)
    TPMA_CC_INITIALIZER(0x0121, 0, 1, 1, 0, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_NV.UndefineSpace)
    TPMA_CC_INITIALIZER(0x0122, 0, 1, 0, 2, 0, 0, 0),
  #endif
  #if (PAD_LIST )
    TPMA_CC_INITIALIZER(0x0123, 0, 0, 0, 0, 0, 0, 0),
  #endif
  #if (PAD_LIST || CCChangeEPS)
    TPMA_CC_INITIALIZER(0x0124, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_ChangePPS)
    TPMA_CC_INITIALIZER(0x0125, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_Clear)
    TPMA_CC_INITIALIZER(0x0126, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_ClearControl)
    TPMA_CC_INITIALIZER(0x0127, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_ClockSet)
    TPMA_CC_INITIALIZER(0x0128, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_HierarchyChangeAuth)
    TPMA_CC_INITIALIZER(0x0129, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_NV_DefineSpace)
    TPMA_CC_INITIALIZER(0x012a, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_PCR_Allocate)
    TPMA_CC_INITIALIZER(0x012b, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_PCR_SetAuthPolicy)
    TPMA_CC_INITIALIZER(0x012c, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_PP_Commands)
    TPMA_CC_INITIALIZER(0x012d, 0, 1, 0, 1, 0, 0, 0),
  #endif
  #if (PAD_LIST || CC_SetPrimaryPolicy)
    TPMA_CC_INITIALIZER(0x012e, 0, 1, 0, 1, 0, 0, 0),
  #endif
};
```
#if (PAD_LIST || CC_FieldUpgradeStart)
  TPMA_CC_INITIALIZER(0x012F, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_ClockRateAdjust)
  TPMA_CC_INITIALIZER(0x0130, 0, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_CreatePrimary)
  TPMA_CC_INITIALIZER(0x0131, 0, 0, 0, 1, 1, 0, 0),
#endif

#if (PAD_LIST || CC_NV_GlobalWriteLock)
  TPMA_CC_INITIALIZER(0x0132, 0, 1, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_GetCommandAuditDigest)
  TPMA_CC_INITIALIZER(0x0133, 0, 1, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_NV_Increment)
  TPMA_CC_INITIALIZER(0x0134, 0, 1, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_NV_SetBits)
  TPMA_CC_INITIALIZER(0x0135, 0, 1, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_NV_Extend)
  TPMA_CC_INITIALIZER(0x0136, 0, 1, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_NV_Write)
  TPMA_CC_INITIALIZER(0x0137, 0, 1, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_NV_WriteLock)
  TPMA_CC_INITIALIZER(0x0138, 0, 1, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_DictionaryAttackLockReset)
  TPMA_CC_INITIALIZER(0x0139, 0, 1, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_DictionaryAttackParameters)
  TPMA_CC_INITIALIZER(0x013A, 0, 1, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_NV_ChangeAuth)
  TPMA_CC_INITIALIZER(0x013B, 0, 1, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PCR_Event)
  TPMA_CC_INITIALIZER(0x013C, 0, 1, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PCR_Reset)
  TPMA_CC_INITIALIZER(0x013D, 0, 1, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_SequenceComplete)
  TPMA_CC_INITIALIZER(0x013E, 0, 0, 0, 1, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_SetAlgorithmSet)
  TPMA_CC_INITIALIZER(0x013F, 0, 1, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_SetCommandCodeAuditStatus)
  TPMA_CC_INITIALIZER(0x0140, 0, 1, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_FieldUpgradeData)
  TPMA_CC_INITIALIZER(0x0141, 0, 1, 0, 0, 0, 0, 0, 0),
#endif

#if (PAD_LIST || CC_IncrementalSelfTest)
  TPMA_CC_INITIALIZER(0x0142, 0, 1, 0, 0, 0, 0, 0, 0),
#endif

#if (PAD_LIST || CC_SelfTest)
  TPMA_CC_INITIALIZER(0x0143, 0, 1, 0, 0, 0, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Startup)
  TPMA_CC_INITIALIZER(0x0144, 0, 1, 0, 0, 0, 0, 0, 0),
#endif
#if (PAD_LIST || CC_Shutdown)
   TPMA_CC_INITIALIZER(0x0145, 0, 1, 0, 0, 0, 0, 0, 0),
#endif

#if (PAD_LIST || CC_StirRandom)
   TPMA_CC_INITIALIZER(0x0146, 0, 1, 0, 0, 0, 0, 0, 0),
#endif

#if (PAD_LIST || CC_ActivateCredential)
   TPMA_CC_INITIALIZER(0x0147, 0, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Certify)
   TPMA_CC_INITIALIZER(0x0148, 0, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PolicyNV)
   TPMA_CC_INITIALIZER(0x0149, 0, 0, 0, 0, 3, 0, 0, 0),
#endif

#if (PAD_LIST || CC_CertifyCreation)
   TPMA_CC_INITIALIZER(0x014A, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Duplicate)
   TPMA_CC_INITIALIZER(0x014B, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_GetTime)
   TPMA_CC_INITIALIZER(0x014C, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_GetSessionAuditDigest)
   TPMA_CC_INITIALIZER(0x014D, 0, 0, 0, 3, 0, 0, 0),
#endif

#if (PAD_LIST || CC_NV_Read)
   TPMA_CC_INITIALIZER(0x014E, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_NV_ReadLock)
   TPMA_CC_INITIALIZER(0x014F, 0, 1, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_ObjectChangeAuth)
   TPMA_CC_INITIALIZER(0x0150, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PolicySecret)
   TPMA_CC_INITIALIZER(0x0151, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Rewrap)
   TPMA_CC_INITIALIZER(0x0152, 0, 0, 0, 2, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Create)
   TPMA_CC_INITIALIZER(0x0153, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_ECDH_ZGen)
   TPMA_CC_INITIALIZER(0x0154, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || (CC_HMAC || CC_MAC))
   TPMA_CC_INITIALIZER(0x0155, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Import)
   TPMA_CC_INITIALIZER(0x0156, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Load)
   TPMA_CC_INITIALIZER(0x0157, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Quote)
   TPMA_CC_INITIALIZER(0x0158, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || (CC_RSA_Decrypt)
   TPMA_CC_INITIALIZER(0x0159, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST )
   TPMA_CC_INITIALIZER(0x015A, 0, 0, 0, 0, 0, 0, 0),
#endif
#if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
TPMA_CC_INITIALIZER(0x015B, 0, 0, 0, 1, 1, 0, 0),
#endif

#if (PAD_LIST || CC_SequenceUpdate)
TPMA_CC_INITIALIZER(0x015C, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_Sign)
TPMA_CC_INITIALIZER(0x015D, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_Unseal)
TPMA_CC_INITIALIZER(0x015E, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST)
TPMA_CC_INITIALIZER(0x015F, 0, 0, 0, 0, 0, 0, 0, 0),
#endif
#if (PAD_LIST || CC_PolicySigned)
TPMA_CC_INITIALIZER(0x0160, 0, 0, 0, 2, 0, 0, 0),
#endif
#if (PAD_LIST || CC_ContextLoad)
TPMA_CC_INITIALIZER(0x0161, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_ContextSave)
TPMA_CC_INITIALIZER(0x0162, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_ECDH_KeyGen)
TPMA_CC_INITIALIZER(0x0163, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_EncryptDecrypt)
TPMA_CC_INITIALIZER(0x0164, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_FlushContext)
TPMA_CC_INITIALIZER(0x0165, 0, 0, 0, 0, 0, 0, 0, 0),
#endif
#if (PAD_LIST)
TPMA_CC_INITIALIZER(0x0166, 0, 0, 0, 0, 0, 0, 0, 0),
#endif
#if (PAD_LIST || CC_LoadExternal)
TPMA_CC_INITIALIZER(0x0167, 0, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_MakeCredential)
TPMA_CC_INITIALIZER(0x0168, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_NV_ReadPublic)
TPMA_CC_INITIALIZER(0x0169, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_PolicyAuthorize)
TPMA_CC_INITIALIZER(0x016A, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_PolicyAuthValue)
TPMA_CC_INITIALIZER(0x016B, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_PolicyCommandCode)
TPMA_CC_INITIALIZER(0x016C, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_PolicyCounterTimer)
TPMA_CC_INITIALIZER(0x016D, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_PolicyCpHash)
TPMA_CC_INITIALIZER(0x016E, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_PolicyLocality)
TPMA_CC_INITIALIZER(0x016F, 0, 0, 0, 1, 0, 0),
#endif
#if (PAD_LIST || CC_PolicyNameHash)
TPMA_CC_INITIALIZER(0x0170, 0, 0, 0, 1, 0, 0),
#endif
255  #if (PAD_LIST || CC_PolicyOR)
256   TPMA_CC_INITIALIZER(0x0171, 0, 0, 0, 1, 0, 0, 0),
257  #endif
258  #if (PAD_LIST || CC_PolicyTicket)
259   TPMA_CC_INITIALIZER(0x0172, 0, 0, 0, 1, 0, 0, 0),
260  #endif
261  #if (PAD_LIST || CC_ReadPublic)
262   TPMA_CC_INITIALIZER(0x0173, 0, 0, 0, 1, 0, 0, 0),
263  #endif
264  #if (PAD_LIST || CC_RSA_Encrypt)
265   TPMA_CC_INITIALIZER(0x0174, 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, 2, 1, 0, 0),
272  #endif
273  #if (PAD_LIST || CC_VerifySignature)
274   TPMA_CC_INITIALIZER(0x0177, 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),
289  #endif
290  #if (PAD_LIST || CC_Hash)
291   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, 1, 0, 0, 0, 0),
299  #endif
300  #if (PAD_LIST || CC_PolicyRestart)
301   TPMA_CC_INITIALIZER(0x0180, 0, 0, 0, 1, 0, 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, 1, 0, 0, 0, 0),
311  #endif
312  #if (PAD_LIST || CC_NV_Certify)
313   TPMA_CC_INITIALIZER(0x0184, 0, 0, 0, 3, 0, 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, 1, 0, 0, 0),
320  #endif
#if (PAD_LIST || CC_PolicyPhysicalPresence)
TPMA_CC_INITIALIZER(0x0187, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PolicyDuplicationSelect)
TPMA_CC_INITIALIZER(0x0188, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PolicyGetDigest)
TPMA_CC_INITIALIZER(0x0189, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_TestParms)
TPMA_CC_INITIALIZER(0x018A, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PolicyPassword)
TPMA_CC_INITIALIZER(0x018C, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_ZGen_2Phase)
TPMA_CC_INITIALIZER(0x018D, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_EC_Ephemeral)
TPMA_CC_INITIALIZER(0x018E, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PolicyNvWritten)
TPMA_CC_INITIALIZER(0x018F, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PolicyTemplate)
TPMA_CC_INITIALIZER(0x0190, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_CreateLoaded)
TPMA_CC_INITIALIZER(0x0191, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_PolicyAuthorizeNV)
TPMA_CC_INITIALIZER(0x0192, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_EncryptDecrypt2)
TPMA_CC_INITIALIZER(0x0193, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_AC_GetCapability)
TPMA_CC_INITIALIZER(0x0194, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_AC_Send)
TPMA_CC_INITIALIZER(0x0195, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Policy_AC_SendSelect)
TPMA_CC_INITIALIZER(0x0196, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_CertifyX509)
TPMA_CC_INITIALIZER(0x0197, 0, 0, 0, 1, 0, 0, 0),
#endif

#if (PAD_LIST || CC_Vendor_TCG_Test)
TPMA_CC_INITIALIZER(0x0000, 0, 0, 0, 1, 0, 0, 0),
#endif

{0}

};

This is the command code attribute structure.

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

#include "ccdefs.h"

#define COMMAND_ATTRIBUTES

#define IS_IMPLEMENTED
#define HANDLE_1_USER
#define PP_COMMAND

#define PAD_LIST
#define CC_EVICTCONTROL
#define CC_HIERARCHYCONTROL
#define CC_NV_UNDEFINESPACE
#define CC_CHANGEEPS
#define CC_CHANGEPPS
#define CC_CLEAR
#define CC_CLEARCONTROL
#define CC_CLOCKSET
#define CC_HIERARCHYCHANGEAUTH
#define CC_NV_DEFINESPACE
#define CC_PCR_ALLOCATE
#define CC_PCR_SETAUTHPOLICY
#define CC_PP_COMMANDS
#define CC_SETPRIMARYPOLICY
#define CC_FIELDUPGRADESTART
#define CC_CLOCKRATEADJUST

#define (COMMAND_ATTRIBUTES)(CC_EvictControl
     * // 0x0120
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_HierarchyControl)
(COMMAND_ATTRIBUTES)(CC_HierarchyControl
     * // 0x0121
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_NV_UndefineSpace)
(COMMAND_ATTRIBUTES)(CC_NV_UndefineSpace
     * // 0x0122
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST )
(COMMAND_ATTRIBUTES)(0), // 0x0123
#endif
#if (PAD_LIST || CC_ChangeEPS)
(COMMAND_ATTRIBUTES)(CC_ChangeEPS
     * // 0x0124
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_ChangePPS)
(COMMAND_ATTRIBUTES)(CC_ChangePPS
     * // 0x0125
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_Clear)
(COMMAND_ATTRIBUTES)(CC_Clear
     * // 0x0126
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_ClearControl)
(COMMAND_ATTRIBUTES)(CC_ClearControl
     * // 0x0127
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_ClockSet)
(COMMAND_ATTRIBUTES)(CC_ClockSet
     * // 0x0128
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_HierarchyChangeAuth)
(COMMAND_ATTRIBUTES)(CC_HierarchyChangeAuth
     * // 0x0129
     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_NV_DefineSpace)
(COMMAND_ATTRIBUTES)(CC_NV_DefineSpace
     * // 0x012A
     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_PCR_Allocate)
(COMMAND_ATTRIBUTES)(CC_PCR_Allocate
     * // 0x012B
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_PCR_SetAuthPolicy)
(COMMAND_ATTRIBUTES)(CC_PCR_SetAuthPolicy
     * // 0x012C
     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_PP_Commands)
(COMMAND_ATTRIBUTES)(CC_PP_Commands
     * // 0x012D
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_REQUIRED)),
#endif
#if (PAD_LIST || CC_SetPrimaryPolicy)
(COMMAND_ATTRIBUTES)(CC_SetPrimaryPolicy
     * // 0x012E
     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_FieldUpgradeStart)
(COMMAND_ATTRIBUTES)(CC_FieldUpgradeStart
     * // 0x012F
     (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_ClockRateAdjust)
(COMMAND_ATTRIBUTES)(CC_ClockRateAdjust
     * // 0x0130
     (IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif
#if (PAD_LIST || CC_CreatePrimary)
(COMMAND_ATTRIBUTES)(CC_CreatePrimary * // 0x0131
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+PP_COMMAND+ENCRYPT_2+R_HANDLE)),
#endif

#if (PAD_LIST || CC_NV_GlobalWriteLock)
(COMMAND_ATTRIBUTES)(CC_NV_GlobalWriteLock * // 0x0132
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif

#if (PAD_LIST || CC_GetCommandAuditDigest)
(COMMAND_ATTRIBUTES)(CC_GetCommandAuditDigest * // 0x0133
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_NV_Increment)
(COMMAND_ATTRIBUTES)(CC_NV_Increment                * // 0x0134
(IS_IMPLEMENTED+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_NV_SetBits)
(COMMAND_ATTRIBUTES)(CC_NV_SetBits                  * // 0x0135
(IS_IMPLEMENTED+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_NV_Extend)
(COMMAND_ATTRIBUTES)(CC_NV_Extend                   * // 0x0136
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_NV_Write)
(COMMAND_ATTRIBUTES)(CC_NV_Write                    * // 0x0137
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_NV_WriteLock)
(COMMAND_ATTRIBUTES)(CC_NV_WriteLock                * // 0x0138
(ISIMPLEMENTED+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_DictionaryAttackLockReset)
(COMMAND_ATTRIBUTES)(CC_DictionaryAttackLockReset   * // 0x0139
(IS_IMPLEMENTED+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_DictionaryAttackParameters)
(COMMAND_ATTRIBUTES)(CC_DictionaryAttackParameters  * // 0x013A
(IS_IMPLEMENTED+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_NV_ChangeAuth)
(COMMAND_ATTRIBUTES)(CC_NV_ChangeAuth               * // 0x013B
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN)),
#endif

#if (PAD_LIST || CC_PCR_Event)
(COMMAND_ATTRIBUTES)(CC_PCR_Event                   * // 0x013C
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_PCR_Reset)
(COMMAND_ATTRIBUTES)(CC_PCR_Reset                   * // 0x013D
(IS_IMPLEMENTED+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_SequenceComplete)
(COMMAND_ATTRIBUTES)(CC_SequenceComplete            * // 0x013E
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_SetAlgorithmSet)
(COMMAND_ATTRIBUTES)(CC_SetAlgorithmSet             * // 0x013F
(IS_IMPLEMENTED+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_SetCommandCodeAuditStatus)
(COMMAND_ATTRIBUTES)(CC_SetCommandCodeAuditStatus   * // 0x0140
(IS_IMPLEMENTED+HANDLE_1_USER+PP_COMMAND)),
#endif

#if (PAD_LIST || CC_FieldUpgradeData)
(COMMAND_ATTRIBUTES)(CC_FieldUpgradeData            * // 0x0141

(IS_IMPLEMENTED+DECRYPT_2)),
#endif
#if (PAD_LIST || CC_IncrementalSelfTest)
 (COMMAND_ATTRIBUTES)(CC_IncrementalSelfTest
 * // 0x0142
 (IS_IMPLEMENTED)),
#endif
#if (PAD_LIST || CC_SelfTest)
 (COMMAND_ATTRIBUTES)(CC_SelfTest
 * // 0x0143
 (IS_IMPLEMENTED)),
#endif
#if (PAD_LIST || CC_Startup)
 (COMMAND_ATTRIBUTES)(CC_Startup
 * // 0x0144
 (IS_IMPLEMENTED+NO_SESSIONS)),
#endif
#if (PAD_LIST || CC_Shutdown)
 (COMMAND_ATTRIBUTES)(CC_Shutdown
 * // 0x0145
 (IS_IMPLEMENTED)),
#endif
#if (PAD_LIST || CC_StirRandom)
 (COMMAND_ATTRIBUTES)(CC_StirRandom
 * // 0x0146
 (IS_IMPLEMENTED+DECRYPT_2)),
#endif
#if (PAD_LIST || CC_ActivateCredential)
 (COMMAND_ATTRIBUTES)(CC_ActivateCredential
 * // 0x0147
 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
#endif
#if (PAD_LIST || CC_Certify)
 (COMMAND_ATTRIBUTES)(CC_Certify
 * // 0x0148
 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
#endif
#if (PAD_LIST || CC_PolicyNV)
 (COMMAND_ATTRIBUTES)(CC_PolicyNV
 * // 0x0149
 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL)),
#endif
#if (PAD_LIST || CC_CertifyCreation)
 (COMMAND_ATTRIBUTES)(CC_CertifyCreation
 * // 0x014A
 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
#endif
#if (PAD_LIST || CC_Duplicate)
 (COMMAND_ATTRIBUTES)(CC_Duplicate
 * // 0x014B
 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+ENCRYPT_2)),
#endif
#if (PAD_LIST || CC_GetTime)
 (COMMAND_ATTRIBUTES)(CC_GetTime
 * // 0x014C
 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
#endif
#if (PAD_LIST || CC_GetSessionAuditDigest)
 (COMMAND_ATTRIBUTES)(CC_GetSessionAuditDigest
 * // 0x014D
 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
#endif
#if (PAD_LIST || CC_NV_Read)
 (COMMAND_ATTRIBUTES)(CC_NV_Read
 * // 0x014E
 (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
#endif
#if (PAD_LIST || CC_NV_ReadLock)
 (COMMAND_ATTRIBUTES)(CC_NV_ReadLock
 * // 0x014F
 (IS_IMPLEMENTED+HANDLE_1_USER)),
#endif
#if (PAD_LIST || CC_ObjectChangeAuth)
 (COMMAND_ATTRIBUTES)(CC_ObjectChangeAuth
 * // 0x0150
 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+ENCRYPT_2)),
#endif
#if (PAD_LIST || CC_PolicySecret)
 (COMMAND_ATTRIBUTES)(CC_PolicySecret
 * // 0x0151
 (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ALLOW_TRIAL+ENCRYPT_2)),
#endif
#if (PAD_LIST || CC_Rewrap)
(COMMAND_ATTRIBUTES)(CC_Rewrap
    * // 0x0152
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_Create)
(COMMAND_ATTRIBUTES)(CC_Create
    * // 0x0153
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_ECDH_ZGen)
(COMMAND_ATTRIBUTES)(CC_ECDH_ZGen
    * // 0x0154
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || (CC_HMAC || CC_MAC))
(COMMAND_ATTRIBUTES)((CC_HMAC || CC_MAC)
    * // 0x0155
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_Import)
(COMMAND_ATTRIBUTES)(CC_Import
    * // 0x0156
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_Load)
(COMMAND_ATTRIBUTES)(CC_Load
    * // 0x0157
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2+R_HANDLE)),
#endif

#if (PAD_LIST || CC_Quote)
(COMMAND_ATTRIBUTES)(CC_Quote
    * // 0x0158
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_RSA_Decrypt)
(COMMAND_ATTRIBUTES)(CC_RSA_Decrypt
    * // 0x0159
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST )
(COMMAND_ATTRIBUTES)(0),
#endif

#if (PAD_LIST || (CC_HMAC_Start || CC_MAC_Start))
(COMMAND_ATTRIBUTES)((CC_HMAC_Start || CC_MAC_Start)
    * // 0x015B
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER+R_HANDLE)),
#endif

#if (PAD_LIST || CC_SequenceUpdate)
(COMMAND_ATTRIBUTES)(CC_SequenceUpdate
    * // 0x015C
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_Sign)
(COMMAND_ATTRIBUTES)(CC_Sign
    * // 0x015D
(IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_Unseal)
(COMMAND_ATTRIBUTES)(CC_Unseal
    * // 0x015E
(IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST )
(COMMAND_ATTRIBUTES)(0),
#endif

#if (PAD_LIST || CC_PolicySigned)
(COMMAND_ATTRIBUTES)(CC_PolicySigned
    * // 0x0160
(IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_ContextLoad)
(COMMAND_ATTRIBUTES)(CC_ContextLoad
    * // 0x0161
(ISIMPLEMENTED+NO_SESSIONS+R_HANDLE)),
#endif

#if (PAD_LIST || CC_ContextSave)
(COMMAND_ATTRIBUTES)(CC_ContextSave
    * // 0x0162
(ISIMPLEMENTED+NO_SESSIONS)),
#endif
#if (PAD_LIST || CC_ECDH_KeyGen)
   (COMMAND_ATTRIBUTES)(CC_ECDH_KeyGen  * // 0x0163
    (IS_IMPLEMENTED+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_EncryptDecrypt)
   (COMMAND_ATTRIBUTES)(CC_EncryptDecrypt  * // 0x0164
    (IS_IMPLEMENTED+HANDLE_1_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_FlushContext)
   (COMMAND_ATTRIBUTES)(CC_FlushContext  * // 0x0165
    (IS_IMPLEMENTED+NO_SESSIONS)),
#endif

#if (PAD_LIST )
   (COMMAND_ATTRIBUTES)(0), // 0x0166
#endif

#if (PAD_LIST || CC_LoadExternal)
   (COMMAND_ATTRIBUTES)(CC_LoadExternal  * // 0x0167
    (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
#endif

#if (PAD_LIST || CC_MakeCredential)
   (COMMAND_ATTRIBUTES)(CC_MakeCredential  * // 0x0168
    (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_NV_ReadPublic)
   (COMMAND_ATTRIBUTES)(CC_NV_ReadPublic  * // 0x0169
    (IS_IMPLEMENTED+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_PolicyAuthorize)
   (COMMAND_ATTRIBUTES)(CC_PolicyAuthorize  * // 0x016A
    (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_PolicyAuthValue)
   (COMMAND_ATTRIBUTES)(CC_PolicyAuthValue  * // 0x016B
    (IS_IMPLEMENTED+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_PolicyCommandCode)
   (COMMAND_ATTRIBUTES)(CC_PolicyCommandCode  * // 0x016C
    (IS_IMPLEMENTED+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_PolicyCpHash)
   (COMMAND_ATTRIBUTES)(CC_PolicyCpHash  * // 0x016D
    (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_PolicyLocality)
   (COMMAND_ATTRIBUTES)(CC_PolicyLocality  * // 0x016F
    (IS_IMPLEMENTED+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_PolicyNameHash)
   (COMMAND_ATTRIBUTES)(CC_PolicyNameHash  * // 0x0170
    (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_PolicyOR)
   (COMMAND_ATTRIBUTES)(CC_PolicyOR  * // 0x0171
    (IS_IMPLEMENTED+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_PolicyTicket)
   (COMMAND_ATTRIBUTES)(CC_PolicyTicket  * // 0x0172
    (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_ReadPublic)
   (COMMAND_ATTRIBUTES)(CC_ReadPublic  * // 0x0173
    (IS_IMPLEMENTED+ENCRYPT_2)),
#endif
#ifdef

#if (PAD_LIST || CC_RSA_Encrypt)
  (COMMAND_ATTRIBUTES)(CC_RSA_Encrypt * // 0x0174
   (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
#endif

#if (PAD_LIST )
  (COMMAND_ATTRIBUTES)(0), // 0x0175
#endif

#if (PAD_LIST || CC_StartAuthSession)
  (COMMAND_ATTRIBUTES)(CC_StartAuthSession * // 0x0176
   (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2+R_HANDLE)),
#endif

#if (PAD_LIST || CC_VerifySignature)
  (COMMAND_ATTRIBUTES)(CC_VerifySignature * // 0x0177
   (IS_IMPLEMENTED+DECRYPT_2)),
#endif

#if (PAD_LIST || CC_ECC_Parameters)
  (COMMAND_ATTRIBUTES)(CC_ECC_Parameters * // 0x0178
   (IS_IMPLEMENTED)),
#endif

#if (PAD_LIST || CC_FirmwareRead)
  (COMMAND_ATTRIBUTES)(CC_FirmwareRead * // 0x0179
   (IS_IMPLEMENTED+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_GetCapability)
  (COMMAND_ATTRIBUTES)(CC_GetCapability * // 0x017A
   (IS_IMPLEMENTED)),
#endif

#if (PAD_LIST || CC_GetRandom)
  (COMMAND_ATTRIBUTES)(CC_GetRandom * // 0x017B
   (IS_IMPLEMENTED+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_GetTestResult)
  (COMMAND_ATTRIBUTES)(CC_GetTestResult * // 0x017C
   (IS_IMPLEMENTED+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_Hash)
  (COMMAND_ATTRIBUTES)(CC_Hash * // 0x017D
   (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_PCR_Read)
  (COMMAND_ATTRIBUTES)(CC_PCR_Read * // 0x017E
   (IS_IMPLEMENTED)),
#endif

#if (PAD_LIST || CC_PolicyPCR)
  (COMMAND_ATTRIBUTES)(CC_PolicyPCR * // 0x017F
   (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_PolicyRestart)
  (COMMAND_ATTRIBUTES)(CC_PolicyRestart * // 0x0180
   (IS_IMPLEMENTED+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_ReadClock)
  (COMMAND_ATTRIBUTES)(CC_ReadClock * // 0x0181
   (IS_IMPLEMENTED)),
#endif

#if (PAD_LIST || CC_PCR_Extend)
  (COMMAND_ATTRIBUTES)(CC_PCR_Extend * // 0x0182
   (IS_IMPLEMENTED+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_PCR_SetAuthValue)
  (COMMAND_ATTRIBUTES)(CC_PCR_SetAuthValue * // 0x0183
   (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_USER)),
#endif

#if (PAD_LIST || CC_NV_Certify)
  (COMMAND_ATTRIBUTES)(CC_NV_Certify * // 0x0184
   (IS_IMPLEMENTED+ENCRYPT_2)),
#endif

Page 20
(IS_IMPLEMENTED+DECRIPT_2+HANDLE_1_USER+HANDLE_2_USER+ENCRYPT_2)),
#endif

#endif
    
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#endif
#if (PAD_LIST || CC_AC_Send)
  (COMMAND_ATTRIBUTES)(CC_AC_Send * // 0x0195
   (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_DUP+HANDLE_2_USER)),
#endif

#if (PAD_LIST || CC_Policy_AC_SendSelect)
  (COMMAND_ATTRIBUTES)(CC_Policy_AC_SendSelect * // 0x0196
   (IS_IMPLEMENTED+DECRYPT_2+ALLOW_TRIAL)),
#endif

#if (PAD_LIST || CC_CertifyX509)
  (COMMAND_ATTRIBUTES)(CC_CertifyX509 * // 0x0197
   (IS_IMPLEMENTED+DECRYPT_2+HANDLE_1_ADMIN+HANDLE_2_USER+ENCRYPT_2)),
#endif

#if (PAD_LIST || CC_Vendor_TCG_Test)
  (COMMAND_ATTRIBUTES)(CC_Vendor_TCG_Test * // 0x0000
   (IS_IMPLEMENTED+DECRYPT_2+ENCRYPT_2)),
#endif

0

};
#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.

```c
#ifndef COMMAND_ATTRIBUTES_H
#define COMMAND_ATTRIBUTES_H

typedef UINT16 COMMAND_ATTRIBUTES;

#define NOT_IMPLEMENTED (COMMAND_ATTRIBUTES)(0)
#define ENCRYPT_2 ((COMMAND_ATTRIBUTES)1 << 0)
#define ENCRYPT_4 ((COMMAND_ATTRIBUTES)1 << 1)
#define DECRYPT_2 ((COMMAND_ATTRIBUTES)1 << 2)
#define DECRYPT_4 ((COMMAND_ATTRIBUTES)1 << 3)
#define HANDLE_1_USER ((COMMAND_ATTRIBUTES)1 << 4)
#define HANDLE_1_ADMIN ((COMMAND_ATTRIBUTES)1 << 5)
#define HANDLE_1_DUP ((COMMAND_ATTRIBUTES)1 << 6)
#define HANDLE_2_USER ((COMMAND_ATTRIBUTES)1 << 7)
#define PP_COMMAND ((COMMAND_ATTRIBUTES)1 << 8)
#define IS_IMPLEMENTED ((COMMAND_ATTRIBUTES)1 << 9)
#define NO_SESSIONS ((COMMAND_ATTRIBUTES)1 << 10)
#define NV_COMMAND ((COMMAND_ATTRIBUTES)1 << 11)
#define PP_REQUIRED ((COMMAND_ATTRIBUTES)1 << 12)
#define R_HANDLE ((COMMAND_ATTRIBUTES)1 << 13)
#define ALLOW_TRIAL ((COMMAND_ATTRIBUTES)1 << 14)

#endif // COMMAND_ATTRIBUTES_H
```
5.6 CommandDispatchData.h

This file should only be included by CommandCodeAttributes.c

```c
#ifdef _COMMAND_TABLE_DISPATCH_

Define the stop value

#define END_OF_LIST 0xff
#define ADD_FLAG 0x80

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

#define UNMARSHAL_DISPATCH(name) (UNMARSHAL_t)name##_Unmarshal
#define MARSHAL_DISPATCH(name) (MARSHAL_t)##name##_Marshal
#define _UNMARSHAL_T_ UNMARSHAL_t
#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

const _UNMARSHAL_T_ UnmarshalArray[] = {
  #define TPMI_DH_CONTEXT_H_UNMARSHAL 0
  #define TPMI_RH_AC_H_UNMARSHAL (TPMI_DH_CONTEXT_H_UNMARSHAL + 1)
  #define TPMI_RH_CLEAR_H_UNMARSHAL (TPMI_RH_AC_H_UNMARSHAL + 1)
  #define TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL (TPMI_RH_CLEAR_H_UNMARSHAL + 1)
  #define TPMI_RH_LOCKOUT_H_UNMARSHAL (TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL + 1)
  #define TPMI_RH_NV_AUTH_H_UNMARSHAL (TPMI_RH_LOCKOUT_H_UNMARSHAL + 1)
  #define TPMI_RH_NV_INDEX_H_UNMARSHAL (TPMI_RH_NV_AUTH_H_UNMARSHAL + 1)
  #define TPMI_RH_PLATFORM_H_UNMARSHAL (TPMI_RH_NV_INDEX_H_UNMARSHAL + 1)
  #define TPMI_RH_PROVISION_H_UNMARSHAL (TPMI_RH_PLATFORM_H_UNMARSHAL + 1)
  #define TPMI_SH_HMAC_H_UNMARSHAL (TPMI_RH_PROVISION_H_UNMARSHAL + 1)
  #define TPMI_SH_POLICY_H_UNMARSHAL (TPMI_SH_HMAC_H_UNMARSHAL + 1)
  #define TPMI_DH_ENTITY_H_UNMARSHAL (TPMI_SH_POLICY_H_UNMARSHAL + 1)
  #define TPMI_DH_OBJECT_H_UNMARSHAL (TPMI_DH_ENTITY_H_UNMARSHAL + 1)
  #define TPMI_DH_PARENT_H_UNMARSHAL (TPMI_DH_OBJECT_H_UNMARSHAL + 1)
  #define TPMI_DH_PCR_H_UNMARSHAL (TPMI_DH_PARENT_H_UNMARSHAL + 1)
  #define TPMI_RH_ENDORSEMENT_H_UNMARSHAL (TPMI_DH_PCR_H_UNMARSHAL + 1)
  #define TPMI_RH_HIERARCHY_H_UNMARSHAL (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)
  // HANDLE_FIRST_FLAG_TYPE is the first handle that needs a flag when called.
  #define HANDLE_FIRST_FLAG_TYPE (TPMI_SH_POLICY_H_UNMARSHAL + 1)
  #define TPMI_DH_ENTITY_H_UNMARSHAL (TPMI_SH_POLICY_H_UNMARSHAL + 1)
  #define TPMI_DH_OBJECT_H_UNMARSHAL (TPMI_DH_ENTITY_H_UNMARSHAL + 1)
  #define TPMI_DH_PARENT_H_UNMARSHAL (TPMI_DH_OBJECT_H_UNMARSHAL + 1)
  #define TPMI_DH_PCR_H_UNMARSHAL (TPMI_DH_PARENT_H_UNMARSHAL + 1)
  #define TPMI_RH_ENDORSEMENT_H_UNMARSHAL (TPMI_DH_PCR_H_UNMARSHAL + 1)
  // PARAMETER_FIRST_TYPE marks the end of the handle list.
  #define PARAMETER_FIRST_TYPE (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)
  #define TPM2B_DATA_P_UNMARSHAL (TPMI_RH_ENDORSEMENT_H_UNMARSHAL + 1)

#endif
```
UNMARSHAL_DISPATCH(TPM2B_DATA),
#define TPM2B_DIGEST_P_UNMARSHAL (TPM2B_DATA_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_DIGEST),
#define TPM2B_ECC_PARAMETER_P_UNMARSHAL (TPM2B_DIGEST_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_ECC_PARAMETER),
#define TPM2B_ECC_POINT_P_UNMARSHAL (TPM2B_ECC_PARAMETER_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_ECC_POINT),
#define TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL (TPM2B_ECC_POINT_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
#define TPM2B_EVENT_P_UNMARSHAL (TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_EVENT),
#define TPM2B_ID_OBJECT_P_UNMARSHAL (TPM2B_EVENT_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_ID_OBJECT),
#define TPM2B_MAX_BUFFER_P_UNMARSHAL (TPM2B_ID_OBJECT_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
#define TPM2B_MAX_NV_BUFFER_P_UNMARSHAL (TPM2B_MAX_BUFFER_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
#define TPM2B_NAME_P_UNMARSHAL (TPM2B_MAX_NV_BUFFER_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_NAME),
#define TPM2B_NV_PUBLIC_P_UNMARSHAL (TPM2B_NAME_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
#define TPM2B_PRIVATE_P_UNMARSHAL (TPM2B_NV_PUBLIC_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_PRIVATE),
#define TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL (TPM2B_PRIVATE_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
#define TPM2B_SENSITIVE_P_UNMARSHAL (TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_SENSITIVE),
#define TPM2B_SENSITIVE_CREATE_P_UNMARSHAL (TPM2B_SENSITIVE_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_CREATE),
#define TPM2B_SENSITIVE_DATA_P_UNMARSHAL (TPM2B_SENSITIVE_CREATE_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
#define TPM2B_TEMPLATE_P_UNMARSHAL (TPM2B_SENSITIVE_DATA_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_TEMPLATE),
#define TPM2B_TIMEOUT_P_UNMARSHAL (TPM2B_TEMPLATE_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPM2B_TIMEOUT),
#define TPMI_DH_CONTEXT_P_UNMARSHAL (TPM2B_TIMEOUT_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_DH_CONTEXT),
#define TPMI_DH_PERSISTENT_P_UNMARSHAL (TPMI_DH_CONTEXT_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_DH_PERSISTENT),
#define TPMI_ECC_CURVE_P_UNMARSHAL (TPMI_DH_PERSISTENT_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_ECC_CURVE),
#define TPMI_YES_NO_P_UNMARSHAL (TPMI_ECC_CURVE_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_YES_NO),
#define TPMI_ALG_P_UNMARSHAL (TPMI_YES_NO_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_ALG),
#define TPMI_CC_P_UNMARSHAL (TPMI_ALG_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_CC),
#define TPMI_DIGEST_P_UNMARSHAL (TPMI_CC_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_DIGEST),
#define TPMI_DIGEST_VALUES_P_UNMARSHAL (TPMI_DIGEST_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_DIGEST_VALUES),
#define TPMI_PCR_SELECTION_P_UNMARSHAL (TPMI_DIGEST_VALUES_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_PCR_SELECTION),
#define TPMI_CONTEXT_P_UNMARSHAL (TPMI_PCR_SELECTION_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_CONTEXT),
#define TPMI_PUBLIC_PARMS_P_UNMARSHAL (TPMI_CONTEXT_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_PUBLIC_PARMS),
#define TPMI_TK_AUTH_P_UNMARSHAL (TPMI_PUBLIC_PARMS_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_TK_AUTH),
#define TPMI_TK_CREATION_P_UNMARSHAL (TPMI_TK_AUTH_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_TK_CREATION),
#define TPMI_TK_HASHCHECK_P_UNMARSHAL (TPMI_TK_CREATION_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMI_TK_HASHCHECK),
#define TPMI_TK_VERIFIED_P_UNMARSHAL (TPMI_TK_HASHCHECK_P_UNMARSHAL + 1)
UNMARSHAL_DISPATCH(TPMT_TK_VERIFIED),
#define TPM_AT_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPM_AT),
#define TPM_CAP_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPM_CAP),
#define TPM_CLOCK_ADJUST_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPM_CLOCK_ADJUST),
#define TPM_EO_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPM_EO),
#define TPM_SE_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPM_SE),
#define TPM_SU_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPM_SU),
#define UINT16_P_UNMARSHAL
UNMARSHAL_DISPATCH(UINT16),
#define UINT32_P_UNMARSHAL
UNMARSHAL_DISPATCH(UINT32),
#define UINT64_P_UNMARSHAL
UNMARSHAL_DISPATCH(UINT64),
#define UINT8_P_UNMARSHAL
UNMARSHAL_DISPATCH(UINT8),
// PARAMETER_FIRST_FLAG_TYPE is the first parameter to need a flag.
#define PARAMETER_FIRST_FLAG_TYPE
UNMARSHAL_DISPATCH(TPM2B_PUBLIC),
#define TPM2B_PUBLIC_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPM2B_PUBLIC),
#define TPMI_ALG_CIPHER_MODE_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMI_ALG_CIPHER_MODE),
#define TPMI_ALG_HASH_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMI_ALG_HASH),
#define TPMI_ALG_MAC_SCHEME_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMI_ALG_MAC_SCHEME),
#define TPMI_DH_PCR_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMI_DH_PCR),
#define TPMI_ECC_KEY_EXCHANGE_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMI_ECC_KEY_EXCHANGE),
#define TPMI_RH_ENABLES_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMI_RH_ENABLES),
#define TPMI_RH_HIERARCHY_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMI_RH_HIERARCHY),
#define TPMT_RSA_DECRYPT_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMT_RSA_DECRYPT),
#define TPMT_SIGNATURE_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMT_SIGNATURE),
#define TPMT_SIG_SCHEME_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMT_SIG_SCHEME),
#define TPMT_SYM_DEF_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMT_SYM_DEF),
#define TPMT_SYM_DEF_OBJECT_P_UNMARSHAL
UNMARSHAL_DISPATCH(TPMT_SYM_DEF_OBJECT)
};

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.

const _MARSHAL_T_ MarshalArray[] = {
#define UINT32_H_MARSHAL
0
MASHAL_DISPATCH(UINT32),
// RESPONSE_PARAMETER_FIRST_TYPE marks the end of the response handles.
#define RESPONSE_PARAMETER_FIRST_TYPE
MASHAL_DISPATCH(TPM2B_ATTEST),
#define TPM2B_ATTEST_P_MARSHAL
MASHAL_DISPATCH(TPM2B_ATTEST),
#define TPM2B_CREATION_DATA_P_MARSHAL
MASHAL_DISPATCH(TPM2B_CREATION_DATA)
175 #define TPM2B_DATA_P_MARSHAL (TPM2B_CREATION_DATA_P_MARSHAL + 1)
176 MARSHAL_DISPATCH(TPM2B_DATA),
177 #define TPM2B_DIGEST_P_MARSHAL (TPM2B_DATA_P_MARSHAL + 1)
178 MARSHAL_DISPATCH(TPM2B_DIGEST),
179 #define TPM2B_ECC_POINT_P_MARSHAL (TPM2B_DIGEST_P_MARSHAL + 1)
180 MARSHAL_DISPATCH(TPM2B_ECC_POINT),
181 #define TPM2B_ENCRYPTED_SECRET_P_MARSHAL (TPM2B_ECC_POINT_P_MARSHAL + 1)
182 MARSHAL_DISPATCH(TPM2B_ENCRYPTED_SECRET),
183 #define TPM2B_ID_OBJECT_P_MARSHAL (TPM2B_ENCRYPTED_SECRET_P_MARSHAL + 1)
184 MARSHAL_DISPATCH(TPM2B_ID_OBJECT),
185 #define TPM2B_IV_P_MARSHAL (TPM2B_ID_OBJECT_P_MARSHAL + 1)
186 MARSHAL_DISPATCH(TPM2B_IV),
187 #define TPM2B_MAX_BUFFER_P_MARSHAL (TPM2B_IV_P_MARSHAL + 1)
188 MARSHAL_DISPATCH(TPM2B_MAX_BUFFER),
189 #define TPM2B_MAX_NV_BUFFER_P_MARSHAL (TPM2B_MAX_BUFFER_P_MARSHAL + 1)
190 MARSHAL_DISPATCH(TPM2B_MAX_NV_BUFFER),
191 #define TPM2B_NAME_P_MARSHAL (TPM2B_MAX_NV_BUFFER_P_MARSHAL + 1)
192 MARSHAL_DISPATCH(TPM2B_NAME),
193 #define TPM2B_NV_PUBLIC_P_MARSHAL (TPM2B_NAME_P_MARSHAL + 1)
194 MARSHAL_DISPATCH(TPM2B_NV_PUBLIC),
195 #define TPM2B_PRIVATE_P_MARSHAL (TPM2B_NV_PUBLIC_P_MARSHAL + 1)
196 MARSHAL_DISPATCH(TPM2B_PRIVATE),
197 #define TPM2B_PUBLIC_P_MARSHAL (TPM2B_PRIVATE_P_MARSHAL + 1)
198 MARSHAL_DISPATCH(TPM2B_PUBLIC),
199 #define TPM2B_PUBLIC_KEY_RSA_P_MARSHAL (TPM2B_PUBLIC_P_MARSHAL + 1)
200 MARSHAL_DISPATCH(TPM2B_PUBLIC_KEY_RSA),
201 #define TPM2B_SENSITIVE_DATA_P_MARSHAL (TPM2B_PUBLIC_KEY_RSA_P_MARSHAL + 1)
202 MARSHAL_DISPATCH(TPM2B_SENSITIVE_DATA),
203 #define TPM2B_TIMEOUT_P_MARSHAL (TPM2B_SENSITIVE_DATA_P_MARSHAL + 1)
204 MARSHAL_DISPATCH(TPM2B_TIMEOUT),
205 #define UINT8_P_MARSHAL (TPM2B_TIMEOUT_P_MARSHAL + 1)
206 MARSHAL_DISPATCH(UINT8),
207 #define TPML_AC_CAPABILITIES_P_MARSHAL (UINT8_P_MARSHAL + 1)
208 MARSHAL_DISPATCH(TPML_AC_CAPABILITIES),
209 #define TPML_ALG_P_MARSHAL (TPML_AC_CAPABILITIES_P_MARSHAL + 1)
210 MARSHAL_DISPATCH(TPML_ALG),
211 #define TPML_DIGEST_P_MARSHAL (TPML_ALG_P_MARSHAL + 1)
212 MARSHAL_DISPATCH(TPML_DIGEST),
213 #define TPML_DIGEST_VALUES_P_MARSHAL (TPML_DIGEST_P_MARSHAL + 1)
214 MARSHAL_DISPATCH(TPML_DIGEST_VALUES),
215 #define TPML_PCR_SELECTION_P_MARSHAL (TPML_DIGEST_VALUES_P_MARSHAL + 1)
216 MARSHAL_DISPATCH(TPML_PCR_SELECTION),
217 #define TPMS_AC_OUTPUT_P_MARSHAL (TPML_PCR_SELECTION_P_MARSHAL + 1)
218 MARSHAL_DISPATCH(TPMS_AC_OUTPUT),
219 #define TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL (TPMS_AC_OUTPUT_P_MARSHAL + 1)
220 MARSHAL_DISPATCH(TPMS_ALGORITHM_DETAIL_ECC),
221 #define TPMS_CAPABILITY_DATA_P_MARSHAL (TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL + 1)
222 MARSHAL_DISPATCH(TPMS_CAPABILITY_DATA),
223 #define TPMS_CONTEXT_P_MARSHAL (TPMS_CAPABILITY_DATA_P_MARSHAL + 1)
224 MARSHAL_DISPATCH(TPMS_CONTEXT),
225 #define TPMS_TIME_INFO_P_MARSHAL (TPMS_CONTEXT_P_MARSHAL + 1)
226 MARSHAL_DISPATCH(TPMS_TIME_INFO),
227 #define TPM_TK_AUTH_P_MARSHAL (TPMS_TIME_INFO_P_MARSHAL + 1)
228 MARSHAL_DISPATCH(TPM_TK_AUTH),
229 #define TPM_TK_CREATION_P_MARSHAL (TPM_TK_AUTH_P_MARSHAL + 1)
230 MARSHAL_DISPATCH(TPM_TK_CREATION),
231 #define TPM_TK_HASHCHECK_P_MARSHAL (TPM_TK_CREATION_P_MARSHAL + 1)
232 MARSHAL_DISPATCH(TPM_TK_HASHCHECK),
233 #define TPM_TK_VERIFIED_P_MARSHAL (TPM_TK_HASHCHECK_P_MARSHAL + 1)
234 MARSHAL_DISPATCH(TPM_TK_VERIFIED),
#define UINT32_P_MARSHAL                  (TPMT_TK_VERIFIED_P_MARSHAL + 1)
#define UINT16_P_MARSHAL                   (UINT32_P_MARSHAL + 1)

// RESPONSE_PARAMETER_LAST_TYPE is the end of the response parameter list.
#define RESPONSE_PARAMETER_LAST_TYPE      (UINT16_P_MARSHAL)

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

#define INT32_P_UNMARSHAL                  UINT32_P_UNMARSHAL
#define TPM2B_AUTH_P_UNMARSHAL              TPM2B_DIGEST_P_UNMARSHAL
#define TPM2B_NONCE_P_UNMARSHAL             TPM2B_DIGEST_P_UNMARSHAL
#define TPM2B_OPERAND_P_UNMARSHAL           TPM2B_DIGEST_P_UNMARSHAL
#define TPM_A_LOCALITY_P_UNMARSHAL          UINT8_P_UNMARSHAL
#define TPM_CC_P_UNMARSHAL                  UINT32_P_UNMARSHAL
#define TPMI_DH_CONTEXT_H_MARSHAL           UINT32_H_MARSHAL
#define TPMI_DH_OBJECT_H_MARSHAL            UINT32_H_MARSHAL
#define TPMI_SH_AUTH_SESSION_H_MARSHAL      UINT32_H_MARSHAL
#define TPM_HANDLE_H_MARSHAL                UINT32_H_MARSHAL
#define TPM2B_NONCE_P_MARSHAL               TPM2B_DIGEST_P_MARSHAL
#define TPMI_YES_NO_P_MARSHAL               UINT8_P_MARSHAL
#define TPM_RC_P_MARSHAL                    UINT32_P_MARSHAL

#ifndef CC_Startup
#include "Startup_fp.h"
#endif

typedef TPM_RC  (Startup_Entry)(
  Startup_In                  *in
);

typedef const struct {
  Startup_Entry           *entry;
  UINT16                  inSize;
  UINT16                  outSize;
  UINT16                  offsetOfTypes;
  BYTE                    types[3];
} Startup_COMMAND_DESCRIPTOR_t;

Startup_CommandDescriptor_t _StartupData = {
  /* entry         */ &TPM2_Startup,
  /* inSize        */ (UINT16)(sizeof(Startup_In)),
  /* outSize       */ 0,
  /* offsetOfTypes */ offsetof(Startup_COMMAND_DESCRIPTOR_t, types),
  /* offsets       */ // No parameter offsets;
  /* types         */ {TPM_SU_P_UNMARSHAL,
   END_OF_LIST,
   END_OF_LIST}
};

#define _StartupDataAddress (&_StartupData)

#else
#define _StartupDataAddress 0
#endif // CC_Startup

#ifndef CC_Shutdown
#include "Shutdown_fp.h"
#endif

typedef TPM_RC  (Shutdown_Entry)(
  Shutdown_In                 *in
);

typedef const struct {
  Shutdown_Entry           *entry;
  UINT16                  inSize;
  UINT16                  outSize;
  UINT16                  offsetOfTypes;
  BYTE                    types[3];
} Shutdown_COMMAND_DESCRIPTOR_t;

Shutdown_COMMAND_DESCRIPTOR_t _ShutdownData = {
  /* entry         */ &TPM2_Shutdown,
  /* inSize        */ (UINT16)(sizeof(Shutdown_In)),
};
#define _ShutdownDataAddress (&_ShutdownData)
#else
#define _ShutdownDataAddress 0
#endif

#define _SelfTestDataAddress (&_SelfTestData)
#else
#define _SelfTestDataAddress 0
#endif

#define _IncrementalSelfTestDataAddress (&_IncrementalSelfTestData)
#else
#define _IncrementalSelfTestDataAddress 0
#endif

typedef const struct {  
    SelfTest_Entry *entry;  
    UINT16 inSize;  
    UINT16 outSize;  
    UINT16 offsetOfTypes;  
    BYTE types[3];
} SelfTest_COMMAND_DESCRIPTOR_t;

SelfTest_COMMAND_DESCRIPTOR_t _SelfTestData = {
    /* entry */ &TPM2_SelfTest,
    /* inSize */ (UINT16)(sizeof(SelfTest_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetOf(SelfTest_COMMAND_DESCRIPTOR_t, types),
    /* offsets */  // No parameter offsets;
    /* types */  
        {TPMI_YES_NO_P_UNMARSHAL,
        END_OF_LIST,
        END_OF_LIST}
};
#define _SelfTestDataAddress (&_SelfTestData)
#else
#define _SelfTestDataAddress 0
#endif

typedef const struct {  
    IncrementalSelfTest_Entry *entry;  
    UINT16 inSize;  
    UINT16 outSize;  
    UINT16 offsetOfTypes;  
    BYTE types[4];
} IncrementalSelfTest_COMMAND_DESCRIPTOR_t;

IncrementalSelfTest_COMMAND_DESCRIPTOR_t _IncrementalSelfTestData = {
    /* entry */ &TPM2_IncrementalSelfTest,
    /* inSize */ (UINT16)(sizeof(IncrementalSelfTest_In)),
    /* outSize */ (UINT16)(sizeof(IncrementalSelfTest_Out)),
    /* offsetOfTypes */ offsetOf(IncrementalSelfTest_COMMAND_DESCRIPTOR_t, types),
    /* offsets */  // No parameter offsets;
    /* types */  
        {TPML_ALG_P_UNMARSHAL,
        END_OF_LIST,
        TPML_ALG_P_MARSHAL,
        END_OF_LIST}
};
#define _IncrementalSelfTestDataAddress (&_IncrementalSelfTestData)
#else
#define _IncrementalSelfTestDataAddress 0
#endif

#define _CC_Shutdown
#if CC_SelfTest
#include "SelfTest_fp.h"

typedef TPM_RC (SelfTest_Entry)(
    SelfTest_In *in,
);
#endif

#define _CC_IncrementalSelfTest
#if CC_IncrementalSelfTest
#include "IncrementalSelfTest_fp.h"

typedef TPM_RC (IncrementalSelfTest_Entry)(
    IncrementalSelfTest_In *in,
    IncrementalSelfTest_Out *out
);
#endif
#if CC_GetTestResult
#include "GetTestResult_fp.h"
typedef TPM_RC (GetTestResult_Entry)(
    GetTestResult_Out *out
);
typedef const struct {
    GetTestResult_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[4];
} GetTestResult_COMMAND_DESCRIPTOR_t;
GetTestResult_COMMAND_DESCRIPTOR_t _GetTestResultData = {
    /* entry */ &TPM2_GetTestResult,
    /* inSize */ 0,
    /* outSize */ (UINT16)(sizeof(GetTestResult_Out)),
    /* offsetOfTypes */ offsetof(GetTestResult_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {(UINT16)(offsetof(GetTestResult_Out, testResult))},
    /* types */
    TPM2B_MAX_BUFFER_P_MARSHAL,
    TPM_RC_P_MARSHAL,
    END_OF_LIST
};
#define _GetTestResultDataAddress (&_GetTestResultData)
#else
#define _GetTestResultDataAddress 0
#endif
// CC_GetTestResult

#if CC_StartAuthSession
#include "StartAuthSession_fp.h"
typedef TPM_RC (StartAuthSession_Entry)(
    StartAuthSession_In *in,
    StartAuthSession_Out *out
);
typedef const struct {
    StartAuthSession_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[7];
    BYTE types[11];
} StartAuthSession_COMMAND_DESCRIPTOR_t;
StartAuthSession_COMMAND_DESCRIPTOR_t _StartAuthSessionData = {
    /* entry */ &TPM2_StartAuthSession,
    /* inSize */ (UINT16)(sizeof(StartAuthSession_In)),
    /* outSize */ (UINT16)(sizeof(StartAuthSession_Out)),
    /* offsetOfTypes */ offsetof(StartAuthSession_COMMAND_DESCRIPTOR_t, types),
    /* offsets */
    TPM2B_NONCE_P_UNMARSHAL,
    TPM секр. секретный P UNMARSHAL,
    TPM2B ENCRYPTED_SECRET_P UNMARSHAL,
    TPM_SE P UNMARSHAL,
    TPM2B_MaxBuffer_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
    TPM2B_MAX BUFFER_P Marshal,
    TPM2B_NONCE P Marshal,
#include "PolicyRestart_fp.h"

typedef TPM_RC (PolicyRestart_Entry)(
    PolicyRestart_In *in
);

typedef const struct {
    PolicyRestart_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    BYTE types[3];
} PolicyRestart_COMMAND_DESCRIPTOR_t;

PolicyRestart_COMMAND_DESCRIPTOR_t _PolicyRestartData = {
    /* entry */ &TPM2_PolicyRestart,
    /* inSize */ (UINT16)(sizeof(PolicyRestart_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetof(PolicyRestart_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ // No parameter offsets:
    /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
                  END_OF_LIST,
                  END_OF_LIST}
};

#define _PolicyRestartDataAddress (&_PolicyRestartData)

#define _StartAuthSessionDataAddress (&_StartAuthSessionData)

#define _StartAuthSessionDataAddress 0

#endif

#endif // CC_PolicyRestart

#define _PolicyRestartDataAddress 0

#endif // CC_PolicyRestart

#endif // CC_Create

#define _CreateDataAddress 0

#endif // CC_Create

#define _CreateDataAddress (&_CreateData)

#define _StartAuthSessionDataAddress 0

#endif // CC_StartAuthSession

typedef const struct {
    Create_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[8];
    BYTE types[12];
} Create_COMMAND_DESCRIPTOR_t;

Create_COMMAND_DESCRIPTOR_t _CreateData = {
    /* entry */ &TPM2_Create,
    /* inSize */ (UINT16)(sizeof(Create_In)),
    /* outSize */ (UINT16)(sizeof(Create_Out)),
    /* offsetOfTypes */ offsetof(Create_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {(UINT16)(offsetof(Create_In, inSensitive)),
                   (UINT16)(offsetof(Create_In, inPublic)),
                   (UINT16)(offsetof(Create_In, outsideInfo)),
                   (UINT16)(offsetof(Create_In, creationPCR)),
                   (UINT16)(offsetof(Create_Out, outPublic)),
                   (UINT16)(offsetof(Create_Out, creationData)),
                   (UINT16)(offsetof(Create_Out, creationHash)),
                   (UINT16)(offsetof(Create_Out, creationTicket))},
    /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
                 TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
                 TPM2B_PUBLIC_P_UNMARSHAL,
                 TPM2B_DATA_P_UNMARSHAL,
                 TPM2B_CREATION_DATA_P_MARSHAL,
                END_OF_LIST,
                TPM2B_PRIVATE_P_MARSHAL,
                TPM2B_PUBLIC_P_MARSHAL,
                TPM2B_CREATION_DATA_P_MARSHAL,
                END_OF_LIST}
499                  TPM2B_DIGEST_P_MARSHAL,  
500                  TPM2B_PRIVATE_P_MARSHAL,  
501                  TPM2B_PUBLIC_P_MARSHAL, 
502                  END_OF_LIST}  
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 */ &_TPM_Load,  
523     /* inSize */ (UINT16)(sizeof(Load_In)),  
524     /* outSize */ (UINT16)(sizeof(Load_Out)),  
525     /* offsetOfTypes */ offsetof(Load_COMMAND_DESCRIPTOR_t, types),  
526     /* offsets */ {  
527         (UINT16)(offsetof(Load_In, inPrivate)),  
528         (UINT16)(offsetof(Load_In, inPublic)),  
529         (UINT16)(offsetof(Load_Out, name))},  
530     /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL, 
531                  TPM2B_PRIVATE_P_UNMARSHAL, 
532                  TPM2B_PUBLIC_P_UNMARSHAL, 
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 */ &_TPM_LoadExternal,  
557     /* inSize */ (UINT16)(sizeof(LoadExternal_In)),  
558     /* outSize */ (UINT16)(sizeof(LoadExternal_Out)),  
559     /* offsetOfTypes */ offsetof(LoadExternal_COMMAND_DESCRIPTOR_t, types),  
560     /* offsets */ {  
561         (UINT16)(offsetof(LoadExternal_In, inPublic)),  
562         (UINT16)(offsetof(LoadExternal_In, hierarchy)),  
563         (UINT16)(offsetof(LoadExternal_Out, name))},  
564     /* types */ {TPM2B_SENSITIVE_P_UNMARSHAL, 
565                  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 #define _LoadExternalDataAddress (&_LoadExternalData)
571 #else
572 #define _LoadExternalDataAddress 0
573 #endif // CC_LoadExternal
574 #if CC_ReadPublic
575 #include "ReadPublic_fp.h"
576 typedef TPM_RC (ReadPublic_Entry)(
577  ReadPublic_In *in,
578  ReadPublic_Out *out
579 );
580 typedef const struct {
581  ReadPublic_Entry *entry;
582  UINT16 inSize;
583  UINT16 outSize;
584  UINT16 offsetOfTypes;
585  UINT16 paramOffsets[2];
586  BYTE types[6];
587 } ReadPublic_COMMAND_DESCRIPTOR_t;
588 ReadPublic_COMMAND_DESCRIPTOR_t _ReadPublicData = {
589  /* entry         */ &TPM2_ReadPublic,
590  /* inSize        */ (UINT16)(sizeof(ReadPublic_In)),
591  /* outSize       */ (UINT16)(sizeof(ReadPublic_Out)),
592  /* offsetOfTypes */ offsetof(ReadPublic_COMMAND_DESCRIPTOR_t, types),
593  /* offsets       */ {
594    (UINT16)(offsetof(ReadPublic_Out, name)),
595    (UINT16)(offsetof(ReadPublic_Out, qualifiedName))},
596  /* types         */ {
597    TPMI_DH_OBJECT_H_UNMARSHAL,
598    END_OF_LIST,
599    TPM2B_PUBLIC_P_MARSHAL,
600    TPM2B_NAME_P_MARSHAL,
601    TPM2B_NAME_P_MARSHAL,
602    END_OF_LIST}
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, types),
626  /* offsets       */ {
627    (UINT16)(offsetof(ActivateCredential_In, keyHandle)),
628    (UINT16)(offsetof(ActivateCredential_In, secret))},
629  /* types         */
typedef TPM_RC  (MakeCredential_Entry)(
    MakeCredential_In           *in,
    MakeCredential_Out          *out
);

typedef const struct {
    MakeCredential_Entry    *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[3];
    BYTE                    types[7];
} MakeCredentialCOMMAND_DESCRIPTOR_t;

MakeCredential_COMMAND_DESCRIPTOR_t _MakeCredentialData = {
    /* entry         */ &TPM2_MakeCredential,
    /* inSize        */ (UINT16)(sizeof(MakeCredential_In)),
    /* outSize       */ (UINT16)(sizeof(MakeCredential_Out)),
    /* offsetOfTypes */ offsetof(MakeCredential_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ {
        (UINT16)(offsetof(MakeCredential_In, credential)),
        (UINT16)(offsetof(MakeCredential_In, objectName)),
        (UINT16)(offsetof(MakeCredential_Out, secret))},
    /* types         */ {
        TPMI_DH_OBJECT_H_UNMARSHAL,
        TPM2B_DIGEST_P_UNMARSHAL,
        TPM2B_NAME_P_UNMARSHAL,
        END_OF_LIST,
        TPM2B_ID_OBJECT_P_MARSHAL,
        TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
        END_OF_LIST}
};

#define _MakeCredentialDataAddress (&_MakeCredentialData)
#else
#define _MakeCredentialDataAddress 0
#endif // CC_MakeCredential

typedef TPM_RC  (Unseal_Entry)(
    Unseal_In                   *in,
    Unseal_Out                  *out
);

typedef const struct {
    Unseal_Entry            *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    BYTE                    types[4];
} Unseal_COMMAND_DESCRIPTOR_t;

Unseal_COMMAND_DESCRIPTOR_t _UnsealData = {
    /* entry         */ &TPM2_Unseal,
    /* inSize        */ (UINT16)(sizeof(Unseal_In)),
    /* outSize       */ (UINT16)(sizeof(Unseal_Out)),
    /* offsetOfTypes */ offsetof(Unseal_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ // No parameter offsets;
    /* types         */ {
        TPMI_DH_OBJECT_H_UNMARSHAL,
#define _UnsealDataAddress (&_UnsealData)
#else
#define _UnsealDataAddress 0
#endif
// CC_Unseal

#if CC_ObjectChangeAuth
#include "ObjectChangeAuth_fp.h"
typedef TPM_RC (ObjectChangeAuth_Entry)(
    ObjectChangeAuth_In *in,
    ObjectChangeAuth_Out *out
);
typedef const struct {
    ObjectChangeAuth_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[2];
    BYTE types[6];
} ObjectChangeAuth_COMMAND_DESCRIPTOR_t;
ObjectChangeAuth_COMMAND_DESCRIPTOR_t _ObjectChangeAuthData = {
    /* entry         */ &TPM2_ObjectChangeAuth,
    /* inSize        */ (UINT16)(sizeof(ObjectChangeAuth_In)),
    /* outSize       */ (UINT16)(sizeof(ObjectChangeAuth_Out)),
    /* offsetOfTypes */ offsetOf(ObjectChangeAuth_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ {
        (UINT16)(offsetof(ObjectChangeAuth_In, parentHandle)),
        (UINT16)(offsetof(ObjectChangeAuth_In, newAuth))},
    /* types         */ {TPMI_DH_OBJECT_H_UNMARSHAL,
        TPM2B_PRIVATE_P_MARSHAL,
        END_OF_LIST}
};
#define _ObjectChangeAuthDataAddress (&_ObjectChangeAuthData)
#else
#define _ObjectChangeAuthDataAddress 0
#endif
// CC_ObjectChangeAuth

#if CC_CreateLoaded
#include "CreateLoaded_fp.h"
typedef TPM_RC (CreateLoaded_Entry)(
    CreateLoaded_In *in,
    CreateLoaded_Out *out
);
typedef const struct {
    CreateLoaded_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[5];
    BYTE types[9];
} CreateLoaded_COMMAND_DESCRIPTOR_t;
CreateLoaded_COMMAND_DESCRIPTOR_t _CreateLoadedData = {
    /* entry         */ &TPM2_CreateLoaded,
    /* inSize        */ (UINT16)(sizeof(CreateLoaded_In)),
    /* outSize       */ (UINT16)(sizeof(CreateLoaded_Out)),
    /* offsetOfTypes */ offsetOf(CreateLoaded_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ {
        (UINT16)(offsetof(CreateLoaded_In, inSensitive)),
        (UINT16)(offsetof(CreateLoaded_In, inPublic)),
        (UINT16)(offsetof(CreateLoaded_Out, outPrivate)),
        (UINT16)(offsetof(CreateLoaded_Out, outPublic)),
        (UINT16)(offsetof(CreateLoaded_Out, name))},
    /* types         */ {TPMI_DH_PARENT_H_UNMARSHAL + ADD_FLAG,
        TPM2B_SENSITIVE_CREATE_P_MARSHAL,
        TPM2B_PRIVATE_P_MARSHAL,
        END_OF_LIST}
};
#define _CreateLoadedDataAddress (&_CreateLoadedData)
#else
#define _CreateLoadedDataAddress 0
#endif
// CC_CreateLoaded
TPM2B_TEMPLATE_P_UNMARSHAL,
END_OF_LIST,
TPM_HANDLE_H_MARSHAL,
TPM2B_PRIVATE_P_MARSHAL,
TPM2B_PUBLIC_P_MARSHAL,
TPM2B_NAME_P_MARSHAL,
END_OF_LIST};

#define _CreateLoadedDataAddress (&_CreateLoadedData)
#else
#define _CreateLoadedDataAddress 0
#endif // CC_CreateLoaded

#include "Duplicate_fp.h"
typedef TPM_RC (Duplicate_Entry)(
    Duplicate_In *in,
    Duplicate_Out *out
);
#endif

#define _DuplicateDataAddress (&_DuplicateData)
#else
#define _DuplicateDataAddress 0
#endif // CC_Duplicate

#include "Rewrap_fp.h"
typedef TPM_RC (Rewrap_Entry)(
    Rewrap_In *in,
    Rewrap_Out *out
);
#endif

#define _RewrapDataAddress (&_RewrapData)
#else
#define _RewrapDataAddress 0
#endif // CC_Rewrap
/* inSize */ (UINT16)(sizeof(Rewrap_In)),
/* outSize */ (UINT16)(sizeof(Rewrap_Out)),
/* offsetOfTypes */ offsetof(Rewrap_COMMAND_DESCRIPTOR_t, types),
/* offsets */ {
  (UINT16)(offsetof(Rewrap_In, newParent)),
  (UINT16)(offsetof(Rewrap_In, inDuplicate)),
  (UINT16)(offsetof(Rewrap_In, name)),
  (UINT16)(offsetof(Rewrap_In, inSymSeed)),
  (UINT16)(offsetof(Rewrap_Out, outSymSeed))},
/* types */ {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
TPM2B_PRIVATE_P_UNMARSHAL,
TPM2B_NAME_P_UNMARSHAL,
TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
END_OF_LIST,
TPM2B_PRIVATE_P_MARSHAL,
TPM2B_ENCRYPTED_SECRET_P_MARSHAL,
END_OF_LIST};
#define _RewrapDataAddress (&_RewrapData)
#else
#define _RewrapDataAddress 0
#endif // CC_Rewrap
#if CC_Import
#include "Import_fp.h"
typedef TPM_RC (Import_Entry)(
  Import_In *in,
  Import_Out *out);
#endif
typedef const struct {
  Import_Entry            *entry;
  UINT16                  inSize;
  UINT16                  outSize;
  UINT16                  offsetOfTypes;
  UINT16                  paramOffsets[5];
  BYTE                    types[9];
} Import_COMMAND_DESCRIPTOR_t;
Import_COMMAND_DESCRIPTOR_t _ImportData = {
/* entry */            &TPM2_Import,
/* inSize */           (UINT16)(sizeof(Import_In)),
/* outSize */          (UINT16)(sizeof(Import_Out)),
/* offsetOfTypes */    offsetof(Import_COMMAND_DESCRIPTOR_t, types),
/* offsets */          {
  (UINT16)(offsetof(Import_In, encryptionKey)),
  (UINT16)(offsetof(Import_In, objectPublic)),
  (UINT16)(offsetof(Import_In, duplicate)),
  (UINT16)(offsetof(Import_In, inSymSeed)),
  (UINT16)(offsetof(Import_In, symmetricAlg))},
/* types */            {TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_DATA_P_UNMARSHAL,
TPM2B_PUBLIC_P_UNMARSHAL,
TPM2B_PRIVATE_P_UNMARSHAL,
TPM2B_ENCRYPTED_SECRET_P_UNMARSHAL,
TPMT_SYM_DEF_OBJECT_P_UNMARSHAL + ADD_FLAG,
END_OF_LIST,
TPM2B_PRIVATE_P_MARSHAL,
END_OF_LIST};
#define _ImportDataAddress (&_ImportData)
#else
#define _ImportDataAddress 0
#endif // CC_Import
#if CC_RSA_Encrypt
#include "RSA_Encrypt_fp.h"
typedef TPM_RC (RSA_Encrypt_Entry)(
  RSA_Encrypt_In       *in,
  RSA_Encrypt_Out      *out);
typedef const struct {
    RSA_Encrypt_Entry       *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[3];
    BYTE                    types[7];
} RSA_Encrypt_COMMAND_DESCRIPTOR_t;

RSA_Encrypt_COMMAND_DESCRIPTOR_t _RSA_EncryptData = {
    /* entry         */ &TPM2_RSA_Encrypt,
    /* inSize        */ (UINT16)(sizeof(RSA_Encrypt_In)),
    /* outSize       */ (UINT16)(sizeof(RSA_Encrypt_Out)),
    /* offsetOfTypes */ offsetof(RSA_Encrypt_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */{(UINT16)(offsetof(RSA_Encrypt_In, message)),
    (UINT16)(offsetof(RSA_Encrypt_In, inScheme)),
    (UINT16)(offsetof(RSA_Encrypt_In, label))},
    /* types         */{TPMI_DH_OBJECT_H_UNMARSHAL,
    TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
    TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
    TPM2B_DATA_P_UNMARSHAL,
    END_OF_LIST,
    TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
    END_OF_LIST}
};
#define _RSA_EncryptDataAddress (&_RSA_EncryptData)
#else
#define _RSA_EncryptDataAddress 0
#endif

#if CC_RSA_Decrypt
#include "RSA_Decrypt_fp.h"
typedef TPM_RC  (RSA_Decrypt_Entry)(
    RSA_Decrypt_In              *in,
    RSA_Decrypt_Out             *out
);
typedef const struct {
    RSA_Decrypt_Entry       *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[3];
    BYTE                    types[7];
} RSA_Decrypt_COMMAND_DESCRIPTOR_t;

RSA_Decrypt_COMMAND_DESCRIPTOR_t _RSA_DecryptData = {
    /* entry         */ &TPM2_RSA_Decrypt,
    /* inSize        */ (UINT16)(sizeof(RSA_Decrypt_In)),
    /* outSize       */ (UINT16)(sizeof(RSA_Decrypt_Out)),
    /* offsetOfTypes */ offsetof(RSA_Decrypt_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */{(UINT16)(offsetof(RSA_Decrypt_In, cipherText)),
    (UINT16)(offsetof(RSA_Decrypt_In, inScheme)),
    (UINT16)(offsetof(RSA_Decrypt_In, label))},
    /* types         */{TPMI_DH_OBJECT_H_UNMARSHAL,
    TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
    TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
    TPM2B_DATA_P_UNMARSHAL,
    END_OF_LIST,
    TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
    END_OF_LIST}
};
#define _RSA_DecryptDataAddress (&_RSA_DecryptData)
#else
#define _RSA_DecryptDataAddress 0
#endif

#if CC_ECDH_KeyGen
#include "ECDH_KeyGen_fp.h"
typedef TPM_RC  (ECDH_KeyGen_Entry)(
    ECDH_KeyGen_In              *in,
    ECDH_KeyGen_Out             *out
);
typedef const struct {
    ECDH_KeyGen_Entry       *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[3];
    BYTE                    types[7];
} ECDH_KeyGen_COMMAND_DESCRIPTOR_t;

ECDH_KeyGen_COMMAND_DESCRIPTOR_t _ECDH_KeyGenData = {
    /* entry         */ &TPM2_ECDH_KeyGen,
    /* inSize        */ (UINT16)(sizeof(ECDH_KeyGen_In)),
    /* outSize       */ (UINT16)(sizeof(ECDH_KeyGen_Out)),
    /* offsetOfTypes */ offsetof(ECDH_KeyGen_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */{(UINT16)(offsetof(ECDH_KeyGen_In, keyMaterial)),
    (UINT16)(offsetof(ECDH_KeyGen_In, keyType)),
    (UINT16)(offsetof(ECDH_KeyGen_In, label))},
    /* types         */{TPMI_DH_OBJECT_H_UNMARSHAL,
    TPM2B_PUBLIC_KEY_RSA_P_UNMARSHAL,
    TPMT_RSA_DECRYPT_P_UNMARSHAL + ADD_FLAG,
    TPM2B_DATA_P_UNMARSHAL,
    END_OF_LIST,
    TPM2B_PUBLIC_KEY_RSA_P_MARSHAL,
    END_OF_LIST}
};
#define _ECDH_KeyGenDataAddress (&_ECDH_KeyGenData)
#else
#define _ECDH_KeyGenDataAddress 0
#endif

#if CC_SCP
#include "SCP_fp.h"
typedef TPM_RC  (SCP_Entry)(
    SCP_In              *in,
typedef const struct {
    ECDH_KeyGen_Entry       *entry;  
    UINT16                  inSize;  
    UINT16                  outSize; 
    UINT16                  offsetOfTypes;  
    UINT16                  paramOffsets[1]; 
    BYTE                    types[5];
} ECDH_KeyGen_COMMAND_DESCRIPTOR_t;

ECDH_KeyGen_COMMAND_DESCRIPTOR_t _ECDH_KeyGenData = {
    /* entry */ &TPM2_ECDH_KeyGen, 
    /* inSize */ (UINT16)(sizeof(ECDH_KeyGen_In)), 
    /* outSize */ (UINT16)(sizeof(ECDH_KeyGen_Out)), 
    /* offsetOfTypes */ offsetof(ECDH_KeyGen_COMMAND_DESCRIPTOR_t, types), 
    /* offsets */ {(UINT16)(offsetof(ECDH_KeyGen_Out, pubPoint))}, 
    /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL, 
                END_OF_LIST, 
                TPM2B_ECC_POINT_P_MARSHAL, 
                TPM2B_ECC_POINT_P_MARSHAL, 
                END_OF_LIST}
};
#define _ECDH_KeyGenDataAddress (&_ECDH_KeyGenData)

typedef const struct {
    ECDH_ZGen_Entry         *entry;  
    UINT16                  inSize;  
    UINT16                  outSize; 
    UINT16                  offsetOfTypes;  
    UINT16                  paramOffsets[1]; 
    BYTE                    types[5];
} ECDH_ZGen_COMMAND_DESCRIPTOR_t;

ECDH_ZGen_COMMAND_DESCRIPTOR_t _ECDH_ZGenData = {
    /* entry */ &TPM2_ECDH_ZGen, 
    /* inSize */ (UINT16)(sizeof(ECDH_ZGen_In)), 
    /* outSize */ (UINT16)(sizeof(ECDH_ZGen_Out)), 
    /* offsetOfTypes */ offsetof(ECDH_ZGen_COMMAND_DESCRIPTOR_t, types), 
    /* offsets */ {(UINT16)(offsetof(ECDH_ZGen_In, inPoint))}, 
    /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL, 
                END_OF_LIST, 
                TPM2B_ECC_POINT_P_UNMARSHAL, 
                TPM2B_ECC_POINT_P_MARSHAL, 
                TPM2B_ECC_POINT_P_MARSHAL, 
                END_OF_LIST}
};
#define _ECDH_ZGenDataAddress (&_ECDH_ZGenData)

typedef const struct {
    ECC_Parameters_Entry    *entry; 
    UINT16                  inSize; 
    UINT16                  outSize; 
    UINT16                  offsetOfTypes; 
    UINT16                  paramOffsets[1]; 
    BYTE                    types[5];
} ECC_Parameters_COMMAND_DESCRIPTOR_t;

ECC_Parameters_COMMAND_DESCRIPTOR_t _ECDH_ParametersData = {
    /* entry */ &TPM2_ECC_Parameters, 
    /* inSize */ (UINT16)(sizeof(ECC_Parameters_In)), 
    /* outSize */ (UINT16)(sizeof(ECC_Parameters_Out)), 
    /* offsetOfTypes */ offsetof(ECC_Parameters_COMMAND_DESCRIPTOR_t, types), 
    /* offsets */ {(UINT16)(offsetof(ECC_Parameters_In, parameters))}, 
    /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL, 
                END_OF_LIST, 
                TPM2B_ECC_POINT_P_UNMARSHAL, 
                TPM2B_ECC_POINT_P_UNMARSHAL, 
                TPM2B_ECC_POINT_P_MARSHAL, 
                TPM2B_ECC_POINT_P_MARSHAL, 
                END_OF_LIST}
};
#define _ECDH_ParametersDataAddress (&_ECDH_ParametersData)
UINT16 OFFSETOFTYPES;
BYTE TYPES[4];
)
ECC_Parameters_COMMAND_DESCRIPTOR_t
ECC_Parameters_COMMAND_DESCRIPTOR_t _ECC_ParametersData = {
/* entry */ &TPM2_ECC_Parameters,
/* inSize */ (UINT16)(sizeof(ECC_Parameters_In)),
/* outSize */ (UINT16)(sizeof(ECC_Parameters_Out)),
/* offsetOfTypes */ offsetof(ECC_Parameters_COMMAND_DESCRIPTOR_t, types),
/* offsets */ // No parameter offsets;
/* types */ {TPMI_ECC_CURVE_P_UNMARSHAL,
END_OF_LIST,
TPMS_ALGORITHM_DETAIL_ECC_P_MARSHAL,
END_OF_LIST}
};
#define _ECC_ParametersDataAddress (&_ECC_ParametersData)
#else
#define _ECC_ParametersDataAddress 0
#endif
// CC_ECC_Parameters
#if CC_ZGen_2Phase
#include "ZGen_2Phase_fp.h"
typedef TPM_RC (ZGen_2Phase_Entry)(
ZGen_2Phase_In *in,
ZGen_2Phase_Out *out
);
typedef const struct {
ZGen_2Phase_Entry *entry;
UINT16 inSize;
UINT16 outSize;
UINT16 offsetOfTypes;
UINT16 paramOffsets[5];
BYTE types[9];
} ZGen_2Phase_COMMAND_DESCRIPTOR_t;
ZGen_2Phase_COMMAND_DESCRIPTOR_t _ZGen_2PhaseData = {
/* entry */ &TPM2_ZGen_2Phase,
/* inSize */ (UINT16)(sizeof(ZGen_2Phase_In)),
/* outSize */ (UINT16)(sizeof(ZGen_2Phase_Out)),
/* offsetOfTypes */ (UINT16)(offsetof(ZGen_2Phase_COMMAND_DESCRIPTOR_t, types)),
/* offsets */ {
(UINT16)(offsetof(ZGen_2Phase_In, inQsB)),
(UINT16)(offsetof(ZGen_2Phase_In, inQeB)),
(UINT16)(offsetof(ZGen_2Phase_In, inScheme)),
(UINT16)(offsetof(ZGen_2Phase_In, counter)),
(UINT16)(offsetof(ZGen_2Phase_Out, outZ2))},
/* types */ {
TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_ECC_POINT_P_UNMARSHAL,
TPM2B_ECC_POINT_P_UNMARSHAL,
TPM2B_ECC_POINT_P_MARSHAL,
TPM2B_ECC_POINT_P_MARSHAL,
END_OF_LIST}
};
#define _ZGen_2PhaseDataAddress (&_ZGen_2PhaseData)
#else
#define _ZGen_2PhaseDataAddress 0
#endif
// CC_EncryptDecrypt
#include "EncryptDecrypt_fp.h"
typedef TPM_RC (EncryptDecrypt_Entry)(
EncryptDecrypt_In *in,
EncryptDecrypt_Out *out
);
typedef const struct {
EncryptDecrypt_Entry *entry;
UINT16 inSize;
UINT16 outSize;
}
```c
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 */ {
1101 (UINT16)(offsetof(EncryptDecrypt_In, decrypt)),
1102 (UINT16)(offsetof(EncryptDecrypt_In, mode)),
1103 (UINT16)(offsetof(EncryptDecrypt_In, ivIn)),
1104 (UINT16)(offsetof(EncryptDecrypt_In, inData)),
1105 (UINT16)(offsetof(EncryptDecrypt_Out, ivOut))},
1106 /* types */ {
1107 TPMI_DH_OBJECT_H_UNMARSHAL,
1108 TPMI_YES_NO_P_UNMARSHAL,
1109 TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1110 TPM2B_IV_P_UNMARSHAL,
1111 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1112 TPM2B_MAX_BUFFER_P_MARSHAL,
1113 TPM2B_MAX_BUFFER_P_MARSHAL,
1114 END_OF_LIST
1115};
1116 #define _EncryptDecryptDataAddress (&_EncryptDecryptData)
1117 #else
1118 #define _EncryptDecryptDataAddress 0
1119 #endif // CC_EncryptDecrypt
1120 #if CC_EncryptDecrypt2
1121 #include "EncryptDecrypt2_fp.h"
1122 typedef TPM_RC (EncryptDecrypt2_Entry)(
1123 EncryptDecrypt2_In *in,
1124 EncryptDecrypt2_Out *out
1125);
1126 typedef const struct {
1127 EncryptDecrypt2_Entry *entry;
1128 UINT16 inSize;
1129 UINT16 outSize;
1130 UINT16 offsetOfTypes;
1131 UINT16 paramOffsets[5];
1132 BYTE types[9];
1133 } EncryptDecrypt2_COMMAND_DESCRIPTOR_t;
1134 EncryptDecrypt2_COMMAND_DESCRIPTOR_t _EncryptDecrypt2Data = {
1135 /* entry */ &TPM2_EncryptDecrypt2,
1136 /* inSize */ (UINT16)(sizeof (EncryptDecrypt2_In)),
1137 /* outSize */ (UINT16)(sizeof (EncryptDecrypt2_Out)),
1138 /* offsetOfTypes */ offsetof(EncryptDecrypt2_COMMAND_DESCRIPTOR_t, types),
1139 /* offsets */ {
1140 (UINT16)(offsetof(EncryptDecrypt2_In, inData)),
1141 (UINT16)(offsetof(EncryptDecrypt2_In, decrypt)),
1142 (UINT16)(offsetof(EncryptDecrypt2_In, mode)),
1143 (UINT16)(offsetof(EncryptDecrypt2_In, ivIn)),
1144 (UINT16)(offsetof(EncryptDecrypt2_Out, ivOut))},
1145 /* types */ {
1146 TPMI_DH_OBJECT_H_UNMARSHAL,
1147 TPM2B_MAX_BUFFER_P_UNMARSHAL,
1148 TPMI_YES_NO_P_UNMARSHAL,
1149 TPMI_ALG_CIPHER_MODE_P_UNMARSHAL + ADD_FLAG,
1150 TPM2B_IV_P_UNMARSHAL,
1151 TPM2B_MAX_BUFFER_P_MARSHAL,
1152 TPM2B_MAX_BUFFER_P_MARSHAL,
1153 END_OF_LIST
1154};
1155 #define _EncryptDecrypt2DataAddress (&_EncryptDecrypt2Data)
1156 #else
1157 #define _EncryptDecrypt2DataAddress 0
1158 #endif // CC_EncryptDecrypt2
```
#if CC_Hash
#include "Hash_fp.h"

typedef TPM_RC (Hash_Entry)(
    Hash_In *in,
    Hash_Out *out
);

typedef const struct {
    Hash_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[3];
    BYTE types[7];
} Hash_COMMAND_DESCRIPTOR_t;

Hash_COMMAND_DESCRIPTOR_t _HashData = {
    /* entry */ &TPM2_Hash,
    /* inSize */ (UINT16)(sizeof(Hash_In)),
    /* outSize */ (UINT16)(sizeof(Hash_Out)),
    /* offsetOfTypes */ offsetOf(Hash_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(Hash_In, hashAlg)),
        (UINT16)(offsetof(Hash_In, hierarchy)),
        (UINT16)(offsetof(Hash_Out, validation))},
    /* types */ {
        TPM2B_MAX_BUFFER_P_UNMARSHAL,
        TPM_ALG_HASH_P_UNMARSHAL,
        TPM_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
        END_OF_LIST,
        TPM2B_DIGEST_P_MARSHAL,
        TPMT_TK_HASHCHECK_P_MARSHAL,
        END_OF_LIST}
};

#define _HashDataAddress (&_HashData)
#else
#define _HashDataAddress 0
#endif // CC_Hash

#endif // CC_Hash
#if CC_MAC
#include "MAC_fp.h"
typedef TPM_RC (MAC_Entry)
  (MAC_In *in,
   MAC_Out *out)
);
typedef const struct
  {
    MAC_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[2];
    BYTE types[6];
  } MAC_COMMAND_DESCRIPTOR_t;
MAC_COMMAND_DESCRIPTOR_t _MACData = {
  /* entry         */ &TPM2_MAC,
  /* inSize        */ (UINT16)(sizeof(MAC_In)),
  /* outSize       */ (UINT16)(sizeof(MAC_Out)),
  /* offsetOfTypes */ offsetof(MAC_COMMAND_DESCRIPTOR_t, types),
  /* offsets */
  (UINT16)(offsetof(MAC_In, buffer)),
  /* types */
  {TPMI_DH_OBJECT_H_UNMARSHAL,
   TPM2_MAX_BUFFER_P_UNMARSHAL,
   TPM_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
   TPM2_DIGEST_P_MARSHAL,
   END_OF_LIST}
};
#define _MACDataAddress (&_MACData)
#else
#define _MACDataAddress 0
#endif // CC_MAC

#if CC_GetRandom
#include "GetRandom_fp.h"
typedef TPM_RC (GetRandom_Entry)
  (GetRandom_In *in,
   GetRandom_Out *out)
);
typedef const struct
  {
    GetRandom_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    BYTE types[4];
  } GetRandom_COMMAND_DESCRIPTOR_t;
GetRandom_COMMAND_DESCRIPTOR_t _GetRandomData = {
  /* entry         */ &TPM2_GetRandom,
  /* inSize        */ (UINT16)(sizeof(GetRandom_In)),
  /* outSize       */ (UINT16)(sizeof(GetRandom_Out)),
  /* offsetOfTypes */ offsetof(GetRandom_COMMAND_DESCRIPTOR_t, types),
  /* offsets */
  // No parameter offsets;
  /* types */
  {UINT16_P_UNMARSHAL,
   TPM2_MAX_BUFFER_P_UNMARSHAL,
   TPM_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
   TPM2_DIGEST_P_MARSHAL,
   END_OF_LIST}
};
#define _GetRandomDataAddress (&_GetRandomData)
#else
#define _GetRandomDataAddress 0
#endif // CC_GetRandom

#if CC_StirRandom
#include "StirRandom_fp.h"
typedef TPM_RC (StirRandom_Entry)
  (StirRandom_In *in)
);
typedef const struct
  {
    StirRandom_In *in
  } StirRandom_COMMAND_DESCRIPTOR_t;
StirRandom_COMMAND_DESCRIPTOR_t _StirRandomData = {
  /* entry */ &TPM2_StirRandom,
  /* inSize */ (UINT16)(sizeof(StirRandom_In)),
  /* outSize */ (UINT16)(sizeof(StirRandom_Out)),
  /* offsetOfTypes */ offsetof(StirRandom_COMMAND_DESCRIPTOR_t, types),
  /* offsets */
  // No parameter offsets;
  /* types */
  {UINT16_P_UNMARSHAL,
   TPM2_MAX_BUFFER_P_UNMARSHAL,
   TPM_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
   TPM2_DIGEST_P_MARSHAL,
   END_OF_LIST}
};
#define _StirRandomDataAddress (&_StirRandomData)
#else
#define _StirRandomDataAddress 0
#endif // CC_StirRandom
typedef struct {
    TPM2_StirRandom         entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    BYTE                    types[3];
} StirRandom_COMMAND_DESCRIPTOR_t;

StirRandom_COMMAND_DESCRIPTOR_t _StirRandomData = {
    /* entry */ &TPM2_StirRandom,
    /* inSize */ (UINT16)(sizeof(StirRandom_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetof(SendRandom_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ // No parameter offsets;
    /* types */ {TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
                  END_OF_LIST,
                  END_OF_LIST}
};
#define _StirRandomDataAddress (&_StirRandomData)
#endif // CC_StirRandom

#define CC_HMAC_Start
#include "HMAC_Start_fp.h"
typedef TPM_RC  (HMAC_Start_Entry)(
    HMAC_Start_In               *in,
    HMAC_Start_Out              *out)
    HMAC_Start_COMMAND_DESCRIPTOR_t _HMAC_StartData = {
    /* entry */ &TPM2_HMAC_Start,
    /* inSize */ (UINT16)(sizeof(HMAC_Start_In)),
    /* outSize */ (UINT16)(sizeof(HMAC_Start_Out)),
    /* offsetOfTypes */ offsetof(HMAC_Start_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(HMAC_Start_In, auth)),
        (UINT16)(offsetof(HMAC_Start_In, hashAlg))},
    /* types */ {
        TPMI_DH_OBJECT_H_UNMARSHAL,
        TPM2B_AUTH_P_UNMARSHAL,
        TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
        END_OF_LIST,
        TPMI_DH_OBJECT_H_MARSHAL,
        END_OF_LIST}
};
#define _HMAC_StartDataAddress (&_HMAC_StartData)
#else
#define _HMAC_StartDataAddress 0
#endif // CC_HMAC_Start

#define CC_MAC_Start
#include "MAC_Start_fp.h"
typedef TPM_RC  (MAC_Start_Entry)(
    MAC_Start_In                *in,
    MAC_Start_Out               *out)
    MAC_Start_COMMAND_DESCRIPTOR_t _MAC_StartData = {
    /* entry */ &TPM2_MAC_Start,
    /* inSize */ (UINT16)(sizeof(MAC_Start_In)),
    /* outSize */ (UINT16)(sizeof(MAC_Start_Out)),
    /* offsetOfTypes */ offsetof(MAC_Start_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(MAC_Start_In, auth)),
        (UINT16)(offsetof(MAC_Start_In, hashAlg))},
    /* types */ {
        TPMI_DH_OBJECT_H_UNMARSHAL,
        TPM2B_AUTH_P_UNMARSHAL,
        TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
        END_OF_LIST,
        TPMI_DH_OBJECT_H_MARSHAL,
        END_OF_LIST}
};
#define _MAC_StartDataAddress (&_MAC_StartData)
#else
#define _MAC_StartDataAddress 0
#endif // CC_MAC_Start
MAC_Start_COMMAND_DESCRIPTOR_t _MAC_StartData = {
    /* entry */ &TPM2_MAC_Start,
    /* inSize */ (UINT16)(sizeof(MAC_Start_In)),
    /* outSize */ (UINT16)(sizeof(MAC_Start_Out)),
    /* offsetOfTypes */ offsetOf(MAC_Start_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ ((UINT16)(offsetof(MAC_Start_In, auth)),
                  (UINT16)(offsetof(MAC_Start_In, inScheme))),
    /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
                 TPM2B_AUTH_P_UNMARSHAL,
                 TPMI_ALG_MAC_SCHEME_P_UNMARSHAL + ADD_FLAG,
                 END_OF_LIST,
                 TPMI_DH_OBJECT_H_MARSHAL,
                 END_OF_LIST}
};
#define _MAC_StartDataAddress (&_MAC_StartData)
#else
#define _MAC_StartDataAddress 0
#endif
// CC_MAC_Start
#if CC_HashSequenceStart
#include "HashSequenceStart_fp.h"
typedef TPM_RC (HashSequenceStart_Entry)(
    HashSequenceStart_In *in,
    HashSequenceStart_Out *out
);
#endif
#define const struct {
    HashSequenceStart_Entry entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[5];
} HashSequenceStart_COMMAND_DESCRIPTOR_t;
HashSequenceStart_COMMAND_DESCRIPTOR_t _HashSequenceStartData = {
    /* entry */ &TPM2_HashSequenceStart,
    /* inSize */ (UINT16)(sizeof(HashSequenceStart_In)),
    /* outSize */ (UINT16)(sizeof(HashSequenceStart_Out)),
    /* offsetOfTypes */ offsetOf(HashSequenceStart_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ ((UINT16)(offsetof(HashSequenceStart_In, hashAlg))),
    /* types */ {TPM2B_AUTH_P_UNMARSHAL,
                  TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
                  END_OF_LIST,
                  TPMI_DH_OBJECT_H_MARSHAL,
                  END_OF_LIST}
};
#define _HashSequenceStartDataAddress (&_HashSequenceStartData)
#else
#define _HashSequenceStartDataAddress 0
#endif
// CC_HashSequenceStart
#if CC_SequenceUpdate
#include "SequenceUpdate_fp.h"
typedef TPM_RC (SequenceUpdate_Entry)(
    SequenceUpdate_In *in
);
#endif
#define const struct {
    SequenceUpdate_Entry entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[4];
} SequenceUpdate_COMMAND_DESCRIPTOR_t;
SequenceUpdate_COMMAND_DESCRIPTOR_t _SequenceUpdateData = {
    /* entry */ &TPM2_SequenceUpdate,
    /* inSize */ (UINT16)(sizeof(SequenceUpdate_In)),
    /* outSize */ 0,
};
Trusted Platform Module Library
Part 4: Supporting Routines

#define offsetOfTypes offsetof(SequenceUpdate_COMMAND_DESCRIPTOR_t, types),
#define offsets {(UINT16)(offsetof(SequenceUpdate_In, buffer))},
#define types {TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_MAX_BUFFER_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST}
#endif
// CC_SequenceUpdate

#define _SequenceUpdateDataAddress (&_SequenceUpdateData)
#define _SequenceCompleteDataAddress (&_SequenceCompleteData)
#define _EventSequenceCompleteDataAddress (&_EventSequenceCompleteData)

#define offsetOfTypes offsetof(SequenceUpdate_COMMAND_DESCRIPTOR_t, types),
#define offsets {(UINT16)(offsetof(SequenceComplete_In, buffer))},
#define types {TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_MAX_BUFFER_P_UNMARSHAL,
TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
END_OF_LIST,
TPM2B_DIGEST_P_MARSHAL,
TPMT_TK_HASHCHECK_P_MARSHAL,
END_OF_LIST}
#endif
// CC_EventSequenceComplete

typedef const struct {
    EventSequenceComplete_Entry *entry;
    UINT16                        inSize;
    UINT16                        outSize;
    UINT16                        offsetOfTypes;
    UINT16                        paramOffsets[3];
    BYTE                          types[6];
} EventSequenceComplete_COMMAND_DESCRIPTOR_t;

typedef TPM_RC (EventSequenceComplete_Entry)(
    EventSequenceComplete_In *in,
    EventSequenceComplete_Out *out
);

EventSequenceComplete_COMMAND_DESCRIPTOR_t _EventSequenceCompleteData = {
    /* entry */ &TPM2_EventSequenceComplete,
    /* inSize */ (UINT16)(sizeof(EventSequenceComplete_In)),
    /* outSize */ (UINT16)(sizeof(EventSequenceComplete_Out)),
    /* offsetOfTypes */ offsetof(EventSequenceComplete_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {(UINT16)(offsetof(EventSequenceComplete_In, buffer))},
    /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_MAX_BUFFER_P_UNMARSHAL,
TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
END_OF_LIST,
TPM2B_DIGEST_P_MARSHAL,
TPMT_TK_HASHCHECK_P_MARSHAL,
END_OF_LIST}
};
#define _EventSequenceCompleteDataAddress (&_EventSequenceCompleteData)
#define _SequenceCompleteDataAddress 0
#endif // CC_SequenceComplete

#include "EventSequenceComplete_fp.h"

define TPM_RC (SequenceComplete_Entry)(
    SequenceComplete_In *in,
    SequenceComplete_Out *out
);

define const struct {
    SequenceComplete_Entry *entry;
    UINT16                        inSize;
    UINT16                        outSize;
    UINT16                        offsetOfTypes;
    UINT16                        paramOffsets[3];
    BYTE                          types[7];
} SequenceComplete_COMMAND_DESCRIPTOR_t;

SequenceComplete_COMMAND_DESCRIPTOR_t _SequenceCompleteData = {
    /* entry */ &TPM2_SequenceComplete,
    /* inSize */ (UINT16)(sizeof(SequenceComplete_In)),
    /* outSize */ (UINT16)(sizeof(SequenceComplete_Out)),
    /* offsetOfTypes */ offsetof(SequenceComplete_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {(UINT16)(offsetof(SequenceComplete_In, buffer))},
    /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_MAX_BUFFER_P_UNMARSHAL,
TPMI_RH_HIERARCHY_P_UNMARSHAL + ADD_FLAG,
END_OF_LIST,
END_OF_LIST}
};
#define _SequenceCompleteDataAddress 0
#endif // CC_SequenceUpdate

#include "SequenceComplete_fp.h"

define TPM_RC (SequenceComplete_Entry)(
    SequenceComplete_In *in,
    SequenceComplete_Out *out
);

define const struct {
    SequenceComplete_Entry *entry;
    UINT16                        inSize;
    UINT16                        outSize;
    UINT16                        offsetOfTypes;
    UINT16                        paramOffsets[3];
    BYTE                          types[7];
} SequenceUpdate_COMMAND_DESCRIPTOR_t;

SequenceUpdate_COMMAND_DESCRIPTOR_t _SequenceUpdateData = {
    /* entry */ offsetOfTypes,/* offsets */ {(UINT16)(offsetof(SequenceUpdate_In, buffer))},
    /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_MAX_BUFFER_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST}
};
#define _SequenceUpdateDataAddress (&_SequenceUpdateData)
/* offsets       */
{(UINT16)(offsetof(EventSequenceComplete_In,
sequenceHandle)),
(UINT16)(offsetof(EventSequenceComplete_In,
buffer))},
/* types         */
{TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_MAX_BUFFER_P_UNMARSHAL,
END_OF_LIST,
TPML_DIGEST_VALUES_P_MARSHAL,
END_OF_LIST
};
#define _EventSequenceCompleteDataAddress (&_EventSequenceCompleteData)
#else
#define _EventSequenceCompleteDataAddress 0
#endif // CC_EventSequenceComplete
#if CC_Certify
#include "Certify_fp.h"
typedef TPM_RC  (Certify_Entry)(
Certify_In                  *in,
Certify_Out                 *out
);
typedef const struct {
    Certify_Entry           *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[4];
    BYTE                    types[8];
} Certify_COMMAND_DESCRIPTOR_t;
Certify_COMMAND_DESCRIPTOR_t _CertifyData = {
/* entry         */&TPM2_Certify,
/* inSize        */(UINT16)(sizeof(Certify_In)),
/* outSize       */(UINT16)(sizeof(Certify_Out)),
/* offsetOfTypes */(UINT16)(offsetof(Certify_COMMAND_DESCRIPTOR_t, types)),
/* offsets       */{(UINT16)(offsetof(Certify_In, signHandle)),
(UINT16)(offsetof(Certify_In, qualifyingData)),
(UINT16)(offsetof(Certify_In, inScheme)),
(UINT16)(offsetof(Certify_Out, signature)))},
/* types         */{TPMI_DH_OBJECT_H_UNMARSHAL,
TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
TPM2B_DATA_P_UNMARSHAL,
TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
END_OF_LIST,
TPM2B_ATTEST_P_MARSHAL,
TPMT_SIGNATURE_P_MARSHAL,
END_OF_LIST
};
#define _CertifyDataAddress (&_CertifyData)
#else
#define _CertifyDataAddress 0
#endif // CC_Certify
#if CC_CertifyCreation
#include "CertifyCreation_fp.h"
typedef TPM_RC  (CertifyCreation_Entry)(
CertifyCreation_In          *in,
CertifyCreation_Out         *out
);
typedef const struct {
    CertifyCreation_Entry   *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[6];
    BYTE                    types[10];
} CertifyCreation_COMMAND_DESCRIPTOR_t;
CertifyCreation_COMMAND_DESCRIPTOR_t _CertifyCreationData = {

/* entry */ &TPM2_CertifyCreation,
/* inSize */ (UINT16)(sizeof(CertifyCreation_In)),
/* outSize */ (UINT16)(sizeof(CertifyCreation_Out)),
/* offsetOfTypes */ offsetof(CertifyCreation_COMMAND_DESCRIPTOR_t, types),
/* offsets */ {
  (UINT16)(offsetof(CertifyCreation_In, objectHandle)),
  (UINT16)(offsetof(CertifyCreation_In, qualifyingData)),
  (UINT16)(offsetof(CertifyCreation_In, creationHash)),
  (UINT16)(offsetof(CertifyCreation_In, inScheme)),
  (UINT16)(offsetof(CertifyCreation_In, creationTicket)),
  (UINT16)(offsetof(CertifyCreation_Out, signature))},
/* types */ {
  TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
  TPMI_DH_OBJECT_H_UNMARSHAL,
  TPM2B_DATA_P_UNMARSHAL,
  TPM2B_DIGEST_P_UNMARSHAL,
  TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
  TPMT_TK_CREATION_P_UNMARSHAL,
  END_OF_LIST,
  TPM2B_ATTEST_P_MARSHAL,
  TPMT_SIGNATURE_P_MARSHAL,
  END_OF_LIST
};
#define _CertifyCreationDataAddress (&_CertifyCreationData)
#else
#define _CertifyCreationDataAddress 0
#endif // CC_CertifyCreation
#if CC_Quote
#include "Quote_fp.h"
typedef TPM_RC (Quote_Entry)(
  Quote_In                    *in,
  Quote_Out                   *out
);
typedef const struct {
  Quote_Entry             *entry;
  UINT16                  inSize;
  UINT16                  outSize;
  UINT16                  offsetOfTypes;
  UINT16                  paramOffsets[4];
  BYTE                    types[8];
} Quote_COMMAND_DESCRIPTOR_t;
Quote_COMMAND_DESCRIPTOR_t _QuoteData = {
  /* entry */ &TPM2_Quote,
  /* inSize */ (UINT16)(sizeof(Quote_In)),
  /* outSize */ (UINT16)(sizeof(Quote_Out)),
  /* offsetOfTypes */ offsetof(Quote_COMMAND_DESCRIPTOR_t, types),
  /* offsets */ {
    (UINT16)(offsetof(Quote_In, qualifyingData)),
    (UINT16)(offsetof(Quote_In, inScheme)),
    (UINT16)(offsetof(Quote_In, PCRselect)),
    (UINT16)(offsetof(Quote_Out, signature))},
  /* types */ {
    TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
    TPM2B_DATA_P_UNMARSHAL,
    TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
    TPML_PCR_SELECTION_P_UNMARSHAL,
    END_OF_LIST,
    TPM2B_ATTEST_P_MARSHAL,
    TPMT_SIGNATURE_P_MARSHAL,
    END_OF_LIST
};
#define _QuoteDataAddress (&_QuoteData)
#else
#define _QuoteDataAddress 0
#endif // CC_Quote
#if CC_GetSessionAuditDigest
#include "GetSessionAuditDigest_fp.h"
typedef TPM_RC (GetSessionAuditDigest_Entry)(
  GetSessionAuditDigest_In        *in,
  GetSessionAuditDigest_Out      *out
typedef const struct {
    GetSessionAuditDigest_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[5];
    BYTE types[9];
} GetSessionAuditDigest_COMMAND_DESCRIPTOR_t;

GetSessionAuditDigest_COMMAND_DESCRIPTOR_t _GetSessionAuditDigestData = {
    /* entry */ &TPM2_GetSessionAuditDigest,
    /* inSize */ (UINT16) (offsetof(GetSessionAuditDigest_In, signHandle)),
    /* outSize */ (UINT16) (offsetof(GetSessionAuditDigest_In, sessionHandle)),
    /* offsetOfTypes */ offsetof(GetSessionAuditDigest_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16) (offsetof(GetSessionAuditDigest_In, qualifyingData))},
    /* types */ {
        TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
        TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
        TPMI_SH_HMAC_H_UNMARSHAL,
        TPM2B_DATA_P_UNMARSHAL,
        TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
        END_OF_LIST,
        TPM2B_ATTEST_P_MARSHAL,
        TPMT_SIGNATURE_P_MARSHAL,
        END_OF_LIST
    }
};

#define _GetSessionAuditDigestDataAddress (&_GetSessionAuditDigestData)
#else
#define _GetSessionAuditDigestDataAddress 0
#endif

/* GetCommandAuditDigest */
typedef TPM_RC (GetCommandAuditDigest_Entry)(
    GetCommandAuditDigest_In *in,
    GetCommandAuditDigest_Out *out
);
(UINT16)(offsetof(GetCommandAuditDigest_Out, signature))},

    /* types */
    {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
     TPM2_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
     TPM2B_DATA_P_UNMARSHAL,
     TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
     END_OF_LIST,
     TPM2B_ATTEST_P_MARSHAL,
     TPMT_SIGNATURE_P_MARSHAL,
     END_OF_LIST}
};
#define _GetCommandAuditDigestDataAddress (_GetCommandAuditDigestData)
#else
#define _GetCommandAuditDigestDataAddress 0
#endif
// CC_GetCommandAuditDigest

#define _GetTimeDataAddress (&_GetTimeData)
#else
#define _GetTimeDataAddress 0
#define _GetTimeDataAddress 0
#endif
// CC_GetTime

#define _CertifyX509DataAddress (&_CertifyX509Data)
#else
#define _CertifyX509DataAddress 0
#endif
// CC_CertifyX509

typedef TPM_RC (CertifyX509_Entry)(
    CertifyX509_In *in,
    CertifyX509_Out *out
);
typedef const struct {
    CertifyX509_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[6];
    BYTE types[10];
} CertifyX509_COMMAND_DESCRIPTOR_t;

CertifyX509_COMMAND_DESCRIPTOR_t _CertifyX509Data = {
    /* entry */ &TPM2_CertifyX509,
    /* inSize */ (UINT16)(sizeof(CertifyX509_In)),
    /* outSize */ (UINT16)(sizeof(CertifyX509_Out)),
    /* offsetOfTypes */ (UINT16)(offsetof(CertifyX509_COMMAND_DESCRIPTOR_t, types)),
    /* offsets */
    /* types */
    {TPMI_RH_ENDORSEMENT_H_UNMARSHAL,
     TPM2_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
     TPM2B_DATA_P_UNMARSHAL,
     TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
     END_OF_LIST,
     TPM2B_ATTEST_P_MARSHAL,
     TPMT_SIGNATURE_P_MARSHAL,
     END_OF_LIST}
};
#define _CertifyX509DataAddress (_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 */  {
1739     (UINT16)(offsetof(CertifyX509_In, signHandle)),
1740     (UINT16)(offsetof(CertifyX509_In, qualifyingData)),
1741     (UINT16)(offsetof(CertifyX509_In, inScheme)),
1742     (UINT16)(offsetof(CertifyX509_Out, partialCertificate)),
1743     (UINT16)(offsetof(CertifyX509_Out, tbsDigest)),
1744     (UINT16)(offsetof(CertifyX509_Out, signature))},
1745 /* types */    {
1746     TPMI_DH_OBJECT_H_UNMARSHAL,
1747     TPM2B_DATA_P_UNMARSHAL,
1748     TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
1749     TPM2B_MAX_BUFFER_P_UNMARSHAL,
1750     END_OF_LIST,
1751     TPM2B_MAX_BUFFER_P_MARSHAL,
1752     TPM2B_DIGEST_P_MARSHAL,
1753     TPMT_SIGNATURE_P_MARSHAL,
1754     END_OF_LIST}
1755 
1756 #define _CertifyX509DataAddress (&_CertifyX509Data)
1757 #else
1758 #define _CertifyX509DataAddress 0
1759 #endif /* CC_CertifyX509 */
1760 #if CC_Commit
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 */  {
1779     (UINT16)(offsetof(Commit_In, P1)),
1780     (UINT16)(offsetof(Commit_In, s2)),
1781     (UINT16)(offsetof(Commit_In, y2)),
1782     (UINT16)(offsetof(Commit_Out, L)),
1783     (UINT16)(offsetof(Commit_Out, E)),
1784     (UINT16)(offsetof(Commit_Out, counter))},
1785     /* types */    {
1786     TPMI_DH_OBJECT_H_UNMARSHAL,
1787     TPM2B_ECC_POINT_P_UNMARSHAL,
1788     TPM2B_SENSITIVE_DATA_P_UNMARSHAL,
1789     TPM2B_ECC_PARAMETER_P_UNMARSHAL,
1790     END_OF_LIST,
1791     TPM2B_ECC_POINT_P_MARSHAL,
1792     TPM2B_ECC_POINT_P_MARSHAL,
1793     TPM2B_ECC_POINT_P_MARSHAL,
1794     UINT16_P_MARSHAL,
1795     END_OF_LIST},
1796 
1797 #define _CommitDataAddress (&_CommitData)
1798 #else
1799 #define _CommitDataAddress 0
1800 #endif /* CC_Commit */
1801 #if CC_EC_Ephemeral
```c
#include "EC_Ephemeral_fp.h"

typedef TPM_RC (EC_Ephemeral_Entry)(
    EC_Ephemeral_In             *in,
    EC_Ephemeral_Out            *out
);

typedef const struct {
    EC_Ephemeral_Entry         *entry;
    UINT16                      inSize;
    UINT16                      outSize;
    UINT16                      offsetOfTypes;
    UINT16                      paramOffsets[1];
    BYTE                        types[5];
} EC_Ephemeral_COMMAND_DESCRIPTOR_t;

EC_Ephemeral_COMMAND_DESCRIPTOR_t _EC_EphemeralData = {
    /* entry         */ &TPM2_EC_Ephemeral,
    /* inSize        */ (UINT16)(sizeof(EC_Ephemeral_In)),
    /* outSize       */ (UINT16)(sizeof(EC_Ephemeral_Out)),
    /* offsetOfTypes */ offsetof(EC_Ephemeral_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ {(UINT16)(offsetof(EC_Ephemeral_Out, counter))},
    /* types         */ {TPMI_ECC_CURVE_P_UNMARSHAL,
                             TPM2B_ECC_POINT_P_MARSHAL,
                             UINT16_P_MARSHAL,
                             TPM2B_ECC_POINT_P_MARSHAL,
                             UINT16_P_MARSHAL,
                             TPM2B_ECC_POINT_P_MARSHAL,
                             TPM2B_ECC_POINT_P_MARSHAL,
                             END_OF_LIST
};

#define _EC_EphemeralDataAddress (&_EC_EphemeralData)

typedef TPM_RC (VerifySignature_Entry)(
    VerifySignature_In          *in,
    VerifySignature_Out         *out
);

typedef const struct {
    VerifySignature_Entry       *entry;
    UINT16                      inSize;
    UINT16                      outSize;
    UINT16                      offsetOfTypes;
    UINT16                      paramOffsets[2];
    BYTE                        types[6];
} VerifySignature_COMMAND_DESCRIPTOR_t;

VerifySignature_COMMAND_DESCRIPTOR_t _VerifySignatureData = {
    /* entry         */ &TPM2_VerifySignature,
    /* inSize        */ (UINT16)(sizeof(VerifySignature_In)),
    /* outSize       */ (UINT16)(sizeof(VerifySignature_Out)),
    /* offsetOfTypes */ offsetof(VerifySignature_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ {(UINT16)(offsetof(VerifySignature_In, digest))},
    /* types         */ {TPMI_DH_OBJECT_H_UNMARSHAL,
                             TPM2B_DIGEST_P_UNMARSHAL,
                             TPMT_SIGNATURE_P_UNMARSHAL,
                             TPM2B_DIGEST_P_MARSHAL,
                             TPM2B_DIGEST_P_MARSHAL,
                             END_OF_LIST
};

#define _VerifySignatureDataAddress (&_VerifySignatureData)
```

```c
#include "Sign_fp.h"

typedef TPM_RC (Sign_Entry)(
    Sign_In                     *in,
    Sign_Out                    *out
);```
typedef const struct {
    Sign_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[3];
    BYTE types[7];
} Sign_COMMAND_DESCRIPTOR_t;

Sign_COMMAND_DESCRIPTOR_t _SignData = {
    /* entry */ &TPM2_Sign,
    /* inSize */ (UINT16)(sizeof(Sign_In)),
    /* outSize */ (UINT16)(sizeof(Sign_Out)),
    /* offsetOfTypes */ offsetof(Sign_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(Sign_In, digest)),
        (UINT16)(offsetof(Sign_In, inScheme)),
        (UINT16)(offsetof(Sign_In, validation))},
    /* types */ {
        TPMI_DH_OBJECT_H_UNMARSHAL,
        TPM2B_DIGEST_P_UNMARSHAL,
        TPM_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
        TPM_TK_HASHCHECK_P_UNMARSHAL,
        END_OF_LIST,
        TPM_SIGNATURE_P_MARSHAL,
        END_OF_LIST}
};

#define _SignDataAddress (&_SignData)

#endif // CC_Sign

#endif // CC_SetCommandCodeAuditStatus

typedef TPM_RC (SetCommandCodeAuditStatus_Entry)(
    SetCommandCodeAuditStatus_In *in
);
#if CC_PCR_Extend
#include "PCR_Extend_fp.h"
typedef TPM_RC (PCR_Extend_Entry)(
    PCR_Extend_In               *in,
    PCR_Extend_Out               *out
);
typedef const struct {
    PCR_Extend_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[4];
} PCR_Extend_COMMAND_DESCRIPTOR_t;
PCR_Extend_COMMAND_DESCRIPTOR_t _PCR_ExtendData = {
    /* entry */ &TPM2_PCR_Extend,
    /* inSize */ (UINT16)(sizeof(PCR_Extend_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetof(PCR_Extend_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {(UINT16)(offsetof(PCR_Extend_In, digests))},
    /* types */ {
        TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
        TPML_DIGEST_VALUES_P_UNMARSHAL,
        END_OF_LIST,
    }
};
#define _PCR_ExtendDataAddress (&_PCR_ExtendData)
#else
#define _PCR_ExtendDataAddress 0
#endif // CC_PCR_Extend

#if CC_PCR_Event
#include "PCR_Event_fp.h"
typedef TPM_RC (PCR_Event_Entry)(
    PCR_Event_In                *in,
    PCR_Event_Out               *out
);
typedef const struct {
    PCR_Event_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[5];
} PCR_Event_COMMAND_DESCRIPTOR_t;
PCR_Event_COMMAND_DESCRIPTOR_t _PCR_EventData = {
    /* entry */ &TPM2_PCR_Event,
    /* inSize */ (UINT16)(sizeof(PCR_Event_In)),
    /* outSize */ (UINT16)(sizeof(PCR_Event_Out)),
    /* offsetOfTypes */ offsetof(PCR_Event_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {(UINT16)(offsetof(PCR_Event_In, eventData))},
    /* types */ {
        TPMI_DH_PCR_H_UNMARSHAL + ADD_FLAG,
        TPM2B_EVENT_P_UNMARSHAL,
        END_OF_LIST,
        TPML_DIGEST_VALUES_P_MARSHAL,
        END_OF_LIST}
};
#define _PCR_EventDataAddress (&_PCR_EventData)
#else
#define _PCR_EventDataAddress 0
#endif // CC_PCR_Event

#if CC_PCR_Read
#include "PCR_Read_fp.h"
typedef TPM_RC (PCR_Read_Entry)(
    PCR_Read_In *in,
    PCR_Read_Out *out
);
typedef const struct {
    PCR_Read_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[4];
} PCR_Read_COMMAND_DESCRIPTOR_t;
PCR_Read_COMMAND_DESCRIPTOR_t _PCR_ReadData = {
    /* entry */ &TPM2_PCR_Read,
    /* inSize */ (UINT16)(sizeof(PCR_Read_In)),
    /* outSize */ (UINT16)(sizeof(PCR_Read_Out)),
    /* offsetOfTypes */ offsetof(PCR_Read_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {(UINT16)(offsetof(PCR_Read_In, data))},
    /* types */ {
        TPML_DIGEST_VALUES_P_UNMARSHAL,
        END_OF_LIST,
    }
};
#define _PCR_ReadDataAddress (&_PCR_ReadData)
#else
#define _PCR_ReadDataAddress 0
#endif // CC_PCR_Read
```c
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 */ {
2005    (UINT16)(offsetof(PCR_Read_Out, pcrSelectionOut)),
2006    (UINT16)(offsetof(PCR_Read_Out, pcrValues))},
2007 /* types */ {TPML_PCR_SELECTION_P_UNMARSHAL,
2008    END_OF_LIST,
2009    UINT32_P_MARSHAL,
2010    TPML_PCR_SELECTION_P_MARSHAL,
2011    TPML_DIGEST_P_MARSHAL,
2012    END_OF_LIST}
2013 }
2014 #define _PCR_ReadDataAddress (&_PCR_ReadData)
2015 #else
2016 #define _PCR_ReadDataAddress 0
2017 #endif  // CC_PCR_Read
2018 #ifdef  // CC_PCR_Allocate
2019 #include "PCR_Allocate_fp.h"
2020 typedef TPM_RC (PCR_Allocate_Entry)(
2021 PCR_Allocate_In *in,
2022 PCR_Allocate_Out *out
2023 );
2024 typedef const struct {
2025     PCR_Allocate_Entry  *entry;
2026  UINT16         inSize;
2027  UINT16         outSize;
2028  UINT16         offsetOfTypes;
2029  UINT16         paramOffsets[4];
2030  BYTE                types[8];
2031 } PCR_Allocate_COMMAND_DESCRIPTOR_t;
2032 PCR_Allocate_COMMAND_DESCRIPTOR_t _PCR_AllocateData = {
2033 /* entry */ &TPM2_PCR_Allocate,
2034 /* inSize */ (UINT16)(sizeof(PCR_Allocate_In)),
2035 /* outSize */ (UINT16)(sizeof(PCR_Allocate_Out)),
2036 /* offsetOfTypes */ offsetof(PCR_Allocate_COMMAND_DESCRIPTOR_t, types),
2037 /* offsets */ {
2038    (UINT16)(offsetof(PCR_Allocate_In, pcrAllocation)),
2039    (UINT16)(offsetof(PCR_Allocate_Out, maxPCR)),
2040    (UINT16)(offsetof(PCR_Allocate_Out, sizeNeeded)),
2041    (UINT16)(offsetof(PCR_Allocate_Out, sizeAvailable))},
2042 /* types */ {TPMI_RH_PLATFORM_H_UNMARSHAL,
2043    END_OF_LIST,
2044    TPMI_YES_NO_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 {
```
```c
2059  PCRL_SetAuthPolicy_Entry *entry;
2060  UINT16 inSize;
2061  UINT16 outSize;
2062  UINT16 offsetOfTypes;
2063  UINT16 paramOffsets[3];
2064  BYTE types[6];
2065 } PCRL_SetAuthPolicyCOMMAND_DESCRIPTOR_t;
2066 PCRL_SetAuthPolicyCOMMAND_DESCRIPTOR_t _PCRL_SetAuthPolicyData = {
2067  /* entry */ &TPM2_PCR_SetAuthPolicy,
2068  /* inSize */ (UINT16)(sizeof(PCRL_SetAuthPolicy_In)),
2069  /* outSize */ 0,
2070  /* offsetOfTypes */ offsetof(PCRL_SetAuthPolicy_COMMAND_DESCRIPTOR_t,
2071  types),
2072  /* offsets */ {(UINT16)(offsetof(PCRL_SetAuthPolicy_In, authPolicy)),
2073  (UINT16)(offsetof(PCRL_SetAuthPolicy_In, hashAlg)),
2074  (UINT16)(offsetof(PCRL_SetAuthPolicy_In, pcrNum))},
2075  /* types */ {TPMI_RH_PLATFORM_H_UNMARSHAL,
2076  TPM2B_DIGEST_P_UNMARSHAL,
2077  TPM2_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
2078  TPM2_DH_PCR_P_UNMARSHAL,
2079  END_OF_LIST,
2080  END_OF_LIST}"
2081 #define _PCRL_SetAuthPolicyDataAddress (_PCRL_SetAuthPolicyData)
2082 #else
2083 #define _PCRL_SetAuthPolicyDataAddress 0
2084 #endif // CC_PCR_SetAuthPolicy
2085 #if CC_PCR_SetAuthValue
2086 #include "PCR_SetAuthValue_fp.h"
2087 typedef TPM_RC (PCRL_SetAuthValue_Entry)(
2088  PCRL_SetAuthValue_In *in
2089 ) ;
2090 typedef const struct {
2091  PCRL_SetAuthValue_Entry *entry;
2092  UINT16 inSize;
2093  UINT16 outSize;
2094  UINT16 offsetOfTypes;
2095  UINT16 paramOffsets[1];
2096  BYTE types[4];
2097 } PCRL_SetAuthValue_COMMAND_DESCRIPTOR_t;
2098 PCRL_SetAuthValue_COMMAND_DESCRIPTOR_t _PCRL_SetAuthValueData = {
2099  /* entry */ &TPM2_PCR_SetAuthValue,
2100  /* inSize */ (UINT16)(sizeof(PCRL_SetAuthValue_In)),
2101  /* outSize */ 0,
2102  /* offsetOfTypes */ offsetof(PCRL_SetAuthValue_COMMAND_DESCRIPTOR_t, types),
2103  /* offsets */ {(UINT16)(offsetof(PCRL_SetAuthValue_In, auth))},
2104  /* types */ {TPMI_DH_PCR_H_UNMARSHAL,
2105  TPM2B_DIGEST_P_UNMARSHAL,
2106  END_OF_LIST,
2107  END_OF_LIST}"
2108 #define _PCRL_SetAuthValueDataAddress (_PCRL_SetAuthValueData)
2109 #else
2110 #define _PCRL_SetAuthValueDataAddress 0
2111 #endif // CC_PCR_SetAuthValue
2112 #if CC_PCR_Reset
2113 #include "PCR_Reset_fp.h"
2114 typedef TPM_RC (PCR_Reset_Entry)(
2115  PCR_Reset_In *in
2116 ) ;
2117 typedef const struct {
2118  PCR_Reset_Entry *entry;
2119  UINT16 inSize;
2120  UINT16 outSize;
2121  UINT16 offsetOfTypes;
2122  BYTE types[3];
2123 } PCR_Reset_COMMAND_DESCRIPTOR_t;
2124 PCR_Reset_COMMAND_DESCRIPTOR_t _PCR_ResetData = {
2125  /* entry */ &TPM2_PCR_Reset,
2126  /* inSize */ (UINT16)(sizeof(PCR_Reset_In)),
2127  /* outSize */ 0,
2128  /* offsetOfTypes */ offsetof(PCR_Reset_COMMAND_DESCRIPTOR_t, types),
2129  /* offsets */ {(UINT16)(offsetof(PCR_Reset_In, auth))},
2130  /* types */ {TPMI_DH_PCR_H_UNMARSHAL,
2131  TPM2B_DIGEST_P_UNMARSHAL,
2132  END_OF_LIST,
2133  END_OF_LIST}"
2134 #define _PCR_ResetDataAddress (_PCR_ResetData)
2135 #else
2136 #define _PCR_ResetDataAddress 0
2137 #endif // CC_PCR_Reset
```
```c
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 #ifdef /* CC_PCR_Reset */
2137 #define _PCR_ResetDataAddress 0
2138 #endif
2139 #include "PolicySigned_fp.h"
2140 typedef TPM_RC (PolicySigned_Entry)(
2141 PolicySigned_In *in,
2142 PolicySigned_Out *out
2143 );
2144 typedef const struct {
2145 PolicySigned_Entry *entry;
2146 UINT16 inSize;
2147 UINT16 outSize;
2148 UINT16 offsetOfTypes;
2149 UINT16 paramOffsets[7];
2150 BYTE types[11];
2151 } PolicySigned_COMMAND_DESCRIPTOR_t;
2152 PolicySigned_COMMAND_DESCRIPTOR_t _PolicySignedData = {
2153 /* entry */ &TPM2_PolicySigned,
2154 /* inSize */ (UINT16)(sizeof (PolicySigned_In)),
2155 /* outSize */ (UINT16)(sizeof (PolicySigned_Out)),
2156 /* offsetOfTypes */ offsetof(PolicySigned_COMMAND_DESCRIPTOR_t, types),
2157 /* offsets */ {(UINT16)(offsetof(PolicySigned_In, policySession)),
2158 (UINT16)(offsetof(PolicySigned_In, nonceTPM)),
2159 (UINT16)(offsetof(PolicySigned_In, cpHashA)),
2160 (UINT16)(offsetof(PolicySigned_In, policyRef)),
2161 (UINT16)(offsetof(PolicySigned_In, expiration)),
2162 (UINT16)(offsetof(PolicySigned_In, auth)),
2163 (UINT16)(offsetof(PolicySigned_Out, policyTicket))},
2164 /* types */ {TPMI_DH_OBJECT_H_UNMARSHAL,
2165 TPMI_SH_POLICY_H_UNMARSHAL,
2166 TPM2B_NONCE_P_UNMARSHAL,
2167 TPM2B_DIGEST_P_UNMARSHAL,
2168 TPM2B_NONCE_P_UNMARSHAL,
2169 INT32_P_UNMARSHAL,
2170 TPMT_SIGNATURE_P_UNMARSHAL,
2171 TPMT_TIMEOUT_P_MARSHAL,
2172 END_OF_LIST,
2173 END_OF_LIST)
2174 );
2175 #define _PolicySignedDataAddress (&_PolicySignedData)
2176 #ifdef /* CC_PolicySigned */
2177 #define _PolicySignedDataAddress 0
2178 #endif
2179 #include "PolicySecret_fp.h"
2180 typedef TPM_RC (PolicySecret_Entry)(
2181 PolicySecret_In *in,
2182 PolicySecret_Out *out
2183 );
2184 typedef const struct {
2185 PolicySecret_Entry *entry;
2186 UINT16 inSize;
```
Trusted Platform Module Library

Part 4: Supporting Routines

Page 58

TCG PUBLIC REVIEW

Family “2.0”

April 15, 2019

Copyright © TCG 2006-2019

Level 00 Revision 01.55

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        INT32_P_UNMARSHAL,
2211        TPM2B_TIMEOUT_P_MARSHAL,
2212        TPMT_TK_AUTH_P_MARSHAL,
2213        END_OF_LIST,
2214        END_OF_LIST}},
2215
2216 };#define _PolicySecretDataAddress (&_PolicySecretData)
2217 #else
2218 #define _PolicySecretDataAddress 0
2219 #endif
2220 // 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
typedef TPM_RC (PolicyOR_Entry)(
    PolicyOR_In *in
);

typedef const struct {
    PolicyOR_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[4];
} PolicyOR_COMMAND_DESCRIPTOR_t;

PolicyOR_COMMAND_DESCRIPTOR_t _PolicyORData = {
    /* entry */ &TPM2_PolicyOR,
    /* inSize */ (UINT16)(sizeof(PolicyOR_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetOf(PolicyOR_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(PolicyOR_In, pHashList))},
    /* types */ {
        TPMI_SH_POLICY_H_UNMARSHAL,
        TPML_DIGEST_P_UNMARSHAL,
        END_OF_LIST,
        END_OF_LIST}
};

#define _PolicyORDataAddress (&_PolicyORData)

#endif // CC_PolicyOR

#endif // CC_PolicyPCR

#endif // CC_PolicyLocality

} PolicyOR_COMMAND_DESCRIPTOR_t;

PolicyOR_COMMAND_DESCRIPTOR_t _PolicyORData = {
    /* entry */ &TPM2_PolicyOR,
    /* inSize */ (UINT16)(sizeof(PolicyOR_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetOf(PolicyOR_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(PolicyOR_In, pHashList))},
    /* types */ {
        TPMI_SH_POLICY_H_UNMARSHAL,
        TPM2B_DIGEST_P_UNMARSHAL,
        TPM2_PCR_SELECTION_P_UNMARSHAL,
        END_OF_LIST,
        END_OF_LIST}
};

#define _PolicyPCRDataAddress (&_PolicyPCRData)

#endif // CC_PolicyPCR

} PolicyPCR_COMMAND_DESCRIPTOR_t;

PolicyPCR_COMMAND_DESCRIPTOR_t _PolicyPCRData = {
    /* entry */ &TPM2_PolicyPCR,
    /* inSize */ (UINT16)(sizeof(PolicyPCR_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetOf(PolicyPCR_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(PolicyPCR_In, pcrDigest))},
    /* types */ {
        TPMI_SH_POLICY_H_UNMARSHAL,
        TPM2B_DIGEST_P_UNMARSHAL,
        TPM2_PCR_SELECTION_P_UNMARSHAL,
        END_OF_LIST,
        END_OF_LIST}
};

#define _PolicyPCRDDataAddress (&_PolicyPCRDData)

#else
#endif // CC_PolicyLocality

} PolicyLOCALITY_COMMAND_DESCRIPTOR_t;

PolicyLOCALITY_COMMAND_DESCRIPTOR_t _PolicyPCRDData = {
    /* entry */ &TPM2_PolicyLocality,
    /* inSize */ (UINT16)(sizeof(PolicyPCRDData_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetOf(PolicyLOCALITY_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(PolicyPCRDData_In, pHashList))},
    /* types */ {
        TPMI_SH_POLICY_H_UNMARSHAL,
        TPM2B_DIGEST_P_UNMARSHAL,
        TPM2_PCR_SELECTION_P_UNMARSHAL,
        END_OF_LIST,
        END_OF_LIST}
};

#define _PolicyLocalityDataAddress (&_PolicyPCRDData)
UINT16 inSize;
UINT16 outSize;
UINT16 offsetsOfTypes;
UINT16 paramOffsets[1];
BYTE types[4];

PolicyLocality_COMMAND_DESCRIPTOR_t _PolicyLocalityData = {
  /* entry */ &TPM2_PolicyLocality,
  /* inSize */ (UINT16)(sizeof(PolicyLocality_In)),
  /* outSize */ 0,
  /* offsetOfTypes */ offsetOfPolicyLocality_COMMAND_DESCRIPTOR_t, types,
  /* offsets */ {(UINT16)(offsetof(PolicyLocality_In, locality))},
  /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
                TPMA_LOCALITY_P_UNMARSHAL,
                END_OF_LIST, END_OF_LIST} 
};
#define _PolicyLocalityDataAddress (&_PolicyLocalityData)
#endif // CC_PolicyLocality

#elif CC_PolicyNV
#include "PolicyNV_fp.h"

typedef TPM_RC (PolicyNV_Entry)(
  PolicyNV_In *in 
);

typedef const struct 
{
  PolicyNV_Entry      *entry;
  UINT16              inSize;
  UINT16              outSize;
  UINT16              offsetsOfTypes;
  UINT16              paramOffsets[5];
  BYTE                types[8];
} PolicyNV_COMMAND_DESCRIPTOR_t;

PolicyNV_COMMAND_DESCRIPTOR_t _PolicyNVData = {
  /* entry */ &TPM2_PolicyNV,
  /* inSize */ (UINT16)(sizeof(PolicyNV_In)),
  /* outSize */ 0,
  /* offsetOfTypes */ offsetOfPolicyNV_COMMAND_DESCRIPTOR_t, types,
  /* offsets */ {(UINT16)(offsetof(PolicyNV_In, nvIndex)),
                 (UINT16)(offsetof(PolicyNV_In, policySession)),
                 (UINT16)(offsetof(PolicyNV_In, operandB)),
                 (UINT16)(offsetof(PolicyNV_In, offset)),
                 (UINT16)(offsetof(PolicyNV_In, operation))},
  /* types */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,
                TPM2B_OPERAND_P_UNMARSHAL,
                TPM_EO_P_UNMARSHAL,
                TPM16_P_UNMARSHAL,
                TPM2B_OPERAND_P_UNMARSHAL,
                END_OF_LIST, END_OF_LIST} 
};
#define _PolicyNVDataAddress (&_PolicyNVData)
#elif
#endif // CC_PolicyNV

#endif // CC_PolicyCounterTimer

#include "PolicyCounterTimer_fp.h"

typedef TPM_RC (PolicyCounterTimer_Entry)(
  PolicyCounterTimer_In *in 
);

typedef const struct 
{
  PolicyCounterTimer_Entry     *entry;
  UINT16              inSize;
  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,
2397      types),
2398      /* offsets */ {(UINT16)(offsetof(PolicyCounterTimer_In, operandB)),
2399      (UINT16)(offsetof(PolicyCounterTimer_In, offset)),
2400      (UINT16)(offsetof(PolicyCounterTimer_In, operation))},
2401      /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2402      TPM2B_OPERAND_P_UNMARSHAL,
2403      UINT16_P_UNMARSHAL,
2404      TPM_EO_P_UNMARSHAL,
2405      END_OF_LIST,
2406      END_OF_LIST};
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,
2429      types),
2430      /* offsets */ {(UINT16)(offsetof(PolicyCommandCode_In, code))},
2431      /* types */ {TPMI_SH_POLICY_H_UNMARSHAL,
2432      TPM2B_CC_P_UNMARSHAL,
2433      TPM_EO_P_UNMARSHAL,
2434      END_OF_LIST,
2435      END_OF_LIST};
2436 #define _PolicyCommandCodeDataAddress (&_PolicyCommandCodeData)
2437 #else
2438 #define _PolicyCommandCodeDataAddress 0
2439 #endif // CC_PolicyCommandCode
2440 #if CC_PolicyPhysicalPresence
2441 #include "PolicyPhysicalPresence_fp.h"
2442 typedef TPM_RC (PolicyPhysicalPresence_Entry)(
2443      PolicyPhysicalPresence_In *in
2444      );
2445 typedef const struct {
2446      PolicyPhysicalPresence_Entry *entry;
2447      UINT16       inSize;
2448      UINT16       outSize;
2449      UINT16       offsetOfTypes;
2450      BYTE         types[3];
2451 } PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t;
PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t _PolicyPhysicalPresenceData = {
    /* entry */ &TPM2_PolicyPhysicalPresence,
    /* inSize */ (UINT16)(sizeof(PolicyPhysicalPresence_In)),
    /* outSize */ 0,
    /* offsetOfTypes */
    offsetof(PolicyPhysicalPresence_COMMAND_DESCRIPTOR_t, types),
    /* offsets */  // No parameter offsets;
    /* types */
    {TPMI_SH_POLICY_H_UNMARSHAL,
     END_OF_LIST,
     END_OF_LIST}
};
#define _PolicyPhysicalPresenceDataAddress (&_PolicyPhysicalPresenceData)
#else
#define _PolicyPhysicalPresenceDataAddress 0
#endif // CC_PolicyPhysicalPresence
#if CC_PolicyCpHash
#include "PolicyCpHash_fp.h"
typedef TPM_RC (PolicyCpHash_Entry)(
    PolicyCpHash_In  *in
);
#endif // CC_PolicyCpHash
#if CC_PolicyNameHash
#include "PolicyNameHash_fp.h"
typedef TPM_RC (PolicyNameHash_Entry)(
    PolicyNameHash_In  *in
);
#endif // CC_PolicyNameHash
PolicyCpHash_COMMAND_DESCRIPTOR_t _PolicyCpHashData = {
    /* entry */ &TPM2_PolicyCpHash,
    /* inSize */ (UINT16)(sizeof(PolicyCpHash_In)),
    /* outSize */ 0,
    /* offsetOfTypes */
    offsetof(PolicyCpHash_COMMAND_DESCRIPTOR_t, types),
    /* offsets */
    {(UINT16)(offsetof(PolicyCpHash_In, cpHashA))},
    /* types */
    {TPMI_SH_POLICY_H_UNMARSHAL,
     TPM2B_DIGEST_P_UNMARSHAL,
     END_OF_LIST,
     END_OF_LIST}
};
#define _PolicyCpHashDataAddress (&_PolicyCpHashData)
#else
#define _PolicyCpHashDataAddress 0
#endif // CC_PolicyCpHash
#if CC_PolicyNameHash
#include "PolicyNameHash_fp.h"
typedef TPM_RC (PolicyNameHash_Entry)(
    PolicyNameHash_In  *in
);
#endif // CC_PolicyNameHash
PolicyNameHash_COMMAND_DESCRIPTOR_t _PolicyNameHashData = {
    /* entry */ &TPM2_PolicyNameHash,
    /* inSize */ (UINT16)(sizeof(PolicyNameHash_In)),
    /* outSize */ 0,
    /* offsetOfTypes */
    offsetof(PolicyNameHash_COMMAND_DESCRIPTOR_t, types),
    /* offsets */
    {(UINT16)(offsetof(PolicyNameHash_In, nameHash))},
    /* types */
    {TPMI_SH_POLICY_H_UNMARSHAL,
     TPM2B_DIGEST_P_UNMARSHAL,
     END_OF_LIST,
     END_OF_LIST}
};
#define _PolicyNameHashDataAddress (&_PolicyNameHashData)
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 */
2541   {(UINT16)(offsetof(PolicyDuplicationSelect_In, objectName)),
2542    (UINT16)(offsetof(PolicyDuplicationSelect_In, newParentName)),
2543    (UINT16)(offsetof(PolicyDuplicationSelect_In, includeObject))},
2544   /* types */
2545   {TPMI_SH_POLICY_H_UNMARSHAL,
2546    TPM2B_NAME_P_UNMARSHAL,
2547    TPM2B_NAME_P_UNMARSHAL,
2548    TPM2B_NAME_P_UNMARSHAL,
2549    TPM_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_PolicyAuthorize
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 */
2577   {(UINT16)(offsetof(PolicyAuthorize_In, approvedPolicy)),
2578    (UINT16)(offsetof(PolicyAuthorize_In, policyRef)),
2579    (UINT16)(offsetof(PolicyAuthorize_In, keySign)),
2580    (UINT16)(offsetof(PolicyAuthorize_In, checkTicket))},
2581   /* types */
2582   {TPMI_SH_POLICY_H_UNMARSHAL,
2583    TPM2B_DIGEST_P_UNMARSHAL,
2584    TPM2B_NONCE_P_UNMARSHAL,
```c
#define _PolicyAuthorizeDataAddress (&_PolicyAuthorizeData)
#else
#define _PolicyAuthorizeDataAddress 0
#endif
#endif
// CC_PolicyAuthorize

#define _PolicyAuthValueDataAddress (&_PolicyAuthValueData)
#else
#define _PolicyAuthValueDataAddress 0
#endif
// CC_PolicyAuthValue

#define _PolicyPasswordDataAddress (&_PolicyPasswordData)
#else
#define _PolicyPasswordDataAddress 0
#endif
// CC_PolicyPassword

#define _PolicyGetDigestDataAddress (&_PolicyGetDigestData)
#else
#define _PolicyGetDigestDataAddress 0
#endif
// CC_PolicyGetDigest
```
typedef const struct {
    PolicyGetDigest_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    BYTE types[4];
} PolicyGetDigest_COMMAND_DESCRIPTOR_t;
PolicyGetDigest_COMMAND_DESCRIPTOR_t _PolicyGetDigestData = {
    /* entry         */ &TPM2_PolicyGetDigest,
    /* inSize        */ (UINT16)(sizeof(PolicyGetDigest_In)),
    /* outSize       */ (UINT16)(sizeof(PolicyGetDigest_Out)),
    /* offsetOfTypes */ offset(PolicyGetDigest_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ /* No parameter offsets: */
    /* types         */ {TPMI_SH_POLICY_H_UNMARSHAL,
        TPM2B_DIGEST_P_MARSHAL,
        END_OF_LIST}
};

#define _PolicyGetDigestDataAddress (&_PolicyGetDigestData)
#endif // CC_PolicyGetDigest

#define _PolicyGetDigestDataAddress 0
#endif // CC_PolicyNvWritten

#include "PolicyNvWritten_fp.h"

typedef TPM_RC (PolicyNvWritten_Entry)(
    PolicyNvWritten_In *in
); typedef const struct {
    PolicyNvWritten_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[4];
} PolicyNvWritten_COMMAND_DESCRIPTOR_t;
PolicyNvWritten_COMMAND_DESCRIPTOR_t _PolicyNvWrittenData = {
    /* entry         */ &TPM2_PolicyNvWritten,
    /* inSize        */ (UINT16)(sizeof(PolicyNvWritten_In)),
    /* outSize       */ 0,
    /* offsetOfTypes */ offset(PolicyNvWritten_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ {(UINT16)(offsetof(PolicyNvWritten_In, writtenSet))},
    /* types         */ {TPMI_SH_POLICY_H_UNMARSHAL,
        TPMI_YES_NO_P_UNMARSHAL,
        END_OF_LIST} _
    END_OF_LIST
};
#define _PolicyNvWrittenDataAddress (&_PolicyNvWrittenData)
#if CC_PolicyTemplate
#include "PolicyTemplate_fp.h"

typedef TPM_RC (PolicyTemplate_Entry)(
    PolicyTemplate_In *in
); typedef const struct {
    PolicyTemplate_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[4];
} PolicyTemplate_COMMAND_DESCRIPTOR_t;
PolicyTemplate_COMMAND_DESCRIPTOR_t _PolicyTemplateData = {
    /* 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, types),
2741 /* offsets */ {{(UINT16)(offsetof(PolicyAuthorizeNV_In, nvIndex))},
2742 (UINT16)(offsetof(PolicyAuthorizeNV_In, policySession))},
2743 /* types */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,
2744 TPMI_RH_NV_INDEX_H_UNMARSHAL,
2745 TPMI_SH_POLICY_H_UNMARSHAL,
2746 END_OF_LIST,
2747 END_OF_LIST}
2748
2749 #define _PolicyAuthorizeNVDataAddress (&_PolicyAuthorizeNVData)
2750 #else
2751 #define _PolicyAuthorizeNVDataAddress 0
2752 #endif // CC_PolicyAuthorizeNV
2753 #if CC_CreatePrimary
2754 #include "CreatePrimary_fp.h"
2755 typedef TPM_RC (CreatePrimary_Entry)(
2756 CreatePrimary_In *in,
2757 CreatePrimary_Out *out
2758 );
2759 typedef const struct {
2760 CreatePrimary_Entry *entry;
2761 UINT16 inSize;
2762 UINT16 outSize;
2763 UINT16 offsetOfTypes;
2764 UINT16 paramOffsets[9];
2765 BYTE types[13];
2766 } CreatePrimary_COMMAND_DESCRIPTOR_t;
2767 CreatePrimary_COMMAND_DESCRIPTOR_t _CreatePrimaryData = {
2768 /* entry */ &TPM2_CreatePrimary,
2769 /* inSize */ (UINT16)(sizeof(CreatePrimary_In)),
2770 /* outSize */ (UINT16)(sizeof(CreatePrimary_Out)),
2771 /* offsetOfTypes */ offsetOf(CreatePrimary_COMMAND_DESCRIPTOR_t, types),
2772 /* offsets */ {{(UINT16)(offsetof(CreatePrimary_In, inSensitive))},
2773 (UINT16)(offsetof(CreatePrimary_In, inPublic)),
typedef TPM_RC (HierarchyControl_Entry)(
    HierarchyControl_In *in
);

typedef const struct {
    HierarchyControl_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[2];
    BYTE types[5];
} HierarchyControl_COMMAND_DESCRIPTOR_t;

HierarchyControl_COMMAND_DESCRIPTOR_t _HierarchyControlData = {
    /* entry */ &TPM2_HierarchyControl,
    /* inSize */ (UINT16)(sizeof(HierarchyControl_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetof(HierarchyControl_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(HierarchyControl_In, enable)),
        (UINT16)(offsetof(HierarchyControl_In, state))},
    /* types */ {
        TPM2B_SENSITIVE_CREATE_P_UNMARSHAL,
        TPM2B_PUBLIC_P_UNMARSHAL,
        TPM2B_DATA_P_UNMARSHAL,
        TPM2B_DIGEST_P_MARSHAL,
        TPM2B_CREATION_DATA_P_MARSHAL,
        TPM2B_PUBLIC_P_MARSHAL,
        TPM2B_NAME_P_MARSHAL,
        TPM2B_TK_CREATION_P_MARSHAL,
        TPM2B_SENSITIVE_CREATE_P_MARSHAL,
        TPM2B_PUBLIC_P_MARSHAL,
        TPM2B_DATA_P_MARSHAL,
        TPM2B_DIGEST_P_MARSHAL,
        TPM2B_CREATION_DATA_P_MARSHAL,
        TPM2B_PUBLIC_P_MARSHAL,
        TPM2B_NAME_P_MARSHAL,
        TPM2B_TK_CREATION_P_MARSHAL
    },
    /* endOfList */ 0
};

#define _HierarchyControlDataAddress (&_HierarchyControlData)
BYTE types[5];

SetPrimaryPolicy_COMMAND_DESCRIPTOR_t _SetPrimaryPolicyData = {
  /* entry */ &TPM2_SetPrimaryPolicy,
  /* inSize */ (UINT16)(sizeof(SetPrimaryPolicy_In)),
  /* outSize */ 0,
  /* offsetOfTypes */ offsetof(SetPrimaryPolicy_COMMAND_DESCRIPTOR_t, types),
  /* offsets */ {
    (UINT16)(offsetof(SetPrimaryPolicy_In, authPolicy)),
    (UINT16)(offsetof(SetPrimaryPolicy_In, hashAlg))},
  /* types */ {
    TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
    TPM2B_DIGEST_P_UNMARSHAL,
    TPMI_ALG_HASH_P_UNMARSHAL + ADD_FLAG,
    END_OF_LIST,
    END_OF_LIST}
};

#define _SetPrimaryPolicyDataAddress (&_SetPrimaryPolicyData)

else
#define _SetPrimaryPolicyDataAddress 0
#endif // CC_SetPrimaryPolicy

#if CC_ChangePPS
#include "ChangePPS_fp.h"
typedef TPM_RC (ChangePPS_Entry)(
  ChangePPS_In *in
);
typedef const struct {
  ChangePPS_Entry *entry;
  UINT16 inSize;
  UINT16 outSize;
  UINT16 offsetOfTypes;
  BYTE types[3];
} ChangePPS_COMMAND_DESCRIPTOR_t;

ChangePPS_COMMAND_DESCRIPTOR_t _ChangePPSData = {
  /* entry */ &TPM2_ChangePPS,
  /* inSize */ (UINT16)(sizeof(ChangePPS_In)),
  /* outSize */ 0,
  /* offsetOfTypes */ offsetof(ChangePPS_COMMAND_DESCRIPTOR_t, types),
  /* offsets */ // No parameter offsets;
  /* types */ {
    TPMI_RH_PLATFORM_H_UNMARSHAL,
    END_OF_LIST,
    END_OF_LIST}
};

#define _ChangePPSDataAddress (&_ChangePPSData)
#else
#define _ChangePPSDataAddress 0
#endif // CC_ChangePPS

#if CC_ChangeEPS
#include "ChangeEPS_fp.h"
typedef TPM_RC (ChangeEPS_Entry)(
  ChangeEPS_In *in
);
typedef const struct {
  ChangeEPS_Entry *entry;
  UINT16 inSize;
  UINT16 outSize;
  UINT16 offsetOfTypes;
  BYTE types[3];
} ChangeEPS_COMMAND_DESCRIPTOR_t;

ChangeEPS_COMMAND_DESCRIPTOR_t _ChangeEPSData = {
  /* entry */ &TPM2_ChangeEPS,
  /* inSize */ (UINT16)(sizeof(ChangeEPS_In)),
  /* outSize */ 0,
  /* offsetOfTypes */ offsetof(ChangeEPS_COMMAND_DESCRIPTOR_t, types),
  /* offsets */ // No parameter offsets;
  /* types */ {
    TPMI_RH_PLATFORM_H_UNMARSHAL,
    END_OF_LIST,
    END_OF_LIST}
};

#define _ChangeEPSDataAddress (&_ChangeEPSData)
#else
#define _ChangeEPSDataAddress 0
#endif // CC_ChangeEPS
2905 };
2906 #define _ChangeEPSDataAddress (&_ChangeEPSData)
2907#endif
2908#endif
2909#define _ChangeEPSDataAddress 0
2910#endif // CC_ChangeEPS
2911#if CC_Clear
2912#include "Clear_fp.h"
2913typedef TPM_RC (Clear_Entry)(
2914 Clear_In                    *in
2915 );
2916typedef const struct {
2917     Clear_Entry            *entry;
2918     UINT16                inSize;
2919     UINT16                outSize;
2920     UINT16                offsetOfTypes;
2921     BYTE                  types[3];
2922 } Clear_COMMAND_DESCRIPTOR_t;
2923Clear_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 */ 0, // No parameter offsets:
2929    /* types */ {TPMI_RH_CLEAR_H_UNMARSHAL,
2930             END_OF_LIST, __
2931             END_OF_LIST}
2932};
2933#define _ClearDataAddress (&_ClearData)
2934#endif
2935#define _ClearDataAddress 0
2936#endif // CC_Clear
2937#if CC_ClearControl
2938#include "ClearControl_fp.h"
2939typedef TPM_RC (ClearControl_Entry)(
2940 ClearControl_In             *in
2941 );
2942typedef 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;
2950ClearControl_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#endif
2963#define _ClearControlDataAddress 0
2964#endif // CC_ClearControl
2965#if CC_HierarchyChangeAuth
2966#include "HierarchyChangeAuth_fp.h"
2967typedef TPM_RC (HierarchyChangeAuth_Entry)(
2968 HierarchyChangeAuth_In             *in
2969 );
2970typedef const struct {
2971     HierarchyChangeAuth_Entry         *entry;
UINT16 inSize;
UINT16 outSize;
UINT16 offsetOfTypes;
UINT16 paramOffsets[1];
BYTE types[4];

HierarchyChangeAuth_COMMAND_DESCRIPTOR_t _HierarchyChangeAuthData = {
    /* entry */ &TPM2_HierarchyChangeAuth,
    /* inSize */ (UINT16)(sizeof(HierarchyChangeAuth_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offset(HierarchyChangeAuth_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {(UINT16)(offsetof(HierarchyChangeAuth_In, newAuth))},
    /* types */ {TPMI_RH_HIERARCHY_AUTH_H_UNMARSHAL,
                 TPM2B_AUTH_P_UNMARSHAL,
                 END_OF_LIST,
                 END_OF_LIST}];

#define _HierarchyChangeAuthDataAddress (&_HierarchyChangeAuthData)
#else
#define _HierarchyChangeAuthDataAddress 0
#endif // CC_HierarchyChangeAuth

typedef TPM_RC (DictionaryAttackLockReset_Entry)(
    DictionaryAttackLockReset_In *in
) DictionaryAttackLockReset_In;

typedef const struct {
    DictionaryAttackLockReset_Entry    *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    BYTE types[3];
} DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t;

DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t _DictionaryAttackLockResetData = {
    /* entry */ &TPM2_DictionaryAttackLockReset,
    /* inSize */ (UINT16)(sizeof(DictionaryAttackLockReset_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offset(DictionaryAttackLockReset_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ /* No parameter offsets; */
    /* types */ {TPMI_RH_LOCKOUT_H_UNMARSHAL,
                 END_OF_LIST,
                 END_OF_LIST}];

#define _DictionaryAttackLockResetDataAddress (&_DictionaryAttackLockResetData)
#else
#define _DictionaryAttackLockResetDataAddress 0
#endif // CC_DictionaryAttackLockReset

typedef TPM_RC (DictionaryAttackParameters_Entry)(
    DictionaryAttackParameters_In *in
) DictionaryAttackParameters_In;

typedef const struct {
    DictionaryAttackParameters_Entry    *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    BYTE types[6];
} DictionaryAttackParameters_COMMAND_DESCRIPTOR_t;

DictionaryAttackParameters_COMMAND_DESCRIPTOR_t _DictionaryAttackParametersData = {
    /* entry */ &TPM2_DictionaryAttackParameters,
/* inSize */
(UINT16)(sizeof(DictionaryAttackParameters_In)),
/* outSize */
0,
/* offsetOfTypes */
offsetof(DictionaryAttackParameters_COMMAND_DESCRIPTOR_t, types),
/* offsets */
{(UINT16)(offsetof(DictionaryAttackParameters_In, newMaxTries)),
(UINT16)(offsetof(DictionaryAttackParameters_In, newRecoveryTime)),
(UINT16)(offsetof(DictionaryAttackParameters_In, lockoutRecovery))},
/* types */
{TPMI_RH_LOCKOUT_H_UNMARSHAL,
UINT32_P_UNMARSHAL,
UINT32_P_UNMARSHAL,
UINT32_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST};
#define _DictionaryAttackParametersDataAddress (&_DictionaryAttackParametersData)
#else
#define _DictionaryAttackParametersDataAddress 0
#endif // CC_DictionaryAttackParameters
#if CC_PP_Commands
#include "PP_Commands_fp.h"
typedef TPM_RC (PP_Commands_Entry)(
PP_Commands_In *in
);
typedef const struct {
PP_Commands_Entry *entry;
UINT16 inSize;
UINT16 outSize;
UINT16 offsetOfTypes;
UINT16 paramOffsets[2];
BYTE types[5];
} PP_Commands_COMMAND_DESCRIPTOR_t;
PP_Commands_COMMAND_DESCRIPTOR_t _PP_CommandsData = {
/* entry */
&TPM2_PP_Commands,
/* inSize */
(UINT16)(sizeof(PP_Commands_In)),
/* outSize */
0,
/* offsetOfTypes */
offsetof(PP_Commands_COMMAND_DESCRIPTOR_t, types),
/* offsets */
{(UINT16)(offsetof(PP_Commands_In, setList)),
(UINT16)(offsetof(PP_Commands_In, clearList))},
/* types */
{TPMI_RH_PLATFORM_H_UNMARSHAL,
TPML_CC_P_UNMARSHAL,
TPML_CC_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST};
#define _PP_CommandsDataAddress (&_PP_CommandsData)
#else
#define _PP_CommandsDataAddress 0
#endif // CC_PP_Commands
#if CC_SetAlgorithmSet
#include "SetAlgorithmSet_fp.h"
typedef TPM_RC (SetAlgorithmSet_Entry)(
SetAlgorithmSet_In *in
);
typedef const struct {
SetAlgorithmSet_Entry *entry;
UINT16 inSize;
UINT16 outSize;
UINT16 offsetOfTypes;
UINT16 paramOffsets[1];
BYTE types[4];
} SetAlgorithmSet_COMMAND_DESCRIPTOR_t;
SetAlgorithmSet_COMMAND_DESCRIPTOR_t _SetAlgorithmSetData = {
/* entry */
&TPM2_SetAlgorithmSet,
/* inSize */
(UINT16)(sizeof(SetAlgorithmSet_In)),
/* outSize */
0,
/* offsetOfTypes */
offsetof(SetAlgorithmSet_COMMAND_DESCRIPTOR_t, types),
/* offsets */
{(UINT16)(offsetof(SetAlgorithmSet_In, setList)),
(UINT16)(offsetof(SetAlgorithmSet_In, clearList))},
/* types */
{TPMI_RH_PLATFORM_H_UNMARSHAL,
TPML_CC_P_UNMARSHAL,
TPML_CC_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST};
#define _SetAlgorithmSetDataAddress (&_SetAlgorithmSetData)
#else
#define _SetAlgorithmSetDataAddress 0
#endif // CC_SetAlgorithmSet
/* entry */ &TPM2_SetAlgorithmSet,
/* inSize */ (UINT16)(sizeof(SetAlgorithmSet_In)),
/* outSize */ 0,
/* offsetOfTypes */ offsetof(SetAlgorithmSet_COMMAND_DESCRIPTOR_t, types),
/* offsets */ {(UINT16)(offsetof(SetAlgorithmSet_In, algorithmSet))},
/* types */ {TPMI_RH_PLATFORM_H_UNMARSHAL,
UINT32_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST};
#define _SetAlgorithmSetDataAddress (&_SetAlgorithmSetData)
#else
#define _SetAlgorithmSetDataAddress 0
#endif // CC_SetAlgorithmSet
#if CC_FieldUpgradeStart
#include "FieldUpgradeStart_fp.h"
typedef TPM_RC (FieldUpgradeStart_Entry)(
FieldUpgradeStart_In *in
);
typedef const struct {
    FieldUpgradeStart_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[3];
    BYTE types[6];
} FieldUpgradeStart_COMMAND_DESCRIPTOR_t;
FieldUpgradeStart_COMMAND_DESCRIPTOR_t _FieldUpgradeStartData = {
/* entry */ &TPM2_FieldUpgradeStart,
/* inSize */ (UINT16)(sizeof(FieldUpgradeStart_In)),
/* outSize */ 0,
/* offsetOfTypes */ offsetof(FieldUpgradeStart_COMMAND_DESCRIPTOR_t, types),
/* offsets */ {(UINT16)(offsetof(FieldUpgradeStart_In, keyHandle)),
(UINT16)(offsetof(FieldUpgradeStart_In, fuDigest)),
(UINT16)(offsetof(FieldUpgradeStart_In, manifestSignature))},
/* types */ {TPMI_RH_PLATFORM_H_UNMARSHAL,
TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_DIGEST_P_UNMARSHAL,
TPMT_SIGNATURE_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST};
#define _FieldUpgradeStartDataAddress (&_FieldUpgradeStartData)
#else
#define _FieldUpgradeStartDataAddress 0
#endif // CC_FieldUpgradeStart
#if CC_FieldUpgradeData
#include "FieldUpgradeData_fp.h"
typedef TPM_RC (FieldUpgradeData_Entry)(
FieldUpgradeData_In *in,
FieldUpgradeData_Out *out
);
typedef const struct {
    FieldUpgradeData_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    UINT16 paramOffsets[1];
    BYTE types[5];
} FieldUpgradeData_COMMAND_DESCRIPTOR_t;
FieldUpgradeData_COMMAND_DESCRIPTOR_t _FieldUpgradeDataData = {
/* entry */ &TPM2_FieldUpgradeData,
/* inSize */ (UINT16)(sizeof(FieldUpgradeData_In)),
/* outSize */ (UINT16)(sizeof(FieldUpgradeData_Out)),
/* offsetOfTypes */ offsetof(FieldUpgradeData_COMMAND_DESCRIPTOR_t, types),
/* offsets */ {(UINT16)(offsetof(FieldUpgradeData_In, fuDigest)),
(UINT16)(offsetof(FieldUpgradeData_In, manifestSignature))},
/* types */ {TPMI_RH_PLATFORM_H_UNMARSHAL,
TPMI_DH_OBJECT_H_UNMARSHAL,
TPM2B_DIGEST_P_UNMARSHAL,
TPMT_SIGNATURE_P_UNMARSHAL,
END_OF_LIST,
END_OF_LIST};
#define _FieldUpgradeDataDataAddress (&_FieldUpgradeDataData)
#else
#define _FieldUpgradeDataDataAddress 0
#endif // CC_FieldUpgradeData
/* offsetOfTypes */ offsetof(FieldUpgradeData_COMMAND_DESCRIPTOR_t, types),
/* offsets */
{(UINT16)(offsetof(FieldUpgradeData_Out, firstDigest))},
/* types */
{TPM2B_MAX_BUFFER_P_UNMARSHAL,
END_OF_LIST,
TPM2_P_MARSHAL,
TPMT_HA_P_MARSHAL,
END_OF_LIST}
};
#define _FieldUpgradeDataDataAddress (&_FieldUpgradeDataData)
#else
#define _FieldUpgradeDataDataAddress 0
#endif // CC_FieldUpgradeData
#endif // CC_FirmwareRead
#endif // CC_ContextSave

typedef TPM_RC (FirmwareRead_Entry)(
        FirmwareRead_In *in,
        FirmwareRead_Out *out
);
typedef const struct {
        FirmwareRead_Entry *entry;
        UINT16 inSize;
        UINT16 outSize;
        UINT16 offsetOfTypes;
        BYTE types[4];
    } FirmwareRead_COMMAND_DESCRIPTOR_t;
FirmwareRead_COMMAND_DESCRIPTOR_t _FirmwareReadData = {
/* entry */ &TPM2_FirmwareRead,
/* inSize */ (UINT16)(sizeof(FirmwareRead_In)),
/* outSize */ (UINT16)(sizeof(FirmwareRead_Out)),
/* offsetOfTypes */ offsetof(FirmwareRead_COMMAND_DESCRIPTOR_t, types),
/* offsets */ // No parameter offsets;
/* types */
{UINT32_P_UNMARSHAL,
END_OF_LIST,
TPM2B_MAX_BUFFER_P_MARSHAL,
END_OF_LIST}
};
#define _FirmwareReadDataAddress (&_FirmwareReadData)
#else
#define _FirmwareReadDataAddress
#endif // CC_FirmwareRead

typedef TPM_RC (ContextSave_Entry)(
        ContextSave_In *in,
        ContextSave_Out *out
);
typedef const struct {
        ContextSave_Entry *entry;
        UINT16 inSize;
        UINT16 outSize;
        UINT16 offsetOfTypes;
        BYTE types[4];
    } ContextSave_COMMAND_DESCRIPTOR_t;
ContextSave_COMMAND_DESCRIPTOR_t _ContextSaveData = {
/* entry */ &TPM2_ContextSave,
/* inSize */ (UINT16)(sizeof(ContextSave_In)),
/* outSize */ (UINT16)(sizeof(ContextSave_Out)),
/* offsetOfTypes */ offsetof(ContextSave_COMMAND_DESCRIPTOR_t, types),
/* offsets */ // No parameter offsets;
/* types */
{TPMI_DH_CONTEXT_H_UNMARSHAL,
END_OF_LIST,
TPMS_CONTEXT_P_MARSHAL,
END_OF_LIST}
};
#define _ContextSaveDataAddress (&_ContextSaveData)
#else

#define _ContextSaveDataAddress 0
#endif // CC_ContextSave

#define CC_ContextLoad

#include "ContextLoad_fp.h"

typedef TPM_RC (ContextLoad_Entry)(
    ContextLoad_In *in,
    ContextLoad_Out *out
);

typedef const struct {
    ContextLoad_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    BYTE types[4];
} ContextLoad_COMMAND_DESCRIPTOR_t;

ContextLoad_COMMAND_DESCRIPTOR_t _ContextLoadData = {
    /* entry */ &TPM2_ContextLoad,
    /* inSize */ (UINT16)(sizeof(ContextLoad_In)),
    /* outSize */ (UINT16)(sizeof(ContextLoad_Out)),
    /* offsetOfTypes */ offsetof(ContextLoad_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ // No parameter offsets;
    /* types */ {TPMS_CONTEXT_P_UNMARSHAL,
    TPMI_DH_CONTEXT_H_MARSHAL,
    END_OF_LIST}
};

#define _ContextLoadDataAddress (&_ContextLoadData)
#endif // CC_ContextLoad

#define CC_FlushContext

#include "FlushContext_fp.h"

typedef TPM_RC (FlushContext_Entry)(
    FlushContext_In *in
);

typedef const struct {
    FlushContext_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
    BYTE types[3];
} FlushContext_COMMAND_DESCRIPTOR_t;

FlushContext_COMMAND_DESCRIPTOR_t _FlushContextData = {
    /* entry */ &TPM2_FlushContext,
    /* inSize */ (UINT16)(sizeof(FlushContext_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetof(FlushContext_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ // No parameter offsets;
    /* types */ {TPMI_DH_CONTEXT_P_UNMARSHAL,
    TPMI_DH CONTEXT_H MARSHAL,
    END_OF_LIST}
};

#define _FlushContextDataAddress (&_FlushContextData)
#endif // CC_FlushContext

#define CC_EvictControl

#include "EvictControl_fp.h"

typedef TPM_RC (EvictControl_Entry)(
    EvictControl_In *in
);

typedef const struct {
    EvictControl_Entry *entry;
    UINT16 inSize;
    UINT16 outSize;
    UINT16 offsetOfTypes;
} EvictControl_COMMAND_DESCRIPTOR_t;
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 */ {
3300 (UINT16)(offsetof(EvictControl_IN, objectHandle)),
3301 (UINT16)(offsetof(EvictControl_IN, persistentHandle))},
3302 /* types */ {
3303 TPMI_RH_PROVISION_H_UNMARSHAL,
3304 TPMI_DH_OBJECT_H_UNMARSHAL,
3305 TPMI_DH_PERSISTENT_P_UNMARSHAL,
3306 END_OF_LIST,
3307 END_OF_LIST
3308 
3309 
3310 }; // 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 offsetOfType;
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 */ {
3329 (UINT16)(offsetof(ClockSet_In, newTime))},
3330 /* types */ {
3331 TPMI_RH_PROVISION_H_UNMARSHAL,
3332 TPMI_DH_OBJECT_H_UNMARSHAL,
3333 TPMI_DH_PERSISTENT_P_UNMARSHAL,
3334 END_OF_LIST,
3335 END_OF_LIST
3336 
3337 #define _ReadClockDataAddress (&_ReadClockData)
3338 #else
3339 #define _ReadClockDataAddress 0
3340 #endif // CC_ReadClock
3341 #if CC_ClockSet
3342 #include "ClockSet_fp.h"
3343 typedef TPM_RC (ClockSet_Entry)(
3344 ClockSet_In *in
3345 
3346 typedef const struct {
3347 ClockSet_Entry *entry;
3348 UINT16 inSize;
3349 UINT16 outSize;
3350 UINT16 offsetOfType;
3351 UINT16 paramOffsets[1];
3352 BYTE types[4];
3353 } ClockSet_COMMAND_DESCRIPTOR_t;
3354 ClockSet_COMMAND_DESCRIPTOR_t _ClockSetData = {
3355 /* entry */ &TPM2_ClockSet,
3356 /* inSize */ (UINT16)(sizeof(ClockSet_In)),
3357 /* outSize */ 0,
3358 /* offsetOfTypes */ offsetof(ClockSet_COMMAND_DESCRIPTOR_t, types),
3359 /* offsets */ {
3360 (UINT16)(offsetof(ClockSet_In, newTime))},
3361 /* types */ {
3362 TPMI_RH_PROVISION_H_UNMARSHAL,
typedef TPM_RC (ClockRateAdjust_Entry)(
    ClockRateAdjust_In *in
)

typedef const struct {
    ClockRateAdjust_Entry   *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetsOfTypes;
    UINT16                  paramOffsets[1];
    BYTE                    types[4];
} ClockRateAdjust_COMMAND_DESCRIPTOR_t;

ClockRateAdjust_COMMAND_DESCRIPTOR_t _ClockRateAdjustData = {
    /* entry */ &TPM2_ClockRateAdjust,
    /* inSize */ (UINT16)(sizeof(ClockRateAdjust_In)),
    /* outSize */ 0,
    /* offsetsOfTypes */ offsetOf(ClockRateAdjust_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(ClockRateAdjust_In, rateAdjust))},
    /* types */ {
        TPMI_RH_PROVISION_H_UNMARSHAL,
        TPM_CLOCK_ADJUST_P_UNMARSHAL,
        END_OF_LIST,
        END_OF_LIST
    }
};

#define _ClockRateAdjustDataAddress (&_ClockRateAdjustData)

#define _ClockSetDataAddress ((&_ClockSetData)
else
#define _ClockSetDataAddress 0
#endif // CC_ClockSet
#if CC_ClockRateAdjust
#include "ClockRateAdjust_fp.h"

typedef TPM_RC (GetCapability_Entry)(
    GetCapability_In *in,
    GetCapability_Out *out
)

typedef const struct {
    GetCapability_Entry     *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetsOfTypes;
    UINT16                  paramOffsets[3];
    BYTE                    types[7];
} GetCapability_COMMAND_DESCRIPTOR_t;

GetCapability_COMMAND_DESCRIPTOR_t _GetCapabilityData = {
    /* entry */ &TPM2_GetCapability,
    /* inSize */ (UINT16)(sizeof(GetCapability_In)),
    /* outSize */ (UINT16)(sizeof(GetCapability_Out)),
    /* offsetsOfTypes */ offsetOf(GetCapability_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(GetCapability_In, property)),
        (UINT16)(offsetof(GetCapability_In, propertyCount)),
        (UINT16)(offsetof(GetCapability_Out, capabilityData))},
    /* types */ {
        TPMI_CAP_P_UNMARSHAL,
        UINT32_P_UNMARSHAL,
        UINT32_P_UNMARSHAL,
        END_OF_LIST,
        TPMI_YES_NO_P_MARSHAL,
        TPMS_CAPABILITY_DATA_P_MARSHAL,
        END_OF_LIST
    }
};
#define _GetCapabilityDataAddress (&_GetCapabilityData)
#else
#define _GetCapabilityDataAddress 0
#endif // CC_GetCapability

#include "TestParms_fp.h"

typedef TPM_RC (TestParms_Entry)(
  TestParms_In *in
)
TestParms_COMMAND_DESCRIPTOR_t _TestParmsData = {
  /* entry */ &TPM2_TestParms,
  /* inSize */ (UINT16)(sizeof(TestParms_In)),
  /* offsetOfType */ 0,
  /* offsets */ 0, // No parameter offsets;
  /* types */ {TPMT_PUBLICParms_P_UNMARSHAL, END_OF_LIST}
};
#define _TestParmsDataAddress (&_TestParmsData)
#else
#define _TestParmsDataAddress 0
#endif // CC_TestParms

#include "NV_DefineSpace_fp.h"

typedef NSP_DEFINESPACE (NV_DefineSpace_Entry)(
  NV_DefineSpace_In *in
)
NV_DefineSpace_COMMAND_DESCRIPTOR_t _NV_DefineSpaceData = {
  /* entry */ &TPM2_NV_DefineSpace,
  /* inSize */ (UINT16)(sizeof(NV_DefineSpace_In)),
  /* offsetOfType */ 0,
  /* offsets */ 0,
  /* types */ {TPMI_RH_PROVISION_H_UNMARSHAL, END_OF_LIST}
};
#define _NV_DefineSpaceDataAddress (&_NV_DefineSpaceData)
#else
#define _NV_DefineSpaceDataAddress 0
#endif // CC_NV_DefineSpace

#include "NV_UndefineSpace_fp.h"

typedef TPM_RC (NV_UndefineSpace_Entry)(
  NV_UndefineSpace_In *in
);
```c
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 */ offsetof(NV_UndefineSpaceSpecial_COMMAND_DESCRIPTOR_t, types),
3529  /* offsets */ {{(UINT16)(offsetof(NV_UndefineSpaceSpecial_In, platform))},
3530  /* types */ {TPMI_RH_NV_INDEX_H_UNMARSHAL,
3531                     TPMI_RH_PLATFORM_H_UNMARSHAL,
3532                     END_OF_LIST,
3533                     END_OF_LIST}
3534};
3535  #define _NV_UndefineSpaceSpecialDataAddress (&_NV_UndefineSpaceSpecialData)
3536  #else
3537  #define _NV_UndefineSpaceSpecialDataAddress 0
3538  #endif  // CC_NV_UndefineSpaceSpecial
3539  #if CC_NV_ReadPublic
3540  #include "NV_ReadPublic_fp.h"
3541  typedef TPM_RC  (NV_ReadPublic_Entry)(
3542    NV_ReadPublic_In  *in,
3543    NV_ReadPublic_Out *out
3544  );
3545  typedef const struct {
3546    NV_ReadPublic_Entry  *entry;
3547    UINT16                   inSize;
3548    UINT16                   outSize;
3549    UINT16                   offsetOfTypes;
3550    UINT16                   paramOffsets[1];
3551    BYTE                     types[5];
3552}  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   TPM2B_NV_PUBLIC_P_MARSHAL,
3561   TPM2B_NAME_P_MARSHAL,
3562   END_OF_LIST}
3563};
3564  
3565  #define _NV_ReadPublicDataAddress (&_NV_ReadPublicData)
3566
3567  #else
3568  #define _NV_ReadPublicDataAddress 0
3569  #endif // CC_NV_ReadPublic
3570
3571  #if CC_NV_Write
3572  #include "NV_Write_fp.h"
3573  typedef TPM_RC (NV_Write_Entry)(
3574      NV_Write_In *in)
3575  NV_Write_Entry
3576      *entry;
3577      UINT16 inSize;
3578      UINT16 outSize;
3579      UINT16 offOffsets;
3580      UINT16 paramOffsets[3];
3581      BYTE types[6];
3582  } NV_Write_COMMAND_DESCRIPTOR_t;
3583  
3584  NV_Write_COMMAND_DESCRIPTOR_t _NV_WriteData = {
3585  /* entry */ &TPM2_NV_Write,
3586  /* inSize */ (UINT16)(sizeof(NV_Write_In)),
3587  /* outSize */ 0,
3588  /* offsetOfTypes */ offsetof(NV_Write_COMMAND_DESCRIPTOR_t, types),
3589  /* offsets */ {(UINT16)(offsetof(NV_Write_In, nvIndex))},
3590  /* types */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,
3591   TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3592   UINT16_P_UNMARSHAL,
3593   TPM2B_NV_INDEX_H_UNMARSHAL,
3594   END_OF_LIST,
3595   END_OF_LIST}
3596};
3597  
3598  #define _NV_WriteDataAddress (&_NV_WriteData)
3599  #else
3600  #define _NV_WriteDataAddress 0
3601  #endif // CC_NV_Write
3602
3603  #if CC_NV_Increment
3604  #include "NV_Increment_fp.h"
3605  typedef TPM_RC (NV_Increment_Entry)(
3606      NV_Increment_In *in)
3607  NV_Increment_Entry
3608      *entry;
3609      UINT16 inSize;
3610      UINT16 outSize;
3611      UINT16 offOffsets;
3612      UINT16 paramOffsets[1];
3613      BYTE types[4];
3614  } NV_Increment_COMMAND_DESCRIPTOR_t;
3615  
3616  NV_Increment_COMMAND_DESCRIPTOR_t _NV_IncrementData = {
3617  /* entry */ &TPM2_NV_Increment,
3618  /* inSize */ (UINT16)(sizeof(NV_Increment_In)),
3619  /* outSize */ 0,
3620  /* 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)(offsetof(NV_Extend_In)),
3645 /* outSize */ 0,
3646 /* offsetOfTypes */ offsetof(NV_Extend_COMMAND_DESCRIPTOR_t, types),
3647 /* offsets */{(UINT16)(offsetof(NV_Extend_In, nvIndex))},
3648 /* types */{TPMI_RH_NV_AUTH_H_UNMARSHAL,
3649 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3650 TPM2B_MAX_NV_BUFFER_P_UNMARSHAL,
3651 END_OF_LIST,
3652 END_OF_LIST}.
3653 #define _NV_ExtendDataAddress (&_NV_ExtendData)
3654 #else
3655 #define _NV_ExtendDataAddress 0
3656 #endif // CC_NV_Extend
3657 #if CC_NV_SetBits
3658 #include "NV_SetBits_fp.h"
3659 typedef TPM_RC (NV_SetBits_Entry)(
3660 NV_SetBits_In *in
3661);
3662 typedef const struct {
3663 NV_SetBits_Entry *entry;
3664 UINT16 inSize;
3665 UINT16 outSize;
3666 UINT16 offsetOfTypes;
3667 UINT16 paramOffsets[2];
3668 BYTE types[5];
3669 } NV_SetBits_COMMAND_DESCRIPTOR_t;
3670 NV_SetBits_COMMAND_DESCRIPTOR_t _NV_SetBitsData = {
3671 /* entry */ &TPM2_NV_SetBits,
3672 /* inSize */ (UINT16)(offsetof(NV_SetBits_In)),
3673 /* outSize */ 0,
3674 /* offsetOfTypes */ offsetof(NV_SetBits_COMMAND_DESCRIPTOR_t, types),
3675 /* offsets */{(UINT16)(offsetof(NV_SetBits_In, nvIndex))},
3676 /* types */{TPMI_RH_NV_AUTH_H_UNMARSHAL,
3677 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3678 UINT64_P_UNMARSHAL,
3679 END_OF_LIST,
3680 END_OF_LIST}.
3681 #define _NV_SetBitsDataAddress (&_NV_SetBitsData)
3682 #else
3683 #define _NV_SetBitsDataAddress 0
3684 #endif // CC_NV_SetBits
3685 #endif // CC_NV_SetBits
3686 #endif // CC_NV_Increment
3687 #endif // CC_NV_Extend
```c
#define _NV_SetBitsDataAddress (&_NV_SetBitsData)

#else
#define _NV_SetBitsDataAddress 0
#endif

#include "NV_WriteLock_fp.h"

typedef TPM_RC (NV_WriteLock_Entry)(
    NV_WriteLock_In *in
);

typedef const struct {
    NV_WriteLock_Entry     *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offsetOfTypes;
    UINT16                  paramOffsets[1];
    BYTE                    types[4];
} NV_WriteLock_COMMAND_DESCRIPTOR_t;

NV_WriteLock_COMMAND_DESCRIPTOR_t _NV_WriteLockData = {
    /* entry         */ &TPM2_NV_WriteLock,
    /* inSize        */ (UINT16)(sizeof(NV_WriteLock_In)),
    /* outSize       */ 0,
    /* offsetOfTypes */ offsetOf(NV_WriteLock_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ {{(UINT16)(offsetof(NV_WriteLock_In, nvIndex))},
    /* types         */ {TPMI_RH_NV_AUTH_H_UNMARSHAL,
    TPMI_RH_NV_INDEX_H_UNMARSHAL,
    END_OF_LIST,
    END_OF_LIST}}
};

#define _NV_WriteLockDataAddress (&_NV_WriteLockData)
#endif

#include "NV_GlobalWriteLock_fp.h"

typedef TPM_RC (NV_GlobalWriteLock_Entry)(
    NV_GlobalWriteLock_In *in
);

typedef const struct {
    NV_GlobalWriteLock_Entry    *entry;
    UINT16                      inSize;
    UINT16                      outSize;
    UINT16                      offsetOfTypes;
    BYTE                        types[3];
} NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t;

NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t _NV_GlobalWriteLockData = {
    /* entry         */ &TPM2_NV_GlobalWriteLock,
    /* inSize        */ (UINT16)(sizeof(NV_GlobalWriteLock_In)),
    /* outSize       */ 0,
    /* offsetOfTypes */ offsetOf(NV_GlobalWriteLock_COMMAND_DESCRIPTOR_t, types),
    /* offsets       */ // No parameter offsets;
    /* types        */ {TPMI_RH_PROVISION_H_UNMARSHAL,
    END_OF_LIST,
    END_OF_LIST}
};

#define _NV_GlobalWriteLockDataAddress (&_NV_GlobalWriteLockData)
#endif

#include "NV_Read_fp.h"

typedef TPM_RC (NV_Read_Entry)(
    NV_Read_In *in,
    NV_Read_Out *out
);

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 */ {
3763 (UINT16)(offsetof(NV_Read_In, nvIndex)),
3764 (UINT16)(offsetof(NV_Read_In, size)),
3765 (UINT16)(offsetof(NV_Read_In, offset))},
3766 /* types */ {
3767 TPMI_RH_NV_AUTH_H_UNMARSHAL,
3768 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3769 UINT16_P_UNMARSHAL,
3770 BYTE_P_UNMARSHAL,
3771 TPM2B_MAX_NV_BUFFER_P_MARSHAL,
3772 END_OF_LIST}
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 */ {
3796 (UINT16)(offsetof(NV_ReadLock_In, nvIndex))},
3797 /* types */ {
3798 TPMI_RH_NV_AUTH_H_UNMARSHAL,
3799 TPMI_RH_NV_INDEX_H_UNMARSHAL,
3800 END_OF_LIST,
3801 END_OF_LIST}
3802 #define _NV_ReadLockDataAddress (&_NV_ReadLockData)
3803 #else
3804 #define _NV_ReadLockDataAddress 0
3805 #endif // CC_NV_ReadLock
3806 #if CC_NV_ChangeAuth
3807 #include "NV_ChangeAuth_fp.h"
3808 typedef TPM_RC (NV_ChangeAuth_Entry)(
3809 NV_ChangeAuth_In *in
3810 )
3811 typedef const struct {
3812 NV_ChangeAuth_Entry *entry;
3813 UINT16 inSize;
3814 UINT16 outSize;
3815 UINT16 offsetOfTypes;
BYTE                   types[4];
} NV_ChangeAuth_COMMAND_DESCRIPTOR_t;
NV_ChangeAuth_COMMAND_DESCRIPTOR_t _NV_ChangeAuthData = {
    /* entry */       &TPM2_NV_ChangeAuth,
    /* inSize */      (UINT16)(sizeof(NV_ChangeAuth_In)),
    /* outSize */     0,
    /* offSetOfTypes */  offSetOf(NV_ChangeAuth_COMMAND_DESCRIPTOR_t, types),
    /* offsets */     {{UINT16(offSetOf(NV_ChangeAuth_In, newAuth))},
    /* types */       {TPMI_RH_NV_INDEX_H_UNMARSHAL,
                        TPM2B_AUTH_P_UNMARSHAL,
                        END_OF_LIST,
                        END_OF_LIST}
};
#define _NV_ChangeAuthDataAddress (&_NV_ChangeAuthData)
#else
#define _NV_ChangeAuthDataAddress 0
#endif
// CC_NV_ChangeAuth

#if CC_NV_Certify
#include "NV_Certify_fp.h"
typedef TPM_RC (NV_Certify_Entry)(
    NV_Certify_In               *in,
    NV_Certify_Out              *out
);
typedef const struct
{
    NV_Certify_Entry        *entry;
    UINT16                  inSize;
    UINT16                  outSize;
    UINT16                  offSetOfTypes;
    UINT16                  paramOffsets[7];
    BYTE                    types[11];
} NV_Certify_COMMAND_DESCRIPTOR_t;
NV_Certify_COMMAND_DESCRIPTOR_t _NV_CertifyData = {
    /* entry */       &TPM2_NV_Certify,
    /* inSize */      (UINT16)(sizeof(NV_Certify_In)),
    /* outSize */     (UINT16)(sizeof(NV_Certify_Out)),
    /* offSetOfTypes */  offSetOf(NV_Certify_COMMAND_DESCRIPTOR_t, types),
    /* offsets */     {{UINT16(offSetOf(NV_Certify_In, authHandle))},
                       (UINT16)(offSetOf(NV_Certify_In, nvIndex)),
                       (UINT16)(offSetOf(NV_Certify_In, qualifyingData)),
                       (UINT16)(offSetOf(NV_Certify_In, inScheme)),
                       (UINT16)(offSetOf(NV_Certify_In, size)),
                       (UINT16)(offSetOf(NV_Certify_In, offset)),
                       (UINT16)(offSetOf(NV_Certify_Out, signature))},
    /* types */       {TPMI_DH_OBJECT_H_UNMARSHAL + ADD_FLAG,
                        TPM2B_NV_AUTH_H_UNMARSHAL,
                        TPM2B_NV_INDEX_H_UNMARSHAL,
                        TPMT_DATA_P_UNMARSHAL,
                        TPMT_SIG_SCHEME_P_UNMARSHAL + ADD_FLAG,
                        UINT16_P_UNMARSHAL,
                        END_OF_LIST,
                        TPM2B_ATTEST_P_MARSHAL,
                        TPM2B_ATTEST_P_MARSHAL,
                        END_OF_LIST}
};
#define _NV_CertifyDataAddress (_NV_CertifyData)
#else
#define _NV_CertifyDataAddress 0
#endif
// CC_NV_Certify

#if CC_AC_GetCapability
#include "AC_GetCapability_fp.h"
typedef TPM_RC (AC_GetCapability_Entry)(
    AC_GetCapability_In         *in,
    AC_GetCapability_Out        *out
);
typedef const struct
{
AC_GetCapability_Entry *entry;
UINT16 inSize;
UINT16 outSize;
UINT16 offsetOfTypes;
UINT16 paramOffsets[3];
BYTE types[7];
}

AC_GetCapability_COMMAND_DESCRIPTOR_t

AC_GetCapability_COMMAND_DESCRIPTOR_t _AC_GetCapabilityData = {
/* entry */ &TPM2_AC_GetCapability,
/* inSize */ (UINT16)(sizeof(AC_GetCapability_In)),
/* outSize */ (UINT16)(sizeof(AC_GetCapability_Out)),
/* offsetOfTypes */ &AC_GetCapability_COMMAND_DESCRIPTOR_tetypes,
/* offsets */ {
(UINT16)(offsetof(AC_GetCapability_In, capability)),
(UINT16)(offsetof(AC_GetCapability_In, count)),
(UINT16)(offsetof(AC_GetCapability_Out, capabilitiesData))},
/* types */ {
TPMI_RH_AC_H_UNMARSHAL,
TPM_AT_P_UNMARSHAL,
UINT32_P_UNMARSHAL,
END_OF_LIST,
TPM_YES_NO_P_MARSHAL,
TPML_AC_CAPABILITIES_P_MARSHAL,
END_OF_LIST}

#define _AC_GetCapabilityDataAddress (&_AC_GetCapabilityData)
#else
#define _AC_GetCapabilityDataAddress 0
#endif

/* entry */ &TPM2_AC_Send,
/* inSize */ (UINT16)(sizeof(AC_Send_In)),
/* outSize */ (UINT16)(sizeof(AC_Send_Out)),
/* offsetOfTypes */ &AC_Send_COMMAND_DESCRIPTOR_types,
/* offsets */ {
(UINT16)(offsetof(AC_Send_In, authHandle)),
(UINT16)(offsetof(AC_Send_In, ac)),
(UINT16)(offsetof(AC_Send_In, acDataIn))},
/* types */ {
TPMI_DH_OBJECT_H_UNMARSHAL,
TPMI_RH_NV_AUTH_H_UNMARSHAL,
TPM2B_MAX_BUFFER_P_UNMARSHAL,
END_OF_LIST,
TPMS_AC_OUTPUT_P_MARSHAL,
END_OF_LIST}

#define _AC_SendDataAddress (&_AC_SendData)
#else
#define _AC_SendDataAddress 0
#endif

/* entry */ &TPM2_AC_SendSelect,
/* inSize */ (UINT16)(sizeof(AC_SendSelect_In)),
/* outSize */ (UINT16)(sizeof(AC_SendSelect_Out)),
/* offsetOfTypes */ &AC_SendSelect_COMMAND_DESCRIPTOR_types,
/* offsets */ {
(UINT16)(offsetof(AC_SendSelect_In, authHandle)),
(UINT16)(offsetof(AC_SendSelect_In, ac)),
(UINT16)(offsetof(AC_SendSelect_In, acDataIn))},
/* types */ {
TPMI_DH_OBJECT_H_UNMARSHAL,
TPMI_RH_NV_AUTH_H_UNMARSHAL,
TPMI_RH_AC_H_UNMARSHAL,
TPM2B_MAX_BUFFER_P_UNMARSHAL,
END_OF_LIST,
TPMS_AC_OUTPUT_P_MARSHAL,
END_OF_LIST}

#define _AC_SendSelectDataAddress (&_AC_SendSelectData)
#else
#define _AC_SendSelectDataAddress 0
#endif

/* entry */ &Policy_AC_SendSelect,
/* in */
/* entry */ &TPM2_AC_SendSelect,
/* in */
/* entry */ &TPM2_AC_H

#define _Policy_ACSendSelectDataAddress (&Policy_AC_SendSelectData)
#define _Policy_ACSendSelectDataAddress 0
#endif

#define _Policy_ACSendSelectDataAddress (&Policy_AC_SendSelectData)
#define _Policy_ACSendSelectDataAddress 0
#endif

#define _Policy_ACSendSelectDataAddress (Policy_AC_SendSelect_In)
typedef const struct {
    Policy_AC_SendSelect_Entry  *entry;
    UINT16                      inSize;
    UINT16                      outSize;
    UINT16                      offsetOfTypes;
    UINT16                      paramOffsets[4];
    BYTE                        types[7];
} Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t;

Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t _Policy_AC_SendSelectData = {
    /* entry */ &TPM2_Policy_AC_SendSelect,
    /* inSize */ (UINT16)(sizeof(Policy_AC_SendSelect_In)),
    /* outSize */ 0,
    /* offsetOfTypes */ offsetof(Policy_AC_SendSelect_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(Policy_AC_SendSelect_In, objectName)),
        (UINT16)(offsetof(Policy_AC_SendSelect_In, authHandleName)),
        (UINT16)(offsetof(Policy_AC_SendSelect_In, acName)),
        (UINT16)(offsetof(Policy_AC_SendSelect_In, includeObject))},
    /* types */ {
        TPMI_SH_POLICY_H_UNMARSHAL,
        TPM2B_NAME_P_UNMARSHAL,
        TPM2B_NAME_P_UNMARSHAL,
        TPM2B_NAME_P_UNMARSHAL,
        TPMI_YES_NO_P_UNMARSHAL,
        END_OF_LIST,
        END_OF_LIST}
};

#define _Policy_AC_SendSelectDataAddress (&_Policy_AC_SendSelectData)
#else
#define _Policy_AC_SendSelectDataAddress 0
#endif // CC_Policy_AC_SendSelect
#endif

typedef Vendor_TCG_Test_Entry(Vendor_TCG_Test_In          *in,
    Vendor_TCG_Test_Out         *out) {
    Vendor_TCG_Test_In          *entry;
    UINT16                      inSize;
    UINT16                      outSize;
    UINT16                      offsetOfTypes;
    BYTE                        types[4];
} Vendor_TCG_Test_COMMAND_DESCRIPTOR_t;

Vendor_TCG_Test_COMMAND_DESCRIPTOR_t _Vendor_TCG_TestData = {
    /* entry */ &TPM2_Vendor_TCG_Test,
    /* inSize */ (UINT16)(sizeof(Vendor_TCG_Test_In)),
    /* outSize */ (UINT16)(sizeof(Vendor_TCG_Test_Out)),
    /* offsetOfTypes */ offsetof(Vendor_TCG_Test_COMMAND_DESCRIPTOR_t, types),
    /* offsets */ {
        (UINT16)(offsetof(Vendor_TCG_Test_IN, objectName)),
        (UINT16)(offsetof(Vendor_TCG_Test_IN, authHandleName)),
        (UINT16)(offsetof(Vendor_TCG_Test_IN, acName)),
        (UINT16)(offsetof(Vendor_TCG_Test_IN, includeObject))},
    /* types */ {
        TPM2B_DATA_P_UNMARSHAL,
        TPM2B_DATA_P_MARSHAL,
        END_OF_LIST,
        END_OF_LIST}
};

#define _Vendor_TCG_TestDataAddress (&_Vendor_TCG_TestData)
#else
#define _Vendor_TCG_TestDataAddress 0
#endif // CC_Vendor_TCG_Test

COMMAND_DESCRIPTOR_t *s_CommandDataArray[] = {
    #if (PAD_LIST || CC_NV_UndefineSpaceSpecial)
        (COMMAND_DESCRIPTOR_t *)_NV_UndefineSpaceSpecialDataAddress,
    #endif // CC_NV_UndefineSpaceSpecial
#if (PAD_LIST || CC_EvictControl)
  (COMMAND_DESCRIPTOR_t *)_EvictControlDataAddress,
#endif // CC_EvictControl

#if (PAD_LIST || CC_HierarchyControl)
  (COMMAND_DESCRIPTOR_t *)_HierarchyControlDataAddress,
#endif // CC_HierarchyControl

#if (PAD_LIST || CC_NV.UndefineSpace)
  (COMMAND_DESCRIPTOR_t *)_NV.UndefineSpaceDataAddress,
#endif // CC_NV.UndefineSpace

#if (PAD_LIST)
  (COMMAND_DESCRIPTOR_t *)0,
#endif

#if (PAD_LIST || CC_ChangeEPS)
  (COMMAND_DESCRIPTOR_t *)_ChangeEPSDataAddress,
#endif // CC_ChangeEPS

#if (PAD_LIST || CC_ChangePPS)
  (COMMAND_DESCRIPTOR_t *)_ChangePPSDataAddress,
#endif // CC_ChangePPS

#if (PAD_LIST || CC_Clear)
  (COMMAND_DESCRIPTOR_t *)_ClearDataAddress,
#endif // CC_Clear

#if (PAD_LIST || CC_ClearControl)
  (COMMAND_DESCRIPTOR_t *)_ClearControlDataAddress,
#endif // CC_ClearControl

#if (PAD_LIST || CC_ClockSet)
  (COMMAND_DESCRIPTOR_t *)_ClockSetDataAddress,
#endif // CC_ClockSet

#if (PAD_LIST || CC_HierarchyChangeAuth)
  (COMMAND_DESCRIPTOR_t *)_HierarchyChangeAuthDataAddress,
#endif // CC_HierarchyChangeAuth

#if (PAD_LIST || CC_NV.UndefineSpace)
  (COMMAND_DESCRIPTOR_t *)_NV.UndefineSpaceDataAddress,
#endif // CC_NV.UndefineSpace

#if (PAD_LIST || CC_PCR_Allocate)
  (COMMAND_DESCRIPTOR_t *)_PCR_AllocateDataAddress,
#endif // CC_PCR_Allocate

#if (PAD_LIST || CC_PCR_SetAuthPolicy)
  (COMMAND_DESCRIPTOR_t *)_PCR_SetAuthPolicyDataAddress,
#endif // CC_PCR_SetAuthPolicy

#if (PAD_LIST || CC_PP_Commands)
  (COMMAND_DESCRIPTOR_t *)_PP_CommandsDataAddress,
#endif // CC_PP_Commands

#if (PAD_LIST || CC_SetPrimaryPolicy)
  (COMMAND_DESCRIPTOR_t *)_SetPrimaryPolicyDataAddress,
#endif // CC_SetPrimaryPolicy

#if (PAD_LIST || CC_FieldUpgradeStart)
  (COMMAND_DESCRIPTOR_t *)_FieldUpgradeStartDataAddress,
#endif // CC_FieldUpgradeStart

#if (PAD_LIST || CC_ClockRateAdjust)
  (COMMAND_DESCRIPTOR_t *)_ClockRateAdjustDataAddress,
#endif // CC_ClockRateAdjust

#if (PAD_LIST || CC_CreatePrimary)
  (COMMAND_DESCRIPTOR_t *)_CreatePrimaryDataAddress,
#endif // CC_CreatePrimary

#if (PAD_LIST || CC_NV.GlobalWriteLock)
  (COMMAND_DESCRIPTOR_t *)_NV.GlobalWriteLockDataAddress,
#endif // CC_NV.GlobalWriteLock

#if (PAD_LIST || CC_GetCommandAuditDigest)
  (COMMAND_DESCRIPTOR_t *)_GetCommandAuditDigestDataAddress,
#endif // CC_GetCommandAuditDigest

#if (PAD_LIST || CC_NV.Increment)
  (COMMAND_DESCRIPTOR_t *)_NV.IncrementDataAddress,
#endif // CC_NV.Increment

#if (PAD_LIST || CC_NV.SetBits)
  (COMMAND_DESCRIPTOR_t *)_NV.SetBitsDataAddress,
#endif // CC_NV.SetBits
#if (PAD_LIST || CC_NV_Extend)
    (COMMAND_DESCRIPTOR_t *)_NV_ExtendDataAddress,
#endif // CC_NV_Extend

#if (PAD_LIST || CC_NV_Write)
    (COMMAND_DESCRIPTOR_t *)_NV_WriteDataAddress,
#endif // CC_NV_Write

#if (PAD_LIST || CC_NV_WriteLock)
    (COMMAND_DESCRIPTOR_t *)_NV_WriteLockDataAddress,
#endif // CC_NV_WriteLock

#if (PAD_LIST || CC_DictionaryAttackLockReset)
    (COMMAND_DESCRIPTOR_t *)_DictionaryAttackLockResetDataAddress,
#endif // CC_DictionaryAttackLockReset

#if (PAD_LIST || CC_DictionaryAttackParameters)
    (COMMAND_DESCRIPTOR_t *)_DictionaryAttackParametersDataAddress,
#endif // CC_DictionaryAttackParameters

#if (PAD_LIST || CC_NV_ChangeAuth)
    (COMMAND_DESCRIPTOR_t *)_NV_ChangeAuthDataAddress,
#endif // CC_NV_ChangeAuth

#if (PAD_LIST || CC_PCR_Event)
    (COMMAND_DESCRIPTOR_t *)_PCR_EventDataAddress,
#endif // CC_PCR_Event

#if (PAD_LIST || CC_PCR_Reset)
    (COMMAND_DESCRIPTOR_t *)_PCR_ResetDataAddress,
#endif // CC_PCR_Reset

#if (PAD_LIST || CC_SequenceComplete)
    (COMMAND_DESCRIPTOR_t *)_SequenceCompleteDataAddress,
#endif // CC_SequenceComplete

#if (PAD_LIST || CC_SetAlgorithmSet)
    (COMMAND_DESCRIPTOR_t *)_SetAlgorithmSetDataAddress,
#endif // CC_SetAlgorithmSet

#if (PAD_LIST || CC_SetCommandCodeAuditStatus)
    (COMMAND_DESCRIPTOR_t *)_SetCommandCodeAuditStatusDataAddress,
#endif // CC_SetCommandCodeAuditStatus

#if (PAD_LIST || CC_FieldUpgradeData)
    (COMMAND_DESCRIPTOR_t *)_FieldUpgradeDataDataAddress,
#endif // CC_FieldUpgradeData

#if (PAD_LIST || CC_IncrementalSelfTest)
    (COMMAND_DESCRIPTOR_t *)_IncrementalSelfTestDataAddress,
#endif // CC_IncrementalSelfTest

#if (PAD_LIST || CC_Startup)
    (COMMAND_DESCRIPTOR_t *)_StartupDataAddress,
#endif // CC_Startup

#if (PAD_LIST || CC_Shutdown)
    (COMMAND_DESCRIPTOR_t *)_ShutdownDataAddress,
#endif // CC_Shutdown

#if (PAD_LIST || CC_StirRandom)
    (COMMAND_DESCRIPTOR_t *)_StirRandomDataAddress,
#endif // CC_StirRandom

#if (PAD_LIST || CC_ActivateCredential)
    (COMMAND_DESCRIPTOR_t *)_ActivateCredentialDataAddress,
#endif // CC_ActivateCredential

#if (PAD_LIST || CC_Certify)
    (COMMAND_DESCRIPTOR_t *)_CertifyDataAddress,
#endif // CC_Certify

#if (PAD_LIST || CC_PolicyNV)
    (COMMAND_DESCRIPTOR_t *)_PolicyNVDataAddress,
#endif // CC_PolicyNV

#if (PAD_LIST || CC_CertifyCreation)
    (COMMAND_DESCRIPTOR_t *)_CertifyCreationDataAddress,
#endif // CC_CertifyCreation

#if (PAD_LIST || CC_Duplicate)
    (COMMAND_DESCRIPTOR_t *)_DuplicateDataAddress,
#endif // CC_Duplicate
#if (PAD_LIST || CC_GetTime)
  (COMMAND_DESCRIPTOR_t *)_GetTimeDataAddress,
#endif // CC_GetTime

#if (PAD_LIST || CC_GetSessionAuditDigest)
  (COMMAND_DESCRIPTOR_t *)_GetSessionAuditDigestDataAddress,
#endif // CC_GetSessionAuditDigest

#if (PAD_LIST || CC_NV_Read)
  (COMMAND_DESCRIPTOR_t *)_NV_ReadDataAddress,
#endif // CC_NV_Read

#if (PAD_LIST || CC_NV_ReadLock)
  (COMMAND_DESCRIPTOR_t *)_NV_ReadLockDataAddress,
#endif // CC_NV_ReadLock

#if (PAD_LIST || CC_ObjectChangeAuth)
  (COMMAND_DESCRIPTOR_t *)_ObjectChangeAuthDataAddress,
#endif // CC_ObjectChangeAuth

#if (PAD_LIST || CC_PolicySecret)
  (COMMAND_DESCRIPTOR_t *)_PolicySecretDataAddress,
#endif // CC_PolicySecret

#if (PAD_LIST || CC_Rewrap)
  (COMMAND_DESCRIPTOR_t *)_RewrapDataAddress,
#endif // CC_Rewrap

#if (PAD_LIST || CC_Create)
  (COMMAND_DESCRIPTOR_t *)_CreateDataAddress,
#endif // CC_Create

#if (PAD_LIST || CC_ECDH_ZGen)
  (COMMAND_DESCRIPTOR_t *)_ECDH_ZGenDataAddress,
#endif // CC_ECDH_ZGen

#if (PAD_LIST || (CC_HMAC || CC_MAC))
  # if CC_HMAC
    (COMMAND_DESCRIPTOR_t *)_HMACDataAddress,
  # endif

  # if CC_MAC
    (COMMAND_DESCRIPTOR_t *)_MACDataAddress,
  # endif

  # if (CC_HMAC || CC_MAC) > 1
    # error "More than one aliased command defined"
  # endif
#endif // CC_HMAC CC_MAC

#if (PAD_LIST || CC_Import)
  (COMMAND_DESCRIPTOR_t *)_ImportDataAddress,
#endif // CC_Import

#if (PAD_LIST || CC_Load)
  (COMMAND_DESCRIPTOR_t *)_LoadDataAddress,
#endif // CC_Load

#if (PAD_LIST || CC_Quote)
  (COMMAND_DESCRIPTOR_t *)_QuoteDataAddress,
#endif // CC_Quote

#if (PAD_LIST || CC_RSA_Decrypt)
  (COMMAND_DESCRIPTOR_t *)_RSA_DecryptDataAddress,
#endif // CC_RSA_Decrypt

#if (PAD_LIST) 0,
#endif //

#if (PAD_LIST || (CC_HMAC || CC_MAC))
  # if CC_HMAC_Start
    (COMMAND_DESCRIPTOR_t *)_HMAC_StartDataAddress,
  # endif

  # if CC_MAC_Start
    (COMMAND_DESCRIPTOR_t *)_MAC_StartDataAddress,
  # endif

  # if (CC_HMAC_Start || CC_MAC_Start) > 1
    # error "More than one aliased command defined"
  # endif
#endif // CC_HMAC_Start CC_MAC_Start

#if (PAD_LIST || CC_SequenceUpdate)
  (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_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_t *)_VerifySignatureDataAddress,
4288 #endif // CC_VerifySignature
4289 #if (PAD_LIST || CC_StartAuthSession)
4290 (COMMAND_DESCRIPTOR_t_t *)_ECC_ParametersDataAddress,
4291 #endif // CC_ECC_Parameters
4292 #if (PAD_LIST || CC_GetRandom)
4293 (COMMAND_DESCRIPTOR_t_t *)_GetRandomDataAddress,
4294 #endif // CC_GetRandom
4295 #if (PAD_LIST || CC_GetTestResult)
4296 (COMMAND_DESCRIPTOR_t_t *)_GetTestResultDataAddress,
4297 #endif // CC_GetTestResult
4298 #if (PAD_LIST || CC_PCR_Read)
4299 (COMMAND_DESCRIPTOR_t_t *)_PCR_ReadDataAddress,
4300 #endif // CC_PCR_Read
4301 #if (PAD_LIST || CC_PolicyPCR)
4302 (COMMAND_DESCRIPTOR_t_t *)_PolicyPCRDataAddress,
4303 #endif // CC_PolicyPCR
4304 #if (PAD_LIST || CC_PolicyRestart)
4305 (COMMAND_DESCRIPTOR_t_t *)_PolicyRestartDataAddress,
4306 #endif // CC_PolicyRestart
4307 #if (PAD_LIST || CC_PolicyPhysicalPresence)
4308 (COMMAND_DESCRIPTOR_t_t *)_PolicyPhysicalPresenceDataAddress,
4309 #endif // CC_PolicyPhysicalPresence
4310 #if (PAD_LIST || CC_PolicyDuplicationSelect)
4311 (COMMAND_DESCRIPTOR_t_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 #endif // _COMMAND_TABLE_DISPATCH_
5.7 Commands.h

```c
#ifndef COMMANDS_H_
#define _COMMANDS_H_

Start-up

#ifdef TPM_CC_Startup
#include "Startup_fp.h"
#endif

#ifdef TPM_CC_Shutdown
#include "Shutdown_fp.h"
#endif

Testing

#ifdef TPM_CC_SelfTest
#include "SelfTest_fp.h"
#endif

#ifdef TPM_CC_IncrementalSelfTest
#include "IncrementalSelfTest_fp.h"
#endif

#ifdef TPM_CC_GetTestResult
#include "GetTestResult_fp.h"
#endif

Session Commands

#ifdef TPM_CC_StartAuthSession
#include "StartAuthSession_fp.h"
#endif

#ifdef TPM_CC_PolicyRestart
#include "PolicyRestart_fp.h"
#endif

Object Commands

#ifdef TPM_CC_Create
#include "Create_fp.h"
#endif

#ifdef TPM_CC_Load
#include "Load_fp.h"
#endif

#ifdef TPM_CC_LoadExternal
#include "LoadExternal_fp.h"
#endif

#ifdef TPM_CC_ReadPublic
#include "ReadPublic_fp.h"
#endif

#ifdef TPM_CC_ActivateCredential
#include "ActivateCredential_fp.h"
#endif

#ifdef TPM_CC_MakeCredential
#include "MakeCredential_fp.h"
#endif

#ifdef TPM_CC_Unseal
#include "Unseal_fp.h"
#endif

#ifdef TPM_CC_ObjectChangeAuth
#include "ObjectChangeAuth_fp.h"
#endif

#ifdef TPM_CC_CreateLoaded
#include "CreateLoaded_fp.h"
#endif
#endif
```
Duplication Commands

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

Asymmetric Primitives

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

Symmetric Primitives

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

Random Number Generator

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

Hash/HMAC/Event Sequences
#ifdef TPM_CC_HMAC_Start
#include "HMAC_Start_fp.h"
#endif

#ifdef TPM_CC_MAC_Start
#include "MAC_Start_fp.h"
#endif

#ifdef TPM_CC_HashSequenceStart
#include "HashSequenceStart_fp.h"
#endif

#ifdef TPM_CC_SequenceUpdate
#include "SequenceUpdate_fp.h"
#endif

#ifdef TPM_CC_SequenceComplete
#include "SequenceComplete_fp.h"
#endif

#ifdef TPM_CC_EventSequenceComplete
#include "EventSequenceComplete_fp.h"
#endif

Attestation Commands

#ifdef TPM_CC_Certify
#include "Certify_fp.h"
#endif

#ifdef TPM_CC_CertifyCreation
#include "CertifyCreation_fp.h"
#endif

#ifdef TPM_CC_Quote
#include "Quote_fp.h"
#endif

#ifdef TPM_CC_GetSessionAuditDigest
#include "GetSessionAuditDigest_fp.h"
#endif

#ifdef TPM_CC_GetCommandAuditDigest
#include "GetCommandAuditDigest_fp.h"
#endif

#ifdef TPM_CC_GetTime
#include "GetTime_fp.h"
#endif

#ifdef TPM_CC_CertifyX509
#include "CertifyX509_fp.h"
#endif

Ephemeral EC Keys

#ifdef TPM_CC_Commit
#include "Commit_fp.h"
#endif

#ifdef TPM_CC_EC_Ephemeral
#include "EC_Ephemeral_fp.h"
#endif

Signing and Signature Verification

#ifdef TPM_CC_VerifySignature
#include "VerifySignature_fp.h"
#endif

#ifdef TPM_CC_Sign
#include "Sign_fp.h"
#endif

Command Audit

#ifdef TPM_CC_SetCommandCodeAuditStatus
#include "SetCommandCodeAuditStatus_fp.h"
#endif

Integrity Collection (PCR)

ifdef TPM_CC_PCR_Extend
#include "PCR_Extend_fp.h"
endif

ifdef TPM_CC_PCR_Event
#include "PCR_Event_fp.h"
endif

ifdef TPM_CC_PCR_Read
#include "PCR_Read_fp.h"
endif

ifdef TPM_CC_PCR_Allocate
#include "PCR_Allocate_fp.h"
endif

ifdef TPM_CC_PCR_SetAuthPolicy
#include "PCR_SetAuthPolicy_fp.h"
endif

ifdef TPM_CC_PCR_SetAuthValue
#include "PCR_SetAuthValue_fp.h"
endif

ifdef TPM_CC_PCR_Reset
#include "PCR_Reset_fp.h"
endif

Enhanced Authorization (EA) Commands

ifdef TPM_CC_PolicySigned
#include "PolicySigned_fp.h"
endif

ifdef TPM_CC_PolicySecret
#include "PolicySecret_fp.h"
endif

ifdef TPM_CC_PolicyTicket
#include "PolicyTicket_fp.h"
endif

ifdef TPM_CC_PolicyOR
#include "PolicyOR_fp.h"
endif

ifdef TPM_CC_PolicyPCR
#include "PolicyPCR_fp.h"
endif

ifdef TPM_CC_PolicyLocality
#include "PolicyLocality_fp.h"
endif

ifdef TPM_CC_PolicyNV
#include "PolicyNV_fp.h"
endif

ifdef TPM_CC_PolicyCounterTimer
#include "PolicyCounterTimer_fp.h"
endif

ifdef TPM_CC_PolicyCommandCode
#include "PolicyCommandCode_fp.h"
endif

ifdef TPM_CC_PolicyPhysicalPresence
#include "PolicyPhysicalPresence_fp.h"
endif

ifdef TPM_CC_PolicyCpHash
#include "PolicyCpHash_fp.h"
endif

ifdef TPM_CC_PolicyNameHash
#include "PolicyNameHash_fp.h"
endif
#ifdef TPM_CC_PolicyDuplicationSelect
#include "PolicyDuplicationSelect_fp.h"
#endif

#include "PolicyAuthorize_fp.h"
#endif

#include "PolicyAuthValue_fp.h"
#endif

#include "PolicyPassword_fp.h"
#endif

#include "PolicyGetDigest_fp.h"
#endif

#include "PolicyNvWritten_fp.h"
#endif

#include "PolicyTemplate_fp.h"
#endif

#include "PolicyAuthorizeNV_fp.h"
#endif

Hierarchy Commands

#include "CreatePrimary_fp.h"
#endif

#include "HierarchyControl_fp.h"
#endif

#include "SetPrimaryPolicy_fp.h"
#endif

#include "ChangePPS_fp.h"
#endif

#include "ChangeEPS_fp.h"
#endif

#include "Clear_fp.h"
#endif

#include "ClearControl.fp.h"
#endif

#include "HierarchyChangeAuth_fp.h"
#endif

Dictionary Attack Functions

#include "DictionaryAttackLockReset_fp.h"
#endif

#include "DictionaryAttackParameters_fp.h"
#endif

Miscellaneous Management Functions

#include "TPM_CC_PP_Commands"
#include "PP_Commands_fp.h"
#endif

#ifdef TPM_CC_SetAlgorithmSet
#include "SetAlgorithmSet_fp.h"
#endif

Field Upgrade
#ifdef TPM_CC_FieldUpgradeStart
#include "FieldUpgradeStart_fp.h"
#endif
#ifdef TPM_CC_FieldUpgradeData
#include "FieldUpgradeData_fp.h"
#endif
#ifdef TPM_CC_FirmwareRead
#include "FirmwareRead_fp.h"
#endif

Context Management
#ifdef TPM_CC_ContextSave
#include "ContextSave_fp.h"
#endif
#ifdef TPM_CC_ContextLoad
#include "ContextLoad_fp.h"
#endif
#ifdef TPM_CC_FlushContext
#include "FlushContext_fp.h"
#endif
#ifdef TPM_CC_EvictControl
#include "EvictControl_fp.h"
#endif

Clocks and Timers
#ifdef TPM_CC_ReadClock
#include "ReadClock_fp.h"
#endif
#ifdef TPM_CC_ClockSet
#include "ClockSet_fp.h"
#endif
#ifdef TPM_CC_ClockRateAdjust
#include "ClockRateAdjust_fp.h"
#endif

Capability Commands
#ifdef TPM_CC_GetCapability
#include "GetCapability_fp.h"
#endif
#ifdef TPM_CC_TestParms
#include "TestParms_fp.h"
#endif

Non-volatile Storage
#ifdef TPM_CC_NV_DefineSpace
#include "NV_DefineSpace_fp.h"
#endif
#ifdef TPM_CC_NV_UndefineSpace
#include "NV.UndefineSpace_fp.h"
#endif
#ifdef TPM_CC_NV_UndefineSpaceSpecial
#include "NV_UndefineSpaceSpecial_fp.h"
#endif

ifdef TPM_CC_NV_ReadPublic
#include "NV_ReadPublic_fp.h"
#endif

ifdef TPM_CC_NV_Write
#include "NV_Write_fp.h"
endif

ifdef TPM_CC_NV_Increment
#include "NV_Increment_fp.h"
endif

ifdef TPM_CC_NV_Extend
#include "NV_Extend_fp.h"
endif

ifdef TPM_CC_NV_SetBits
#include "NV_SetBits_fp.h"
endif

ifdef TPM_CC_NV_WriteLock
#include "NV_WriteLock_fp.h"
endif

ifdef TPM_CC_NV_GlobalWriteLock
#include "NV_GlobalWriteLock_fp.h"
endif

ifdef TPM_CC_NV_Read
#include "NV_Read_fp.h"
endif

ifdef TPM_CC_NV_ReadLock
#include "NV_ReadLock_fp.h"
endif

ifdef TPM_CC_NV_ChangeAuth
#include "NV_ChangeAuth_fp.h"
endif

ifdef TPM_CC_NV_Certify
#include "NV_Certify_fp.h"
endif

endif

Attached Components

ifdef TPM_CC_AC_GetCapability
#include "AC_GetCapability_fp.h"
endif

ifdef TPM_CC_AC_Send
#include "AC_Send_fp.h"
endif

ifdef TPM_CC_Policy_AC_SendSelect
#include "Policy_AC_SendSelect_fp.h"
endif

Vendor Specific

ifdef TPM_CC_Vendor_TCG_Test
#include "Vendor_TCG_Test_fp.h"
endif

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.

```c
#define __COMPILER_DEPENDENCIES_H_
#endif
#define GCC
#define _MSC_VER
#define WIN32
#endif
#define REVERSE_ENDIAN_16(_Number) _byteswap_ushort(_Number)
#define REVERSE_ENDIAN_32(_Number) _byteswap_ulong(_Number)
#define REVERSE_ENDIAN_64(_Number) _byteswap_uint64(_Number)
#endif
#endif
#ifndef _MSC_VER
#ifndef WINAPI
#define WINAPI
#define __pragma(x)
#define NO_CRT_STDIO_INLINE
#endif
#define _REDUCE_WARNING_LEVEL_(n) \
__pragma(warning(push, n))
#define _NORMAL_WARNING_LEVEL_ \
__pragma(warning(pop))
#include <stdint.h>
#endif
#endif
#ifndef _MSC_VER
#ifndef WINAPI
#define WINAPI
#define __pragma(x)
#define NO_CRT_STDIO_INLINE
#endif
#define _REDUCE_WARNING_LEVEL_(n) \
__pragma(warning(push, n))
#define _NORMAL_WARNING_LEVEL_ \
__pragma(warning(pop))
#include <stdint.h>
#endif
```

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

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

```
#define REVERSE_ENDIAN_16(_Number) _byteswap_ushort(_Number)
#define REVERSE_ENDIAN_32(_Number) _byteswap_ulong(_Number)
#define REVERSE_ENDIAN_64(_Number) _byteswap_uint64(_Number)
```

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

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

```
#define _REDUCE_WARNING_LEVEL_(n) \
__pragma(warning(push, n))
#define _NORMAL_WARNING_LEVEL_ \
__pragma(warning(pop))
#include <stdint.h>
```

Restore the compiler warning level
# define REVERSE_ENDIAN_32(_Number) __builtin_bswap32(_Number)
# define REVERSE_ENDIAN_64(_Number) __builtin_bswap64(_Number)
#endif
#if defined(__GNUC__)
# define NORETURN __attribute__((noreturn))
#include <stdint.h>
#endif

Things that are not defined should be defined as NULL

#ifndef NORETURN
#define NORETURN
#endif
#ifndef LIB_EXPORT
#define LIB_EXPORT
#endif
#ifndef LIB_IMPORT
#define LIB_IMPORT
#endif
#ifndef _REDUCE_WARNING_LEVEL_
#define _REDUCE_WARNING_LEVEL_(n)
#endif
#ifndef _NORMAL_WARNING_LEVEL_
#define _NORMAL_WARNING_LEVEL_
#endif
#ifndef NOT_REFERENCED
#define NOT_REFERENCED(x) (x = x)
#endif
#ifndef _POSIX_
typedef int SOCKET;
#endif
#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.

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

5.9.2 Includes

#ifndef GLOBAL_H
#define GLOBAL_H
#ifdef GLOBAL_C
#define EXTERN extern
#define INITIALIZER(_name_)
#else
#define EXTERN extern _REDUCE_WARNING_LEVEL_(2)
#define INITIALIZER(_value_) = _value_
#endif

#include <string.h>
#include <stddef.h>

#if SIMULATION
#undef CONTEXT_SLOT
#define CONTEXT_SLOT UINT8
#else
#define INITIALIZER(_value_)
#endif

#include "Capabilities.h"
#include "TpmTypes.h"
#include "CommandAttributes.h"
#include "CryptTest.h"
#include "BnValues.h"
#include "CryptHash.h"
#include "CryptSym.h"
#include "CryptRand.h"
#include "CryptEcc.h"
#include "CryptRsa.h"
#include "CryptTest.h"
#include "TpmError.h"
#include "NV.h"

//** Defines and Types

//*** Size Types
// These types are used to differentiate the two different size values used.
// NUMBYTES is used when a size is a number of bytes (usually a TPM2B)
typedef UINT16   NUMBYTES;

//*** Other Types
// An AUTH_VALUE is a BYTE array containing a digest (TPMU_HA)
typedef BYTE AUTH_VALUE[sizeof(TPMU_HA)];

A TIME_INFO is a BYTE array that can contain a TPMS_TIME_INFO

typedef BYTE TIME_INFO[sizeof(TPMS_TIME_INFO)];

A NAME is a BYTE array that can contain a TPMU_NAME

typedef BYTE NAME[sizeof(TPMU_NAME)];

Definition for a PROOF value

TPM2B_TYPE(PROOF, PROOF_SIZE);

Definition for a Primary Seed value

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.

#if CLOCK_STOPS
typedef UINT64 CLOCK_NONCE;
#else
typedef UINT32 CLOCK_NONCE;
#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.

typedef struct {
    unsigned publicOnly : 1;       //0) SET if only the public portion of an object is loaded
    unsigned epsHierarchy : 1;     //1) SET if the object belongs to EPS Hierarchy
    unsigned ppsHierarchy : 1;     //2) SET if the object belongs to PPS Hierarchy
    unsigned spsHierarchy : 1;     //3) SET if the object belongs to SPS Hierarchy
    unsigned evict : 1;            //4) SET if the object is a platform or owner evict object. Platform-
                                    // evict object belongs to PPS hierarchy, owner-evict object
                                    // belongs to SPS or EPS hierarchy.
}
// This bit is also used to mark a completed sequence object so it will be flush when the SequenceComplete command succeeds.
unsigned primary : 1; //5) SET for a primary object
unsigned temporary : 1; //6) SET for a temporary object
unsigned stClear : 1; //7) SET for an stClear object
unsigned hmacSeq : 1; //8) SET for an HMAC or MAC sequence
unsigned hashSeq : 1; //9) SET for a hash sequence object
unsigned eventSeq : 1; //10) SET for an event sequence object
unsigned ticketSafe : 1; //11) SET if a ticket is safe to create for hash sequence object
unsigned firstBlock : 1; //12) SET if the first block of hash data has been received. It works with ticketSafe bit
unsigned isParent : 1; //13) SET if the key has the proper attributes to be a parent key
unsigned privateExp : 1; //14) SET when the private exponent of an RSA key has been validated.
unsigned not_used_14 : 1; //15) SET when the slot is occupied.
unsigned occupied : 1; //16) SET when the key is a derivation parent
unsigned derivation : 1; //17) SET when the object is loaded with TPM2_LoadExternal();
unsigned external : 1;

#define RSA_prime_flag 0x8000

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.

typedef struct OBJECT
{
    // The attributes field is required to be first followed by the publicArea.
    // This allows the overlay of the object structure and a sequence structure
    OBJECT_ATTRIBUTES attributes; // object attributes
    TPMT_PUBLIC publicArea; // public area of an object
    TPMT_SENSITIVE sensitive; // sensitive area of an object
    TPM2B_NAME qualifiedName; // object qualified name
    TPMI_DH_OBJECT evictHandle; // if the object is an evict object, the original handle is kept here.
    // The 'working' handle will be the handle of an object slot.
    TPM2B_NAME name; // Name of the object name. Kept here
    // to avoid repeatedly computing it.
} 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

```c
typedef struct HASH_OBJECT
{
    OBJECT_ATTRIBUTES attributes; // The attributes of the HASH object
    TPMI_ALG_PUBLIC type; // algorithm
    TPMI_ALG_HASH nameAlg; // name algorithm
    TPMA_OBJECT objectAttributes; // object attributes

    // The data below is unique to a sequence object
    TPM2B_AUTH auth; // authorization for use of sequence

    union
    {
        HASH_STATE hashState[HASH_COUNT];
        HMAC_STATE hmacState;
    }

} 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()

```c
typedef union ANY_OBJECT
{
    OBJECT entity;
    HASH_OBJECT hash;
} ANY_OBJECT;
```

5.9.4 AUTH_DUP Types

These values are used in the authorization processing.

```c
typedef UINT32 AUTH_ROLE;
declare AUTH_NONE ((AUTH_ROLE)(0))
declare AUTH_USER ((AUTH_ROLE)(1))
declare AUTH_ADMIN ((AUTH_ROLE)(2))
declare 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.

```c
typedef struct SESSION_ATTRIBUTES {
    unsigned isPolicy : 1; // (1) SET if the session may only be used for policy
    unsigned isAudit : 1; // (2) SET if the session is used for audit tracking
    unsigned isBound : 1; // (3) SET if the session is bound to with an entity. This attribute will be CLEAR if either isPolicy or isAudit is SET.
    unsigned isCpHashDefined : 1; // (3) SET if the cpHash has been defined
    unsigned isAuthValueNeeded : 1; // (5) SET if the authValue is required for computing the session HMAC. This attribute is not SET unless 'isPolicy' is SET.
    unsigned isPasswordNeeded : 1; // (6) SET if a password authValue is required for authorization. This attribute is not SET unless 'isPolicy' is SET.
    unsigned isPPRequired : 1; // (7) SET if physical presence is required to be asserted when the authorization is checked. This attribute is not SET unless 'isPolicy' is SET.
    unsigned isTrialPolicy : 1; // (8) SET if the policy session is created for trial of the policy's policyHash generation. This attribute is not SET unless 'isPolicy' is SET.
    unsigned isDaBound : 1; // (9) SET if the bind entity had noDA CLEAR. If this is SET, then an authorization failure using this session will count against lockout even if the object being authorized is exempt from DA.
    unsigned isLockoutBound : 1; // (10) SET if the session is bound to lockoutAuth.
    unsigned includeAuth : 1; // (11) This attribute is SET when the authValue of an object is to be included in the computation of the HMAC key for the command and response computations. (was 'requestWasBound')
    unsigned checkNvWritten : 1; // (12) SET if the TPMA_NV_WRITTEN attribute needs to be checked when the policy is used for authorization for NV access. If this is SET for any other type, the policy will fail.
    unsigned nvWrittenState : 1; // (13) SET if TPMA_NV_WRITTEN is required to be SET. Used when 'checkNvWritten' is SET
    unsigned isTemplateSet : 1; // (14) SET if the templateHash needs to be checked for Create, CreatePrimary, or CreateLoaded.
} 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.

```c
typedef struct SESSION
{
    SESSION_ATTRIBUTES attributes;  // session attributes
    UINT32 pcrCounter;             // PCR counter value when PCR is included (policy session)
    // If no PCR is included, this value is 0.
    UINT64 startTime;              // The value in g_time when the session was started (policy session)
    UINT64 timeout;                // The timeout relative to g_time
    // There is no timeout if this value is 0.
    CLOCK_NONCE epoch;             // The g_clockEpoch value when the session was started.
    // If g_clockEpoch does not match this value when the timeout is used, then
    // then the command will fail.
    TPM_CC commandCode;            // command code (policy session)
    TPM_ALG_ID authHashAlg;        // Session hash algorithm
    TPMA_LOCALITY commandLocality;  // command locality (policy session)
    TPM_SYM_DEF symmetric;         // session symmetric algorithm (if any)
    TPM2B_AUTH sessionKey;         // session secret value used for this session
    CLOCK_NONCE epoch;             // The g_clockEpoch value when the
    // session was started. If g_clockEpoch does not match this value when the
    // timeout is used, then
    // then the command will fail.
    union
    {
        TPM2B_NAME boundEntity;  // value used to track the entity to which the session is bound
        TPM2B_DIGEST cpHash;     // the required cpHash value for the command being authorized
        TPM2B_DIGEST nameHash;   // the required nameHash
        TPM2B_DIGEST templateHash;  // the required template for creation
    } u1;
    union
    {
        TPM2B_DIGEST auditDigest;  // audit session digest
        TPM2B_DIGEST policyDigest; // policyHash
    } u2;
} SESSION;
#define EXPIRES_ON_RESET INT32_MIN
#define TIMEOUT_ON_RESET UINT64_MAX
#define EXPIRES_ON_RESTART (INT32_MIN + 1)
#define TIMEOUT_ON_RESTART (UINT64_MAX - 1)
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.

```c
typedef struct PCR_SAVE
{
    #if ALG_SHA1
```
/* This counter increments whenever the PCR are updated. */
/* NOTE: A platform-specific specification may designate 
   certain PCR changes as not causing this counter 
   to increment. */
UINT32 pcrCounter;

5.9.6.2 PCR_POLICY

#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.

typedef struct PCR_POLICY
{
  TPMI_ALG_HASH hashAlg[NUM_POLICY_PCR_GROUP];
  TPM2B_DIGEST a;
  TPM2B_DIGEST policy[NUM_POLICY_PCR_GROUP];
} PCR_POLICY;
#endif

5.9.6.3 PCR_AUTHVALUE

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

typedef struct PCR_AUTH_VALUE
{
  TPM2B_DIGEST auth[NUM_AUTHVALUE_PCR_GROUP];
} 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().

typedef enum
{
  SU_RESET,
  SU_RESTART,
  SU_RESUME
} 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.

```c
typedef struct NV_INDEX {
    TPMS_NV_PUBLIC publicArea;
    TPM2B_AUTH authValue;
} 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.

```c
typedef UINT32 NV_REF;
typedef BYTE *NV_RAM_REF;
```

5.9.8.3 NV_PIN

This structure deals with the possible endianess 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.

```c
#if BIG_ENDIAN_TPM
typedef struct {
    UINT32 pinCount;
    UINT32 pinLimit;
} PIN_DATA;
#else
typedef struct {
    UINT32 pinLimit;
    UINT32 pinCount;
} PIN_DATA;
#endif
typedef union {
    UINT64 intVal;
    PIN_DATA pin;
} 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.

```c
#if ALG_ECC
```
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

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.

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.

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.

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.
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 \texttt{gp.orderlyState}. These are hacks and are being used to avoid having to change the layout of \texttt{gp}. The \texttt{PRE_STARTUP_FLAG} indicates that a \texttt{TPM\_Hash\_Start()/\_Data/\_End} sequence was received after \texttt{TPM\_Init()} but before \texttt{TPM2\_StartUp()}. \texttt{STARTUP\_LOCALITY\_3} indicates that the last \texttt{TPM2\_StartUp()} was received at locality 3. These flags are only relevant if after a \texttt{TPM2\_Shutdown(STATE)}.

\begin{verbatim}
#define PRE_STARTUP_FLAG 0x8000
#define STARTUP_LOCALITY_3 0x4000
\end{verbatim}

5.9.10.15 \texttt{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 \texttt{failedTries} on the next non-orderly startup. This bit is merged with \texttt{gp.orderlyState} when that \texttt{gp.orderly} is set to \texttt{SU\_NONE\_VALUE}.

\begin{verbatim}
EXTERN BOOL                 g_daUsed;
\end{verbatim}

5.9.10.16 \texttt{g\_updateNV}

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

\begin{verbatim}
typedef BYTE        UPDATE_TYPE;
#define UT_NONE     (UPDATE_TYPE)0
#define UT_NV       (UPDATE_TYPE)1
#define UT_ORDERLY  (UPDATE_TYPE)(UT_NV + 2)
EXTERN UPDATE_TYPE          g_updateNV;
\end{verbatim}

5.9.10.17 \texttt{g\_powerWasLost}

This flag is used to indicate if the power was lost. It is SET in \texttt{TPM\_Init()}. This flag is cleared by \texttt{TPM2\_StartUp()} after all power-lost activities are completed.

\begin{verbatim}
EXTERN BOOL             g_powerWasLost;
\end{verbatim}

5.9.10.18 \texttt{g\_clearOrderly}

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

\begin{verbatim}
EXTERN BOOL             g_clearOrderly;
\end{verbatim}
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
359

// Hierarchy authPolicies
TPMI_ALG_HASH ownerAlg;
TPMI_ALG_HASH endorsementAlg;
TPMI_ALG_HASH lockoutAlg;
TPM2B_DIGEST ownerPolicy;
TPM2B_DIGEST endorsementPolicy;
TPM2B_DIGEST lockoutPolicy;

// Hierarchy authValues
TPM2B_AUTH ownerAuth;
TPM2B_AUTH endorsementAuth;
TPM2B_AUTH lockoutAuth;

// Primary Seeds
TPM2B_SEED EPSeed;
TPM2B_SEED SPSeed;
TPM2B_SEED PPSeed;
// Note there is a nullSeed in the state_reset memory.

// Hierarchy proofs
TPM2B_PROOF phProof;
TPM2B_PROOF shProof;
TPM2B_PROOF ehProof;
// Note there is a nullProof in the state_reset memory.

// Reset Events

// This counter increments on each TPM Reset. The counter is reset by
// TPM2_Clear().
UINT32 resetCount;

// PCR

// This structure indicates the allocation of PCR. The structure contains a
// list of PCR allocations for each implemented algorithm. If no PCR are
// allocated for an algorithm, a list entry still exists but the bit map
// will contain no SET bits.
TPML_PCR_SELECTION pcrAllocated;

// Physical Presence

// The PP_LIST type contains a bit map of the commands that require physical
// to be asserted when the authorization is evaluated. Physical presence will be
// checked if the corresponding bit in the array is SET and if the authorization
// handle is TPM_RH_PLATFORM.
BYTE ppList[(COMMAND_COUNT + 7) / 8];
// Dictionary attack values
// These values are used for dictionary attack tracking and control.

UINT32 failedTries; // the current count of unexpired authorization failures

UINT32 maxTries; // number of unexpired authorization failures before the TPM is in lockout

UINT32 recoveryTime; // time between authorization failures before failedTries is decremented

UINT32 lockoutRecovery; // time that must expire between authorization failures associated with lockoutAuth

BOOL lockOutAuthEnabled; // TRUE if use of lockoutAuth is allowed

//****************************************************************************
// Orderly State
//****************************************************************************
// The orderly state for current cycle

TPM_SU orderlyState;

//****************************************************************************
// Command audit values.
//****************************************************************************

BYTE auditCommands[((COMMAND_COUNT + 1) + 7) / 8];

TPMI_ALG_HASH auditHashAlg;

UINT64 auditCounter;

//****************************************************************************
// Algorithm selection
//****************************************************************************

UINT32 algorithmSet;

//****************************************************************************
// Firmware version
//****************************************************************************
// The firmwareV1 and firmwareV2 values are instanced in TimeStamp.c. This is a scheme used in development to allow determination of the linker build time of the TPM. An actual implementation would implement these values in a way that is consistent with vendor needs. The values are maintained in RAM for simplified access with a master version in NV. These values are modified in a vendor-specific way.

// g_firmwareV1 contains the more significant 32-bits of the vendor version number.
// In the reference implementation, if this value is printed as a hex value, it will have the format of YYYYMMDD

UINT32 firmwareV1;

//****************************************************************************
// g_firmwareV2 contains the less significant 32-bits of the vendor version number.
// In the reference implementation, if this value is printed as a hex value, it will have the format of 00 HH MM SS

UINT32 firmwareV2;

//****************************************************************************
// Timer Epoch
//****************************************************************************

// timeEpoch contains a nonce that has a vendor-specific size (should not be
5.9.11.3 ORDERLY_DATA

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

```c
typedef struct orderly_data {
    //***************************************************************************
    // TIME
    //***************************************************************************
    UINT64 clock; // The orderly version of clock
    TPMI_YES_NO clockSafe; // Indicates if the clock value is safe.
    // In many implementations, the quality of the entropy available is not that
    // high. To compensate, the current value of the drbgState can be saved and
    // restored on each power cycle. This prevents the internal state from reverting
    // to the initial state on each power cycle and starting with a limited amount
    // of entropy. By keeping the old state and adding entropy, the entropy will
    // accumulate.
    DRBG_STATE drbgState;
    // These values allow the accumulation of self-healing time across orderly shutdown
    // of the TPM.
    #if ACCUMULATE_SELF_HEAL_TIMER
    UINT64 selfHealTimer; // current value of s_selfHealTimer
    UINT64 lockoutTimer; // current value of s_lockoutTimer
    UINT64 time; // current value of g_time at shutdown
    #endif // ACCUMULATE_SELF_HEAL_TIMER
}
#else
#define s_selfHealTimer go.selfHealTimer
#define s_lockoutTimer go.lockoutTimer
#endif
#define drbgDefault go.drbgState
EXTERN ORDERLY_DATA go;
```

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).
typedef struct state_clear_data
{
    //***********************************************************************
    //           Hierarchy Control
    //***********************************************************************
    BOOL shEnable;    // default reset is SET
    BOOL ehEnable;    // default reset is SET
    BOOL phEnableNV; // default reset is SET
    TPMI_ALG_HASH platformAlg; // default reset is TPM_ALG_NULL
    TPM2B_DIGEST platformPolicy; // default reset is an Empty Buffer
    TPM2B_AUTH platformAuth; // default reset is an Empty Buffer

    //***********************************************************************
    //           PCR
    //***********************************************************************
    PCR_SAVE pcrSave; // default reset is 0...0

    // This structure hold the authorization values for those PCR that have an
    // update authorization.
    // This implementation only supports a single group of PCR controlled by
    // authorization. If more are required, then this structure would be changed to
    // an array.
    PCR_AUTHVALUE pcrAuthValues;
} STATE_CLEAR_DATA;
EXTERN STATE_CLEAR_DATA gc;

5.9.11.5 State Reset Data

This structure contains data 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.

typedef struct state_reset_data
{
    //***********************************************************************
    //           Hierarchy Control
    //***********************************************************************
    TPM2B_PROOF nullProof; // The proof value associated with
    TPM2B_SEED nullSeed; // the TPM_RH_NULL hierarchy. The
    // default reset value is from the RNG.

    //***********************************************************************
    //           Context
    //***********************************************************************
    UINT32 clearCount; // The default reset value is 0.
    UINT64 objectContextID; // This is the context ID for a saved
    // object context. The default reset
    // value is 0.

    #ifndef NDEBUG
    #undef CONTEXT_SLOT
    #define CONTEXT_SLOT BYTE
    #endif
CONTEXT_SLOT contextArray[MAX_ACTIVE_SESSIONS];  // This array contains
// the values used to track
// contexts (see
// Session.c in for details). The
// default reset value is {0}.

CONTEXT_COUNTER contextCounter;  // This is the value from which the
// 'contextID' is derived. The
// default reset value is {0}.

// Command Audit

// When an audited command completes, ExecuteCommand() checks the return
// value. If it is TPM_RC_SUCCESS, and the command is an audited command, the
// TPM will extend the cpHash and rpHash for the command to this value. If this
// digest was the Zero Digest before the cpHash was extended, the audit counter
// is incremented.

TPM2B_DIGEST commandAuditDigest;  // This value is set to an Empty Digest
// by TPM2_GetCommandAuditDigest() or a
// TPM Reset.

//*****************************************************************************
// Boot counter
//*****************************************************************************

UINT32 restartCount;  // This counter counts TPM Restarts.
// The default reset value is 0.

//*****************************************************************************
// PCR
//*****************************************************************************

// This counter increments whenever the PCR are updated. This counter is preserved
// across TPM Resume even though the PCR are not preserved. This is because
// sessions remain active across TPM Restart and the count value in the session
// is compared to this counter so this counter must have values that are unique
// as long as the sessions are active.
// NOTE: A platform-specific specification may designate that certain PCR changes
// do not increment this counter to increment.

UINT32 pcrCounter;  // The default reset value is 0.

#if ALG_ECC
//*****************************************************************************
// ECDAA
//*****************************************************************************

UINT64 commitCounter;  // This counter increments each time
// TPM2_Commit() returns
// TPM_RC_SUCCESS. The default reset
// value is 0.

TPM2B_NONCE commitNonce;  // This random value is used to compute
// the commit values. The default reset
// value is from the RNG.

// This implementation relies on the number of bits in g_commitArray being a
// power of 2 (8, 16, 32, 64, etc.) and no greater than 64K.
BYTE commitArray[16];  // The default reset value is {0}.
#endif ALG_ECC

) STATE_RESET_DATA;
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, "ECDAA 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.

```
EXTERN CRYPTO_SELF_TEST_STATE g_cryptoSelfTestState;
```

5.9.15 From Manufacture.c

```
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.

```
EXTERN BOOL g_initialized;
```

5.9.16 Private data

5.9.16.1 From SessionProcess.c

```
#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

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

Array of authorization session attributes

```
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

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

Array of nonces provided by the caller for the corresponding sessions

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

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

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

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

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

Special value to indicate an undefined session index

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

Index of the session used for encryption of a response parameter
EXTERN UINT32 s_encryptSessionIndex;

Index of the session used for decryption of a command parameter

EXTERN UINT32 s_decryptSessionIndex;

Index of a session used for audit

EXTERN UINT32 s_auditSessionIndex;

The \textit{cpHash} for command audit

#ifdef TPM_CC_GetCommandAuditDigest
EXTERN TPM2B_DIGEST s_cpHashForCommandAudit;
#endif

Flag indicating if NV update is pending for the \textit{lockOutAuthEnabled} or \textit{failedTries} DA parameter

EXTERN BOOL s_DAPendingOnNV;

#endif // SESSION_PROCESS_C

\textbf{5.9.16.2 From DA.c}

\begin{verbatim}
#define DA_C || defined GLOBAL_C || defined MANUFACTURE_C
\end{verbatim}

This variable holds the accumulated time since the last time that \textit{failedTries} was decremented. This value is in millisecond.

\begin{verbatim}
#ifndef ACCUMULATE_SELF_HEAL_TIMER
EXTERN UINT64 s_selfHealTimer;
#endif
\end{verbatim}

This variable holds the accumulated time that the \textit{lockoutAuth} has been blocked.

EXTERN UINT64 s_lockoutTimer;

#endif // ACCUMULATE_SELF_HEAL_TIMER

#endif // DA_C

\textbf{5.9.16.3 From NV.c}

\begin{verbatim}
#define NV_C || defined GLOBAL_C
\end{verbatim}

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

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.

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.

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
typedef struct
792  {
793    #if  ALG_SHA1
794      // SHA1 PCR
795      BYTE sha1Pcr[SHA1_DIGEST_SIZE];
796    #endif
797    #if  ALG_SHA256
798      // SHA256 PCR
799      BYTE sha256Pcr[SHA256_DIGEST_SIZE];
800    #endif
801    #if  ALG_SHA384
802      // SHA384 PCR
803      BYTE sha384Pcr[SHA384_DIGEST_SIZE];
804    #endif
805    #if  ALG_SHA512
806      // SHA512 PCR
807      BYTE sha512Pcr[SHA512_DIGEST_SIZE];
808    #endif
809    #if  ALG_SM3_256
810      // SHA256 PCR
811      BYTE sm3_256Pcr[SM3_256_DIGEST_SIZE];
812    #endif
813  } PCR;
814
typedef struct
815  {
816    unsigned int stateSave : 1;  // if the PCR value should be
817                      // saved in state save
818    unsigned int resetLocality : 5; // The locality that the PCR
819                      // can be reset
820    unsigned int extendLocality : 5; // The locality that the PCR
821                      // can be extend
822 } PCR_Attributes;
823
typedef struct
824  EXTERN PCR s_pcrs[IMPLEMENTATION_PCR];
825  

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 butter 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 UINT64 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  EXTERN UINT32    s_failCode;  // the error was signaled  
850  EXTERN UINT32    s_failCallback;
851  EXTERN FailFunction  *LibFailCallback;
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.

```c
extern const TPMA_CC s_ccAttr[];
extern const COMMAND_ATTRIBUTES s_commandAttributes[];
```

```c
#endif // GLOBAL_H
```
5.10 GpMacros.h

5.10.1 Introduction

This file is a collection of miscellaneous macros.

```c
#define GP_MACROS_H
#define NULL
#include "swap.h"
#include "VendorString.h"
```

5.10.2 For Self-test

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

```c
#if SELF_TEST
# define TEST(alg) if (TEST_BIT(alg, g_toTest)) CryptTestAlgorithm(alg, NULL)
#endif // SELF_TEST
```

5.10.3 For Failures

```c
#define FUNCTION_NAME 0
#define FUNCTION_NAME __FUNCTION__
#endif // FAIL_TRACE
#define FAIL(errorCode) TpmFail(FUNCTION_NAME, __LINE__, errorCode)
#define LOG_FAILURE(errorCode) TpmLogFailure(FUNCTION_NAME, __LINE__, errorCode)
```

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.

```c
#define NO_LONGJMP
#define FAIL_RETURN(returnCode)
#define TPM_FAIL_RETURN NORETURN
#define FAIL_RETURN(returnCode) return (returnCode)
#define TPM_FAIL_RETURN void
```
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.

```c
#define REQUIRE(condition, errorCode, returnCode) 
{ 
  if(!!(condition)) 
  { 
    FAIL(FATAL_ERROR_errorCode); 
    FAIL_RETURN(returnCode); 
  } 
} 
#define PARAMETER_CHECK(condition, returnCode) 
REQUIRE((condition), PARAMETER, returnCode) 
#define PCR_SELECT_MIN          ((PLATFORM_PCR+7)/8) 
#define PCR_SELECT_MAX          ((IMPLEMENTATION_PCR+7)/8) 
#define MAX_ORDERLY_COUNT       ((1 << ORDERLY_BITS) - 1) 
#define PRIVATE_VENDOR_SPECIFIC_BYTES 
  ((MAX_RSA_KEY_BYTES/2) * (3 + CRT_FORMAT_RSA * 2)) 

if (defined EMPTY_ASSERT) & (EMPTY_ASSERT != NO) 
  # define pAssert(a) ((void)0) 
else 
  # define pAssert(a) { if(!(a)) FAIL(FATAL_ERROR_PARAMETER);} 
#endif
```

### 5.10.4 Derived from Vendor-specific values

Values derived from vendor specific settings in Implementation.h

```c
#define PCR_SELECT_MIN          ((PLATFORM_PCR+7)/8) 
#define PCR_SELECT_MAX          ((IMPLEMENTATION_PCR+7)/8) 
#define MAX_ORDERLY_COUNT       ((1 << ORDERLY_BITS) - 1) 
#define PRIVATE_VENDOR_SPECIFIC_BYTES 
  ((MAX_RSA_KEY_BYTES/2) * (3 + CRT_FORMAT_RSA * 2)) 
```

### 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

```c
#define ERROR_RETURN(returnCode) 
{ 
  retVal = returnCode; 
  goto Exit; 
} 
#define MAX(a, b) ((a) > (b) ? (a) : (b)) 
#define MIN(a, b) ((a) < (b) ? (a) : (b)) 
```

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

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)

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.
#define ECC_SECURITY_STRENGTH 0
#endif // ALG_ECC
#define MAX_ASYM_SECURITY_STRENGTH \ 
    MAX(RSA_SECURITY_STRENGTH, ECC_SECURITY_STRENGTH)
#define MAX_HASH_SECURITY_STRENGTH \ 
    ((CONTEXT_INTEGRITY_HASH_SIZE * 8) / 2)

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

#define PROOF_SIZE CONTEXT_INTEGRITY_HASH_SIZE
As required by P1.14.4
#define COMPLIANT_PROOF_SIZE \ 
    (MAX(CONTEXT_INTEGRITY_HASH_SIZE, (2 * MAX_SYM_KEY_BYTES)))
As required by P1.14.3.1
#define COMPLIANT_PRIMARY_SEED_SIZE \ 
    BITS_TO_BYTES(MAX_SECURITY_STRENGTH_BITS * 2)
This is the pre-errata version

#ifdef PRIMARY_SEED_SIZE
#define PRIMARY_SEED_SIZE PROOF_SIZE
#endif
#if USE_SPEC_COMPLIANT_PROOFS
#undef PROOF_SIZE
#define PROOF_SIZE COMPLIANT_PROOF_SIZE
#undef PRIMARY_SEED_SIZE
#define PRIMARY_SEED_SIZE COMPLIAN
T PRIMARY_SEED_SIZE
#endif // USE_SPEC_COMPLIANT_PROOFS
#if !SKIP_PROOF_ERRORS
#if PROOF_SIZE < COMPLIANT_PROOF_SIZE
    error "PROOF_SIZE is not compliant with TPM specification"
#endif
#if PRIMARY_SEED_SIZE < COMPLIANT_PRIMARY_SEED_SIZE
    error "Implementation.h specifies a non-compliant PRIMARY_SEED_SIZE"
#endif
#endif // !SKIP_PROOF_ERRORS

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

#if defined CONTEXT_ENCRYPT_ALG
    undef CONTEXT_ENCRYPT_ALGORITHM
#if CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
    define CONTEXT_ENCRYPT_ALGORITHM AES
#elif CONTEXT_ENCRYPT_ALG == ALG_SM4_VALUE
    define CONTEXT_ENCRYPT_ALGORITHM SM4
#elif CONTEXT_ENCRYPT_ALG == ALG_CAMELLIA_VALUE
    define CONTEXT_ENCRYPT_ALGORITHM CAMELLIA
#elif CONTEXT_ENCRYPT_ALG == ALG_TDES_VALUE
    define CONTEXT_ENCRYPT_ALGORITHM TDES
#else error Are you kidding?
# else
#   error Unknown value for CONTEXT_ENCRYPT_ALG
# endif // CONTEXT_ENCRYPT_ALG == ALG_AES_VALUE
#endif

#define CONTEXT_ENCRYPT_ALG
   CONCAT3(ALG_, CONTEXT_ENCRYPT_ALGORITHM, _VALUE)
#endif // CONTEXT_ENCRYPT_ALG
#define CONTEXT_ENCRYPT_KEY_BITS
   CONCAT(CONTEXT_ENCRYPT_ALGORITHM, _MAX_KEY_SIZE_BITS)
#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.

#ifndef LABEL_MAX_BUFFER
#define LABEL_MAX_BUFFER MIN(32, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE))
#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.

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

Check for consistency of the bit ordering of bit fields

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

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

#if USE_BIT_FIELD_STRUCTURES // The default, old version, with bit fields
#define IS_ATTRIBUTE(a, type, b)    ((a.b != 0))
#define SET_ATTRIBUTE(a, type, b)   (a.b = SET)
#define CLEAR_ATTRIBUTE(a, type, b) (a.b = CLEAR)
#define GET_ATTRIBUTE(a, type, b)    (a.b)
#else
#define IS_ATTRIBUTE(a, type, b)    ((a & type##_##b) != 0)
#define SET_ATTRIBUTE(a, type, b)   (a |= type##_##b)
#define CLEAR_ATTRIBUTE(a, type, b) (a &= ~type##_##b)
#define GET_ATTRIBUTE(a, type, b)   ((type)((a & type##_##b) >> type##_##b##_SHIFT))
#endif
#define VERIFY(_X) if(!_X) goto Error
#define VERIFY(_X) if(!_X) goto Error
#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"
```c
#include "CryptUtil_fp.h"
#include "CryptHash_fp.h"
#include "CryptSym_fp.h"
#include "CryptDes_fp.h"
#include "CryptPrime_fp.h"
#include "CryptRand_fp.h"
#include "CryptSelfTest_fp.h"
#include "MathOnByteBuffers_fp.h"
#include "CryptSym_fp.h"
#include "AlgorithmTests_fp.h"
#include "CryptRsa_fp.h"
#include "CryptPrimeSieve_fp.h"
#endif
#if ALG_RSA
#include "CryptPrimeSieve_fp.h"
#endif
#if ALG_ECC
#include "CryptRsa_fp.h"
#include "CryptPrimeSieve_fp.h"
#endif
#if CC_MAC || CC_MAC_Start
#   include "CryptSmac_fp.h"
#   if ALG_CMAC
#       include "CryptCmac_fp.h"
#   endif
#endif
#include "SupportLibraryFunctionPrototypes_fp.h"
#include "Platform_fp.h"
#endif
```
5.12 LibSupport.h

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

```c
#ifndef _LIB_SUPPORT_H_
#define _LIB_SUPPORT_H_

#ifndef RADIX_BITS
    #if defined(__x86_64__) || defined(_WIN64) || defined(_M_X64) || defined(_M_ARM64)
        define RADIX_BITS 64
    #else
        define RADIX_BITS 32
    #endif
#endif // RADIX_BITS

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

#define LIB_JOIN(x,y) x##y
#define LIB_CONCAT(x,y) LIB_JOIN(x, y)
#define LIB_QUOTE(_STRING_) #_STRING_
#define LIB_INCLUDE2(_LIB_, _TYPE_) LIB_QUOTE(##_LIB_##/TpmTo##_LIB_##_TYPE_.h)
#define LIB_INCLUDE(_LIB_, _TYPE_) LIB_INCLUDE2(_LIB_, _TYPE_)
#define SYM_LIBRARY LIB_CONCAT(SYM_LIB_, SYM_LIB)
#define HASH_LIBRARY(_LIB_) LIB_CONCAT(HASH_LIB_, HASH_LIB)
#define MATH_LIBRARY(_LIB_) LIB_CONCAT(MATH_LIB_, MATH_LIB)

#include LIB.Include(SYM_LIB, Sym)
#include LIB.Include(HASH_LIB, Hash)
#undef MIN
#undef MAX
#endif // _LIB_SUPPORT_H_
```

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

```c
#ifndef _NV_H_
#define _NV_H_
#endif

#define TPM_NT_ORDINARY
#define TPM_NT_COUNTER
#define TPM_NT_BITS
#define TPM_NT_EXTEND

#define GET_TPM_NT(attributes) GET_ATTRIBUTE(attributes, TPMA_NV, TPM_NT)

#define IsNvOrdinary(attributes) (GET_TPM_NT(attributes) == TPM_NT_ORDINARY)
#define IsNvCounter(attributes) (GET_TPM_NT(attributes) == TPM_NT_COUNTER)
#define IsNvBits(attributes) (GET_TPM_NT(attributes) == TPM_NT_BITS)
#define IsNvExtend(attributes) (GET_TPM_NT(attributes) == TPM_NT_EXTEND)
#define IsNvPinPass(attributes) (GET_TPM_NT(attributes) == TPM_NT_PIN_PASS)
#define IsNvPinFail(attributes) (GET_TPM_NT(attributes) == TPM_NT_PIN_FAIL)
```

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.

```c
define NV_INDEX_COUNTER_SIZE

typedef struct {
  UINT32 size;
  TPM_HANDLE handle;
} NV_ENTRY_HEADER;
```

```c
#define NV_EVICT_OBJECT_SIZE
#define NV_INDEX_COUNTER_SIZE
#define NV_RAM_INDEX_COUNTER_SIZE
```
typedef struct {
    UINT32          size;
    TPM_HANDLE      handle;
    TPMA_NV         attributes;
} 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.

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.

#define NV_RAM_REF_INIT         0

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

#define RAM_ORDERLY_START       (&s_indexOrderlyRam[0])

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

#define NV_ORDERLY_START        (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

#define RAM_ORDERLY_END         (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.

#define NV_ORDERLY_END          (NV_ORDERLY_START + sizeof(s_indexOrderlyRam))

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

#define ORDERLY_RAM_ADDRESS_OK(start, offset) \ ((start >= RAM_ORDERLY_START) && ((start + offset - 1) < RAM_ORDERLY_END))

#define RETURN_IF_NV_IS_NOT_AVAILABLE \ { \ if(g_NvStatus != TPM_RC_SUCCESS) \ return g_NvStatus; \ }

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

#define RETURN_IF_ORDERLY \ { \ if(NvClearOrderly() != TPM_RC_SUCCESS) \ return g_NvStatus; \ }

#define NV_IS_AVAILABLE     (g_NvStatus == TPM_RC_SUCCESS)
#define IS_ORDERLY(value)   (value < SU_DA_USED_VALUE)
#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__
```
5.14  PRNG_TestVectors.h

```
#define _MSBN_DRBG_TEST_VECTORS_H

#if DRBG_ALGORITHM == TPM_ALG_AES && DRBG_KEY_BITS == 256
    #define DRBG_TEST_INITIATE_ENTROPY
          0x0d, 0x15, 0xaa, 0x80, 0xb1, 0xc6, 0x3a, 0x10, \
          0x90, 0x6c, 0xfe, 0xdb, 0x79, 0x5d, 0xae, 0xb0, \
          0x5b, 0x81, 0x04, 0x1c, 0x5c, 0x5b, 0xfa, 0xcb, \
          0x37, 0x3d, 0x44, 0x40, 0xd9, 0x12, 0x0f, 0x7e, \
          0x3d, 0x6c, 0xf9, 0x09, 0x86, 0xcf, 0x52, 0xd8, \
          0x5d, 0x3e, 0x94, 0x7d, 0x8c, 0x06, 0x1f, 0x91

#define DRBG_TEST_RESEED_ENTROPY
          0x6e, 0xe7, 0x93, 0xa3, 0x39, 0x55, 0xd7, 0x2a, \
          0xd1, 0x2f, 0xd8, 0x0a, 0x8a, 0x3f, 0xcf, 0x95, \
          0x56, 0x1f, 0xac, 0xae, 0x13, 0xa6, 0x50, 0x42, \
          0xb3, 0x40, 0x09, 0x3c, 0x46, 0x4a, 0x7a, 0x22

#define DRBG_TEST_GENERATED_INTERM
          0x28, 0xe0, 0xeb, 0xb8, 0x21, 0x01, 0x66, 0x50, \
          0x8c, 0x8f, 0x65, 0xf2, 0x20, 0x7b, 0xd0, 0xa3

#define DRBG_TEST_GENERATED
          0x94, 0x6f, 0x51, 0x82, 0xd5, 0x45, 0x10, 0xb9, \
          0x46, 0x12, 0x48, 0xf5, 0x71, 0xca, 0x06, 0xc9

#define DRBG_TEST_INITIATE_ENTROPY
          0x8f, 0xc1, 0x1b, 0xdb, 0x5a, 0xab, 0xb7, 0xe0, \
          0x93, 0x6b, 0x14, 0x28, 0xe0, 0x90, 0x73, 0x03, \
          0xcb, 0x45, 0x9f, 0x3b, 0x60, 0x0d, 0xda, 0x87, \
          0x09, 0x55, 0xfb, 0x2d, 0xa8, 0x0a, 0x44, 0xf8

#define DRBG_TEST_RESEED_ENTROPY
          0x0c, 0xd5, 0x3c, 0xd5, 0xec, 0xcd, 0x5a, 0x10, \
          0xd7, 0xea, 0x26, 0x61, 0x11, 0x25, 0x9b, 0x05, \
          0x57, 0xf, 0xc6, 0xdd, 0xd8, 0xe8, 0xd8, 0xbd, \
          0x72, 0x37, 0x8c, 0xf8, 0x2f, 0x1d, 0xba, 0xa2

#define DRBG_TEST_GENERATED_INTERM
          0xdc, 0x3c, 0xf6, 0xbf, 0xb5, 0xd3, 0x41, 0x13, \
          0x5f, 0x2c, 0x68, 0x11, 0xal, 0x07, 0x1c, 0x87

#define DRBG_TEST_GENERATED
          0xb6, 0x18, 0x50, 0xde, 0xcf, 0x97, 0x10, 0x6d, \
          0x44, 0x76, 0x1a, 0x8c, 0x1a, 0x8c, 0x1a, 0xd4
```

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
5.15 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

```
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 # elif ALG_SHA384
36    # define DEFAULT_TEST_DIGEST_SIZE SHA512_DIGEST_SIZE
37    # define DEFAULT_TEST_HASH_BLOCK_SIZE SHA512_BLOCK_SIZE
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.

```c
#ifndef SUPPORT_LIBRARY_FUNCTION_PROTOTYPES_H
#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.

LIB_EXPORT 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.

```void
```MathLibraryCompatibilityCheck(
```void
```);

5.16.4 BnModMult()

Does op1 \* op2 and divide by modulus returning the remainder of the divide.

```LIB_EXPORT BOOL
BnModMult(bigNum result, bigConst op1, bigConst op2, bigConst modulus);

5.16.5 BnMult()

Multiplies two numbers and returns the result

```LIB_EXPORT BOOL
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.

```LIB_EXPORT BOOL
BnDiv(bigNum quotient, bigNum remainder,
```bigConst dividend, bigConst divisor);
5.16.7 BnMod()

#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.

LIB_EXPORT BOOL
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.

LIB_EXPORT BOOL
BnModExp(bigNum result, bigConst number, bigConst exponent, bigConst modulus);

5.16.10 BnModInverse()

Modular multiplicative inverse. This function is only needed when the TPM implements RSA.

LIB_EXPORT BOOL BnModInverse(bigNum result, bigConst number, 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.

LIB_EXPORT BOOL
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.

LIB_EXPORT BOOL
BnEccModMult2(bigPoint R, pointConst S, bigConst d, 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.

LIB_EXPORT BOOL
BnEccAdd(bigPoint R, pointConst S, pointConst Q, bigCurve E);
5.16.14 BnCurveInitialize()

This function is used to initialize the pointers of a \textit{bnCurve\_t} structure. The structure is a set of pointers to \textit{bigNum} values. The curve-dependent values are set by a different function. This function is only needed if the TPM supports ECC.

\begin{verbatim}
31 LIB_EXPORT bigCurve
32 BnCurveInitialize(bigCurve E, TPM_ECC_CURVE curveId);
\end{verbatim}

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.

\begin{verbatim}
33 LIB_EXPORT void
34 BnCurveFree(bigCurve E);
35 #endif
\end{verbatim}
5.17 TPMB.h

This file contains extra TPM2B structures

```c
#ifndef _TPMB_H
#define _TPMB_H

TPM2B Types

typedef struct {
    UINT16          size;
    BYTE            buffer[1];
} TPM2B, *P2B;

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.

#define TPM2B_TYPE(name, bytes) \
    typedef union {
        struct {
            UINT16  size;
            BYTE    buffer[(bytes)];
        } t;
        TPM2B   b;
    } 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

#define TPM2B_INIT(TYPE, name)  \
    TPM2B_##TYPE name = {sizeof(name.t.buffer), {0}}
#define TPM2B_BYTE_VALUE(bytes) TPM2B_TYPE(bytes##_BYTE_VALUE, bytes)
#endif
```
5.18 Tpm.h

Root header file for building any TPM.lib code

```c
#include "TpmBuildSwitches.h"
#include "BaseTypes.h"
#include "TPMB.h"
#include "MinMax.h"
#include "TpmProfile.h"
#include "TpmAlgorithmDefines.h"
#include "LibSupport.h" // Types from the library. These need to come before
// Global.h because some of the structures in
// that file depend on the structures used by the
// cryptographic libraries.
#include "GpMacros.h" // Define additional macros
#include "Global.h" // Define other TPM types
#include "InternalRoutines.h" // Function prototypes
#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.

```
#ifndef _TPM_BUILD SWITCHES_H_
#define _TPM_BUILD SWITCHES_H_
#undef YES
#define YES 1
#undef NO
#define NO 0

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

#ifndef PROFILE
#define PROFILE QUOTE(a) #a
#define PROFILE INCLUDE(a) PROFILE QUOTE(a)
#include PROFILE INCLUDE(PROFILE)
#endif

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

#ifndef DEBUG
#endif

This definition is required for the re-factored code

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

```c
#include ! (defined SIMULATION) || ((SIMULATION != NO) && (SIMULATION != YES))
# undef SIMULATION
#define SIMULATION YES // Default: Either YES or NO
#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.

```c
#include ! (defined LIBRARY_COMPATABILITY_CHECK) || ((LIBRARY_COMPATABILITY_CHECK != NO) && (LIBRARY_COMPATABILITY_CHECK != YES))
# undef LIBRARY_COMPATABILITY_CHECK
#define LIBRARY_COMPATABILITY_CHECK YES // Default: Either YES or NO
#endif
```

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

```c
#include ! (defined USE_DA_USED) || ((USE_DA_USED != NO) && (USE_DA_USED != YES))
# undef USE_DA_USED
#define USE_DA_USED YES // Default: Either YES or NO
#endif
```

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

```c
#include ! (defined TABLE_DRIVEN_DISPATCH) || (TABLE_DRIVEN_DISPATCH != NO) && (TABLE_DRIVEN_DISPATCH != YES))
# undef TABLE_DRIVEN_DISPATCH
#define TABLE_DRIVEN_DISPATCH YES // Default: Either YES or NO
#endif
```

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

```c
#include ! (defined SELF_TEST) || ((SELF_TEST != NO) && (SELF_TEST != YES))
# undef SELF_TEST
#define SELF_TEST YES // Default: Either YES or NO
#endif
```

Enable the generation of RSA primes using a sieve.

```c
#include ! (defined RSA_KEY_SIEVE) || (RSA_KEY_SIEVE != NO) && (RSA_KEY_SIEVE != YES))
# undef RSA_KEY_SIEVE
#define RSA_KEY_SIEVE YES // Default: Either YES or NO
#endif
```

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

```c
#include ! (defined RSA_INSTRUMENT) || (RSA_INSTRUMENT != NO) && (RSA_INSTRUMENT != YES))
# undef RSA_INSTRUMENT
#define RSA_INSTRUMENT NO // Default: Either YES or NO
#endif
```
This switch enables the RNG state save and restore

```c
#if !(defined _DRBG_STATE_SAVE)
    || ((_DRBG_STATE_SAVE != NO) && (_DRBG_STATE_SAVE != YES))
#   undef _DRBG_STATE_SAVE
#   define _DRBG_STATE_SAVE YES // Default: Either YES or NO
#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.

```c
#if !(defined COMPRESSED_LISTS)
    || ((COMPRESSED_LISTS != NO) && (COMPRESSED_LISTS != YES))
#   undef COMPRESSED_LISTS
#   define COMPRESSED_LISTS YES // Default: Either YES or NO
#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.

```c
#if !(defined CLOCK_STOPS) || ((CLOCK_STOPS != NO) && (CLOCK_STOPS != YES))
#   undef CLOCK_STOPS
#   define CLOCK_STOPS NO // Default: Either YES or NO
#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.

```c
#if (defined USE_MARSHALING_DEFINES) && (USE_MARSHALING_DEFINES != NO)
#   undef USE_MARSHALING_DEFINES
#   define USE_MARSHALING_DEFINES YES
#else
#   define USE_MARSHALING_DEFINES YES // Default: Either YES or NO
#endif
```

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

```c
#if SIMULATION && DEBUG
```

Enables use of the key cache. Default is YES

```c
# if !(defined USE_RSA_KEY_CACHE)
    || ((USE_RSA_KEY_CACHE != NO) && (USE_RSA_KEY_CACHE != YES))
#   undef USE_RSA_KEY_CACHE
#   define USE_RSA_KEY_CACHE YES // Default: Either YES or NO
#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

```c
# if USE RSA KEY CACHE
    || ((USE RSA KEY CACHE_FILE != NO) && (USE RSA KEY CACHE_FILE != YES))
#   undef USE_RSA_KEY_CACHE_FILE
#   define USE_RSA_KEY_CACHE_FILE YES // Default: Either YES or NO
#endif
```
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.

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

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.

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.

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)

If an assertion event it not going to produce any trace information (function and line number) then make
FAIL_TRACE == NO
# if !(defined FAIL_TRACE) || ((FAIL_TRACE != NO) & (FAIL_TRACE != YES))
#   undef FAIL_TRACE
#   define FAIL_TRACE YES  // Default: Either YES or NO
# endif
#endif // DEBUG

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

# if !(defined ACCUMULATE_SELF_HEAL_TIMER) || ((ACCUMULATE_SELF_HEAL_TIMER != NO) & (ACCUMULATE_SELF_HEAL_TIMER != YES))
#   undef ACCUMULATE_SELF_HEAL_TIMER
#   define ACCUMULATE_SELF_HEAL_TIMER YES // Default: Either YES or NO
# endif

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

# if !(defined USE_SPEC_COMPLIANT_PROOFS) || ((USE_SPEC_COMPLIANT_PROOFS != NO) & (USE_SPEC_COMPLIANT_PROOFS != YES))
#   undef USE_SPEC_COMPLIANT_PROOFS
#   define USE_SPEC_COMPLIANT_PROOFS YES // Default: Either YES or NO
# 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.

# if !(defined SKIP_PROOF_ERRORS) || ((SKIP_PROOF_ERRORS != NO) & (SKIP_PROOF_ERRORS != YES))
#   undef SKIP_PROOF_ERRORS
#   define SKIP_PROOF_ERRORS NO  // Default: Either YES or NO
# 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

# if !(define USE_BIT_FIELD_STRUCTURES) || ((USE_BIT_FIELD_STRUCTURES != NO) & (USE_BIT_FIELD_STRUCTURES != YES))
#   undef USE_BIT_FIELD_STRUCTURES
#   define USE_BIT_FIELD_STRUCTURES DEBUG // Default: Either YES or NO
# endif

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

# if !(defined LIBRARY_COMPATIBILITY_CHECK) || ((LIBRARY_COMPATIBILITY_CHECK != NO) & (LIBRARY_COMPATIBILITY_CHECK != YES))
#   undef LIBRARY_COMPATIBILITY_CHECK
#   define LIBRARY_COMPATIBILITY_CHECK NO  // Default: Either YES or NO
# 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.

#define ALG_YES YES
#define ALG_NO NO
#define CC_YES YES
#define CC_NO NO
#endif // _TPM_BUILD SWITCHES_H_
5.20 TpmError.h

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

```c
#ifndef _TPM_TYPES_H_
#define _TPM_TYPES_H_

typedef UINT16 TPM_ALG_ID;
#define TYPE_OF_TPM_ALG_ID UINT16
#define ALG_ERROR_VALUE 0x0000
#define TPM_ALG_ERROR (TPM_ALG_ID)(ALG_ERROR_VALUE)
#define ALG_RSA_VALUE 0x0001
#define TPM_ALG_RSA (TPM_ALG_ID)(ALG_RSA_VALUE)
#define ALG_TDES_VALUE 0x0003
#define TPM_ALG_TDES (TPM_ALG_ID)(ALG_TDES_VALUE)
#define ALG_SHA_VALUE 0x0004
#define TPM_ALG_SHA (TPM_ALG_ID)(ALG_SHA_VALUE)
#define ALG_SHA1_VALUE 0x0004
#define TPM_ALG_SHA1 (TPM_ALG_ID)(ALG_SHA1_VALUE)
#define ALG_HMAC_VALUE 0x0005
#define TPM_ALG_HMAC (TPM_ALG_ID)(ALG_HMAC_VALUE)
#define ALG_AES_VALUE 0x0006
#define TPM_ALG_AES (TPM_ALG_ID)(ALG_AES_VALUE)
#define ALG_MGF1_VALUE 0x0007
#define TPM_ALG_MGF1 (TPM_ALG_ID)(ALG_MGF1_VALUE)
#define ALG_KEYEDHASH_VALUE 0x0008
#define TPM_ALG_KEYEDHASH (TPM_ALG_ID)(ALG_KEYEDHASH_VALUE)
#define ALG_XOR_VALUE 0x000A
#define TPM_ALG_XOR (TPM_ALG_ID)(ALG_XOR_VALUE)
#define ALG_SHA256_VALUE 0x000B
#define TPM_ALG_SHA256 (TPM_ALG_ID)(ALG_SHA256_VALUE)
#define ALG_SHA384_VALUE 0x000C
#define TPM_ALG_SHA384 (TPM_ALG_ID)(ALG_SHA384_VALUE)
#define ALG_SHA512_VALUE 0x000D
#define TPM_ALG_SHA512 (TPM_ALG_ID)(ALG_SHA512_VALUE)
#define ALG_NULL_VALUE 0x0010
#define TPM_ALG_NULL (TPM_ALG_ID)(ALG_NULL_VALUE)
#define ALG_SM3_256_VALUE 0x0012
#define TPM_ALG_SM3_256 (TPM_ALG_ID)(ALG_SM3_256_VALUE)
#define ALG_SM2_VALUE 0x001B
#define TPM_ALG_SM2 (TPM_ALG_ID)(ALG_SM2_VALUE)
#define ALG_ECSCHNORR_VALUE 0x001C
#define TPM_ALG_ECSCHNORR (TPM_ALG_ID)(ALG_ECSCHNORR_VALUE)
#define ALG_ECMQV_VALUE 0x001D
#define TPM_ALG_ECMQV (TPM_ALG_ID)(ALG_ECMQV_VALUE)
#define ALG_KDF1_SP800_56A_VALUE 0x0020
#define TPM_ALG_KDF1_SP800_56A (TPM_ALG_ID)(ALG_KDF1_SP800_56A_VALUE)
#define ALG_KDF2_VALUE 0x0021
```

Table 1.2 - Definition of TPM_ALG_ID Constants
Values derived from Table 1:2

Table 1:3 - Definition of 

Table 2:12 - Definition of 


<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>115</td>
<td>#define TPM_CC_PCR_Allocate (TPM_CC) (0x0000016D)</td>
<td></td>
</tr>
<tr>
<td>116</td>
<td>#define TPM_CC_PCR_SetAuthPolicy (TPM_CC) (0x0000016C)</td>
<td></td>
</tr>
<tr>
<td>117</td>
<td>#define TPM_CC_PP_Commands (TPM_CC) (0x0000016B)</td>
<td></td>
</tr>
<tr>
<td>118</td>
<td>#define TPM_CC_SetPrimaryPolicy (TPM_CC) (0x0000016A)</td>
<td></td>
</tr>
<tr>
<td>119</td>
<td>#define TPM_CC_FieldUpgradeStart (TPM_CC) (0x00000169)</td>
<td></td>
</tr>
<tr>
<td>120</td>
<td>#define TPM_CC_ClockRateAdjust (TPM_CC) (0x00000168)</td>
<td></td>
</tr>
<tr>
<td>121</td>
<td>#define TPM_CC_CreatePrimary (TPM_CC) (0x00000167)</td>
<td></td>
</tr>
<tr>
<td>122</td>
<td>#define TPM_CC_NV_GlobalWriteLock (TPM_CC) (0x00000166)</td>
<td></td>
</tr>
<tr>
<td>123</td>
<td>#define TPM_CC_GetCommandAuditDigest (TPM_CC) (0x00000165)</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>#define TPM_CC_NV_Increment (TPM_CC) (0x00000164)</td>
<td></td>
</tr>
<tr>
<td>125</td>
<td>#define TPM_CC_NV_SetBits (TPM_CC) (0x00000163)</td>
<td></td>
</tr>
<tr>
<td>126</td>
<td>#define TPM_CC_NV_Extend (TPM_CC) (0x00000162)</td>
<td></td>
</tr>
<tr>
<td>127</td>
<td>#define TPM_CC_NV_Write (TPM_CC) (0x00000161)</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>#define TPM_CC_NV_WriteLock (TPM_CC) (0x00000160)</td>
<td></td>
</tr>
<tr>
<td>129</td>
<td>#define TPM_CC_DictionaryAttackLockReset (TPM_CC) (0x0000015F)</td>
<td></td>
</tr>
<tr>
<td>130</td>
<td>#define TPM_CC_DictionaryAttackParameters (TPM_CC) (0x0000015E)</td>
<td></td>
</tr>
<tr>
<td>131</td>
<td>#define TPM_CC_NV_ChangeAuth (TPM_CC) (0x0000015D)</td>
<td></td>
</tr>
<tr>
<td>132</td>
<td>#define TPM_CC_PCR_Event (TPM_CC) (0x0000015C)</td>
<td></td>
</tr>
<tr>
<td>133</td>
<td>#define TPM_CC_PCR_Reset (TPM_CC) (0x0000015B)</td>
<td></td>
</tr>
<tr>
<td>134</td>
<td>#define TPM_CC_SecureBinaryComplete (TPM_CC) (0x0000015A)</td>
<td></td>
</tr>
<tr>
<td>135</td>
<td>#define TPM_CC_SetAlgorithmSet (TPM_CC) (0x00000159)</td>
<td></td>
</tr>
<tr>
<td>136</td>
<td>#define TPM_CC_SetCommandCodeAuditStatus (TPM_CC) (0x00000158)</td>
<td></td>
</tr>
<tr>
<td>137</td>
<td>#define TPM_CC_FieldUpgradeData (TPM_CC) (0x00000157)</td>
<td></td>
</tr>
<tr>
<td>138</td>
<td>#define TPM_CC_IncrementalSelfTest (TPM_CC) (0x00000156)</td>
<td></td>
</tr>
<tr>
<td>139</td>
<td>#define TPM_CC_Startup (TPM_CC) (0x00000155)</td>
<td></td>
</tr>
<tr>
<td>140</td>
<td>#define TPM_CC_StirRandom (TPM_CC) (0x00000154)</td>
<td></td>
</tr>
<tr>
<td>141</td>
<td>#define TPM_CC_ActivateCredential (TPM_CC) (0x00000153)</td>
<td></td>
</tr>
<tr>
<td>142</td>
<td>#define TPM_CC_Shutdown (TPM_CC) (0x00000152)</td>
<td></td>
</tr>
<tr>
<td>143</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000151)</td>
<td></td>
</tr>
<tr>
<td>144</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000150)</td>
<td></td>
</tr>
<tr>
<td>145</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000149)</td>
<td></td>
</tr>
<tr>
<td>146</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000148)</td>
<td></td>
</tr>
<tr>
<td>147</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000147)</td>
<td></td>
</tr>
<tr>
<td>148</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000146)</td>
<td></td>
</tr>
<tr>
<td>149</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000145)</td>
<td></td>
</tr>
<tr>
<td>150</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000144)</td>
<td></td>
</tr>
<tr>
<td>151</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000143)</td>
<td></td>
</tr>
<tr>
<td>152</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000142)</td>
<td></td>
</tr>
<tr>
<td>153</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000141)</td>
<td></td>
</tr>
<tr>
<td>154</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000140)</td>
<td></td>
</tr>
<tr>
<td>155</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000139)</td>
<td></td>
</tr>
<tr>
<td>156</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000138)</td>
<td></td>
</tr>
<tr>
<td>157</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000137)</td>
<td></td>
</tr>
<tr>
<td>158</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000136)</td>
<td></td>
</tr>
<tr>
<td>159</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000135)</td>
<td></td>
</tr>
<tr>
<td>160</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000134)</td>
<td></td>
</tr>
<tr>
<td>161</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000133)</td>
<td></td>
</tr>
<tr>
<td>162</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000132)</td>
<td></td>
</tr>
<tr>
<td>163</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000131)</td>
<td></td>
</tr>
<tr>
<td>164</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000130)</td>
<td></td>
</tr>
<tr>
<td>165</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000129)</td>
<td></td>
</tr>
<tr>
<td>166</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000128)</td>
<td></td>
</tr>
<tr>
<td>167</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000127)</td>
<td></td>
</tr>
<tr>
<td>168</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000126)</td>
<td></td>
</tr>
<tr>
<td>169</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000125)</td>
<td></td>
</tr>
<tr>
<td>170</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000124)</td>
<td></td>
</tr>
<tr>
<td>171</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000123)</td>
<td></td>
</tr>
<tr>
<td>172</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000122)</td>
<td></td>
</tr>
<tr>
<td>173</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000121)</td>
<td></td>
</tr>
<tr>
<td>174</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000120)</td>
<td></td>
</tr>
<tr>
<td>175</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000119)</td>
<td></td>
</tr>
<tr>
<td>176</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000118)</td>
<td></td>
</tr>
<tr>
<td>177</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000117)</td>
<td></td>
</tr>
<tr>
<td>178</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000116)</td>
<td></td>
</tr>
<tr>
<td>179</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000115)</td>
<td></td>
</tr>
<tr>
<td>180</td>
<td>#define TPM_CC_PolicyNV (TPM_CC) (0x00000114)</td>
<td></td>
</tr>
</tbody>
</table>
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)
#define SPEC_LEVEL 00
#define TPM_SPEC_LEVEL (TPM_SPEC)(SPEC_LEVEL)
#define SPEC_VERSION 155
#define TPM_SPEC_VERSION (TPM_SPEC)(SPEC_VERSION)
#define SPEC_YEAR 2019
#define TPM_SPEC_YEAR (TPM_SPEC)(SPEC_YEAR)
#define SPEC_DAY_OF_YEAR 107
#define TPM_SPEC_DAY_OF_YEAR (TPM_SPEC)(SPEC_DAY_OF_YEAR)

Table 2:7 - Definition of TPM_GENERATED Constants

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

Table 2:16 - Definition of TPM_RC Constants

typedef UINT32 TPM_RC;
#define TYPE_OF_TPM_RC UINT32
#define TPM_RC_SUCCESS (TPM_RC)(0x000)
#define TPM_RC_BAD_TAG (TPM_RC)(0x01E)
#define RC_VER1 (TPM_RC)(0x100)
#define TPM_RC_INITIALIZE (TPM_RC)(RC_VER1+0x000)
#define TPM_RC_FAILURE (TPM_RC)(RC_VER1+0x001)
#define TPM_RC_SEQUENCE (TPM_RC)(RC_VER1+0x003)
#define TPM_RC_PRIVATE (TPM_RC)(RC_VER1+0x00B)
#define TPM_RC_HMAC (TPM_RC)(RC_VER1+0x019)
#define TPM_RC_DISABLED (TPM_RC)(RC_VER1+0x020)
#define TPM_RC_EXCLUSIVE (TPM_RC)(RC_VER1+0x021)
#define TPM_RC_AUTH_TYPE (TPM_RC)(RC_VER1+0x024)
#define TPM_RC_AUTH_MISSING (TPM_RC)(RC_VER1+0x025)
#define TPM_RC_POLICY (TPM_RC)(RC_VER1+0x026)
#define TPM_RC_PCR (TPM_RC)(RC_VER1+0x027)
#define TPM_RC_PCR_CHANGED (TPM_RC)(RC_VER1+0x028)
#define TPM_RC_UPGRADE (TPM_RC)(RC_VER1+0x02D)
#define TPM_RC_TOO_MANY_CONTEXTS (TPM_RC)(RC_VER1+0x02E)
#define TPM_RC_AUTH_UNAVAILABLE (TPM_RC)(RC_VER1+0x02F)
#define TPM_RC_REBOOT (TPM_RC)(RC_VER1+0x030)
#define TPM_RC_UNBALANCED (TPM_RC)(RC_VER1+0x031)
#define TPM_RC_COMMAND_SIZE (TPM_RC)(RC_VER1+0x042)
#define TPM_RC_COMMAND_CODE (TPM_RC)(RC_VER1+0x043)
#define TPM_RC_AUTHSIZE (TPM_RC)(RC_VER1+0x044)
#define TPM_RC_AUTH_CONTEXT (TPM_RC)(RC_VER1+0x045)
#define TPM_RC_NV_RANGE (TPM_RC)(RC_VER1+0x046)
#define TPM_RC_NV_SIZE (TPM_RC)(RC_VER1+0x047)
#define TPM_RC_NV_LOCKED (TPM_RC)(RC_VER1+0x048)
#define TPM_RC_NV_AUTHORIZATION (TPM_RC)(RC_VER1+0x049)
#define TPM_RC_NV_UNINITIALIZED (TPM_RC)(RC_VER1+0x04A)
#define TPM_RC_NV_SPACE (TPM_RC)(RC_VER1+0x04B)
#define TPM_RC_NV_DEFINED (TPM_RC)(RC_VER1+0x04C)
#define TPM_RC_BAD_CONTEXT (TPM_RC)(RC_VER1+0x050)
#define TPM_RC_CPHASH (TPM_RC)(RC_VER1+0x051)
#define TPM_RC_PARENT (TPM_RC)(RC_VER1+0x052)
#define TPM_RC_NEEDS_TEST (TPM_RC)(RC_VER1+0x053)
#define TPM_RC_NO_RESULT (TPM_RC)(RC_VER1+0x054)
#define TPM_RC_SENSITIVE (TPM_RC)(RC_VER1+0x055)
#define RC_MAX_FM0 (TPM_RC)(RC_VER1+0x07F)
#define RC_FMT1 (TPM_RC)(RC_VER1+0x080)
#define TPM_RC_ASYMMETRIC (TPM_RC)(RC_FMT1+0x001)
#define TPM_RC_ATTRIBS ASYMMETRIC (TPM_RC)(RC_FMT1+0x001)
#define TPM_RC.Attributes (TPM_RC)(RC_FMT1+0x002)
#define TPM_RC_ATTRIBS Attributes (TPM_RC)(RC_FMT1+0x002)
#define TPM_RC_HASH (TPM_RC)(RC_FMT1+0x003)
#define TPM_RCHASH (TPM_RC)(RC_FMT1+0x003)
#define TPM_RC_HASH_HMAC (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+0x0A)
309 \#define TPM_RCS_TYPE (TPM_RC)(RC_FMT1+0x0A)
310 \#define TPM_RC_HANDLE (TPM_RC)(RC_FMT1+0x0B)
311 \#define TPM_RCS_HANDLE (TPM_RC)(RC_FMT1+0x0B)
312 \#define TPM_RC_KDF (TPM_RC)(RC_FMT1+0x0C)
313 \#define TPM_RCS_KDF (TPM_RC)(RC_FMT1+0x0C)
314 \#define TPM_RC_RANGE (TPM_RC)(RC_FMT1+0x0D)
315 \#define TPM_RCS_RANGE (TPM_RC)(RC_FMT1+0x0D)
316 \#define TPM_RC_AUTH_FAIL (TPM_RC)(RC_FMT1+0x0E)
317 \#define TPM_RCS_AUTH_FAIL (TPM_RC)(RC_FMT1+0x0E)
318 \#define TPM_RC_RESERVED_BITS (TPM_RC)(RC_FMT1+0x0F)
319 \#define TPM_RCS_RESERVED_BITS (TPM_RC)(RC_FMT1+0x0F)
320 \#define TPM_RC_PP (TPM_RC)(RC_FMT1+0x10)
321 \#define TPM_RCS_PP (TPM_RC)(RC_FMT1+0x10)
322 \#define TPM_RC_SCHEME (TPM_RC)(RC_FMT1+0x12)
323 \#define TPM_RCS_SCHEME (TPM_RC)(RC_FMT1+0x12)
324 \#define TPM_RC_SIZE (TPM_RC)(RC_FMT1+0x15)
325 \#define TPM_RCS_SIZE (TPM_RC)(RC_FMT1+0x15)
326 \#define TPM_RC_SYMMETRIC (TPM_RC)(RC_FMT1+0x16)
327 \#define TPM_RCS_SYMMETRIC (TPM_RC)(RC_FMT1+0x16)
328 \#define TPM_RC_TAG (TPM_RC)(RC_FMT1+0x17)
329 \#define TPM_RCS_TAG (TPM_RC)(RC_FMT1+0x17)
330 \#define TPM_RC_SELECTOR (TPM_RC)(RC_FMT1+0x18)
331 \#define TPM_RCS_SELECTOR (TPM_RC)(RC_FMT1+0x18)
332 \#define TPM_RC_INSUFFICIENT (TPM_RC)(RC_FMT1+0x1A)
333 \#define TPM_RCS_INSUFFICIENT (TPM_RC)(RC_FMT1+0x1A)
334 \#define TPM_RC_SIGNATURE (TPM_RC)(RC_FMT1+0x1B)
335 \#define TPM_RCS_SIGNATURE (TPM_RC)(RC_FMT1+0x1B)
336 \#define TPM_RC_KEY (TPM_RC)(RC_FMT1+0x1C)
337 \#define TPM_RCS_KEY (TPM_RC)(RC_FMT1+0x1C)
338 \#define TPM_RC_POLICY_FAIL (TPM_RC)(RC_FMT1+0x1D)
339 \#define TPM_RCS_POLICY_FAIL (TPM_RC)(RC_FMT1+0x1D)
340 \#define TPM_RC_INTEGRITY (TPM_RC)(RC_FMT1+0x1F)
341 \#define TPM_RCS_INTEGRITY (TPM_RC)(RC_FMT1+0x1F)
342 \#define TPM_RC_TICKET (TPM_RC)(RC_FMT1+0x20)
343 \#define TPM_RCS_TICKET (TPM_RC)(RC_FMT1+0x20)
344 \#define TPM_RC_RESERVED_BITS (TPM_RC)(RC_FMT1+0x21)
345 \#define TPM_RCS_RESERVED_BITS (TPM_RC)(RC_FMT1+0x21)
346 \#define TPM_RC_BAD_AUTH (TPM_RC)(RC_FMT1+0x22)
347 \#define TPM_RCS_BAD_AUTH (TPM_RC)(RC_FMT1+0x22)
348 \#define TPM_RC_EXPIRED (TPM_RC)(RC_FMT1+0x23)
349 \#define TPM_RCS_EXPIRED (TPM_RC)(RC_FMT1+0x23)
350 \#define TPM_RC_POLICY_CC (TPM_RC)(RC_FMT1+0x24)
351 \#define TPM_RCS_POLICY_CC (TPM_RC)(RC_FMT1+0x24)
352 \#define TPM_RC_BINDING (TPM_RC)(RC_FMT1+0x25)
353 \#define TPM_RCS_BINDING (TPM_RC)(RC_FMT1+0x25)
354 \#define TPM_RC_CURVE (TPM_RC)(RC_FMT1+0x26)
355 \#define TPM_RCS_CURVE (TPM_RC)(RC_FMT1+0x26)
356 \#define TPM_RC_ECC_POINT (TPM_RC)(RC_FMT1+0x27)
357 \#define TPM_RCS_ECC_POINT (TPM_RC)(RC_FMT1+0x27)
358 \#define TPM_RC_OBJECT_HANDLES (TPM_RC)(RC_FMT1+0x00)
359 \#define TPM_RCS_OBJECT_HANDLES (TPM_RC)(RC_FMT1+0x00)
360 \#define TPM_RC_OBJECT_MEMORY (TPM_RC)(RC_FMT1+0x02)
361 \#define TPM_RCS_OBJECT_MEMORY (TPM_RC)(RC_FMT1+0x02)
362 \#define TPM_RC_MEMORY (TPM_RC)(RC_FMT1+0x03)
363 \#define TPM_RCS_MEMORY (TPM_RC)(RC_FMT1+0x03)
364 \#define TPM_RCS_SESSION_HANDLES (TPM_RC)(RC_FMT1+0x05)
365 \#define TPM_RCS_SESSION_HANDLES (TPM_RC)(RC_FMT1+0x05)
#define TPM_RC_LOCALITY (TPM_RC)(RC_WARN+0x007)
#define TPM_RC_YIELDED (TPM_RC)(RC_WARN+0x008)
#define TPM_RC_CANCELED (TPM_RC)(RC_WARN+0x009)
#define TPM_RC_TESTING (TPM_RC)(RC_WARN+0x00A)
#define TPM_RC_REFERENCE_H0 (TPM_RC)(RC_WARN+0x010)
#define TPM_RC_REFERENCE_H1 (TPM_RC)(RC_WARN+0x011)
#define TPM_RC_REFERENCE_H2 (TPM_RC)(RC_WARN+0x012)
#define TPM_RC_REFERENCE_H3 (TPM_RC)(RC_WARN+0x013)
#define TPM_RC_REFERENCE_H4 (TPM_RC)(RC_WARN+0x014)
#define TPM_RC_REFERENCE_H5 (TPM_RC)(RC_WARN+0x015)
#define TPM_RC_REFERENCE_H6 (TPM_RC)(RC_WARN+0x016)
#define TPM_RC_REFERENCE_S0 (TPM_RC)(RC_WARN+0x018)
#define TPM_RC_REFERENCE_S1 (TPM_RC)(RC_WARN+0x019)
#define TPM_RC_REFERENCE_S2 (TPM_RC)(RC_WARN+0x01A)
#define TPM_RC_REFERENCE_S3 (TPM_RC)(RC_WARN+0x01B)
#define TPM_RC_REFERENCE_S4 (TPM_RC)(RC_WARN+0x01C)
#define TPM_RC_REFERENCE_S5 (TPM_RC)(RC_WARN+0x01D)
#define TPM_RC_REFERENCE_S6 (TPM_RC)(RC_WARN+0x01E)
#define TPM_RC_NV_RATE (TPM_RC)(RC_WARN+0x020)
#define TPM_RC_LOCKOUT (TPM_RC)(RC_WARN+0x021)
#define TPM_RC_RETRY (TPM_RC)(RC_WARN+0x022)
#define TPM_RC_NV_UNAVAILABLE (TPM_RC)(RC_WARN+0x023)
#define TPM_RC_NOT_USED (TPM_RC)(RC_WARN+0x7F)
#define TPM_RC_H (TPM_RC) (0x000)
#define TPM_RC_P (TPM_RC) (0x040)
#define TPM_RC_S (TPM_RC) (0x800)
#define TPM_RC_1 (TPM_RC) (0x100)
#define TPM_RC_2 (TPM_RC) (0x200)
#define TPM_RC_3 (TPM_RC) (0x300)
#define TPM_RC_4 (TPM_RC) (0x400)
#define TPM_RC_5 (TPM_RC) (0x500)
#define TPM_RC_6 (TPM_RC) (0x600)
#define TPM_RC_7 (TPM_RC) (0x700)
#define TPM_RC_8 (TPM_RC) (0x800)
#define TPM_RC_9 (TPM_RC) (0x900)
#define TPM_RC_A (TPM_RC) (0xA00)
#define TPM_RC_B (TPM_RC) (0xB00)
#define TPM_RC_C (TPM_RC) (0xC00)
#define TPM_RC_D (TPM_RC) (0xD00)
#define TPM_RC_E (TPM_RC) (0xE00)
#define TPM_RC_F (TPM_RC) (0xF00)
#define TPM_RC_N_MASK (TPM_RC) (0xF00)

Table 2:17 - Definition of TPM_CLOCK_ADJUST Constants

typedef INT8 TPM_CLOCK_ADJUST;
#define TYPE_OF_TPM_CLOCK_ADJUST UINT8
#define TPM_CLOCK_COARSE_SLOWER (TPM_CLOCK_ADJUST)(-3)
#define TPM_CLOCK_MEDIUM_SLOWER (TPM_CLOCK_ADJUST)(-2)
#define TPM_CLOCK_FINE_SLOWER (TPM_CLOCK_ADJUST)(-1)
#define TPM_CLOCK_NO_CHANGE (TPM_CLOCK_ADJUST)(0)
#define TPM_CLOCK_FINE_FASTER (TPM_CLOCK_ADJUST)(1)
#define TPM_CLOCK_MEDIUM_FASTER (TPM_CLOCK_ADJUST)(2)
#define TPM_CLOCK_COARSE_FASTER (TPM_CLOCK_ADJUST)(3)

Table 2:18 - Definition of TPM_EO Constants

typedef UINT16 TPM_EO;
#define TYPE_OF_TPM_EO UINT16
#define TPM_EO_EQ (TPM_EO)(0x0000)
#define TPM_EO_NEQ (TPM_EO)(0x0001)
#define TPM_EO_SIGNED_GT (TPM_EO)(0x0002)
#define TPM_EO_UNSIGNED_GT (TPM_EO)(0x0003)
#define TPM_EO_SIGNED_LT (TPM_EO)(0x0004)
#define TPM_EO_UNSIGNED_LT (TPM_EO)(0x0005)
#define TPM_EO_SIGNED_GE (TPM_EO)(0x0006)
#endif
#define TPM_EO_UNSIGNED_GE (TPM_EO)(0x0007)
#define TPM_EO_SIGNED_LE (TPM_EO)(0x0008)
#define TPM_EO_UNSIGNED_LE (TPM_EO)(0x0009)
#define TPM_EO_BITSET (TPM_EO)(0x000A)
#define TPM_EO_BITCLEAR (TPM_EO)(0x000B)

Table 2:19 - Definition of TPM_ST Constants
typedef UINT16 TPM_ST;
#define TYPE_OF_TPM_ST UINT16
#define TPM_ST_RSP_COMMAND (TPM_ST)(0x00C4)
#define TPM_ST_NULL (TPM_ST)(0x8000)
#define TPM_ST_NO_SESSIONS (TPM_ST)(0x8001)
#define TPM_ST_SESSIONS (TPM_ST)(0x8002)
#define TPM_ST_ATTEST_NV (TPM_ST)(0x8014)
#define TPM_ST_ATTEST_COMMAND_AUDIT (TPM_ST)(0x8015)
#define TPM_ST_ATTEST_SESSION_AUDIT (TPM_ST)(0x8016)
#define TPM_ST_ATTEST_CERTIFY (TPM_ST)(0x8017)
#define TPM_ST_ATTEST_QUOTE (TPM_ST)(0x8018)
#define TPM_ST_ATTEST_TIME (TPM_ST)(0x8019)
#define TPM_ST_ATTEST_CREATION (TPM_ST)(0x801A)
#define TPM_ST_ATTEST_NV_DIGEST (TPM_ST)(0x801C)
#define TPM_ST_CREATION (TPM_ST)(0x8021)
#define TPM_ST_VERIFIED (TPM_ST)(0x8022)
#define TPM_ST_AUTH_SECRET (TPM_ST)(0x8023)
#define TPM_ST_HASHCHECK (TPM_ST)(0x8024)
#define TPM_ST_AUTH_SIGNED (TPM_ST)(0x8025)
#define TPM_ST_FU_MANIFEST (TPM_ST)(0x8029)

Table 2:20 - Definition of TPM_SU Constants
typedef UINT16 TPM_SU;
define TYPE_OF_TPM_SU UINT16
#define TPM_SU_CLEAR (TPM_SU)(0x0000)
#define TPM_SU_STATE (TPM_SU)(0x0001)

Table 2:21 - Definition of TPM_SE Constants
typedef UINT8 TPM_SE;
define TYPE_OF_TPM_SE UINT8
#define TPM_SE_HMAC (TPM_SE)(0x00)
#define TPM_SE_POLICY (TPM_SE)(0x01)
#define TPM_SE_TRIAL (TPM_SE)(0x03)

Table 2:22 - Definition of TPM_CAP Constants
typedef UINT32 TPM_CAP;
define TYPE_OF_TPM_CAP UINT32
#define TPM_CAP_FIRST (TPM_CAP)(0x00000000)
#define TPM_CAP_ALGS (TPM_CAP)(0x00000000)
#define TPM_CAP_HANDLES (TPM_CAP)(0x00000001)
#define TPM_CAP_COMMANDS (TPM_CAP)(0x00000002)
#define TPM_CAP_PP_COMMANDS (TPM_CAP)(0x00000003)
#define TPM_CAP_AUDIT_COMMANDS (TPM_CAP)(0x00000004)
#define TPM_CAP_PCRS (TPM_CAP)(0x00000005)
#define TPM_CAP_TPM_PROPERTIES (TPM_CAP)(0x00000006)
#define TPM_CAP_PCR_PROPERTIES (TPM_CAP)(0x00000007)
#define TPM_CAP_ECC_CURVES (TPM_CAP)(0x00000008)
#define TPM_CAP_AUTH_POLICIES (TPM_CAP)(0x00000009)
#define TPM_CAP_LAST (TPM_CAP)(0x00000009)
#define TPM_CAP_VENDOR_PROPERTY (TPM_CAP)(0x00000100)
typedef UINT32 TPM_PT;
#define TYPE_OF_TPM_PT (TPM_PT) (0x00000000)
#define TPM_PT_NONE (TPM_PT) (0x00000010)
#define PT_GROUP (TPM_PT) (PT_GROUP*1)
#define TPM_PT_FAMILY_INDICATOR (TPM_PT) (PT_FIXED+0)
#define TPM_PT_LEVEL (TPM_PT) (PT_FIXED+1)
#define TPM_PT_REVISION (TPM_PT) (PT_FIXED+2)
#define TPM_PT_DAY_OF_YEAR (TPM_PT) (PT_FIXED+3)
#define TPM_PT_YEAR (TPM_PT) (PT_FIXED+4)
#define TPM_PT_MANUFACTURER (TPM_PT) (PT_FIXED+5)
#define TPM_PT_VENDOR_STRING_1 (TPM_PT) (PT_FIXED+6)
#define TPM_PT_VENDOR_STRING_2 (TPM_PT) (PT_FIXED+7)
#define TPM_PT_VENDOR_STRING_3 (TPM_PT) (PT_FIXED+8)
#define TPM_PT_VENDOR_STRING_4 (TPM_PT) (PT_FIXED+9)
#define TPM_PT_VENDOR_TPM_TYPE (TPM_PT) (PT_FIXED+10)
#define TPM_PT_FIRMWARE_VERSION_1 (TPM_PT) (PT_FIXED+11)
#define TPM_PT_FIRMWARE_VERSION_2 (TPM_PT) (PT_FIXED+12)
#define TPM_PT_INPUT_BUFFER (TPM_PT) (PT_FIXED+13)
#define TPM_PT_HR_TRANSIENT_MIN (TPM_PT) (PT_FIXED+14)
#define TPM_PT_HR_PERSISTENT_MIN (TPM_PT) (PT_FIXED+15)
#define TPM_PT_HR_LOADED_MIN (TPM_PT) (PT_FIXED+16)
#define TPM_PT_HR_LOADED_MAX (TPM_PT) (PT_FIXED+17)
#define TPM_PT_HR_TRANSIENT_MAX (TPM_PT) (PT_FIXED+18)
#define TPM_PT_HR_PERSISTENT_MAX (TPM_PT) (PT_FIXED+19)
#define TPM_PT_NV_COUNTERS_MAX (TPM_PT) (PT_FIXED+20)
#define TPM_PT_NV_INDEX_MAX (TPM_PT) (PT_FIXED+21)
#define TPM_PT_MEMORY (TPM_PT) (PT_FIXED+22)
#define TPM_PT_CLOCK_UPDATE (TPM_PT) (PT_FIXED+23)
#define TPM_PT_CONTEXT_HASH (TPM_PT) (PT_FIXED+24)
#define TPM_PT_CONTEXT_SYM (TPM_PT) (PT_FIXED+25)
#define TPM_PT_CONTEXT_SYM_SIZE (TPM_PT) (PT_FIXED+26)
#define TPM_PT_ORDERLY_COUNT (TPM_PT) (PT_FIXED+27)
#define TPM_PT_MAX_COMMAND_SIZE (TPM_PT) (PT_FIXED+28)
#define TPM_PT_MAX_RESPONSE_SIZE (TPM_PT) (PT_FIXED+29)
#define TPM_PT_MAX_DIGEST (TPM_PT) (PT_FIXED+30)
#define TPM_PT_MAX_OBJECT_CONTEXT (TPM_PT) (PT_FIXED+31)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+32)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+33)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+34)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+35)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+36)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+37)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+38)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+39)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+40)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+41)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+42)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+43)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+44)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+45)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+46)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+47)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+48)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+49)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+50)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+51)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+52)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+53)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+54)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+55)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+56)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+57)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+58)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+59)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+60)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+61)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+62)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+63)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+64)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+65)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+66)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+67)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+68)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+69)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+70)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+71)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+72)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+73)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+74)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+75)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+76)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+77)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+78)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+79)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+80)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+81)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+82)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+83)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+84)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+85)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+86)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+87)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+88)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+89)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+90)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+91)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+92)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+93)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+94)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+95)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+96)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+97)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+98)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+99)
#define TPM_PT_MAX_SESSION_CONTEXT (TPM_PT) (PT_FIXED+100)
#define TPM_PT_ALGORITHM_SET (TPM_PT)(PT_VAR+12)
#define TPM_PT_LOADED_CURVES (TPM_PT)(PT_VAR+13)
#define TPM_PT_LOCKOUT_COUNTER (TPM_PT)(PT_VAR+14)
#define TPM_PT_MAX_AUTH_FAIL (TPM_PT)(PT_VAR+15)
#define TPM_PT_LOCKOUT_INTERVAL (TPM_PT)(PT_VAR+16)
#define TPM_PT_LOCKOUT_RECOVERY (TPM_PT)(PT_VAR+17)
#define TPM_PT_NV_WRITE_RECOVERY (TPM_PT)(PT_VAR+18)
#define TPM_PT_AUDIT_COUNTER_0 (TPM_PT)(PT_VAR+19)
#define TPM_PT_AUDIT_COUNTER_1 (TPM_PT)(PT_VAR+20)

Table 2:24 - Definition of TPM_PT_PCR Constants

typedef UINT32 TPM_PT_PCR;
#define TYPE_OF_TPM_PT_PCR UINT32
#define TPM_PT_PCR_FIRST (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_SAVE (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_EXTEND_L0 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_RESET_L0 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_EXTEND_L1 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_RESET_L1 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_EXTEND_L2 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_RESET_L2 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_EXTEND_L3 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_RESET_L3 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_EXTEND_L4 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_RESET_L4 (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_NO_INCREMENT (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_DRTM_RESET (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_POLICY (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_AUTH (TPM_PT_PCR)(0x00000000)
#define TPM_PT_PCR_LAST (TPM_PT_PCR)(0x00000000)

Table 2:25 - Definition of TPM_PS Constants

typedef UINT32 TPM_PS;
#define TYPE_OF_TPM_PS UINT32
#define TPM_PS_MAIN (TPM_PS)(0x00000000)
#define TPM_PS_PC (TPM_PS)(0x00000001)
#define TPM_PS_PDA (TPM_PS)(0x00000002)
#define TPM_PS_CELL_PHONE (TPM_PS)(0x00000003)
#define TPM_PS_SERVER (TPM_PS)(0x00000004)
#define TPM_PS_SERVER (TPM_PS)(0x00000004)
#define TPM_PS_PERIPHERAL (TPM_PS)(0x00000000)
#define TPM_PS_TSS (TPM_PS)(0x00000000)
#define TPM_PS_STORAGE (TPM_PS)(0x00000000)
#define TPM_PS_AUTHENTICATION (TPM_PS)(0x00000000)
#define TPM_PS_EMBEDDED (TPM_PS)(0x00000000)
#define TPM_PS_HARDCOPY (TPM_PS)(0x00000000)
#define TPM_PS_INFRASTRUCTURE (TPM_PS)(0x00000000)
#define TPM_PS_VIRTUALIZATION (TPM_PS)(0x00000000)
#define TPM_PS_TNC (TPM_PS)(0x00000000)
#define TPM_PS_MULTI_TENANT (TPM_PS)(0x00000000)
#define TPM_PS_TC (TPM_PS)(0x00000000)

Table 2:26 - Definition of Types for Handles

typedef UINT32 TPM_HANDLE;
#define TYPE_OF_TPM_HANDLE UINT32

Table 2:27 - Definition of TPM_HT Constants

typedef UINT32 TPM_HT;
#define TYPE_OF_TPM_HT UINT32
#define TPM_HT_PCR (TPM_HT)(0x00)
#define TPM_HT_NV_INDEX (TPM_HT)(0x01)
#define TPM_HT_HMAC_SESSION (TPM_HT)(0x02)
#define TPM_HT_LOADED_SESSION  (TPM_HT)(0x02)
#define TPM_HT_POLICY_SESSION  (TPM_HT)(0x03)
#define TPM_HT_SAVED_SESSION   (TPM_HT)(0x03)
#define TPM_HT_PERMANENT      (TPM_HT)(0x40)
#define TPM_HT_TRANSIENT      (TPM_HT)(0x80)
#define TPM_HT_PERSISTENT     (TPM_HT)(0x81)
#define TPM_HT_AC             (TPM_HT)(0x90)

Table 2:28 - Definition of TPM_RH Constants

typedef TPM_HANDLE          TPM_RH;
#define TPM_RH_FIRST        (TPM_RH)(0x40000000)
#define TPM_RH_SRK          (TPM_RH)(0x40000000)
#define TPM_RH_OWNER        (TPM_RH)(0x40000001)
#define TPM_RH_REVOKE       (TPM_RH)(0x40000002)
#define TPM_RH_TRANSPORT    (TPM_RH)(0x40000003)
#define TPM_RH_OPERATOR     (TPM_RH)(0x40000004)
#define TPM_RH_ADMIN        (TPM_RH)(0x40000005)
#define TPM_RH_EK           (TPM_RH)(0x40000006)
#define TPM_RH_NULL         (TPM_RH)(0x40000007)
#define TPM_RH_UNASSIGNED   (TPM_RH)(0x40000008)
#define TPM_RS_PW           (TPM_RH)(0x40000009)
#define TPM_RH_LOCKOUT      (TPM_RH)(0x4000000A)
#define TPM_RH_ENDORSEMENT  (TPM_RH)(0x4000000B)
#define TPM_RH_PLATFORM     (TPM_RH)(0x4000000C)
#define TPM_RH_PLATFORM_NV  (TPM_RH)(0x4000000D)
#define TPM_RH_ADMIN_00     (TPM_RH)(0x40000010)
#define TPM_RH_AUTH_00      (TPM_RH)(0x4000010F)
#define TPM_RH_LAST         (TPM_RH)(0x4000010F)

Table 2:29 - Definition of TPM_HC Constants

typedef TPM_HANDLE              TPM_HC;
#define HR_HANDLE_MASK          (TPM_HC)(0x00FFFFFF)
#define HR_RANGE_MASK           (TPM_HC)(0xFF000000)
#define HR_SHIFT                (TPM_HC)(24)
#define HR_PCR                  (TPM_HC)((TPM_HT_PCR<<HR_SHIFT))
#define HR_HMAC_SESSION         (TPM_HC)((TPM_HT_HMAC_SESSION<<HR_SHIFT))
#define HR_POLICY_SESSION       (TPM_HC)((TPM_HT_POLICY_SESSION<<HR_SHIFT))
#define HR_TRANSIENT            (TPM_HC)((TPM_HT_TRANSIENT<<HR_SHIFT))
#define HR_PERSISTENT           (TPM_HC)((TPM_HT_PERSISTENT<<HR_SHIFT))
#define HR_NV_INDEX             (TPM_HC)((TPM_HT_NV_INDEX<<HR_SHIFT))
#define HR_PERMANENT            (TPM_HC)((TPM_HT_PERMANENT<<HR_SHIFT))
#define PCR_FIRST               (TPM_HC)((HR_PCR+0))
#define PCR_LAST                (TPM_HC)((PCR_FIRST+IMPLEMENTATION_PCR-1))
#define HMAC_SESSION_FIRST      (TPM_HC)((HR_HMAC_SESSION+0))
#define HMAC_SESSION_LAST       (TPM_HC)((HMAC_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
#define LOADED_SESSION_FIRST    (TPM_HC)(HMAC_SESSION_FIRST)
#define LOADED_SESSION_LAST     (TPM_HC)(HMAC_SESSION_LAST)
#define POLICY_SESSION_FIRST    (TPM_HC)((HR_POLICY_SESSION+0))
#define POLICY_SESSION_LAST     (TPM_HC)((POLICY_SESSION_FIRST+MAX_ACTIVE_SESSIONS-1))
#define TRANSIENT_FIRST         (TPM_HC)((HR_TRANSIENT+0))
#define ACTIVE_SESSION_FIRST    (TPM_HC)(POLICY_SESSION_FIRST)
#define ACTIVE_SESSION_LAST     (TPM_HC)(POLICY_SESSION_LAST)
#define TRANSIENT_LAST          (TPM_HC)((TRANSIENT_FIRST+MAX_LOADED_OBJECTS-1))
#define PERSISTENT_FIRST        (TPM_HC)((HR_PERSISTENT+0))
#define PERSISTENT_LAST         (TPM_HC)((PERSISTENT_FIRST+0x00FFFFFF))
#define PLATFORM_PERSISTENT     (TPM_HC)((PERSISTENT_FIRST+0x00000000))
#define NV_INDEX_FIRST          (TPM_HC)(HR_NV_INDEX+0)
#define NV_INDEX_LAST           (TPM_HC)(NV_INDEX_FIRST+0x00FFFFFF)
#define PERMANENT_FIRST         (TPM_HC)(TPM_HT_FIRST)
#define PERMANENT_LAST          (TPM_HC)(TPM_RH_FIRST)
#define PERMANENT_AC            (TPM_HC)((TPM_HT_NV_INDEX<<HR_SHIFT)+0xD000000)

#define TPM_HT_HMAC_SESSION (TPM_HT)(0x02)
#define TPM_HT_LOADED_SESSION (TPM_HT)(0x02)
#define TPM_HT_POLICY_SESSION (TPM_HT)(0x03)
#define TPM_HT_SAVED_SESSION  (TPM_HT)(0x03)
#define TPM_HT_PERMANENT     (TPM_HT)(0x40)
#define TPM_HT_TRANSIENT     (TPM_HT)(0x80)
#define TPM_HT_PERSISTENT    (TPM_HT)(0x81)
#define TPM_HT_AC            (TPM_HT)(0x90)
#define NV_AC_FIRST (TPM_HC)((HR_NV_AC+0))

#define NV_AC_LAST (TPM_HC)((HR_NV_AC+0x0000FFFF))

#define AC_FIRST (TPM_HC)((HR_AC+0))

#define AC_LAST (TPM_HC)((HR_AC+0x0000FFFF))

#define TYPE_OF_TPMA_ALGORITHM UINT32

#define TPMA_ALGORITHM_TO_UINT32(a)  (*((UINT32 *)&(a)))

#define UINT32_TO_TPMA_ALGORITHM(a)  (*((TPMA_ALGORITHM *)&(a)))

#define TPMA_ALGORITHM_TO_BYTE_ARRAY(i, a)  
                  {UINT32 x = BYTE_ARRAY_TO_UINT32(a);
                   i = UINT32_TO_TPMA_ALGORITHM(x);  
                }

#if USE_BIT_FIELD_STRUCTURES
typedef struct TPMA_ALGORITHM {

  unsigned asymmetric : 1;
  unsigned symmetric : 1;
  unsigned hash : 1;
  unsigned object : 1;
  unsigned Reserved_bits_at_4 : 4;
  unsigned signing : 1;
  unsigned encrypting : 1;
  unsigned method : 1;
  unsigned Reserved_bits_at_11 : 21;
} TPMA_ALGORITHM;
/* Bits */
#else  
This implements Table 2:30 TPMA_ALGORITHM using bit masking

typedef UINT32                      TPMA_ALGORITHM;
#define TYPE_OF_TPMA_ALGORITHM      UINT32
#define TPMA_ALGORITHM_asymmetric   ((TPMA_ALGORITHM)1 << 0)
#define TPMA_ALGORITHM_symmetric    ((TPMA_ALGORITHM)1 << 1)
#define TPMA_ALGORITHM_hash         ((TPMA_ALGORITHM)1 << 2)
#define TPMA_ALGORITHM_object       ((TPMA_ALGORITHM)1 << 3)
#define TPMA_ALGORITHM_signing      ((TPMA_ALGORITHM)1 << 8)
#define TPMA_ALGORITHM_encrypting   ((TPMA_ALGORITHM)1 << 9)
#define TPMA_ALGORITHM_method       ((TPMA_ALGORITHM)1 << 10)
#endif  
#define TYPE_OF_TPMA_OBJECT UINT32
#define TPMA_OBJECT_TO_UINT32(a)     (*((UINT32 *)&(a)))
#define UINT32_TO_TPMA_OBJECT(a)     (*((TPMA_OBJECT *)&(a)))
#define TPMA_OBJECT_TO_BYTE_ARRAY(i, a)  
                  { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_OBJECT(x);  
                }

This is the initializer for a TPMA_ALGORITHM structure

#define TPMA_ALGORITHM_INITIALIZER(       
  asymmetric, symmetric, hash, object, bits_at_4, 
  signing, encrypting, method, bits_at_11)  
{asymmetric, symmetric, hash, object, bits_at_4, 
  signing, encrypting, method, bits_at_11}
#endif  
else  // USE_BIT_FIELD_STRUCTURES
This is the initializer for a TPMA_ALGORITHM bit array.

#define TPMA_ALGORITHM_INITIALIZER(       
  asymmetric, symmetric, hash, object, bits_at_4, 
  signing, encrypting, method, bits_at_11)  
{(asymmetric << 0) + (symmetric << 1) + (hash << 2) + 
  (object << 3) + (signing << 8) + (encrypting << 9) + 
  (method << 10)}
#endif  // USE_BIT_FIELD_STRUCTURES
#if USE_BIT_FIELD_STRUCTURES
typedef struct TPMA_OBJECT {  // Table 2:31
  unsigned Reserved_bit_at_0        : 1;
  unsigned fixedTPM                : 1;
  unsigned stClear                 : 1;
  unsigned Reserved_bit_at_3       : 1;
  unsigned fixedParent             : 1;
  unsigned sensitiveDataOrigin     : 1;
  unsigned userWithAuth            : 1;
  unsigned adminWithPolicy         : 1;
  unsigned Reserved_bits_at_8      2;
  unsigned noDA                    : 1;
  unsigned encryptedDuplication    : 1;
  unsigned Reserved_bits_at_12     4;
  unsigned restricted              : 1;
  unsigned decrypt                 : 1;
  unsigned sign                    : 1;
  unsigned x509sign                : 1;
  unsigned Reserved_bits_at_20     12;
} TPMA_OBJECT;
/* Bits */
#endif

This is the initializer for a TPMA_OBJECT structure:
#define TPMA_OBJECT_INITIALIZER(
  bit_at_0,             fixedtpm,             stclear,
  bit_at_3,             fixedparent,          sensitivedataorigin,
  userwithauth,         adminwithpolicy,      bits_at_8,
  noda,                 encryptedduplication, bits_at_12,
  restricted,           decrypt,              sign,
  x509sign,             bits_at_20)

This implements Table 2:31 TPMA_OBJECT using bit masking:
typedef UINT32 TPMA_OBJECT;
#define TYPE_OF_TPMA_OBJECT UINT32
#define TPMA_OBJECT_fixedTPM        ((TPMA_OBJECT)1 << 1)
#define TPMA_OBJECT_stClear         ((TPMA_OBJECT)1 << 2)
#define TPMA_OBJECT_fixedParent     ((TPMA_OBJECT)1 << 4)
#define TPMA_OBJECT_sensitiveDataOrigin ((TPMA_OBJECT)1 << 5)
#define TPMA_OBJECT_userWithAuth    ((TPMA_OBJECT)1 << 6)
#define TPMA_OBJECT_adminWithPolicy ((TPMA_OBJECT)1 << 7)
#define TPMA_OBJECT_noDA            ((TPMA_OBJECT)1 << 10)
#define TPMA_OBJECT_encryptedDuplication ((TPMA_OBJECT)1 << 11)
#define TPMA_OBJECT_decrypt         ((TPMA_OBJECT)1 << 17)
#define TPMA_OBJECT_sign            ((TPMA_OBJECT)1 << 18)
#define TPMA_OBJECT_x509sign        ((TPMA_OBJECT)1 << 19)

This is the initializer for a TPMA_OBJECT bit array:
#define TPMA_OBJECT_INITIALIZER(
  bit_at_0,             fixedtpm,             stclear,
  bit_at_3,             fixedparent,          sensitivedataorigin,
  userwithauth,         adminwithpolicy,      bits_at_8,
  noda,                 encryptedduplication, bits_at_12,
  restricted,           decrypt,              sign,
  x509sign,             bits_at_20)
{(fixedtpm << 1) + (stclear << 2) + \
(fixedparent << 4) + (sensitivedataorigin << 5) + \
(userwithauth << 6) + (adminwithpolicy << 7) + \
(noda << 10) + (encryptedduplication << 11) + \
(restricted << 16) + (decrypt << 17) + \
(sign << 18) + (x509sign << 19)}

#endif // USE_BIT_FIELD_STRUCTURES

#define TYPE_OF_TPMA_SESSION    UINT8
#define TPMA_SESSION_TO_UINT8(a) (*((UINT8 *)&(a)))
#define UINT8_TO_TPMA_SESSION(a) (*((TPMA_SESSION *)&(a)))
#define TPMA_SESSION_TO_BYTE_ARRAY(i, a) \n    UINT8_TO_BYTE_ARRAY((TPMA_SESSION_TO_UINT8(i)), (a))
#define BYTE_ARRAY_TO_TPMA_SESSION(i, a) \n    { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_SESSION(x); }

#if USE_BIT_FIELD_STRUCTURES
typedef struct TPMA_SESSION { // Table 2:32
    unsigned continueSession      : 1;
    unsigned auditExclusive       : 1;
    unsigned auditReset           : 1;
    unsigned Reserved_bits_at_3   : 2;
    unsigned decrypt              : 1;
    unsigned encrypt              : 1;
    unsigned audit                : 1;
} TPMA_SESSION;
/* Bits */
#endif // USE_BIT_FIELD_STRUCTURES

This is the initializer for a TPMA_SESSION structure

#define TPMA_SESSION_INITIALIZER( \n    continuesession, auditexclusive, auditreset, bits_at_3, \n    decrypt, encrypt, audit) \n    {(continuesession << 0) + (auditexclusive << 1) + \n    (auditreset << 2) + (decrypt << 5) + \n    (encrypt << 6) + (audit << 7)}

#else // USE_BIT_FIELD_STRUCTURES
typedef UINT8                           TPMA_SESSION;
#define TYPE_OF_TPMA_SESSION            UINT8
#define TPMA_SESSION_continueSession    ((TPMA_SESSION)1 << 0)
#define TPMA_SESSION_auditExclusive     ((TPMA_SESSION)1 << 1)
#define TPMA_SESSION_auditReset         ((TPMA_SESSION)1 << 2)
#define TPMA_SESSION_decrypt            ((TPMA_SESSION)1 << 5)
#define TPMA_SESSION_encrypt            ((TPMA_SESSION)1 << 6)
#define TPMA_SESSION_audit              ((TPMA_SESSION)1 << 7)
This is the initializer for a TPMA_SESSION bit array.

#define TPMA_SESSION_INITIALIZER( \n    continuesession, auditexclusive, auditreset, bits_at_3, \n    decrypt, encrypt, audit) \n    {(continuesession << 0) + (auditexclusive << 1) + \n    (auditreset << 2) + (decrypt << 5) + \n    (encrypt << 6) + (audit << 7)}
#endif // USE_BIT_FIELD_STRUCTURES

#define TYPE_OF_TPMA_LOCALITY   UINT8
#define TPMA_LOCALITY_TO_UINT8(a)    (*((UINT8 *)&(a)))
#define UINT8_TO_TPMA_LOCALITY(a)    (*((TPMA_LOCALITY *)&(a)))
#define TPMA_LOCALITY_TO_BYTE_ARRAY(i, a) \n    UINT8_TO_BYTE_ARRAY((TPMA_LOCALITY_TO_UINT8(i)), (a))
#define BYTE_ARRAY_TO_TPMA_LOCALITY(i, a) \n    { UINT8 x = BYTE_ARRAY_TO_UINT8(a); i = UINT8_TO_TPMA_LOCALITY(x); }

typedef struct TPMA_LOCALITY { // Table 2:33
    unsigned TPM_LOC_ZERO         : 1;

This implements Table 2:32 TPMA_SESSION using bit masking

typedef UINT8                         TPMA_SESSION;
#define TYPE_OF_TPMA_SESSION           UINT8
#define TPMA_SESSION_continueSession   ((TPMA_SESSION)1 << 0)
#define TPMA_SESSION_auditExclusive    ((TPMA_SESSION)1 << 1)
#define TPMA_SESSION_auditReset        ((TPMA_SESSION)1 << 2)
#define TPMA_SESSION_decrypt           ((TPMA_SESSION)1 << 5)
#define TPMA_SESSION_encrypt           ((TPMA_SESSION)1 << 6)
#define TPMA_SESSION_audit             ((TPMA_SESSION)1 << 7)
This is the initializer for a TPMA_SESSION bit array.
This is the initializer for a TPMA_LOCALITY structure

```c
#define TPMA_LOCALITY_INITIALIZER(
    tpm_loc_zero, tpm_loc_one, tpm_loc_two, tpm_loc_three,
    tpm_loc_four, extended)
    {tpm_loc_zero << 0} + (tpm_loc_one << 1) + (tpm_loc_two << 2) +
    (tpm_loc_three << 3) + (tpm_loc_four << 4) + (extended << 5)
```

This implements Table 2:33 TPMA_LOCALITY using bit masking

This is the initializer for a TPMA_PERMANENT structure

```c
#define TPMA_PERMANENT_INITIALIZER(
    ownerAuthSet, endorsementAuthSet, lockoutAuthSet,
    bits_at_3, disableClear, inLockout,
    tpmGeneratedEPS, bits_at_11)
    {ownerAuthSet, endorsementAuthSet, lockoutAuthSet, bits_at_3, disableClear, inLockout, tpmGeneratedEPS, bits_at_11}
```

This is the initializer for a TPMA_PERMANENT structure
866 bits_at_3, disableclear, inlockout, \n867 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, lockauthset,  
879 bits_at_3, disableclear, inlockout,  
880 tpmgeneratedeps, bits_at_11)  
881 {
882 (ownerauthset << 0) + (endorsementauthset << 1) +  
883 (lockauthset << 2) + (disableclear << 8) +  
884 (inlockout << 9) + (tpmgeneratedeps << 10})
885 #endif // USE_BIT_FIELD_STRUCTURES
886 #define TYPE_OF_TPMA_STARTUP_CLEAR UINT32  
887 #define TPMA_STARTUP_CLEAR_TO_UINT32(a) (*((UINT32 *)&(a)))  
888 #define UINT32_TO_TPMA_STARTUP_CLEAR(a) (*((TPMA_STARTUP_CLEAR *)&(a)))  
889 #define TPMA_STARTUP_CLEAR_TO_BYTE_ARRAY(i, a)  
890 {UINT32 x = BYTE_ARRAY_TO_UINT32(a);  
891 i = UINT32_TO_TPMA_STARTUP_CLEAR(x);  
892}
893 #if USE_BIT_FIELD_STRUCTURES
894 typedef struct TPMA_STARTUP_CLEAR {  
895 unsigned phEnable             : 1;  
896 unsigned shEnable             : 1;  
897 unsigned ehEnable             : 1;  
898 unsigned phEnableNV           : 1;  
899 unsignedReserved_bits_at_4   : 27;  
900 unsigned orderly             : 1;  
901 } TPMA_STARTUP_CLEAR;  
902 /* Bits */
903 #if USE_BIT_FIELD_STRUCTURES
904 #define TPMA_STARTUP_CLEAR_INITIALIZER(  
905 phenable, shenable, ehenable, phenablenv, bits_at_4, orderly)  
906 {phenable, shenable, ehenable, phenablenv, bits_at_4, orderly}
907 #else // USE_BIT_FIELD_STRUCTURES
908 typedef UINT32 TPMA_STARTUP_CLEAR;  
909 #define TYPE_OF_TPMA_STARTUP_CLEAR UINT32  
910 #define TPMA_STARTUP_CLEAR_phEnable ((TPMA_STARTUP_CLEAR)1 << 0)  
911 #define TPMA_STARTUP_CLEAR_shEnable ((TPMA_STARTUP_CLEAR)1 << 1)  
912 #define TPMA_STARTUP_CLEAR_ehEnable ((TPMA_STARTUP_CLEAR)1 << 2)  
913 #define TPMA_STARTUP_CLEAR_phEnableNV ((TPMA_STARTUP_CLEAR)1 << 3)  
914 #define TPMA_STARTUP_CLEAR_orderly ((TPMA_STARTUP_CLEAR)1 << 31)  

This implements Table 2:35 TPMA_STARTUP_CLEAR using bit masking

915 #define TPMA_STARTUP_CLEAR_INITIALIZER(  
916 phenable, shenable, ehenable, phenablenv, bits_at_4, orderly)  
917 {phenable, shenable, ehenable, phenablenv, bits_at_4, orderly}
918 #else // USE_BIT_FIELD_STRUCTURES
919 typedef UINT32 TPMA_STARTUP_CLEAR;  
920 #define TYPE_OF_TPMA_STARTUP_CLEAR UINT32  
921 #define TPMA_STARTUP_CLEAR_phEnable ((TPMA_STARTUP_CLEAR)1 << 0)  
922 #define TPMA_STARTUP_CLEAR_shEnable ((TPMA_STARTUP_CLEAR)1 << 1)  
923 #define TPMA_STARTUP_CLEAR_ehEnable ((TPMA_STARTUP_CLEAR)1 << 2)  
924 #define TPMA_STARTUP_CLEAR_phEnableNV ((TPMA_STARTUP_CLEAR)1 << 3)  
925 #define TPMA_STARTUP_CLEAR_orderly ((TPMA_STARTUP_CLEAR)1 << 31)  

This is the initializer for a TPMA_STARTUP_CLEAR bit array.

926 typedef struct TPMA_STARTUP_CLEAR {  
927 unsigned phEnable             : 1;  
928 unsigned shEnable             : 1;  
929 unsigned ehEnable             : 1;  
930 unsigned phEnableNV           : 1;  
931 unsignedReserved_bits_at_4   : 27;  
932 unsigned orderly             : 1;  
933 } TPMA_STARTUP_CLEAR;  
934 /* Bits */
935 #define TPMA_STARTUP_CLEAR_INITIALIZER(  
936 phenable, shenable, ehenable, phenablenv, bits_at_4, orderly)  
937 {phenable, shenable, ehenable, phenablenv, bits_at_4, orderly}
938 #else // USE_BIT_FIELD_STRUCTURES
939 typedef UINT32 TPMA_STARTUP_CLEAR;  
940 #define TYPE_OF_TPMA_STARTUP_CLEAR UINT32  
941 #define TPMA_STARTUP_CLEAR_phEnable ((TPMA_STARTUP_CLEAR)1 << 0)  
942 #define TPMA_STARTUP_CLEAR_shEnable ((TPMA_STARTUP_CLEAR)1 << 1)  
943 #define TPMA_STARTUP_CLEAR_ehEnable ((TPMA_STARTUP_CLEAR)1 << 2)  
944 #define TPMA_STARTUP_CLEAR_phEnableNV ((TPMA_STARTUP_CLEAR)1 << 3)  
945 #define TPMA_STARTUP_CLEAR_orderly ((TPMA_STARTUP_CLEAR)1 << 31)  

This is the initializer for a TPMA_STARTUP_CLEAR bit array.
#define TPMA_STARTUP_CLEAR_INITIALIZER(
  phenable, shenable, ehenable, phenablenv, bits_at_4, orderly) 
{(phenable << 0) + (shenable << 1) + (ehenable << 2) + 
(phenablenv << 3) + (orderly << 31)}
#endif // USE_BIT_FIELD_STRUCTURES
#define TYPE_OF_TPMA_MEMORY UINT32
#define TPMA_MEMORY_TO_UINT32(a)     (*((UINT32 *)&(a)))
#define UINT32_TO_TPMA_MEMORY(a)     (*((TPMA_MEMORY *)&(a)))
#define TPMA_MEMORY_TO_BYTE_ARRAY(i, a) 
  UINT32_TO_BYTE_ARRAY((TPMA_MEMORY_TO_UINT32(i)), (a))
#if USE_BIT_FIELD_STRUCTURES
typedef struct
  // Table 2:36
  unsigned sharedRAM    : 1;
  unsigned sharedNV     : 1;
  unsigned objectCopiedToRam    : 1;
  unsigned Reserved_bits_at_3  : 29;
) TPMA_MEMORY;
/* Bits */
This is the initializer for a TPMA_MEMORY structure
#endif // USE_BIT_FIELD_STRUCTURES
#define TPMA_MEMORY_INITIALIZER(
  sharedram, sharednv, objectcopiedtoram, bits_at_3)
{sharedram, sharednv, objectcopiedtoram, bits_at_3}
#define TYPE_OF_TPMA_CC     UINT32
#define TPMA_CC_TO_UINT32(a)     (*((UINT32 *)&(a)))
#define UINT32_TO_TPMA_CC(a)     (*((TPMA_CC *)&(a)))
#define TPMA_CC_TO_BYTE_ARRAY(i, a) 
  UINT32_TO_BYTE_ARRAY((TPMA_CC_TO_UINT32(i)), (a))
#if USE_BIT_FIELD_STRUCTURES
typedef struct
  // Table 2:37
  unsigned commandIndex     : 16;
  unsigned Reserved_bits_at_16 : 6;
  unsigned nv               : 1;
  unsigned extensive        : 1;
  unsigned flushed          : 1;
  unsigned cHandles         : 3;
  unsigned rHandle          : 1;
  unsigned V                : 1;
  unsigned Reserved_bits_at_30 : 2;
) TPMA_CC;
/* Bits */
This is the initializer for a TPMA_CC structure
#endif // USE_BIT_FIELD_STRUCTURES
#define TPMA_CC_INITIALIZER(
  commandIndex, reserved_bits_at_16, nv, extensive, flushed, cHandles, rHandle, V, reserved_bits_at_30)
{commandIndex, reserved_bits_at_16, nv, extensive, flushed, cHandles, rHandle, V, reserved_bits_at_30}
This implements Table 2:37 TPMA_CC using bit masking.

```c
typedef UINT32                      TPMA_CC;
#define TYPE_OF_TPMA_CC             UINT32
#define TPMA_CC_commandIndex_SHIFT  0
#define TPMA_CC_commandIndex        ((TPMA_CC)0xffff << 0)
#define TPMA_CC_nv                  ((TPMA_CC)1 << 22)
#define TPMA_CC_extensive           ((TPMA_CC)1 << 23)
#define TPMA_CC_flushed             ((TPMA_CC)1 << 24)
#define TPMA_CC_cHandles_SHIFT      25
#define TPMA_CC_cHandles            ((TPMA_CC)0x7 << 25)
#define TPMA_CC_rHandle             ((TPMA_CC)1 << 28)
#define TPMA_CC_V                   ((TPMA_CC)1 << 29)
```

This is the initializer for a TPMA_CC bit array.

```c
#define TPMA_CC_INITIALIZER(
    commandindex, bits_at_16,   nv,           extensive,    flushed,  
    chandles,     rhandle,      v,            bits_at_30)
```

This implements Table 2:38 TPMA_MODES using bit masking.

```c
typedef UINT32                  TPMA_MODES;
#define TYPE_OF_TPMA_MODES UINT32
#define TPMA_MODES_FIPS_140_2   ((TPMA_MODES)1 << 0)
```

This is the initializer for a TPMA_MODES bit array.

```c
#define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {fips_140_2, bits_at_1}
```

This implements Table 2:38 TPMA_MODES using bit masking.

```c
typedef UINT32                  TPMA_MODES;
#define TYPE_OF_TPMA_MODES UINT32
#define TPMA_MODES_FIPS_140_2   ((TPMA_MODES)1 << 0)
```

This is the initializer for a TPMA_MODES bit array.

```c
#define TPMA_MODES_INITIALIZER(fips_140_2, bits_at_1) {{fips_140_2 << 0}}
```
typedef struct TPMA_X509_KEY_USAGE {  
  unsigned digitalSignature : 1;
  unsigned nonrepudiation : 1;
  unsigned keyEncipherment : 1;
  unsigned dataEncipherment : 1;
  unsigned keyAgreement : 1;
  unsigned keyCertSign : 1;
  unsigned crlSign : 1;
  unsigned encipherOnly : 1;
  unsigned decipherOnly : 1;
  unsigned Reserved_bits_at_9 : 23;
} TPMA_X509_KEY_USAGE;

#define TPMA_X509_KEY_USAGE_INITIALIZER(digitalsignature, nonrepudiation, keyencipherment, dataencipherment, keyagreement, keycertsign, crlsign, encipheronly, decipheronly, bits_at_9)

typedef UINT32 TPMA_X509_KEY_USAGE;
#define TYPE_OF_TPMA_X509_KEY_USAGE UINT32
#define TPMA_X509_KEY_USAGE_digitalSignature ((TPMA_X509_KEY_USAGE)1 << 0)
#define TPMA_X509_KEY_USAGE_nonrepudiation ((TPMA_X509_KEY_USAGE)1 << 1)
#define TPMA_X509_KEY_USAGE_keyEncipherment ((TPMA_X509_KEY_USAGE)1 << 2)
#define TPMA_X509_KEY_USAGE_dataEncipherment ((TPMA_X509_KEY_USAGE)1 << 3)
#define TPMA_X509_KEY_USAGE_keyAgreement ((TPMA_X509_KEY_USAGE)1 << 4)
#define TPMA_X509_KEY_USAGE_keyCertSign ((TPMA_X509_KEY_USAGE)1 << 5)
#define TPMA_X509_KEY_USAGE_crlSign ((TPMA_X509_KEY_USAGE)1 << 6)
#define TPMA_X509_KEY_USAGE_encipherOnly ((TPMA_X509_KEY_USAGE)1 << 7)
#define TPMA_X509_KEY_USAGE_decipherOnly ((TPMA_X509_KEY_USAGE)1 << 8)

typedef BYTE TPMI_YES_NO;
#define TPMI_DH_OBJECT TPM_HANDLE
#define TPMI_DH_PERSISTENT TPM_HANDLE
#define TPMI_DH_ENTITY TPM_HANDLE
#define TPMI_SH_AUTH_SESSION TPM_HANDLE

This is the initializer for a TPMA_X509_KEY_USAGE structure.

#define TPMA_X509_KEY_USAGE_INITIALIZER(  
  digitalsignature, nonrepudiation, keyencipherment,  
  dataencipherment, keyagreement, keycertsign,  
  crlsign, encipheronly, decipheronly,  
  bits_at_9)  

This implements Table 2:39 TPMA_X509_KEY_USAGE using bit masking.

define TPMI_YES_NO TPMI_DH_OBJECT;  // Table 2:40 /* Interface */
define TPMI_DH_OBJECT TPM_HANDLE;  // Table 2:41 /* Interface */
define TPMI_DH_PERSISTENT TPM_HANDLE;  // Table 2:42 /* Interface */
define TPMI_DH_ENTITY TPM_HANDLE;  // Table 2:43 /* Interface */
define TPMI_SH_AUTH_SESSION TPM_HANDLE;  // Table 2:45 /* Interface */
define TPMI_DH_PC TPM_HANDLE;  // Table 2:46 /* Interface */

This is the initializer for a TPMA_X509_KEY_USAGE bit array.
typedef TPM_HANDLE TPMI_SH_HMAC; // Table 2:47 /* Interface */
typedef TPM_HANDLE TPMI_SH_POLICY; // Table 2:48 /* Interface */
typedef TPM_HANDLE TPMI_DH_CONTEXT; // Table 2:49 /* Interface */
typedef TPM_HANDLE TPMI_DH_SAVED; // Table 2:50 /* Interface */
typedef TPM_HANDLE TPMI_RH_HIERACHY; // Table 2:51 /* Interface */
typedef TPM_HANDLE TPMI_RH_HIERACHY_AUTH; // Table 2:53 /* Interface */
typedef TPM_HANDLE TPMI_RH_PLATFORM; // Table 2:54 /* Interface */
typedef TPM_HANDLE TPMI_RH_OWNER; // Table 2:55 /* Interface */
typedef TPM_HANDLE TPMI_RH_ENDORSEMENT; // Table 2:56 /* Interface */
typedef TPM_HANDLE TPMI_RH_PROVISION; // Table 2:57 /* Interface */
typedef TPM_HANDLE TPMI_RH_CLEAR; // Table 2:58 /* Interface */
typedef TPM_HANDLE TPMI_RH_NV_AUTH; // Table 2:59 /* Interface */
typedef TPM_HANDLE TPMI_RH_LOCKOUT; // Table 2:60 /* Interface */
typedef TPM_HANDLE TPMI_RH_NV_INDEX; // Table 2:61 /* Interface */
typedef TPM_HANDLE TPMI_RH_AC; // Table 2:62 /* Interface */
typedef TPM_ALG_ID TPMI_ALG_HASH; // Table 2:63 /* Interface */
typedef TPM_ALG_ID TPMI_ALG_ASYM; // Table 2:64 /* Interface */
typedef TPM_ALG_ID TPMI_ALG_SYM; // Table 2:65 /* Interface */
typedef TPM_ALG_ID TPMI_ALG_SYM_OBJECT; // Table 2:66 /* Interface */
typedef TPM_ALG_ID TPMI_ALG_SYM_MODE; // Table 2:67 /* Interface */
typedef TPM_ALG_ID TPMI_ALG_KDF; // Table 2:68 /* Interface */
typedef TPM_ALG_ID TPMI_ALG_SIG_SCHEME; // Table 2:69 /* Interface */
typedef TPM_ALG_ID TPMI_ECC_KEY_EXCHANGE; // Table 2:70 /* Interface */
typedef TPM_ST TPMI_ST_COMMAND_TAG; // Table 2:71 /* Interface */
typedef TPM_ALG_ID TPMI_ALG_MAC_SCHEME; // Table 2:72 /* Interface */
typedef TPM_ALG_ID TPMI_ALG_CIPHER_MODE; // Table 2:73 /* Interface */
typedef BYTE TPMS_EMPTY; // Table 2:74

typedef struct {
    TPM_ALG_ID alg;
    TPMA_ALGORITHM attributes;
} TPMS_ALGORITHM_DESCRIPTION; /* Structure */

typedef union {
    struct {
        UINT16 size;
        BYTE buffer[sizeof(TPMU_HA)];
    }
} TPMU_HA; /* Structure */

typedef struct {
    TPM_ALG_HASH hashAlg;
    TPMU_HA digest;
} TPMT_HA; /* Structure */
typedef union {
  struct {
    UINT16              size;
    BYTE                buffer[sizeof(TPMT_HA)];
  } t;
  TPM2B        b;
} TPM2B_DIGEST;  /* Structure */

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

typedef union {
  struct {
    UINT16              size;
    BYTE                buffer[MAX_DIGEST_BUFFER];
  } t;
  TPM2B        b;
} TPM2B_MAX_DIGEST;  /* Structure */

typedef union {
  struct {
    UINT16              size;
    BYTE                buffer[MAX_NV_BUFFER_SIZE];
  } t;
  TPM2B        b;
} TPM2B_MAX_NV_DIGEST;  /* Structure */

typedef union {
  struct {
    UINT16              size;
    BYTE                buffer[MAX_SYM_BLOCK_SIZE];
  } t;
  TPM2B        b;
} TPM2B_MAX_SYM;  /* Structure */

typedef TPM2B_DIGEST TPM2_NONCE;

typedef TPM2B_DIGEST TPM2_AUTH;

typedef TPM2B_DIGEST TPM2_OPERAND;

Table 2:80 - Definition of Types for TPM2B_NONCE

Table 2:81 - Definition of Types for TPM2B_AUTH

Table 2:82 - Definition of Types for TPM2B_OPERAND

Table 2:83 - Definition of Types for TPM2B_DIGEST

Table 2:84 - Definition of Types for TPM2B_MAX_BUFFER

Table 2:85 - Definition of Types for TPM2B_MAX_NV_BUFFER

Table 2:86 - Definition of Types for TPM2B_MAX_SYM

Table 2:87 - Definition of Types for TPM2B_DIGEST

Table 2:88 - Definition of Types for TPM2B_MAX_NV_BUFFER

Table 2:89 - Definition of Types for TPM2B_MAX_SYM

Table 2:90 - Definition of Types for TPM2B_DIGEST
BYTE t;

typedef struct {
    TPM2B b;
} TPM2B_NAME;

typedef struct {
    TPM2B_NAME;
} TPM2_SELECT {
    sizeofSelect;
}

typedef struct {
    TPMI_ALG_HASH hash;
    sizeofSelect;
}

typedef struct {
    TPM_ST tag;
    TPMI_RH_HIERARCHY hierarchy;
    TPM2B_DIGEST digest;
} TPMT_TK_CREATION;

typedef struct {
    TPM_ST tag;
    TPMI_RH_HIERARCHY hierarchy;
    TPM2B_DIGEST digest;
} TPMT_TK_VERIFIED;

typedef struct {
    TPM_ST tag;
    TPMI_RH_HIERARCHY hierarchy;
    TPM2B_DIGEST digest;
} TPMT_TK_AUTH;

typedef struct {
    TPM_ST tag;
    TPMI_RH_HIERARCHY hierarchy;
    TPM2B_DIGEST digest;
} TPMT_TK_HASHCHECK;

typedef struct {
    TPM_ALG_ID alg;
    TPM_ALGORITHM algProperties;
} TPM_ALG_PROPERTY;

typedef struct {
    TPM_PT property;
    TPM32 value;
} TPMS_TAGGED_PROPERTY;

typedef struct {
    TPM_PT_PCR tag;
    sizeofSelect;
    TPM_HANDLE handle;
} TPMS_TAGGED_PCR_SELECT;

typedef struct {
    TPM_HANDLE handle;
    TPM HA policyHash;
} TPMS_TAGGED_POLICY;

typedef struct {
    TPM_HANDLE handle[
    MAX_CAP_HANDLES];
} TPML_HANDLE;

typedef struct {
    TPM_CC commandCodes[MAX_CAP_CC];
} TPML_CC;

typedef struct {
    TPM_CC;
} TPMS_CC;

typedef struct {
    TPM32 count;
    TPM_CC commandAttributes[MAX_CAP_CC];
} TPML_CCA;

typedef struct {
    TPM32 count;
    TPM32 commandAttributes[MAX_CAP_CC];
} TPML_CCA;

typedef struct {
    TPM_ALG_ID algorithms[MAX_ALG_LIST_SIZE];
} TPML_ALG;

typedef struct {
    TPM_ALG;
} TPML_ALG;

typedef struct {
    TPM_HANDLE handle[MAX_CAP_HANDLES];
} TPML_HANDLE;
typedef struct {
    UINT32 count;
    TPM2B_DIGEST digests[8];
} TPML_DIGEST;  /* Structure */

typedef struct {
    UINT32 count;
    TPM_HA digests[HASH_COUNT];
} TPML_DIGEST_VALUES;  /* Structure */

typedef struct {
    UINT32 count;
    TPMS_PCR_SELECTION pcrSelections[HASH_COUNT];
} TPML_PCR_SELECTION;  /* Structure */

typedef struct {
    UINT32 count;
    TPMS_ALG_PROPERTY algProperties[MAX_CAP_ALGS];
} TPML_ALG_PROPERTY;  /* Structure */

typedef struct {
    UINT32 count;
    TPMS_TAGGED_PROPERTY tpmProperty[MAX_TPM_PROPERTIES];
} TPML_TAGGED_TPM_PROPERTY;  /* Structure */

typedef struct {
    UINT32 count;
    TPM_ECC_CURVE eccCurves[MAX_ECC_CURVES];
} TPML_ECC_CURVE;  /* Structure */

typedef struct {
    UINT32 count;
    TPMS_TAGGED_PCR_SELECT pcrProperty[MAX_PCR_PROPERTIES];
} TPML_TAGGED_PCR_PROPERTY;  /* Structure */

typedef union {
    TPML_ALG_PROPERTY algorithms;
    TPML_HANDLE handles;
    TPML_CCA command;
    TPML_CC ppCommands;
    TPML_CC auditCommands;
    TPML_PCR_SELECTION assignedPCR;
    TPML_TAGGED_TPM_PROPERTY tpmProperties;
    TPML_TAGGED_PCR_PROPERTY pcrProperties;
    #if ALG_ECC
    TPML_ECC_CURVE eccCurves;
    #endif // ALG_ECC
    TPML_TAGGED_POLICY authPolicies;
} TPMU_CAPABILITIES;  /* Structure */

typedef struct {
    TPM_CAP capability;
    TPMS_CAPABILITIES data;
} TPMS_CAPABILITY_DATA;  /* Structure */

typedef struct {
    UINT64 time;
    TPMU_CAPABILITIES data;
} TPMS_TIME_ATTEST_INFO;  /* Structure */

typedef struct {
    TPM_CLOCK_INFO;
    TPMU_CAPABILITIES data;
} TPMS_CLOCK_INFO;  /* Structure */

typedef struct {
    TPM2B_NAME name;
} TPM2B_PUBLIC;  /* Structure */
typedef struct { /* Structure */
    TPM2B_NAME              qualifiedName;
} TPMS_CERTIFY_INFO; /* Structure */

typedef struct { /* Table 2:120 */
    TPML_PCR_SELECTION      pcrSelect;
    TPM2B_DIGEST            pcrDigest;
} TPMS_QUOTE_INFO; /* Structure */

typedef struct { /* Table 2:121 */
    UINT64                  auditCounter;
    TPM_ALG_ID              digestAlg;
    TPM2B_DIGEST            auditDigest;
    TPM2B_DIGEST            commandDigest;
} TPMS_COMMAND_AUDIT_INFO; /* Structure */

typedef struct { /* Table 2:122 */
    TPMI_YES_NO             exclusiveSession;
    TPM2B_DIGEST            sessionDigest;
} TPMS_SESSION_AUDIT_INFO; /* Structure */

typedef struct { /* Table 2:123 */
    TPM2B_NAME              objectName;
    TPM2B_DIGEST            creationHash;
} TPMS_CREATION_INFO; /* Structure */

typedef struct { /* Table 2:124 */
    TPM2B_NAME                  indexName;
    UINT16                      offset;
    TPM2B_MAX_NV_BUFFER         nvContents;
} TPMS_NV_CERTIFY_INFO; /* Structure */

typedef struct { /* Table 2:125 */
    TPM2B_NAME              indexName;
    TPM2B_DIGEST            nvDigest;
} TPMS_NV_DIGEST_CERTIFY_INFO; /* Structure */

typedef TPM_ST              TPMI_ST_ATTEST; /* Interface */

typedef union { /* Structure */
    struct { /* Table 2:126 */
        TPM2B_NAME              qualifiedSigner;
        TPM2B_DATA              extraData;
        TPM2B_DIGEST            clockInfo;
        TPM2B_ATTEST            attestationData;
        TPMI_SH_AUTH_SESSION    sessionHandle;
        TPM2B_NONCE             nonce;
        TPM2B_NONCE             sessionAttributes;
        TPM2B_AUTH              hmac;
    } t;
    TPM2B b;
} TPMU_ATTEST; /* Structure */

typedef union { /* Table 2:127 */
    struct { /* Table 2:128 */
        TPM2B_NAME              indexName;
        TPM2B_DIGEST            nvDigest;
    } t;
    TPM b;
} TPMU_ATTEST; /* Structure */

typedef union { /* Table 2:129 */
    struct { /* Table 2:130 */
        TPM2B_NAME              indexName;
        TPM2B_DIGEST            nvDigest;
    } t;
    TPM b;
} TPMU_ATTEST; /* Structure */

typedef union { /* Table 2:131 */
    struct { /* Table 2:132 */
        TPM2B_NAME              indexName;
        TPM2B_DIGEST            nvDigest;
    } t;
    TPM b;
} TPMU_ATTEST; /* Structure */
typedef TPM_KEY_BITS TPMI_TDES_KEY_BITS; // Table 2:132 /* Interface */
typedef TPM_KEY_BITS TPMI_AES_KEY_BITS; // Table 2:132 /* Interface */
typedef TPM_KEY_BITS TPMI_SM4_KEY_BITS; // Table 2:132 /* Interface */
typedef TPM_KEY_BITS TPMI_CAMELLIA_KEY_BITS; // Table 2:132 /* Interface */

typedef union {
  #if ALG_TDES
    TPMI_TDES_KEY_BITS tdes;
  #endif // ALG_TDES

  #if ALG_AES
    TPMI_AES_KEY_BITS aes;
  #endif // ALG_AES

  #if ALG_SM4
    TPMI_SM4_KEY_BITS sm4;
  #endif // ALG_SM4

  #if ALG_CAMELLIA
    TPMI_CAMELLIA_KEY_BITS camellia;
  #endif // ALG_CAMELLIA

  TPMI_KEY_BITS sym;

  #if ALG_XOR
    TPMI_ALG_HASH xor;
  #endif // ALG_XOR

} TPMU_SYM_KEY_BITS;

typedef union {
  #if ALG_TDES
    TPMI_ALG_SYM_MODE tdes;
  #endif // ALG_TDES

  #if ALG_AES
    TPMI_ALG_SYM_MODE aes;
  #endif // ALG_AES

  #if ALG_SM4
    TPMI_ALG_SYM_MODE sm4;
  #endif // ALG_SM4

  #if ALG_CAMELLIA
    TPMI_ALG_SYM_MODE camellia;
  #endif // ALG_CAMELLIA

  TPMI_ALG_SYM_MODE sym;

} TPMU_SYM_MODE;

typedef struct {
  TPMI_ALG_SYM algorithm;
  TPMU_SYM_KEY_BITS keyBits;
  TPMU_SYM_MODE mode;
} TPMT_SYM_DEF;

typedef struct {
  TPMI_ALG_SYM_OBJECT algorithm;
  TPMT_SYM_DEF_OBJECT keyBits;
  TPMT_SYM_DEF mode;
} TPMT_SYM_DEF_OBJECT;

typedef union {
  struct {
    UINT16 size;
    BYTE buffer[MAX_SYM_KEY_BYTES];
  } t;
  TPM2B b;
} TPM2B_SYM_KEY;

typedef struct {
  TPM2B_SYM_KEY;
} TPM2B_SYM_KEY_PARMS;

typedef struct {
  struct {
    UINT16 size;
    BYTE buffer[MAX_SYM_KEY_BYTES];
  } t;
  TPM2B b;
} TPM2B_LABEL;

typedef struct {
  struct {
    UINT16 size;
    BYTE buffer[MAX_SYM_BUFFER];
  } t;
  TPM2B b;
} TPM2B_LABEL_PARMS;
typedef union {
    struct {
        UINT16 size;
        BYTE buffer[sizeof(TPMS_DERIVE)];
    } t;
    TPM2B b;
} TPM2B_DERIVE;
/* Structure */

typedef union {
    BYTE create[MAX_SYM_DATA];
    TPMS_DERIVE derive;
} TPMU_SENSITIVE_CREATE;
/* Structure */

typedef union {
    struct {
        UINT16 size;
        BYTE buffer[sizeof(TPMU_SENSITIVE_CREATE)];
    } t;
    TPM2B b;
} TPM2B_SENSITIVE_DATA;
/* Structure */

typedef struct {
    TPM2B_AUTH userAuth;
    TPM2B_SENSITIVE_DATA data;
} TPM2B_SENSITIVE_CREATE;
/* Structure */

typedef struct {
    UINT16 size;
    TPM2B_SENSITIVE_CREATE sensitive;
} TPM2B_SENSITIVE_DATA;
/* Structure */

typedef struct {
    TPMI_ALG_HASH hashAlg;
    TPMI_ALG_HASH hmac;
    TPMI_ALG_KDF kdf;
} TPMU_SCHEME_KEYEDHASH;
/* Structure */

typedef struct {
    TPMI_ALG_HASH hmac;
    TPMI_ALG_HASH hashAlg;
    TPMI_ALG_KDF kdf;
} TPMU_SCHEME_KEYEDHASH;
/* Structure */

typedef struct {
    TPMI_ALG_HASH hashAlg;
    TPMI_ALG_KDF kdf;
} TPMU_SCHEME_KEYEDHASH;
/* Structure */

Table 2:150 - Definition of Types for HMAC_SIG_SCHEME

typedef TPMI_ALG_KEYEDHASH_SCHEME TPMI_ALG_KEYEDHASH_SCHEME;

typedef struct {
    TPMI_ALG_HASH hashAlg;
    TPMI_ALG_KDF kdf;
} TPMU_SCHEME_KEYEDHASH;
/* Structure */

typedef struct {
    TPMI_ALG_HASH hmac;
    TPMI_ALG_HASH hashAlg;
    TPMI_ALG_KDF kdf;
} TPMU_SCHEME_KEYEDHASH;
/* Structure */

typedef struct {
    TPMI_ALG_HASH hmac;
    TPMI_ALG_HASH hashAlg;
    TPMI_ALG_KDF kdf;
} TPMU_SCHEME_KEYEDHASH;
/* Structure */

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

typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_RSASSA;

typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_RSAPSS;
/* Structure */

Table 2:155 - Definition of Types for ECC Signature Schemes
typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_ECDSA;
typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_SM2;
typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_ECSCHNORR;
typedef TPMS_SCHEME_HASH TPMS_SIG_SCHEME_ECDAA;
typedef union {
    #if ALG_EC
    TPMS_SIG_SCHEME_ECDAA ecdaa;
    #endif // ALG_EC
    #if ALG_RSASSA
    TPMS_SIG_SCHEME_RSASSA rsassa;
    #endif // ALG_RSASSA
    #if ALG_RSAPSS
    TPMS_SIG_SCHEME_RSAPSS rsapss;
    #endif // ALG_RSAPSS
    #if ALG_ECDSA
    TPMS_SIG_SCHEME_ECDSA ecdsa;
    #endif // ALG_ECDSA
    #if ALG_SM2
    TPMS_SIG_SCHEME_SM2 sm2;
    #endif // ALG_SM2
    #if ALG_ECSCHNORR
    TPMS_SIG_SCHEME_ECSCHNORR ecschnorr;
    #endif // ALG_ECSCHNORR
    #if ALG_HMAC
    TPMS_SCHEME_HMAC hmac;
    #endif // ALG_HMAC
    TPMS_SCHEME_HASH any;
} TPMU_SIG_SCHEME;
typedef struct {
    TPMI_ALG_SIG_SCHEME scheme;
    TPMU_SIG_SCHEME details;
} TPMT_SIG_SCHEME;

typedef TPMS_SCHEME_HASH TPMS_ENC_SCHEME_OAEP;
typedef TPMS_EMPTY TPMS_ENC_SCHEME_RSAES;

typedef TPMS_SCHEME_HASH TPMS_KEY_SCHEME_ECDH;
typedef TPMS_SCHEME_HASH TPMS_KEY_SCHEME_ECMQV;

typedef TPMS_SCHEME_HASH TPMS_SCHEME_MGF1;
typedef TPMS_SCHEME_HASH TPMS_SCHEME_KDF1_SP800_56A;
typedef TPMS_SCHEME_HASH TPMS_SCHEME_KDF2;
typedef TPMS_SCHEME_HASH TPMS_SCHEME_KDF1_SP800_108;
typedef union {
    #if ALG_MGF1
    TPMS_SCHEME_MGF1 mgf1;
    #endif // ALG_MGF1
    #if ALG_KDF1_SP800_56A
    TPMS_SCHEME_KDF1_SP800_56A kdf1_sp800_56a;
    #endif // ALG_KDF1_SP800_56A
    #if ALG_KDF2
    TPMS_SCHEME_KDF2 kdf2;
    #endif // ALG_KDF2
    #if ALG_KDF1_SP800_108
    TPMS_SCHEME_KDF1_SP800_108 kdf1_sp800_108;
    #endif // ALG_KDF1_SP800_108
} TPMU_KDF_SCHEME;
typedef struct {
    /* Structure */
} TPMT_KDF_SCHEME;

Table 2:158 - Definition of Types for Encryption Schemes

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

Table 2:160 - Definition of Types for KDF Schemes
typedef TPM_ALG_ID TPMI_ALG_ASYM_SCHEME;    // Table 2:163 /* Interface */
typedef union {
    /* Structure */
    TPMI_ALG_ASYM_SCHEME scheme;
    TPMU_ASYM_SCHEME details;
} TPMT_ASYM_SCHEME;

typedef TPM_ALG_ID TPMI_ALG_RSA_SCHEME;    // Table 2:166 /* Interface */
typedef union {
    /* Structure */
    TPMI_ALG_RSA_SCHEME scheme;
    TPMU_ASYM_SCHEME details;
} TPMT_RSA_SCHEME;

typedef TPM_ALG_ID TPMI_ALG_RSA_DECRYPT;    // Table 2:168 /* Interface */
typedef union {
    /* Structure */
    TPMI_ALG_RSA_DECRYPT scheme;
    TPMU_ASYM_SCHEME details;
} TPMT_RSA_DECRYPT;

typedef struct {
    /* Structure */
    TPMI_ALG_ASYM_SCHEME scheme;
    TPMU_ASYM_SCHEME details;
} TPMT_KDF_SCHEME;

typedef union {
    /* Structure */
    TPMI_ALG_KDF scheme;
    TPMU_KDF_SCHEME details;
} TPMT_KDF_SCHEME;

typedef union {
    /* Structure */
    TPMI_ALG_RSA_SCHEME scheme;
    TPMU_ASYM_SCHEME details;
} TPMT_RSA_SCHEME;

typedef struct {
    /* Structure */
    TPMI_ALG_KEY_BITS size;
    BYTE buffer[MAX_RSA_KEY_BYTES];
} TPM2B_PUBLIC_KEY_RSA;

typedef union {
    /* Structure */
    TPMI_RSA_KEY_BITS size;
    BYTE buffer[MAX_RSA_KEY_BITS];
} TPM2B_PRIVATE_KEY_RSA;
typedef union {
    struct {
        UINT16 size;
        BYTE buffer[MAX_ECC_KEY_BYTES];
    } t;
    TPM2B b;
} TPM2B_ECC_PARAMETER; /* Structure */

typedef struct {
    TPM2B_ECC_PARAMETER x;
    TPM2B_ECC_PARAMETER y;
} TPM_ECC_POINT; /* Structure */

typedef struct {
    TPM2B_ECC_PARAMETER point;
} TPM2B_ECC_POINT; /* Structure */

typedef TPM_ALG_ID TPMI_ALG_ECC_SCHEME; /* Interface */

typedef TPM_ECC_CURVE TPMI_ECC_CURVE; /* Interface */

typedef struct {
    TPMI_ALG_ECC_SCHEME scheme;
    TPMU_ASYM_SCHEME details;
} TPMT_ECC_SCHEME; /* Structure */

typedef struct {
    UINT16 keySize;
    TPMT_KDF_SCHEME kdf;
    TPM2B_ECC_PARAMETER p;
    TPM2B_ECC_PARAMETER q;
    TPM2B_ECC_PARAMETER g;
    TPM2B_ECC_PARAMETER a;
    TPM2B_ECC_PARAMETER b;
    TPM2B_ECC_PARAMETER n;
    TPM2B_ECC_PARAMETER h;
} TPMS_ALGORITHM_DETAIL_ECC; /* Structure */

typedef struct {
    TPM_ECC_CURVE curveID;
    TPM2B_PUBLIC_KEY_RSA sig;
} TPMS_SIGNATURE_RSA; /* Structure */

typedef TPMS_SIGNATURE_RSA TPMS_SIGNATURE_RSASSA;

typedef TPMS_SIGNATURE_RSA TPMS_SIGNATURE_RSAPSS;

typedef struct {
    TPMI_ALG_HASH hash;
    TPM2B_ECC_PARAMETER signatureR;
    TPM2B_ECC_PARAMETER signatureS;
} TPMS_SIGNATURE_ECC; /* Structure */

typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECDAA;

typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_ECDSA;

typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_SM2;

typedef TPMS_SIGNATURE_ECC TPMS_SIGNATURE_EC Schnorr;

typedef union {
    #if ALG_ECC
    TPM2B_ECC_PARAMETER ecdaa;
    #endif // ALG_ECC
    #if ALG_RSA
    TPM2B_ECC_PARAMETER rsassa;
    #endif // ALG_RSA
    #if ALG_RSA
    TPM2B_ECC_PARAMETER rsapss;
    #endif // ALG_RSA
}
typedef struct {
    TPMI_ALG_SIG_SCHEME sigAlg;
    TPMU_SIGNATURE signature;
} TPMT_SIGNATURE; /* Structure */

typedef union {
    #if ALG_ECC
    BYTE ecc[sizeof(TPMS_ECC_POINT)];
    #endif // ALG_ECC
    #if ALG_RSA
    BYTE rsa[MAX_RSA_KEY_BYTES];
    #endif // ALG_RSA
    #if ALG_SYMCIPHER
    BYTE symmetric[sizeof(TPM2B_DIGEST)];
    #endif // ALG_SYMCIPHER
    #if ALG_KEYEDHASH
    BYTE keyedHash[sizeof(TPM2B_DIGEST)];
    #endif // ALG_KEYEDHASH
} TPMU_ENCRYPTED_SECRET; /* Structure */

typedef union {
    struct {
        UINT16 size;
        BYTE secret[sizeof(TPMU_ENCRYPTED_SECRET)];
    } t;
    TPM2B b;
} TPM2B_ENCRYPTED_SECRET; /* Structure */

typedef TPMI_ALG_PUBLIC; // Table 2:188 /* Interface */

typedef union {
    #if ALG_KEYEDHASH
    TPM2B_DIGEST keyedHash;
    #endif
    #if ALG_SYMCIPHER
    TPM2B_DIGEST sym;
    #endif
    #if ALG_RSA
    TPM2B_PUBLIC_KEY_RSA rsa;
    #endif
} TPMU_PUBLIC_ID; /* Structure */

typedef struct {
    TPMI_ALG_PUBLIC; // Table 2:189 /* Interface */
} TPMT_PUBLIC_ID; /* Structure */

typedef struct {
    TPMT_KEYEDHASH_SCHEME scheme;
} TPMT_KEYEDHASH_PARMS; /* Structure */

typedef struct {
    TPMT_SYM_DEF_OBJECT symmetric;
    TPMT_ASYM_SCHEME scheme;
} TPMT_ASYM_PARMS; /* Structure */

typedef struct {
    TPMT_SYM_DEF_OBJECT symmetric;
    TPMT_RSA_SCHEME scheme;
} TPMT_RSA_PARMS; /* Structure */
# Part 4: Supporting Routines

## Trusted Platform Module Library

**Family “2.0”**

**TCG PUBLIC REVIEW**

**Page 181**

**Level 00 Revision 01.55**

Copyright © TCG 2006-2019

April 15, 2019

```c
1756 UINT32 exponent; /* Structure */
1757 ) TPMS_RSA_PARMS;
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 } TPMU_PUBLIC_PARMS; /* Structure */
1783 typedef struct { // Table 2:196
1784 TPMI_ALG_PUBLIC type;
1785 TPMI_ALG_HASH nameAlg;
1786 TPMI_OBJECT objectAttributes;
1787 TPM2B_DIGEST authPolicy;
1788 TPMU_PUBLIC_PARMS parameters;
1789 TPMU_PUBLIC_ID unique;
1790 } TPM_PUBLIC; /* Structure */
1791 typedef struct { // Table 2:197
1792 UINT16 size;
1793 TPMU_PUBLIC publicArea;
1794 } TPM2_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 TPM2S SENSITIVE DATA bits;
1818 #endif // ALG_KEYEDHASH
1819 #if ALG_SYMCIPHER
1820 TPM2B SYM KEY sym;
1821 #endif // ALG_SYMCIPHER
```
1822   TPMI_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   TPMU_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 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPM_NV_INDEX(x); }
1862 #if USE_BIT_FIELD_STRUCTURES
1863 typedef struct TPM_NV_INDEX {       // Table 2:207
1864   unsigned index : 24;
1865   unsigned RH_NV : 8;
1866 ) TPM_NV_INDEX;        /* Bits */
1867 #endif
1868 #else // USE_BIT_FIELD_STRUCTURES
1869 #define TPM_NV_INDEX_INITIALIZE(index, rh_nv) {index, rh_nv}
1870 #else // USE_BIT_FIELD_STRUCTURES
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_INDEX_RH_NV_INDEX_SHIFT 24
1876 #define TPM_NV_INDEX_INDEX_RH_NV ((TPM_NV_INDEX)0xff << 24)
1877 #endif
1878 #endif
1879 #define TPM_NV_INDEX_INDEX(index, rh_nv) {index, rh_nv}
1880 #else // USE_BIT_FIELD_STRUCTURES
1881 typedef UINT32 TPM_NV_INDEX;
1882 #define TYPE_OF_TPM_NV_INDEX UINT32
1883 #define TPM_NV_INDEX_INDEX_SHIFT 0
1884 #define TPM_NV_INDEX_INDEX ((TPM_NV_INDEX)0xffffffff << 0)
1885 #define TPM_NV_INDEX_INDEX_RH_NV_INDEX_SHIFT 24
1886 #define TPM_nv_INDEX_INDEX_RH_NV ((TPM_NV_INDEX)0xff << 24)
1887 #endif
1888 #endif
1889 #define TPM_NV_INDEX_INDEX(index, rh_nv) {index, rh_nv}
```c
#define TPM_NV_INDEX_INITIALIZER(index, rh_nv) ((index << 0) + (rh_nv << 24))
#endif // USE_BIT_FIELD_STRUCTURES

Table 2: Definition of TPM_NT Constants

typedef UINT32              TPM_NT;
#define TYPE_OF_TPM_NT      UINT32
#define TPM_NT_ORDINARY     (TPM_NT)(0x0)
#define TPM_NT_COUNTER      (TPM_NT)(0x1)
#define TPM_NT_BITS         (TPM_NT)(0x2)
#define TPM_NT_EXTEND       (TPM_NT)(0x4)
#define TPM_NT_PIN_FAIL     (TPM_NT)(0x8)
#define TPM_NT_PIN_PASS     (TPM_NT)(0x9)

typedef struct {
    UINT32 pinCount;
    UINT32 pinLimit;
} TPMS_NV_PIN_COUNTER_PARAMETERS;

#define TYPE_OF_TPMA_NV     UINT32
#define TPMA_NV_TO_UINT32(a)     (*((UINT32 *)&(a)))
#define UINT32_TO_TPMA_NV(a)     (*((TPMA_NV *)&(a)))
#define TPMA_NV_TO_BYTE_ARRAY(i, a) UINT32_TO_BYTE_ARRAY((TPMA_NV_TO_UINT32(i)), (a))
#define BYTE_ARRAY_TO_TPMA_NV(i, a) { UINT32 x = BYTE_ARRAY_TO_UINT32(a); i = UINT32_TO_TPMA_NV(x); }

#if USE_BIT_FIELD_STRUCTURES
typedef struct
    {
        unsigned PPWRITE              : 1;
        unsigned OWNERWRITE           : 1;
        unsigned AUTHWRITE            : 1;
        unsigned POLICYWRITE          : 1;
        unsigned TPM_NT               : 4;
        unsigned Reserved_bits_at_8   : 2;
        unsigned POLICY_DELETE        : 1;
        unsigned WRITELOCKED          : 1;
        unsigned WRITEALL             : 1;
        unsigned WRITEDEFINE          : 1;
        unsigned WRITE_STCLEAR        : 1;
        unsigned GLOBALLOCK           : 1;
        unsigned PPREAD               : 1;
        unsigned OWNERREAD            : 1;
        unsigned AUTHREAD             : 1;
        unsigned POLICYREAD           : 1;
        unsigned Reserved_bits_at_20  : 5;
        unsigned NO_DA                : 1;
        unsigned ORDERY               : 1;
        unsigned CLEAR_STCLEAR        : 1;
        unsigned READLOCKED           : 1;
        unsigned WRITTEN              : 1;
        unsigned PLATFORMCREATE       : 1;
        unsigned READ_STCLEAR         : 1;
    } TPMA_NV;
#endif

This is the initializer for a TPMA_NV structure

#define TPMA_NV_INITIALIZER(
    ppwrite, ownerwrite, authwrite, policywrite, 
    tm_nt, bits_at_8, policy_delete, written, 
    writeall, write#define, write_stclear, globallock, 
    pread, ownerread, authread, policyread, 
    bits_at_20, no_da, orderly, clear_stclear, 
    readlocked, written, platformcreate, read_stclear) \ 
#define TPM_NV_INDEX_INITIALIZER(index, rh_nv) ((index << 0) + (rh_nv << 24))```
This implements Table 2:210 TPMA_NV using bit masking.

```c
typedef UINT32 TPMA_NV;
define TYPE_OF_TPMA_NV UINT32
define TPMA_NV_PPWRITE ((TPMA_NV)1 << 0)
define TPMA_NV.OwnerWrite ((TPMA_NV)1 << 1)
define TPMA_NV.AuthWrite ((TPMA_NV)1 << 2)
define TPMA_NV.PolicyWrite ((TPMA_NV)1 << 3)
define TPMA_NV_TPM_NT_SHIFT 4
#define TPMA_NV_TPM_NT ((TPMA_NV)0xf << 4)
define TPMA_NV.PolicyDelete ((TPMA_NV)1 << 10)
define TPMA_NV.WriteLocked ((TPMA_NV)1 << 11)
define TPMA_NV.WriteAll ((TPMA_NV)1 << 12)
define TPMA_NV.WriteDefine ((TPMA_NV)1 << 13)
define TPMA_NV.WriteStClear ((TPMA_NV)1 << 14)
define TPMA_NV.GlobalLock ((TPMA_NV)1 << 15)
define TPMA_NV.OwnerRead ((TPMA_NV)1 << 17)
define TPMA_NV.AuthRead ((TPMA_NV)1 << 18)
define TPMA_NV.PolicyRead ((TPMA_NV)1 << 19)
define TPMA_NV.NoDa ((TPMA_NV)1 << 25)
define TPMA_NV.Orderly ((TPMA_NV)1 << 26)
define TPMA_NV.ClearStClear ((TPMA_NV)1 << 27)
define TPMA_NV.ReadLocked ((TPMA_NV)1 << 28)
define TPMA_NV.Written ((TPMA_NV)1 << 29)
define TPMA_NV.PlatformCreate ((TPMA_NV)1 << 30)
define TPMA_NV.ReadStClear ((TPMA_NV)1 << 31)
```

This is the initializer for a TPMA_NV bit array.

```c
#define TPMA_NV_INITIALIZER( ppwrite, ownerwrite, authwrite, policywrite, tpm_nt, bits_at_8, policy_delete, writelocked, writeall, written, platformcreate, read_stclear) {
    (ppwrite << 0)         + (ownerwrite << 1)      +
    (authwrite << 2)       + (policywrite << 3)   +
    (tpm_nt << 4)          + (policy_delete << 10) +
    (writelocked << 11)    + (writeall << 12)     +
    (writeall << 13)       + (write_stclear << 14) +
    (globalock << 15)      + (platformcreate, read_stclear)
    readlocked, no_da, orderly,)
}
define TPM2_B_NV_PUBLIC; /* Structure */
typedef struct {
    TPM2_B_DIGEST authPolicy;
    UINT16 dataSize;
} TPM2_B_NV_PUBLIC;
```

This implements Table 2:210 TPMA_NV using bit masking.
struct {  
    UINT16 size;  
    BYTE buffer[MAX_CONTEXT_SIZE];  
} t;  
TPM2B b;  
} TPM2B_CONTEXT_SENSITIVE;  
/* Structure */

typedef struct {  
    TPM2B_DIGEST integrity;  
    TPM2B_CONTEXT_SENSITIVE encrypted;  
} TPMS_CONTEXT_DATA;  
/* Structure */

typedef union {  
    UINT16 size;  
    BYTE buffer[sizeof(TPMS_CONTEXT_DATA)];  
} t;  
TPM2B b;  
} TPM2B_CONTEXT_DATA;  
/* Structure */

typedef struct {  
    UINT64 sequence;  
    TPMI_DH_SAVED savedHandle;  
    TPMI_RH_HIERARCHY hierarchy;  
    TPM2B_CONTEXT_DATA contextBlob;  
} TPMS_CONTEXT;  
/* Structure */

typedef struct {  
    TPMI_PCR_SELECTION pcrSelect;  
    TPM2B_DIGEST pcrDigest;  
    TPM_LOCALITY locality;  
    TPM_ALG_ID parentNameAlg;  
    TPM2B_NAME parentName;  
    TPM2B_NAME parentQualifiedName;  
    TPM2B_DATA outsideInfo;  
} TPMS_CREATION_DATA;  
/* Structure */

typedef struct {  
    UINT16 size;  
    TPMS_CREATION_DATA creationData;  
} TPM2B_CREATION_DATA;  
/* Structure */

Table 2:220 - Definition of TPM_AT Constants

typedef UINT32 TPM_AT;  
#define TYPE_OF_TPM_AT UINT32  
#define TPM_AT_ANY (TPM_AT)(0x00000000)  
#define TPM_AT_ERROR (TPM_AT)(0x00000001)  
#define TPM_AT_PV1 (TPM_AT)(0x00000002)  
#define TPM_AT_VEND (TPM_AT)(0x80000000)

Table 2:221 - Definition of TPM_AE Constants

typedef UINT32 TPM_AE;  
#define TYPE_OF_TPM_AE UINT32  
#define TPM_AE_NONE (TPM_AE)(0x00000000)  
#define TPM_AE_PV1 (TPM_AE)(0x00000001)  
#define TPM_AE_VEND (TPM_AE)(0x80000000)
5.22 VendorString.h

```c
#ifndef _VENDOR_STRING_H
#define _VENDOR_STRING_H

#define MANUFACTURER    "MSFT"

#ifndef MANUFACTURER
#error MANUFACTURER is not provided.
Please modify include\VendorString.h to provide a specific manufacturer name.
#endif

#define VENDOR_STRING_1       "xCG 
#define VENDOR_STRING_2       "fTPM"
// #define VENDOR_STRING_3
// #define VENDOR_STRING_4

#ifndef VENDOR_STRING_1
#error VENDOR_STRING_1 is not provided.
Please modify include\VendorString.h to provide a vendor-specific string.
#endif

#define FIRMWARE_V1         (0x20170619)
#define FIRMWARE_V2         (0x00163636)

#ifndef FIRMWARE_V1
#error FIRMWARE_V1 is not provided.
Please modify include\VendorString.h to provide a vendor specific firmware version.
#endif
```

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.

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.

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.

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

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

```c
#ifndef _SWAP_H
#define _SWAP_H

#if LITTLE_ENDIAN_TPM
#define TO_BIG_ENDIAN_UINT16(i) REVERSE_ENDIAN_16(i)
#define FROM_BIG_ENDIAN_UINT16(i) REVERSE_ENDIAN_16(i)
#define TO_BIG_ENDIAN_UINT32(i) REVERSE_ENDIAN_32(i)
#define FROM_BIG_ENDIAN_UINT32(i) REVERSE_ENDIAN_32(i)
#define TO_BIG_ENDIAN_UINT64(i) REVERSE_ENDIAN_64(i)
#define FROM_BIG_ENDIAN_UINT64(i) REVERSE_ENDIAN_64(i)
#else
#define TO_BIG_ENDIAN_UINT16(i) (i)
#define FROM_BIG_ENDIAN_UINT16(i) (i)
#define TO_BIG_ENDIAN_UINT32(i) (i)
#define FROM_BIG_ENDIAN_UINT32(i) (i)
#define TO_BIG_ENDIAN_UINT64(i) (i)
#define FROM_BIG_ENDIAN_UINT64(i) (i)
#endif

#define AUTO_ALIGN == NO

The aggregation macros for machines that do not allow unaligned access or for little-endian machines.
Aggregate bytes into a UINT

#define BYTE_ARRAY_TO_UINT8(b)  (uint8_t)((b)[0])
#define BYTE_ARRAY_TO_UINT16(b) ByteArrayToUint16((BYTE *)(b))
#define BYTE_ARRAY_TO_UINT32(b) ByteArrayToUint32((BYTE *)(b))
#define BYTE_ARRAY_TO_UINT64(b) ByteArrayToUint64((BYTE *)(b))
#define UINT8_TO_BYTE_ARRAY(i, b) {(*(uint8_t *)(b)) = (i);} 
#define UINT16_TO_BYTE_ARRAY(i, b) {(*(uint16_t *)(b)) = (i);} 
#define UINT32_TO_BYTE_ARRAY(i, b) {(*(uint32_t *)(b)) = (i);} 
#define UINT64_TO_BYTE_ARRAY(i, b) {(*(uint64_t *)(b)) = (i);} 
#else
// AUTO_ALIGN
#define BYTE_ARRAY_TO_UINT8(b) *((uint8_t *)(b))
#define BYTE_ARRAY_TO_UINT16(b) REVERSE_ENDIAN_16(*((uint16_t *)(b)))
#define BYTE_ARRAY_TO_UINT32(b) REVERSE_ENDIAN_32(*((uint32_t *)(b)))
#define BYTE_ARRAY_TO_UINT64(b) REVERSE_ENDIAN_64(*((uint64_t *)(b)))
#define UINT8_TO_BYTE_ARRAY(i, b) {(*(uint8_t *)(b)) = (i);} 
#define UINT16_TO_BYTE_ARRAY(i, b) {REVERSE_ENDIAN_16(*((uint16_t *)(b))) = (i);} 
#define UINT32_TO_BYTE_ARRAY(i, b) {REVERSE_ENDIAN_32(*((uint32_t *)(b))) = (i);} 
#define UINT64_TO_BYTE_ARRAY(i, b) {REVERSE_ENDIAN_64(*((uint64_t *)(b))) = (i);} 
#endif

```
#define UINT16_TO_BYTE_ARRAY(i, b)    (*((uint16_t *)(b)) = REVERSE_ENDIAN_16(i));
#define UINT32_TO_BYTE_ARRAY(i, b)    (*((uint32_t *)(b)) = REVERSE_ENDIAN_32(i));
#define UINT64_TO_BYTE_ARRAY(i, b)    (*((uint64_t *)(b)) = REVERSE_ENDIAN_64(i));
#endif // BIG_ENDIAN_TPM
#endif // AUTO_ALIGN == NO
#endif // __SWAP_H
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"
3  // #include "CommandResponseSizes_fp.h"
```

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

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().

LIB_EXPORT void ExecuteCommand(
        uint32_t         requestSize,  // IN: command buffer size
        unsigned char *request,       // IN: command buffer
        uint32_t        *responseSize, // IN/OUT: response buffer size
        unsigned char **response       // IN/OUT: response buffer
)
{
    // Command local variables
    UINT32               commandSize;
    COMMAND              command;

    // Response local variables
    UINT32               maxResponse = *responseSize;
    TPM_RC               result;     // return code for the command

    // This next function call is used in development to size the command and response
    // buffers. The values printed are the sizes of the internal structures and
    // not the sizes of the canonical forms of the command response structures. Also,
    // the sizes do not include the tag, command.code, requestSize, or the authorization
    // fields.
    //CommandResponseSizes();

    // Set flags for NV access state. This should happen before any other
    // operation that may require a NV write. Note, that this needs to be done
// even when in failure mode. Otherwise, g_updateNV would stay SET while in
// Failure mode and the NV would be written on each call.
g_updateNV = UT_NONE;
g_clearOrderly = FALSE;
if(g_inFailureMode)
{
    // Do failure mode processing
    TpmFailureMode(requestSize, request, responseSize, response);
    return;
}
// Query platform to get the NV state. The result state is saved internally
// and will be reported by NvIsAvailable(). The reference code requires that
// accessibility of NV does not change during the execution of a command.
// Specifically, if NV is available when the command execution starts and then
// is not available later when it is necessary to write to NV, then the TPM
// will go into failure mode.
NvCheckState();

// Due to the limitations of the simulation, TPM clock must be explicitly
// synchronized with the system clock whenever a command is received.
// This function call is not necessary in a hardware TPM. However, taking
// a snapshot of the hardware timer at the beginning of the command allows
// the time value to be consistent for the duration of the command execution.
TimeUpdateToCurrent();

// Any command through this function will unceremoniously end the
// TPM_Hash_Data/_TPM_Hash_End sequence.
if(g_DRTMHandle != TPM_RH_UNASSIGNED)
    ObjectTerminateEvent();

// Get command buffer size and command buffer.
command.parameterBuffer = request;
command.parameterSize = requestSize;

// Parse command header: tag, commandSize and command.code.
// First parse the tag. The unmarshaling routine will validate
// that it is either TPM_ST_SESSIONS or TPM_ST_NO_SESSIONS.
result = TPMI_ST_COMMAND_TAG_Unmarshal(&command.tag,
                                        &command.parameterBuffer,
                                        &command.parameterSize);
if(result != TPM_RC_SUCCESS)
    goto Cleanup;

// Unmarshal the commandSize indicator.
result = UINT32_Unmarshal(&commandSize,
                          &command.parameterBuffer,
                          &command.parameterSize);
if(result != TPM_RC_SUCCESS)
    goto Cleanup;

// On a TPM that receives bytes on a port, the number of bytes that were
// received on that port is requestSize it must be identical to commandSize.
// In addition, commandSize must not be larger than MAX_COMMAND_SIZE allowed
// by the implementation. The check against MAX_COMMAND_SIZE may be redundant
// as the input processing (the function that receives the command bytes and
// places them in the input buffer) would likely have the input truncated when
// it reaches MAX_COMMAND_SIZE, and requestSize would not equal commandSize.
if(commandSize != requestSize || commandSize > MAX_COMMAND_SIZE)
{
    result = TPM_RC_COMMAND_SIZE;
    goto Cleanup;
}

// Unmarshal the command code.
result = TPM_CC_Unmarshal(&command.code, &command.parameterBuffer,
                          &command.parameterSize);
if(result != TPM_RC_SUCCESS)
    goto Cleanup;

// Check to see if the command is implemented.
command.index = CommandCodeToCommandIndex(command.code);
if (UNIMPLEMENTED_COMMAND_INDEX == command.index)
{
    result = TPM_RC_COMMAND_CODE;
    goto Cleanup;
}
#else
    // If the field upgrade is implemented, then the only allowed command is TPM_CC_FieldUpgradeData.
#else
    // Excepting FUM, the TPM only accepts TPM2_Startup() after _TPM_Init(). After getting a TPM2_Startup(), TPM2_Startup() is no longer allowed.
#endif
    if (command.code != TPM_CC_FieldUpgradeData)
    {
        result = TPM_RC_UPGRADE;
        goto Cleanup;
    }
    else
    {
        // The actions of ParseSessionBuffer() are described in the introduction.
        // As the sessions are parsed command.parameterBuffer is advanced so, on a successful return, command.parameterBuffer should be pointing at the first byte of the parameters
        result = ParseSessionBuffer(&command);
        if (result != TPM_RC_SUCCESS)
            goto Cleanup;
    }
    // Authorization session handling for the command.
    ClearCpRpHashes(&command);
    if (command.tag == TPM_ST_SESSIONS)
    {
        // Find out session buffer size.
        result = UINT32_Unmarshal((UINT32 *)&command.authSize,
                              &command.parameterBuffer,
                              &command.parameterSize);
        if (result != TPM_RC_SUCCESS)
            goto Cleanup;
        // Perform sanity check on the unmarshaled value. If it is smaller than the smallest possible session or larger than the remaining size of the command, then it is an error. NOTE: This check could pass but the session size could still be wrong. That will be determined after the sessions are unmarshaled.
        if (command.authSize < 9 || command.authSize > command.parameterSize)
        {
            result = TPM_RC_SIZE;
            goto Cleanup;
        }
        command.parameterSize -= command.authSize;
        if (result != TPM_RC_SUCCESS)
            goto Cleanup;
    }
    else
    {
        // Start regular command process.
        NvIndexCacheInit();
        // Parse handle buffer.
        result = bParseHandleBuffer(&command);
        if (result != TPM_RC_SUCCESS)
            goto Cleanup;
        // All handles in the handle area are required to reference TPM-resident entities.
        result = EntityGetLoadStatus(&command);
        if (result != TPM_RC_SUCCESS)
            goto Cleanup;
        // Authorization session handling for the command.
        ClearCpRpHashes(&command);
        if (command.tag == TPM_ST_SESSIONS)
        {
            // The actions of ParseSessionBuffer() are described in the introduction.
            // As the sessions are parsed command.parameterBuffer is advanced so, on a successful return, command.parameterBuffer should be pointing at the first byte of the parameters
            result = ParseSessionBuffer(&command);
            if (result != TPM_RC_SUCCESS)
                goto Cleanup;
        }
    }
    // Parse parameter and command area.
    result = ParseCommandArea(&command);
    if (result != TPM_RC_SUCCESS)
        goto Cleanup;
    // Parse the parameter block.
    result = ParseParameterBlock(&command);
    if (result != TPM_RC_SUCCESS)
        goto Cleanup;
    // Perform the action.
    result = CallCommand(&command);
    if (result != TPM_RC_SUCCESS)
        goto Cleanup;
    // Authorization session handling for the command.
    ClearCpRpHashes(&command);
    if (command.tag == TPM_ST_SESSIONS)
    {
        // The actions of ParseSessionBuffer() are described in the introduction.
        // As the sessions are parsed command.parameterBuffer is advanced so, on a successful return, command.parameterBuffer should be pointing at the first byte of the parameters
        result = ParseSessionBuffer(&command);
        if (result != TPM_RC_SUCCESS)
            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 // leave space for the parameter size field if needed
177 if(command.tag == TPM_ST_SESSIONS)
178     command.responseBuffer += sizeof(UINT32);
179     command.responseBuffer += sizeof(TPM_HANDLE);
180 // CommandDispatcher returns a response handle buffer and a response parameter
181 // buffer if it succeeds. It will also set the parameterSize field in the
182 // buffer if the tag is TPM_RC_SESSIONS.
183 result = CommandDispatcher(&command);
184 if(result != TPM_RC_SUCCESS)
185     goto Cleanup;
186 // CommandDispatcher returns a response handle buffer and a response parameter
187 // buffer if it succeeds. It will also set the parameterSize field in the
188 // buffer if the tag is TPM_RC_SESSIONS.
189 result = CommandDispatcher(&command);
190 if(result != TPM_RC_SUCCESS)
191     goto Cleanup;
192 Cleanup:
193     if(g_clearOrderly == TRUE
194         && NV_IS_ORDERLY)
195         {
196 #if USE_DA_USED
197         gp.orderlyState = g_daUsed ? SU_DA_USED_VALUE : SU_NONE_VALUE;
198 #else
199         gp.orderlyState = SU_NONE_VALUE;
200 #endif
201         NV_SYNC_PERSISTENT(orderlyState);
202     // This implementation loads an "evict" object to a transient object slot in
203     // RAM whenever an "evict" object handle is used in a command so that the
204     // access to any object is the same. These temporary objects need to be
205     // cleared from RAM whether the command succeeds or fails.
206     ObjectCleanupEvict();
207 // The parameters and sessions have been marshaled. Now tack on the header and
208 // set the sizes
209     BuildResponseHeader(&command, *response, result);
210     // Try to commit all the writes to NV if any NV write happened during this
211     // command execution. This check should be made for both succeeded and failed
212     // commands, because a failed one may trigger a NV write in DA logic as well.
213     // This is the only place in the command execution path that may call the NV
214     // commit. If the NV commit fails, the TPM should be put in failure mode.
215     if((g_updateNV != UT_NONE) && !g_inFailureMode)
216         {
217             if(g_updateNV == UT_ORDERLY)
218                 NvUpdateIndexOrderlyData();
219             if(!NvCommit())
220                 FAIL(FATAL_ERROR_INTERNAL);
221             g_updateNV = UT_NONE;
pAssert((UINT32)command.parameterSize <= maxResponse);

// Clear unused bits in response buffer.
MemorySet(*response + *responseSize, 0, maxResponse - *responseSize);

// as a final act, and not before, update the response size.
*responseSize = (UINT32)command.parameterSize;

return;
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

```c
#include "Tpm.h"

// if TABLE_DRIVEN_DISPATCH
typedef TPM_RC(NoFlagFunction)(void *target, BYTE **buffer, INT32 *size);
typedef TPM_RC(FlagFunction)(void *target, BYTE **buffer, INT32 *size, BOOL flag);
typedef INT16(MarshalFunction)(void *source, BYTE **buffer, INT32 *size);
typedef MarshalFunction *UNMARSHAL_t;
typedef COMMAND_t {
    COMMAND_NO_ARGS *noArgs;
    COMMAND_IN_ARG *inArg;
    COMMAND_OUT_ARG *outArg;
    COMMAND_INOUT_ARG *inOutArg;
} COMMAND_t;
```

This structure is used by ParseHandleBuffer() and CommandDispatcher(). The parameters in this structure are unique for each command. The parameters are:
<table>
<thead>
<tr>
<th>command</th>
<th>holds the address of the command processing function that is called by Command Dispatcher.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>inSize</strong></td>
<td>this is the size of the command-dependent input structure. The input structure holds the unmarshaled handles and command parameters. If the command takes no arguments (handles or parameters) then inSize will have a value of 0.</td>
</tr>
<tr>
<td><strong>outSize</strong></td>
<td>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 then the marshaled version. This is because the marshaled version contains the exact same number of significant bytes but with padding removed.</td>
</tr>
<tr>
<td><strong>typesOffsets</strong></td>
<td>this parameter points to the list of data types that are to be marshaled or unmarshaled. The list of types follows the offsets array. The offsets array is variable sized so the typesOffset field is necessary for the handle and command processing to be able to find the types that are being handled. The offsets array may be empty. The types structure is described below.</td>
</tr>
<tr>
<td><strong>offsets</strong></td>
<td>this is an array of offsets of each of the parameters in the command or response. When processing the command parameters (not handles) the list contains the offset of the next parameter. For example, if the first command parameter has a size of 4 and there is a second command parameter, then the offset would be 4, indicating that the second parameter starts at 4. If the second parameter has a size of 8, and there is a third parameter, then the second entry in offsets is 12 (4 for the first parameter and 8 for the second). An offset value of 0 in the list indicates the start of the response parameter list. When CommandDispatcher() hits this value, it will stop unmarshaling the parameters and call command. If a command has no response parameters and only one command parameter, then offsets can be an empty list.</td>
</tr>
</tbody>
</table>

```c
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:
### 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.

```c
TPM_RC ParseHandleBuffer(
    COMMAND     *command
) {
    TPM_RC                   result;
    #if TABLE_DRIVEN_DISPATCH
    COMMAND_DESCRIPTOR_t    *desc;
    BYTE                    *types;
    BYTE                     type;
    BYTE                     dtype;
    #else
    #include "Commands.h"
    #endif

    // Make sure that nothing strange has happened
    pAssert(command->index < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
    // Get the address of the descriptor for this command
    desc = s_CommandDataArray[command->index];
    pAssert(desc != NULL);
    // Get the associated list of unmarshaling data types.
    types = &((BYTE *)desc)[desc->typesOffset];
    // if(s_ccAttr[commandIndex].commandIndex == TEST_COMMAND)
    // No handles yet
    command->handleNum = 0;
```
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         // Call it setting the flag to the appropriate value
89         result = f(&command->handles[command->handleNum]),
90             &command->parameterBuffer,
91             &command->parameterSize, (type & 0x80) != 0);
92     }
93     // Got a handle
94     // We do this first so that the match for the handle offset of the
95     // response code works correctly.
96     command->handleNum += 1;
97     if (result != TPM_RC_SUCCESS)
98         // if the unmarshaling failed, return the response code with the
99         // handle indication set
100        return result + TPM_RC_H + (command->handleNum * TPM_RC_1);
101     }
102 }else
103 }
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.

TPM_RC
CommandDispatcher(
    COMMAND *command
    )
```c
{  
  #if !TABLE_DRIVEN_DISPATCH
  TPM_RC result;
  BYTE **paramBuffer = &command->parameterBuffer;
  INT32 *paramBufferSize = &command->parameterSize;
  BYTE **responseBuffer = &command->responseBuffer;
  INT32 *respParmSize = &command->parameterSize;
  INT32 rSize;
  TPM_HANDLE *handles = &command->handles[0];

  // command->handleNum = 0;  // The command-specific code knows how
  // many handles there are. This is for
  // cataloging the number of response
  // handles
  MemoryIoBufferAllocationReset();  // Initialize so that allocation will
  // work properly
  switch(GetCommandCode(command->index))  
  {
    #include "CommandDispatcher.h"

    default:
      FAIL(FATAL_ERROR_INTERNAL);
      break;
  }
  Exit:
  MemoryIoBufferZero();
  return result;

  #else
  COMMAND_DESCRIPTOR_t *desc;
  BYTE *types;
  BYTE type;
  UINT16 *offsets;
  UINT16 offset = 0;
  UINT32 maxInSize;
  BYTE *commandIn;
  INT32 maxOutSize;
  BYTE *commandOut;
  COMMAND_t cmd;
  UINT32 hasInParameters = 0;
  BOOL hasOutParameters = FALSE;
  UINT32 pNum = 0;
  BYTE dType;  // dispatch type
  TPM_RC result;

  // Get the address of the descriptor for this command
  pAssert(command->index < sizeof(s_CommandDataArray) / sizeof(COMMAND_DESCRIPTOR_t *));
  desc = s_CommandDataArray[command->index];

  // Get the list of parameter types for this command
  pAssert(desc != NULL);
  types = &((BYTE *)desc)[desc->typesOffset];

  // Get a pointer to the list of parameter offsets
  offsets = &desc->offsets[0];
  // pointer to handles
  handles = command->handles;

  // Get the size required to hold all the unmarshaled parameters for this command
  maxInSize = desc->inSize;
  // and the size of the output parameter structure returned by this command
  maxOutSize = desc->outSize;

  MemoryIoBufferAllocationReset();

  // Get a buffer for the input parameters
```
commandIn = MemoryGetInBuffer(maxInSize);
// And the output parameters
commandOut = (BYTE *)MemoryGetOutBuffer((UINT32)maxOutSize);
// Get the address of the action code dispatch
cmd = desc->command;

// Copy any handles into the input buffer
for(type = *types++; (type & 0x7F) < PARAMETER_FIRST_TYPE; type = *types++)
{
    // 'offset' was initialized to zero so the first unmarshaling will always
    // be to the start of the data structure
    *(TPM_HANDLE *)&(commandIn[offset]) = *handles++;
    // This check is used so that we don't have to add an additional offset
    // value to the offsets list to correspond to the stop value in the
    // command parameter list.
    if(*types != 0xFF)
        offset = *offsets++;
    // maxInSize -= sizeof(TPM_HANDLE);
    hasInParameters++;
}

// Exit loop with type containing the last value read from types
// maxInSize has the amount of space remaining in the command action input
// buffer. Make sure that we don't have more data to unmarshal than is going to
// fit.

// type contains the last value read from types so it is not necessary to
// reload it, which is good because *types now points to the next value
for(; (dType = (type & 0x7F)) <= PARAMETER_LAST_TYPE; type = *types++)
{
    pNum++;
    if(dType < PARAMETER_FIRST_FLAG_TYPE)
    {
        NoFlagFunction       *f = (NoFlagFunction *)UnmarshalArray[dType];
        result = f(&commandIn[offset], &command->parameterBuffer,
        &command->parameterSize);
    }
    else
    {
        FlagFunction        *f = UnmarshalArray[dType];
        result = f(&commandIn[offset], &command->parameterBuffer,
        &command->parameterSize,
        (type & 0x80) != 0);
    }
    if(result != TPM_RC_SUCCESS)
    {
        result += TPM_RC_P + (TPM_RC_1 * pNum);
        goto Exit;
    }

    // This check is used so that we don't have to add an additional offset
    // value to the offsets list to correspond to the stop value in the
    // command parameter list.
    if(*types != 0xFF)
        offset = *offsets++;
    hasInParameters++;
}

// Should have used all the bytes in the input
if(command->parameterSize != 0)
{
    result = TPM_RC_SIZE;
    goto Exit;
}

// The command parameter unmarshaling stopped when it hit a value that was out
// of range for unmarshaling values and left *types pointing to the first
// marshaling type. If that type happens to be the STOP value, then there
// are no response parameters. So, set the flag to indicate if there are
// output parameters.
hasOutParameters = *types != 0xFF;

// There are four cases for calling, with and without input parameters and with
// and without output parameters.
if(hasInParameters > 0)
{
    if(hasOutParameters)
        result = cmd.inOutArg(commandIn, commandOut);
    else
        result = cmd.inArg(commandIn);
}
else
{
    if(hasOutParameters)
        result = cmd.outArg(commandOut);
    else
        result = cmd.noArgs();
}
if(result != TPM_RC_SUCCESS)
goto Exit;

// Offset in the marshaled output structure
offset = 0;

// Process the return handles, if any
command->handleNum = 0;

// Could make this a loop to process output handles but there is only ever
// one handle in the outputs (for now).
type = *types++;
if((dType = (type & 0x7F)) < RESPONSE_PARAMETER_FIRST_TYPE)
{
    // The out-handle value was referenced as TPM_HANDLE in the
    // action code so it has to be properly aligned.
    command->handles[command->handleNum++] =
        *((TPM_HANDLE *)&(commandOut[offset]));
    maxOutSize= sizeof(UINT32);
    type = *types++;
    offset = *offsets++;
}
// Use the size of the command action output buffer as the maximum for the
// number of bytes that can get marshaled. Since the marshaling code has
// no pointers to data, all of the data being returned has to be in the
// command action output buffer. If we try to marshal more bytes than
// could fit into the output buffer, we need to fail.
for(;(dType = (type & 0x7F)) <= RESPONSE_PARAMETER_LAST_TYPE
    && !_g_inFailureMode; type = *types++)
{
    const MARSHAL_t     f = MarshalArray[dType];
    command->parameterSize += f(&commandOut[offset], &command->responseBuffer,
        &maxOutSize);
    offset = *offsets++;
}
result = (maxOutSize < 0) ? TPM_RC_FAILURE : TPM_RC_SUCCESS;
Exit:
    MemoryIoBufferZero();
    return result;
#endif}
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

```c
#define SESSION_PROCESS_C
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>handle is exempted from DA logic</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>handle is not exempted from DA logic</td>
</tr>
</tbody>
</table>

```c
BOOL IsDAExempted(TPM_HANDLE handle) // IN: entity handle
{
    BOOL result = FALSE;
    //
    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            // All permanent handles, other than TPM_RH_LOCKOUT, are exempt from
            // DA protection.
            result = (handle != TPM_RH_LOCKOUT);
            break;
        // When this function is called, a persistent object will have been loaded
        // into an object slot and assigned a transient handle.
        case TPM_HT_TRANSIENT:
        {
            TPMA_OBJECT attributes = ObjectGetPublicAttributes(handle);
            result = IS_ATTRIBUTE(attributes, TPMA_OBJECT, noDA);
            break;
        }
        case TPM_HT_NV_INDEX:
        {
            NV_INDEX *nvIndex = NvGetIndexInfo(handle, NULL);
            result = IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, NO_DA);
            break;
        }
    }
    return result;
}
```
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_AUTH_FAIL</td>
<td>authorization failure that caused DA lockout to increment</td>
</tr>
<tr>
<td>TPM_RC_BAD_AUTH</td>
<td>authorization failure did not cause DA lockout to increment</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
IncrementLockout(
    UINT32           sessionIndex
) {
    TPM_HANDLE       handle = s_associatedHandles[sessionIndex];
    TPM_HANDLE       sessionHandle = s_sessionHandles[sessionIndex];
    SESSION         *session = NULL;

    // Don't increment lockout unless the handle associated with the session
    // is DA protected or the session is bound to a DA protected entity.
    if(sessionHandle == TPM_RS_PW)
    {
        if(IsDAExempted(handle))
        return TPM_RC_BAD_AUTH;
    }
    else
    {
        session = SessionGet(sessionHandle);
        // If the session is bound to lockout, then use that as the relevant
        // handle. This means that an authorization failure with a bound session
        // bound to lockoutAuth will take precedence over any other
        // lockout check
        if(session->attributes.isLockoutBound == SET)
            handle = TPM_RH_LOCKOUT;
        if(session->attributes.isDaBound == CLEAR
            && (IsDAExempted(handle) || session->attributes.includeAuth == CLEAR))
            // If the handle was changed to TPM_RH_LOCKOUT, this will not return
            // TPM_RC_BAD_AUTH
            return TPM_RC_BAD_AUTH;
    }    
    if(handle == TPM_RH_LOCKOUT)
    {
        pAssert(gp.lockOutAuthEnabled == TRUE);

        // lockout is no longer enabled
        gp.lockOutAuthEnabled = FALSE;

        // For TPM_RH_LOCKOUT, if lockoutRecovery is 0, no need to update NV since
        // the lockout authorization will be reset at startup.
        if(gp.lockoutRecovery != 0)
        {
```
if(NV_IS_AVAILABLE)
    // Update NV.
    NV_SYNC_PERSISTENT(lockOutAuthEnabled);
else
    // No NV access for now. Put the TPM in pending mode.
    s_DAPendingOnNV = TRUE;
}
else
{
    if(gp.recoveryTime != 0)
    {
        gp.failedTries++;
        if(NV_IS_AVAILABLE)
            // Record changes to NV. NvWrite will SET g_updateNV
            NV_SYNC_PERSISTENT(failedTries);
        else
            // No NV access for now. Put the TPM in pending mode.
            s_DAPendingOnNV = TRUE;
    }
    // Register a DA failure and reset the timers.
    DARegisterFailure(handle);
    return TPM_RC_AUTH_FAIL;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>handle points to the session start entity</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>handle does not point to the session start entity</td>
</tr>
</tbody>
</table>

static BOOL IsSessionBindEntity(
    TPM_HANDLE associatedHandle, // IN: handle to be authorized
    SESSION *session           // IN: associated session
)
{
    TPM2B_NAME entity;         // The bind value for the entity
    // If the session is not bound, return FALSE.
    if(session->attributes.isBound)
    {
        // Compute the bind value for the entity.
        SessionComputeBoundEntity(associatedHandle, &entity);
        // Compare to the bind value in the session.
        return MemoryEqual2B(&entity.b, &session->u1.boundEntity.b);
    }
    return FALSE;
}
6.4.3.4 IsPolicySessionRequired()

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

a) the command requires the DUP role,

b) the command requires the ADMIN role and the authorized entity is an object and its adminWithPolicy bit is SET, or

c) the command requires the ADMIN role and the authorized entity is a permanent handle or an NV Index.

d) The authorized entity is a PCR belonging to a policy group, and has its policy initialized

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>policy session is required</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>policy session is not required</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>authValue is available</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>authValue is not available</td>
</tr>
</tbody>
</table>

```c
static BOOL IsAuthValueAvailable(
    TPM_HANDLE handle,       // IN: handle of entity
    COMMAND_INDEX commandIndex, // IN: command index
    UINT32 sessionIndex       // IN: session index
)
{
    BOOL result = FALSE;
    //
    switch(HandleGetType(handle))
    {
    case TPM_HT_PERMANENT:
        switch(handle)
        {
        // At this point hierarchy availability has already been
        // checked so primary seed handles are always available here
        case TPM_RH_OWNER:
        case TPM_RH_ENDORSEMENT:
        case TPM_RH_PLATFORM:
        #ifdef VENDOR_PERMANENT
        // This vendor defined handle associated with the
        // manufacturer’s shared secret
        case VENDOR_PERMANENT:
        #endif
        // The DA checking has been performed on LockoutAuth but we
        // bypass the DA logic if we are using lockout policy. The
        // policy would allow execution to continue an lockoutAuth
        // could be used, even if direct use of lockoutAuth is disabled
        case TPM_RH_LOCKOUT:
        // NullAuth is always available.
        case TPM_RH_NULL:
            result = TRUE;
            break;
        default:
        // Otherwise authValue is not available.
            break;
        }
        break;
    case TPM_HT_TRANSIENT:
        // A persistent object has already been loaded and the internal
        // handle changed.
        {
            OBJECT *object;
            TPMA_OBJECT attributes;
            //
            object = HandleToObject(handle);
            attributes = object->publicArea.objectAttributes;
```
// authValue is always available for a sequence object.
// An alternative for this is to
// SET_ATTRIBUTE(object->publicArea, TPMA_OBJECT, userWithAuth) when the
// sequence is started.
if (ObjectIsSequence(object))
{
    result = TRUE;
    break;
}
// authValue is available for an object if it has its sensitive
// portion loaded and
// 1. userWithAuth bit is SET, or
// 2. ADMIN role is required
if (object->attributes.publicOnly == CLEAR
    && (IS_ATTRIBUTE(attributes, TPMA_OBJECT, userWithAuth)
        || (CommandAuthRole(commandIndex, sessionIndex) == AUTH_ADMIN
            && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, adminWithPolicy))))
    result = TRUE;
}
break;
case TPM_HT_NV_INDEX:
    // NV Index.
{
    NV_REF locator;
    NV_INDEX *nvIndex = NvGetIndexInfo(handle, &locator);
    TPMA_NV nvAttributes;
    // pAssert(nvIndex != 0);

    nvAttributes = nvIndex->publicArea.attributes;
    if (IsWriteOperation(commandIndex))
    {
        // AuthWrite can't be set for a PIN index
        if (IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHWRITE))
            result = TRUE;
    }
else
{
    // A "read" operation
    // For a PIN Index, the authValue is available as long as the
    // Index has been written and the pinCount is less than pinLimit
    if (IsNvPinFailIndex(nvAttributes)
        || IsNvPinPassIndex(nvAttributes))
        // For non-PIN Indexes, need to allow use of the authValue
        else if (IS_ATTRIBUTE(nvAttributes, TPMA_NV, AUTHREAD))
            result = TRUE;
}
break;
case TPM_HT_PCR:
    // PCR handle.
    // authValue is always allowed for PCR
    result = TRUE;
    break;
default:
    // Otherwise, authValue is not available
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>authPolicy is available</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>authPolicy is not available</td>
</tr>
</tbody>
</table>

```c
static BOOL IsAuthPolicyAvailable(
    TPM_HANDLE       handle, // IN: handle of entity
    COMMAND_INDEX    commandIndex, // IN: command index
    UINT32           sessionIndex // IN: session index
)
{
    BOOL            result = FALSE;
    // switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            switch(handle)
            {
                // At this point hierarchy availability has already been checked.
                case TPM_RH_OWNER:
                    if(gp.ownerPolicy.t.size != 0)
                        result = TRUE;
                    break;
                case TPM_RH_ENDORSEMENT:
                    if(gp.endorsementPolicy.t.size != 0)
                        result = TRUE;
                    break;
                case TPM_RH_PLATFORM:
                    if(gc.platformPolicy.t.size != 0)
                        result = TRUE;
                    break;
                case TPM_RH_LOCKOUT:
                    if(gp.lockoutPolicy.t.size != 0)
                        result = TRUE;
                    break;
                default:
                    break;
            }
            // Object handle.
            // An evict object would already have been loaded and given a
transient object handle by this point.
            OBJECT  *object = HandleToObject(handle);
            // Policy authorization is not available for an object with only
            // public portion loaded.
            if(object->attributes.publicOnly == CLEAR)
                {
                    // Policy authorization is always available for an object but
```
// is never available for a sequence.
if (!ObjectIsSequence(object))
    result = TRUE;
}
break;
}
case TPM_HT_NV_INDEX:
    // An NV Index.
    {
        NV_INDEX         *nvIndex = NvGetIndexInfo(handle, NULL);
        TPMA_NV           nvAttributes = nvIndex->publicArea.attributes;
        // If the policy size is not zero, check if policy can be used.
        if (nvIndex->publicArea.authPolicy.t.size != 0)
            {
                // If policy session is required for this handle, always
                // uses policy regardless of the attributes bit setting
                if (IsPolicySessionRequired(commandIndex, sessionIndex))
                    result = TRUE;
                // Otherwise, the presence of the policy depends on the NV
                // attributes.
                else if (IsWriteOperation(commandIndex))
                    {
                        if (IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYWRITE))
                            result = TRUE;
                        }
                else
                    {
                        if (IS_ATTRIBUTE(nvAttributes, TPMA_NV, POLICYREAD))
                            result = TRUE;
                        }
            }
        break;
    case TPM_HT_PCR:
        // PCR handle.
        if (PCRPolicyIsAvailable(handle))
            result = TRUE;
        break;
    default:
        break;
    }
    return result;

6.4.4 Session Parsing Functions

6.4.4.1 ClearCpRpHashes()

void
ClearCpRpHashes(
    COMMAND         *command
)
{
    #if ALG_SHA1
    command->sha1CpHash.t.size = 0;
    command->sha1RpHash.t.size = 0;
    #endif
    #if ALG_SHA256
    command->sha256CpHash.t.size = 0;
    command->sha256RpHash.t.size = 0;
    #endif
    #if ALG_SHA384
    command->sha384CpHash.t.size = 0;
    }
6.4.4.2 GetCpHashPointer()

Function to get a pointer to the cpHash of the command

```c
static TPM2B_DIGEST *
GetCpHashPointer(
  COMMAND         *command,
  TPMI_ALG_HASH   hashAlg
)
{
  TPM2B_DIGEST     *retVal;

  // switch(hashAlg)
  {
    #if ALG_SHA1
    case ALG_SHA1_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sha1CpHash;
      break;
    #endif
    #if ALG_SHA256
    case ALG_SHA256_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sha256CpHash;
      break;
    #endif
    #if ALG_SHA384
    case ALG_SHA384_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sha384CpHash;
      break;
    #endif
    #if ALG_SHA512
    case ALG_SHA512_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sha512CpHash;
      break;
    #endif
    #if ALG_SM3_256
    case ALG_SM3_256_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sm3_256CpHash;
      break;
    #endif
    default:
      retVal = NULL;
    break;
  }
  return retVal;
}
```

6.4.4.3 GetRpHashPointer()

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

```c
static TPM2B_DIGEST *
GetRpHashPointer()
```
GetRpHashPointer(
  COMMAND *command,
  TPMI_ALG_HASH hashAlg
)
{
  TPM2B_DIGEST *retVal;

  switch(hashAlg)
  {
    #if ALG_SHA1
    case ALG_SHA1_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sha1RpHash;
      break;
    #endif
    #if ALG_SHA256
    case ALG_SHA256_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sha256RpHash;
      break;
    #endif
    #if ALG_SHA384
    case ALG_SHA384_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sha384RpHash;
      break;
    #endif
    #if ALG_SHA512
    case ALG_SHA512_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sha512RpHash;
      break;
    #endif
    #if ALG_SM3_256
    case ALG_SM3_256_VALUE:
      retVal = (TPM2B_DIGEST *)&command->sm3_256RpHash;
      break;
    #endif
    default:
      retVal = NULL;
      break;
  }
  return retVal;
}

6.4.4.4 ComputeCpHash()

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

static TPM2B_DIGEST *
ComputeCpHash(
  COMMAND *command,    // IN: command parsing structure
  TPMI_ALG_HASH hashAlg    // IN: hash algorithm
)
{
  UINT32 i;
  HASH_STATE hashState;
  TPM2B_NAME name;
  TPM2B_DIGEST *cpHash;

  // cpHash = hash(commandCode [ || authName1
  // [ || authName2
  // [ || authName 3 ]]
  // [ || parameters])

  // A cpHash can contain just a commandCode only if the lone session is
  // an audit session.
  // Get pointer to the hash value
  cpHash = GetCpHashPointer(command, hashAlg);
if (cpHash->t.size == 0) {
    cpHash->t.size = CryptHashStart(&hashState, hashAlg);
    // Add commandCode.
    CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
    // Add authNames for each of the handles.
    for (i = 0; i < command->handleNum; i++)
        CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
                           &name)->b);
    // Add the parameters.
    CryptDigestUpdate(&hashState, command->parameterSize,
                     command->parameterBuffer);
    // Complete the hash.
    CryptHashEnd2B(&hashState, &cpHash->b);
}
return cpHash;

6.4.4.5 GetCpHash()

This function is used to access a precomputed cpHash.

static TPM2B_DIGEST *
GetCpHash(COMMAND *command,
           TPMI_ALG_HASH hashAlg)
{
    TPM2B_DIGEST *cpHash = GetCpHashPointer(command, hashAlg);
    //
    pAssert(cpHash->t.size != 0);
    return cpHash;

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().

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>template hash equal to session-&gt;templateHash</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>template hash not equal to session-&gt;templateHash</td>
</tr>
</tbody>
</table>

static BOOL CompareTemplateHash(
    COMMAND *command, // IN: parsing structure
    SESSION *session  // IN: session data
){
    BYTE *pBuffer = command->parameterBuffer;
    INT32 pSize = command->parameterSize;
    TPM2B_DIGEST tHash;
    UINT16 size;
    //
    // Only try this for the three commands for which it is intended
    if (command->code != TPM_CC_Create
        && command->code != TPM_CC_CreatePrimary
        && command->code != TPM_CC_CreateLoaded
    )
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif
        #endif

Page 212
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 // Advance the pointer
562 pBuffer += size;
563 // Get the size of what should be the template
564 if(UINT16_Unmarshal(&size, &pBuffer, &pSize) != TPM_RC_SUCCESS)
565   return FALSE;
566 // See if this is reasonable
567 if(size > pSize)
568   return FALSE;
569 // Hash the template data
570 tHash.t.size = CryptHashBlock(session->authHashAlg, size, pBuffer,
571 sizeof(tHash.t.buffer), tHash.t.buffer);
572 return (MemoryEqual2B(&session->ul.templateHash.b, &tHash.b));
573 }
574
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 // nameHash.t.size = CryptHashStart(&hashState, session->authHashAlg);
587 // Add names.
588 for(i = 0; i < command->handleNum; i++)
589   CryptDigestUpdate2B(&hashState, &EntityGetName(command->handles[i],
590     &name)->b);
591 // Complete hash.
592 CryptHashEnd2B(&hashState, &nameHash.b);
593 // and compare
594 return (MemoryEqual2B(session->ul.nameHash.t.buffer, nameHash.t.buffer,
595   nameHash.t.size));
596 }
597
6.4.4.8 CheckPWAuthSession()

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

<table>
<thead>
<tr>
<th>Error Return</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_AUTH_FAIL</td>
<td>authorization fails and increments DA failure count</td>
</tr>
<tr>
<td>TPM_RC_BAD_AUTH</td>
<td>authorization fails but DA does not apply</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
CheckPWAuthSession(
    UINT32 sessionIndex  // IN: index of session to be processed
)
{
    TPM2B_AUTH authValue;
    TPM_HANDLE associatedHandle = s_associatedHandles[sessionIndex];
    // Strip trailing zeros from the password.
    MemoryRemoveTrailingZeros(&s_inputAuthValues[sessionIndex]);
    // Get the authValue with trailing zeros removed
    EntityGetAuthValue(associatedHandle, &authValue);
    // Success if the values are identical.
    if (MemoryEqual2B(&s_inputAuthValues[sessionIndex].b, &authValue.b))
    {
        return TPM_RC_SUCCESS;
    }
    else // if the digests are not identical
    {
        // Invoke DA protection if applicable.
        return IncrementLockout(sessionIndex);
    }
}
```

#### 6.4.4.9 ComputeCommandHMAC()

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

```c
static TPM2B_DIGEST *
ComputeCommandHMAC(
    COMMAND *command,   // IN: primary control structure
    UINT32 sessionIndex, // IN: index of session to be processed
    TPM2B_DIGEST *hmac   // OUT: authorization HMAC
)
{
    TPM2B_TYPE(KEY, (sizeof(AUTH_VALUE) * 2));
    TPM2B_KEY key;
    BYTE *marshalBuffer = malloc(sizeof(TPMA_SESSION));
    BYTE *buffer;
    UINT32 marshalSize;
    HMAC_STATE hmacState;
    TPM2B_NONCE *nonceDecrypt;
    TPM2B_NONCE *nonceEncrypt;
    SESSION *session;
    // Determine if extra nonceTPM values are going to be required.
    // If this is the first session (sessionIndex = 0) and it is an authorization
    // session that uses an HMAC, then check if additional session nonces are to be
    // included.
    if (sessionIndex == 0 && s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
    {
        // If there is a decrypt session and if this is not the decrypt session,
```
if (s_decryptSessionIndex != UNDEFINED_INDEX && s_decryptSessionIndex != sessionIndex) {
    // Will add the nonce for the decrypt session.
    SESSION *decryptSession = SessionGet(s_sessionHandles[s_decryptSessionIndex]);
    nonceDecrypt = &decryptSession->nonceTPM;
}

if (s_encryptSessionIndex != UNDEFINED_INDEX && s_encryptSessionIndex != sessionIndex && s_encryptSessionIndex != s_decryptSessionIndex) {
    // Have to have the nonce for the encrypt session.
    SESSION *encryptSession = SessionGet(s_sessionHandles[s_encryptSessionIndex]);
    nonceEncrypt = &encryptSession->nonceTPM;
}

// Continue with the HMAC processing.
session = SessionGet(s_sessionHandles[sessionIndex]);

// Generate HMAC key.
MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));

// Check if the session has an associated handle and if the associated entity
// is the one to which the session is bound. If not, add the authValue of
// this entity to the HMAC key.
if (session->attributes.includeAuth == SET) {
    TPM2B_AUTH authValue;
    // Get the entity authValue with trailing zeros removed
    EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);
    // add the authValue to the HMAC key
    MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));
}

// if the HMAC key size is 0, a NULL string HMAC is allowed
if (key.t.size == 0 && s_inputAuthValues[sessionIndex].t.size == 0) {
    hmac->t.size = 0;
    return hmac;
}

// Start HMAC
hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);

// Add cpHash
CryptDigestUpdate2B(&hmacState.hashState,
    &ComputeCpHash(command, session->authHashAlg)->b);

// Add nonces as required
CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);
CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
if (nonceDecrypt != NULL)柔软
    CryptDigestUpdate2B(&hmacState.hashState, &nonceDecrypt->b);
if (nonceEncrypt != NULL)柔软
    CryptDigestUpdate2B(&hmacState.hashState, &nonceEncrypt->b);

// Add sessionAttributes
buffer = marshalBuffer;
marshalSize = TPMA_SESSION_Marshal(&(s_attributes[sessionIndex]),
Trusted Platform Module Library

Part 4: Supporting Routines

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_AUTH_FAIL</td>
<td>authorization failure caused failureCount increment</td>
</tr>
<tr>
<td>TPM_RC_BAD_AUTH</td>
<td>authorization failure did not cause failureCount increment</td>
</tr>
</tbody>
</table>

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

6.4.4.11 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:

a) compare policyDigest in session with authPolicy associated with the entity to be authorized;
b) compare timeout if applicable;
c) compare commandCode if applicable;
d) compare cpHash if applicable; and
e) 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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_PCR_CHANGED</td>
<td>PCR value is not current</td>
</tr>
<tr>
<td>TPM_RC_POLICY_FAIL</td>
<td>policy session fails</td>
</tr>
<tr>
<td>TPM_RC_LOCALITY</td>
<td>command locality is not allowed</td>
</tr>
<tr>
<td>TPM_RC_POLICY_CC</td>
<td>CC doesn't match</td>
</tr>
<tr>
<td>TPM_RC_EXPIRED</td>
<td>policy session has expired</td>
</tr>
<tr>
<td>TPM_RC_PP</td>
<td>PP is required but not asserted</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available for write</td>
</tr>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
CheckPolicyAuthSession(
    COMMAND *command,  // IN: primary parsing structure
    UINT32 sessionIndex // IN: index of session to be processed
) {
    SESSION *session;
    TPM2B_DIGEST authPolicy;
    TPM_ALG_HASH policyAlg;
    UINT8 locality;

    // Initialize pointer to the authorization session.
    session = SessionGet(s_sessionHandles[sessionIndex]);

    // If the command is TPM2_PolicySecret(), make sure that
    // either password or authValue is required
    if(command->code == TPM_CC_PolicySecret
        && session->attributes.isPasswordNeeded == CLEAR
        && session->attributes.isAuthValueNeeded == CLEAR)
        return TPM_RC_MODE;

    // See if the PCR counter for the session is still valid.
    if(!SessionPCRValueIsCurrent(session))
        return TPM_RC_PCR_CHANGED;

    // Get authPolicy.
    policyAlg = EntityGetAuthPolicy(s_associatedHandles[sessionIndex],
        &authPolicy);
    // Check authPolicy.
    if(!MemoryEqual2B(&session->u2.policyDigest.b, &authPolicy.b))
        return TPM_RC_POLICY_FAIL;

    // Policy is OK so check if the other factors are correct
    // Compare policy hash algorithm.
    if(policyAlg != session->authHashAlg)
        return TPM_RC_POLICY_FAIL;

    // Compare timeout.
    if(session->timeout != 0)
        { // Cannot compare time if clock stop advancing. An TPM_RC_NV_UNAVAILABLE
            // or TPM_RC_NV_RATE error may be returned here. This doesn't mean that
            // a new nonce will be created just that, because TPM time can't advance
            // we can't do time-based operations.
            RETURN_IF_NV_IS_NOT_AVAILABLE;

            if((session->timeout < g_time)
                || (session->epoch != g_timeEpoch))
                return TPM_RC_EXPIRED;
        }

    // If command code is provided it must match
```
if (session->commandCode != 0)
{
    if (session->commandCode != command->code)
        return TPM_RC_POLICY_CC;
}
else
    // If command requires a DUP or ADMIN authorization, the session must have
    // command code set.
    AUTH_ROLE role = CommandAuthRole(command->index, sessionIndex);
    if (role == AUTH_ADMIN || role == AUTH_DUP)
        return TPM_RC_POLICY_FAIL;

    // Check command locality.
    BYTE  sessionLocality[sizeof(TPMA_LOCALITY)];
    BYTE  *buffer = sessionLocality;

    // Get existing locality setting in canonical form
    sessionLocality[0] = 0;
    TPMA_LOCALITY_Marshal(&session->commandLocality, &buffer, NULL);

    // See if the locality has been set
    if (sessionLocality[0] != 0)
    {
        // If so, get the current locality
        locality = _plat__LocalityGet();
        if (locality < 5)
        {
            if ((sessionLocality[0] & (1 << locality)) == 0)
                || sessionLocality[0] > 31)
                return TPM_RC_LOCALITY;
        }
        else if (locality > 31)
        {
            if (sessionLocality[0] != locality)
                return TPM_RC_LOCALITY;
        }
        else
        {
            // Could throw an assert here but a locality error is just
            // as good. It just means that, whatever the locality is, it isn't
            // the locality requested so...
            return TPM_RC_LOCALITY;
        }
    }

    // end of locality check

} // Check physical presence.
if (session->attributes.isPPRequired == SET
    && !_plat__PhysicalPresenceAsserted())
    return TPM_RC_PP;

    // Compare cpHash/nameHash if defined, or if the command requires an ADMIN or
    // DUP role for this handle.
    if (session->u1.cpHash.b.size != 0)
    {
        BOOL OK;
        if (session->attributes.isCpHashDefined)
            // Compare cpHash.
            OK = MemoryEqual2B(&session->u1.cpHash.b,
                &ComputeCpHash(command, session->authHashAlg)->b);
        else if (session->attributes.isTemplateSet)
            OK = CompareTemplateHash(command, session);
        else
            OK = CompareNameHash(command, session);

        if (!OK)
            return TPM_RC_POLICY_FAIL;
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SUCCESS</td>
<td>unmarshaled without error</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>the number of bytes unmarshaled is not the same as the value for authorizationSize in the command</td>
</tr>
</tbody>
</table>

```c
static TPM_RC RetrieveSessionData(COMMAND *command) {
    int i;
    TPM_RC result;
    SESSION *session;
    TPMA_SESSION sessionAttributes;
    TPM_HT sessionType;
    INT32 sessionIndex;
    TPM_RC errorIndex;

    // s_decryptSessionIndex = UNDEFINED_INDEX;
    // s_encryptSessionIndex = UNDEFINED_INDEX;
    // s_auditSessionIndex = UNDEFINED_INDEX;

    for(sessionIndex = 0; command->authSize > 0; sessionIndex++) {
        errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];

        // If maximum allowed number of sessions has been parsed, return a size
        // error with a session number that is larger than the number of allowed
        // sessions
        if(sessionIndex == MAX_SESSION_NUM)
            return TPM_RC_SIZE + errorIndex;

        // make sure that the associated handle for each session starts out
        // unassigned
        s_associatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;

        // First parameter: Session handle.
        result = TPMI_SH_AUTH_SESSION_Unmarshal(
```
&s_sessionHandles[sessionIndex],
&command->parameterBuffer,
&command->authSize, TRUE);
if(result != TPM_RC_SUCCESS)
    return result + TPM_RC_S + g_rcIndex[sessionIndex];
// Second parameter: Nonce.
result = TPM2B_NONCE_Unmarshal(&s_nonceCaller[sessionIndex],
    &command->parameterBuffer,
    &command->authSize);
if(result != TPM_RC_SUCCESS)
    return result + TPM_RC_S + g_rcIndex[sessionIndex];
// Third parameter: sessionAttributes.
result = TPMA_SESSION_Unmarshal(&s_attributes[sessionIndex],
    &command->parameterBuffer,
    &command->authSize);
if(result != TPM_RC_SUCCESS)
    return result + TPM_RC_S + g_rcIndex[sessionIndex];
// Fourth parameter: authValue (PW or HMAC).
result = TPM2B_AUTH_Unmarshal(&s_inputAuthValues[sessionIndex],
    &command->parameterBuffer,
    &command->authSize);
if(result != TPM_RC_SUCCESS)
    return result + errorIndex;

sessionAttributes = s_attributes[sessionIndex];
if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
{
    // A PWAP session needs additional processing.
    // Can't have any attributes set other than continueSession bit
    if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt)
        || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt)
        || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit)
        || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)
        || IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset))
        return TPM_RCS_ATTRIBUTES + errorIndex;
    // The nonce size must be zero.
    if(s_nonceCaller[sessionIndex].t.size != 0)
        return TPM_RCS_NONCE + errorIndex;
    continue;
}
// For not password sessions...
// Find out if the session is loaded.
if(!SessionIsLoaded(s_sessionHandles[sessionIndex]))
    return TPM_RC_REFERENCE_S0 + sessionIndex;
sessionType = HandleGetType(s_sessionHandles[sessionIndex]);
session = SessionGet(s_sessionHandles[sessionIndex]);

// Check if the session is an HMAC/policy session.
if((session->attributes.isPolicy == SET
    && sessionType == TPM_HT_HMAC_SESSION)
    || (session->attributes.isPolicy == CLEAR
        && sessionType == TPM_HT_POLICY_SESSION))
    return TPM_RCS_HANDLE + errorIndex;
// Check that this handle has not previously been used.
for(i = 0; i < sessionIndex; i++)
    { if(s_sessionHandles[i] == s_sessionHandles[sessionIndex])
        return TPM_RCS_HANDLE + errorIndex;
    }
// If the session is used for parameter encryption or audit as well, set
// the corresponding Indexes.
// First process decrypt.
if(IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, decrypt))
{ // Check if the commandCode allows command parameter encryption.
if (DecryptSize(command->index) == 0)  
  return TPM_RCS_ATTRIBUTES + errorIndex;
// Encrypt attribute can only appear in one session
if (s_decryptSessionIndex != UNDEFINED_INDEX)  
  return TPM_RCS_ATTRIBUTES + errorIndex;
// Can't decrypt if the session's symmetric algorithm is TPM_ALG_NULL
if (session->symmetric.algorithm == TPM_ALG_NULL)  
  return TPM_RCS_SYMMETRIC + errorIndex;
// All checks passed, so set the index for the session used to decrypt
// a command parameter.
  s_decryptSessionIndex = sessionIndex;
}
// Now process encrypt.
if (IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, encrypt))  
{
  // Check if the commandCode allows response parameter encryption.
  if (EncryptSize(command->index) == 0)  
    return TPM_RCS_ATTRIBUTES + errorIndex;
  // Encrypt attribute can only appear in one session.
  if (s_encryptSessionIndex != UNDEFINED_INDEX)  
    return TPM_RCS_ATTRIBUTES + errorIndex;
  // Can't encrypt if the session's symmetric algorithm is TPM_ALG_NULL
  if (session->symmetric.algorithm == TPM_ALG_NULL)  
    return TPM_RCS_SYMMETRIC + errorIndex;
  // All checks passed, so set the index for the session used to encrypt
  // a response parameter.
  s_encryptSessionIndex = sessionIndex;
}
// At last process audit.
if (IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, audit))  
{
  // Audit attribute can only appear in one session.
  if (sAuditSessionIndex != UNDEFINED_INDEX)  
    return TPM_RCS_ATTRIBUTES + errorIndex;
  // An audit session can not be policy session.
  if (HandleGetType(s_sessionHandles[sessionIndex])  
      == TPM_HT_POLICY_SESSION)  
    return TPM_RCS_ATTRIBUTES + errorIndex;
  // If this is a reset of the audit session, or the first use
  // of the session as an audit session, it doesn't matter what
  // the exclusive state is. The session will become exclusive.
  if (!IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditReset)  
      && session->attributes.isAudit == SET)  
    {
      // Not first use or reset. If auditExclusive is SET, then this
      // session must be the current exclusive session.
      if (IS_ATTRIBUTE(sessionAttributes, TPMA_SESSION, auditExclusive)  
          && g_exclusiveAuditSession != s_sessionHandles[sessionIndex])  
        return TPM_RC_EXCLUSIVE;
    }
  s_auditSessionIndex = sessionIndex;
}
// Initialize associated handle as undefined. This will be changed when
// the handles are processed.
  s AssociatedHandles[sessionIndex] = TPM_RH_UNASSIGNED;
command->sessionNum = sessionIndex;
return TPM_RC_SUCCESS;

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available at this time</td>
</tr>
<tr>
<td>TPM_RC_LOCKOUT</td>
<td>TPM is in lockout</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
CheckLockedOut(
    BOOL lockoutAuthCheck // IN: TRUE if checking is for lockoutAuth
)
{
    // If NV is unavailable, and current cycle state recorded in NV is not
    // SU_NONE_VALUE, refuse to check any authorization because we would
    // not be able to handle a DA failure.
    if(!NV_IS_AVAILABLE && NV_IS_ORDERLY)
        return g_NvStatus;
    // Check if DA info needs to be updated in NV.
    if(s_DAPendingOnNV)
        {
            // If NV is accessible,
            RETURN_IF_NV_IS_NOT_AVAILABLE;
            // write the pending DA data and proceed.
            NV_SYNC_PERSISTENT(lockOutAuthEnabled);
            NV_SYNC_PERSISTENT(failedTries);
            s_DAPendingOnNV = FALSE;
        }
    // Lockout is in effect if checking for lockoutAuth and use of lockoutAuth
    // is disabled...
    if(lockoutAuthCheck)
        {
            if(gp.lockOutAuthEnabled == FALSE)
                return TPM_RC_LOCKOUT;
        }
    else
        {
            // ... or if the number of failed tries has been maxed out.
            if(gp.failedTries >= gp.maxTries)
                return TPM_RC_LOCKOUT;
        }
#endif
return TPM_RC_SUCCESS;
```

6.4.4.14 CheckAuthSession()

This function checks that the authorization session properly authorizes the use of the associated handle.
<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_LOCKOUT</td>
<td>entity is protected by DA and TPM is in lockout, or TPM is locked out</td>
</tr>
<tr>
<td></td>
<td>on NV update pending on DA parameters</td>
</tr>
<tr>
<td>TPM_RC_PP</td>
<td>Physical Presence is required but not provided</td>
</tr>
<tr>
<td>TPM_RC_AUTH_FAIL</td>
<td>HMAC or PW authorization failed with DA side-effects (can be a policy</td>
</tr>
<tr>
<td></td>
<td>session)</td>
</tr>
<tr>
<td>TPM_RC_BAD_AUTH</td>
<td>HMAC or PW authorization failed without DA side-effects (can be a</td>
</tr>
<tr>
<td></td>
<td>policy session)</td>
</tr>
<tr>
<td>TPM_RC_POLICY_FAIL</td>
<td>if policy session fails</td>
</tr>
<tr>
<td>TPM_RC_POLICY_CC</td>
<td>command code of policy was wrong</td>
</tr>
<tr>
<td>TPM_RC_EXPIRED</td>
<td>the policy session has expired</td>
</tr>
<tr>
<td>TPM_RC_PCR</td>
<td>???</td>
</tr>
<tr>
<td>TPM_RC_AUTH_UNAVAILABLE</td>
<td>authValue or authPolicy unavailable</td>
</tr>
</tbody>
</table>

```c
static TPM_RC CheckAuthSession(
    COMMAND *command,       // IN: primary parsing structure
    UINT32 sessionIndex     // IN: index of session to be processed
) {
    TPM_RC result = TPM_RC_SUCCESS;
    SESSION *session = NULL;
    TPM_HANDLE sessionHandle = s_sessionHandles[sessionIndex];
    TPM_HANDLE associatedHandle = s_associatedHandles[sessionIndex];
    TPM_HT sessionHandleType = HandleGetType(sessionHandle);
    // pAssert(sessionHandle != TPM_RH_UNASSIGNED);
    // Take care of physical presence
    if(associatedHandle == TPM_RH_PLATFORM)
        {
            // If the physical presence is required for this command, check for PP
            // assertion. If it isn't asserted, no point going any further.
            if(PhysicalPresenceIsRequired(command->index)
                && !_plat__PhysicalPresenceAsserted())
                return TPM_RC_PP;
        }
    if(sessionHandle != TPM_RS_PW)
        {
            session = SessionGet(sessionHandle);
            // Set includeAuth to indicate if DA checking will be required and if the
            // authValue will be included in any HMAC.
            if(sessionHandleType == TPM_HT_POLICY_SESSION)
            {
                // For a policy session, will check the DA status of the entity if either
                // isAuthValueNeeded or isPasswordNeeded is SET.
                session->attributes.includeAuth =
                    session->attributes.isAuthValueNeeded
                || session->attributes.isPasswordNeeded;
            }
            else
                {
                    // For an HMAC session, need to check unless the session
                    // is bound.
                    session->attributes.includeAuth =
                        !IsSessionBindEntity(s_associatedHandles[sessionIndex], session);
            }
```
// If the authorization session is going to use an authValue, then make sure
// that access to that authValue isn't locked out.
if (session == NULL || session->attributes.includeAuth) {
    // See if entity is subject to lockout.
    if (!IsDAExempted(associatedHandle)) {
        // See if in lockout
        result = CheckLockedOut(associatedHandle == TPM_RH_LOCKOUT);
        if (result != TPM_RC_SUCCESS)
            return result;
    }
}

// Policy or HMAC+PW?
if (sessionHandleType != TPM_HT_POLICY_SESSION) {
    // for non-policy session make sure that a policy session is not required
    if (IsPolicySessionRequired(command->index, sessionIndex))
        return TPM_RC_AUTH_TYPE;

    // The authValue must be available.
    if (!IsAuthValueAvailable(associatedHandle, command->index, sessionIndex))
        return TPM_RC_AUTH_UNAVAILABLE;
}
else {
    // ... see if the entity has a policy, ...
    // Note: IsAuthPolicyAvailable will return FALSE if the sensitive area of the
    // object is not loaded
    if (!IsAuthPolicyAvailable(associatedHandle, command->index, sessionIndex))
        return TPM_RC_AUTH_UNAVAILABLE;

    // ... and check the policy session.
    result = CheckPolicyAuthSession(command, sessionIndex);
    if (result != TPM_RC_SUCCESS)
        return result;
}

// Check authorization according to the type
if (session == NULL || session->attributes.isPasswordNeeded == SET)
    result = CheckPWAuthSession(sessionIndex);
else
    result = CheckSessionHMAC(command, sessionIndex);

// Do processing for PIN Indexes are only three possibilities for 'result' at
// this point: TPM_RC_SUCCESS, TPM_RC_AUTH_FAIL, and TPM_RC_BAD_AUTH.
// For all these cases, we would have to process a PIN index if the
// authValue of the index was used for authorization.
// See if we need to do anything to a PIN index
if (TPM_HT_NV_INDEX == HandleGetType(associatedHandle)) {
    NV_REF locator;
    NV_INDEX *nvIndex = NvGetIndexInfo(associatedHandle, &locator);
    NV_PIN pinData;
    TPMA_NV nvAttributes;

    // pAssert(nvIndex != NULL);
    nvAttributes = nvIndex->publicArea.attributes;
    // If this is a PIN FAIL index and the value has been written
    // then we can update the counter (increment or clear)
    if (IsNvPinFailIndex(nvAttributes) && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN)) {
        pinData.intVal = NvGetUINT64Data(nvIndex, locator);
        if (result != TPM_RC_SUCCESS)
            pinData.pin.pinCount++;
    }
}
else
    pinData.pin.pinCount = 0;
    NvWriteUINT64Data(nvIndex, pinData.intVal);
}

// If this is a PIN PASS Index, increment if we have used the
// authorization value for anything other than NV_Read.
// NOTE: If the counter has already hit the limit, then we
// would not get here because the authorization value would not
// be available and the TPM would have returned before it gets here
else if (IsNvPinPassIndex(nvAttributes) && IS_ATTRIBUTE(nvAttributes, TPMA_NV, WRITTEN) && result == TPM_RC_SUCCESS) {
    // If the access is valid, then increment the use counter
    pinData.intVal = NvGetUINT64Data(nvIndex, locator);
    pinData.pin.pinCount++;
    NvWriteUINT64Data(nvIndex, pinData.intVal);
}
}
return result;
}
#endif

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available for write</td>
</tr>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting</td>
</tr>
</tbody>
</table>

static TPM_RC
CheckCommandAudit(
COMMAND *command
)
{
    // If the audit digest is clear and command audit is required, NV must be
    // available so that TPM2_GetCommandAuditDigest() is able to increment
    // audit counter. If NV is not available, the function bails out to prevent
    // the TPM from attempting an operation that would fail anyway.
    if (gr.commandAuditDigest.t.size == 0 || GetCommandCode(command->index) == TPM_CC_GetCommandAuditDigest)
        RETURN_IF_NV_IS_NOT_AVAILABLE;
    ComputeCpHash(command, gp.auditHashAlg);
    return TPM_RC_SUCCESS;
}
#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

```c
TPM_RC
ParseSessionBuffer(
    COMMAND *command // IN: the structure that contains
)
{
    TPM_RC result;
    UINT32 i;
    INT32 size = 0;
    TPM2B_AUTH extraKey;
    UINT32 sessionIndex;
    TPM_RC errorIndex;
    SESSION *session = NULL;

    // Check if a command allows any session in its session area.
    if(!IsSessionAllowed(command->index))
        return TPM_RC_AUTH_CONTEXT;
    // Default-initialization.
    command->sessionNum = 0;

    result = RetrieveSessionData(command);
    if(result != TPM_RC_SUCCESS)
        return result;

    // There is no command in the TPM spec that has more handles than
    // MAX_SESSION_NUM.
    pAssert(command->handleNum <= MAX_SESSION_NUM);

    // Associate the session with an authorization handle.
    for(i = 0; i < command->handleNum; i++)
    {
        if(CommandAuthRole(command->index, i) != AUTH_NONE)
        {
            // If the received session number is less than the number of handles
            // that requires authorization, an error should be returned.
            // Note: for all the TPM 2.0 commands, handles requiring
            // authorization come first in a command input and there are only ever
            // two values requiring authorization
            if(i > (command->sessionNum - 1))
                return TPM_RC_AUTH_MISSING;
            // Record the handle associated with the authorization session
            s_associatedHandles[i] = command->handles[i];
        }
    }
    // Consistency checks are done first to avoid authorization failure when the
    // command will not be executed anyway.
    for(sessionIndex = 0; sessionIndex < command->sessionNum; sessionIndex++)
    {
        errorIndex = TPM_RC_S + g_rcIndex[sessionIndex];
        // PW session must be an authorization session
        if(s_sessionHandles[sessionIndex] == TPM_RS_PW)
        {
            if(s_associatedHandles[sessionIndex] == TPM_RH_UNASSIGNED)
                return TPM_RCS_HANDLE + errorIndex;
            // a password session can't be audit, encrypt or decrypt
            if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit))
                return TPM_RCS_ATTRIBUTES + errorIndex;
            session = NULL;
        }
        else
        {
```
```c
session = SessionGet(s_sessionHandles[sessionIndex]);

/* A trial session can not appear in session area, because it cannot */
/* be used for authorization, audit or encrypt/decrypt. */
if(session->attributes.isTrialPolicy == SET)
    return TPM_RCS_ATTRIBUTES + errorIndex;

/* See if the session is bound to a DA protected entity */
/* NOTE: Since a policy session is never bound, a policy is still */
/* usable even if the object is DA protected and the TPM is in */
/* lockout. */
if(session->attributes.isDaBound == SET)
{
    result = CheckLockedOut(session->attributes.isLockoutBound == SET);
    if(result != TPM_RC_SUCCESS)
        return result;
}

/* If this session is for auditing, make sure the cpHash is computed. */
if(IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit))
    ComputeCpHash(command, session->authHashAlg);

/* if the session has an associated handle, check the authorization */
if(s_associatedHandles[sessionIndex] != TPM_RH_UNASSIGNED)
{
    result = CheckAuthSession(command, sessionIndex);
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, errorIndex);
}
else
{
    /* a session that is not for authorization must either be encrypt, */
    /* decrypt, or audit */
    if(!IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, audit)  
    && !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, encrypt) 
    && !IS_ATTRIBUTE(s_attributes[sessionIndex], TPMA_SESSION, decrypt))
        return TPM_RCS_ATTRIBUTES + errorIndex;

    /* no authValue included in any of the HMAC computations */
    pAssert(session != NULL);
    session->attributes.includeAuth = CLEAR;

    /* check HMAC for encrypt/decrypt/audit only sessions */
    result = CheckSessionHMAC(command, sessionIndex);
    if(result != TPM_RC_SUCCESS)
        return RcSafeAddToResult(result, errorIndex);
}
#endif

/* Decrypt the first parameter if applicable. This should be the last operation */
/* in session processing. */
if(s_decryptSessionIndex != UNDEFINED_INDEX)
{ }
```
// If this is an authorization session, include the authValue in the
// generation of the decryption key
if (s_associatedHandles[s_decryptSessionIndex] != TPM_RH_UNASSIGNED)
{
    EntityGetAuthValue(s_associatedHandles[s_decryptSessionIndex],
        &extraKey);
}
else
{
    extraKey.b.size = 0;
}
size = DecryptSize(command->index);
result = CryptParameterDecryption(s_sessionHandles[s_decryptSessionIndex],
    s_nonceCaller[s_decryptSessionIndex].b, command->parameterSize, (UINT16)size,
    &extraKey, command->parameterBuffer);
if (result != TPM_RC_SUCCESS)
    return RcSafeAddToResult(result,
        TPM_RC_S + g_rcIndex[s_decryptSessionIndex]);
return TPM_RC_SUCCESS;

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_AUTH_MISSING</td>
<td>failure - one or more handles require authorization</td>
</tr>
</tbody>
</table>

TPM_RC

CheckAuthNoSession(
    COMMAND *command // IN: command parsing structure
)
{
    UINT32 i;
    TPM_RC result = TPM_RC_SUCCESS;
    // Check if the command requires authorization
    for (i = 0; i < command->handleNum; i++)
    {
        if (CommandAuthRole(command->index, i) != AUTH_NONE)
            return TPM_RC_AUTH_MISSING;
    }
    #ifdef TPM_CC_GetCommandAuditDigest
    // Check if the command should be audited.
    if (CommandAuditIsRequired(command->index))
    {
        result = CheckCommandAudit(command);
        if (result != TPM_RC_SUCCESS)
            return result;
    }
    #endif
    // Initialize number of sessions to be 0
    command->sessionNum = 0;
    return TPM_RC_SUCCESS;
}
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 \textit{rpHash} (Response Parameter Hash). The \textit{rpHash} is only computed if there is an HMAC authorization session and the return code is TPM_RC_SUCCESS.

```c
static TPM2B_DIGEST * ComputeRpHash(
    COMMAND *command, // IN: command structure
    TPM_ALG_ID hashAlg // IN: hash algorithm to compute rpHash
) {
    TPM2B_DIGEST *rpHash = GetRpHashPointer(command, hashAlg);
    HASH_STATE hashState;

    if (rpHash->t.size == 0) {
        // rpHash := hash(responseCode || commandCode || parameters)
        rpHash->t.size = CryptHashStart(&hashState, hashAlg);
        // Add hash constituents.
        CryptDigestUpdateInt(&hashState, sizeof(TPM_RC), TPM_RC_SUCCESS);
        CryptDigestUpdateInt(&hashState, sizeof(TPM_CC), command->code);
        CryptDigestUpdate(&hashState, command->parameterSize,
            command->parameterBuffer);
        // Complete hash computation.
        CryptHashEnd2B(&hashState, &rpHash->b);
    }
    return rpHash;
}
```

6.4.5.3 InitAuditSession()

This function initializes the audit data in an audit session.

```c
static void InitAuditSession(
    SESSION *session // session to be initialized
) {
    // Mark session as an audit session.
    session->attributes.isAudit = SET;
    // Audit session can not be bound.
    session->attributes.isBound = CLEAR;
    // Size of the audit log is the size of session hash algorithm digest.
    session->u2.auditDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
    // Set the original digest value to be 0.
    MemorySet(&session->u2.auditDigest.t.buffer, 0,
        session->u2.auditDigest.t.size);
    return;
}
```
6.4.5.4 UpdateAuditDigest

Function to update an audit digest

```c
static void
UpdateAuditDigest(
    COMMAND         *command,
    TPMI_ALG_HASH    hashAlg,
    TPM2B_DIGEST    *digest
)
{
    HASH_STATE       hashState;
    TPM2B_DIGEST    *cpHash = GetCpHash(command, hashAlg);
    TPM2B_DIGEST    *rpHash = ComputeRpHash(command, hashAlg);
    // digestNew := hash (digestOld || cpHash || rpHash)
    digest->t.size = CryptHashStart(&hashState, hashAlg);
    // Add old digest.
    CryptDigestUpdate2B(&hashState, &digest->b);
    // Add cpHash
    CryptDigestUpdate2B(&hashState, &cpHash->b);
    // Add rpHash
    CryptDigestUpdate2B(&hashState, &rpHash->b);
    // Finalize the hash.
    CryptHashEnd2B(&hashState, &digest->b);
}
```

6.4.5.5 Audit()

This function updates the audit digest in an audit session.

```c
static void
Audit(
    COMMAND         *command,  // IN: primary control structure
    SESSION         *auditSession  // IN: loaded audit session
)
{
    UpdateAuditDigest(command, auditSession->authHashAlg,
                       &auditSession->u2.auditDigest);
    return;
}
```

6.4.5.6 CommandAudit()

This function updates the command audit digest.

```c
static void
CommandAudit(
    COMMAND         *command  // IN:
)
{
    // If the digest.size is one, it indicates the special case of changing
    // the audit hash algorithm. For this case, no audit is done on exit.
    // NOTE: When the hash algorithm is changed, g_updateNV is set in order to
    // 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     // Bump the counter and save its value to NV.
1521     gp.auditCounter++;
1522     NV_SYNC_PERSISTENT(auditCounter);
1523 }
1524 UpdateAuditDigest(command, gp.auditHashAlg, &gr.commandAuditDigest);
1525 return;
1526 #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     // Iterate through sessions
1537     for(i = 0; i < command->sessionNum; i++)
1538     {
1539         SESSION     *session;
1540     // PW session do not have a loaded session and can not be an audit
1541     // session either. Skip it.
1542     if(s_sessionHandles[i] == TPM_RS_PW)
1543         continue;
1544     session = SessionGet(s_sessionHandles[i]);
1545     // If a session is used for audit
1546     if(IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, audit))
1547     {
1548         // An audit session has been found
1549         auditSession = s_sessionHandles[i];
1550     // If the session has not been an audit session yet, or
1551     // 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    // If no audit session is found in the command, and the command allows
1583    // a session then, clear the current exclusive
1584    // audit session.
1585    if (auditSession == TPM_RH_UNASSIGNED && IsSessionAllowed(command->index))
1586        {
1587            g_exclusiveAuditSession = TPM_RH_UNASSIGNED;
1588        }
1589    return;
1590

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    // Add the object authValue if required
1611    if (session->attributes.includeAuth == SET)
1612        {
1613            // Note: includeAuth may be SET for a policy that is used in
// UndefineSpaceSpecial(). At this point, the Index has been deleted
// so the includeAuth will have no meaning. However, the
// s_associatedHandles[] value for the session is now set to TPM_RH_NULL so
// this will return the authValue associated with TPM_RH_NULL and that is
// and empty buffer.
  TPM2B_AUTH authValue;

  // Get the authValue with trailing zeros removed
  EntityGetAuthValue(s_associatedHandles[sessionIndex], &authValue);

  // Add it to the key
  MemoryConcat2B(&key.b, &authValue.b, sizeof(key.t.buffer));

  // if the HMAC key size is 0, the response HMAC is computed according to the
  // input HMAC
  if(key.t.size == 0
    && s_inputAuthValues[sessionIndex].t.size == 0)
  {
    hmac->t.size = 0;
    return;
  }

  // Start HMAC computation.
  hmac->t.size = CryptHmacStart2B(&hmacState, session->authHashAlg, &key.b);

  // Add hash components.
  CryptDigestUpdate2B(&hmacState.hashState, &rpHash->b);
  CryptDigestUpdate2B(&hmacState.hashState, &session->nonceTPM.b);
  CryptDigestUpdate2B(&hmacState.hashState, &s_nonceCaller[sessionIndex].b);

  // Add session attributes.
  buffer = marshalBuffer;
  marshallSize = TPMA_SESSION_Marshal(&s_attributes[sessionIndex], &buffer, NULL);
  CryptDigestUpdate(&hmacState.hashState, marshallSize, marshalBuffer);

  // Finalize HMAC.
  CryptHmacEnd2B(&hmacState, &hmac->b);
  return;
}

6.4.5.9 UpdateInternalSession()

Updates internal sessions:

a) Restarts session time.

b) Clears a policy session since nonce is rolling.

static void UpdateInternalSession(
  SESSION *session,       // IN: the session structure
  UINT32 i                 // IN: session number
)
{
  // If nonce is rolling in a policy session, the policy related data
  // will be re-initialized.
  if(HandleGetType(s_sessionHandles[i]) == TPM_HT_POLICY_SESSION
    && IS_ATTRIBUTE(s_attributes[i], TPMA_SESSION, continueSession))
  {
    // When the nonce rolls it starts a new timing interval for the
    // policy session.
    SessionResetPolicyData(session);
    SessionSetStartTime(session);
  }
6.4.5.10 BuildSingleResponseAuth()

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

```c
static TPM2B_NONCE *
BuildSingleResponseAuth(
    COMMAND *command, // IN: command structure
    UINT32 sessionIndex, // IN: session index to be processed
    TPM2B_AUTH *auth // OUT: authHMAC
) {
    // Fill in policy/HMAC based session response.
    SESSION *session = SessionGet(s_sessionHandles[sessionIndex]);
    // If the session is a policy session with isPasswordNeeded SET, the
    // authorization field is empty.
    if(HandleGetType(s_sessionHandles[sessionIndex]) == TPM_HT_POLICY_SESSION
        && session->attributes.isPasswordNeeded == SET)
        auth->t.size = 0;
    else // Compute response HMAC.
        ComputeResponseHMAC(command, sessionIndex, session, auth);
    UpdateInternalSession(session, sessionIndex);
    return &session->nonceTPM;
}
```

6.4.5.11 UpdateAllNonceTPM()

Updates TPM nonce for all sessions in command.

```c
static void
UpdateAllNonceTPM(
    COMMAND *command // IN: controlling structure
) {
    UINT32 i;
    SESSION *session;
    // for(i = 0; i < command->sessionNum; i++)
    { // If not a PW session, compute the new nonceTPM.
        if(s_sessionHandles[i] != TPM_RS_PW)
            { // Compute nonceTPM in both internal session and response.
                session = SessionGet(s_sessionHandles[i]);
                CryptRandomGenerate(session->nonceTPM.t.size,
                        session->nonceTPM.t.buffer);
            }
    }
    return;
}
```

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.

```c
void
BuildResponseSession(
    COMMAND *command    // IN: structure that has relevant command
    // information
) {
    pAssert(command->authSize == 0);

    // Reset the parameter buffer to point to the start of the parameters so that
    // there is a starting point for any rpHash that might be generated and so there
    command->parameterBuffer = command->responseBuffer - command->parameterSize;

    // Session nonces should be updated before parameter encryption
    if(command->tag == TPM_ST_SESSIONS)
        UpdateAllNonceTPM(command);

    // Encrypt first parameter if applicable. Parameter encryption should
    // happen after nonce update and before any rpHash is computed.
    // If the encrypt session is associated with a handle, the authValue of
    // this handle will be concatenated with sessionKey to generate
    // encryption key, no matter if the handle is the session bound entity
    // or not. The authValue is added to sessionKey only when the authValue
    // is available.
    if(s_encryptSessionIndex != UNDEFINED_INDEX)
        {UINT32          size;
         TPM2B_AUTH      extraKey;
         // extraKey.b.size = 0;
         // If this is an authorization session, include the authValue in the
         // generation of the encryption key
         if(s_associatedHandles[s_encryptSessionIndex] != TPM_RH_UNASSIGNED)
             {EntityGetAuthValue(s_associatedHandles[s_encryptSessionIndex],
             &extraKey);
             }

         size = EncryptSize(command->index);
         CryptParameterEncryption(s_sessionHandles[s_encryptSessionIndex],
         &s_nonceCaller[s_encryptSessionIndex].b,
         (UINT16)size,
         &extraKey,
         command->parameterBuffer);
        }

    // Audit sessions should be processed regardless of the tag because
    // a command with no session may cause a change of the exclusivity state.
    UpdateAuditSessionStatus(command);
    #if CC_GetCommandAuditDigest
    // Command Audit
    if(CommandAuditIsRequired(command->index))
        CommandAudit(command);
    #endif
    // Process command with sessions.
    if(command->tag == TPM_ST_SESSIONS)
        {UINT32           i;
         // pAssert(command->sessionNum > 0);

         // 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
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 return;
1811}

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         }
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

```
#include "Tpm.h"
#include "Attest_spt_fp.h"
```

7.2.2 Functions

7.2.2.1 FillInAttestInfo()

Fill in common fields of TPMS_ATTEST structure.

```
void FillInAttestInfo(  
    PMI_DH_OBJECT       signHandle,  // IN: handle of signing object  
    TPM2B_SIG_SCHEME   *scheme,  // IN/OUT: scheme to be used for signing  
    TPM2B_DATA         *data,  // IN: qualifying data  
    TPMS_ATTEST        *attest  // OUT: attest structure  
)
{
    OBJECT *signObject = HandleToObject(signHandle);
    // Magic number
    attest->magic = TPM_GENERATED_VALUE;

    if(signObject == NULL)  
    {
        // The name for a null handle is TPM_RH_NULL
        // This is defined because UINT32_TO_BYTE_ARRAY does a cast. If the
        // size of the cast is smaller than a constant, the compiler warns
        // about the truncation of a constant value.
        TPM_HANDLE nullHandle = TPM_RH_NULL;
        attest->qualifiedSigner.t.size = sizeof(TPM_HANDLE);
        UINT32_TO_BYTE_ARRAY(nullHandle, attest->qualifiedSigner.t.name);
    }
    else  
    {
        // Certifying object qualified name
        if(CryptIsSchemeAnonymous(scheme->scheme))  
            attest->qualifiedSigner.t.size = 0;
        else
            attest->qualifiedSigner = signObject->qualifiedName;
    }
    // current clock in plain text
    TimeFillInfo(&attest->clockInfo);
    // Firmware version in plain text
    attest->firmwareVersion = ((UINT64)gp.firmwareV1 << (sizeof(UINT32) * 8));
    attest->firmwareVersion += gp.firmwareV2;
```
// Check the hierarchy of sign object. For NULL sign handle, the hierarchy
// will be TPM_RH_NULL
if((signObject == NULL)
   || (!signObject->attributes.epsHierarchy
      && !signObject->attributes.ppsHierarchy))
{
   // For signing key that is not in platform or endorsement hierarchy,
   // obfuscate the reset, restart and firmware version information
   UINT64          obfuscation[2];
   CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gp.shProof.b, OBFUSCATE_STRING,
            &attest->qualifiedSigner.b, NULL, 128,
            (BYTE *)&obfuscation[0], NULL, FALSE);
   // Obfuscate data
   attest->firmwareVersion += obfuscation[0];
   attest->clockInfo.resetCount += (UINT32)(obfuscation[1] >> 32);
   attest->clockInfo.restartCount += (UINT32)obfuscation[1];
}
// External data
if(CryptIsSchemeAnonymous(scheme->scheme))
   attest->extraData.t.size = 0;
else
{
   // If we move the data to the attestation structure, then it is not
   // used in the signing operation except as part of the signed data
   attest->extraData = *data;
   data->t.size = 0;
}

7.2.2.2 SignAttestInfo()

Sign a TPMS_ATTEST structure. If signHandle is TPM_RH_NULL, a null signature is returned.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>signHandle references not a signing key</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>scheme is not compatible with signHandle type</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>digest generated for the given scheme is greater than the modulus of</td>
</tr>
<tr>
<td></td>
<td>signHandle (for an RSA key); invalid commit status or failed to</td>
</tr>
<tr>
<td></td>
<td>generate r value (for an ECC key)</td>
</tr>
</tbody>
</table>

TPM_RC

SignAttestInfo(
   OBJECT          *signKey,       // IN: sign object
   TPMT_SIG_SCHEME *scheme,        // IN: sign scheme
   TPMS_ATTEST    *certifyInfo,    // IN: the data to be signed
   TPM2B_DATA     *qualifyingData, // IN: extra data for the signing
   TPM2B_ATTEST   *attest,         // OUT: marshaled attest blob to be
   // signed
   TPMT_SIGNATURE *signature       // OUT: signature
)
{
   BYTE                    *buffer;
   HASH_STATE              hashState;
   TPM2B_DIGEST            digest;
   TPM_RC                  result;
   // Marshal TPMS_ATTEST structure for hash
   buffer = attest->t.attestationData;
   attest->t.size = TPMS_ATTEST_Marshal(certifyInfo, &buffer, NULL);
   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  return result;
120 }

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>object may sign</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>object may not sign</td>
</tr>
</tbody>
</table>

BOOL
IsSigningObject(
    OBJECT  *object     // IN:
)
{
    return ((object == NULL)
        || ((IS_ATTRIBUTE(object->publicArea.objectAttributes, TPMA_OBJECT, sign)
            && object->publicArea.type != TPM_ALG_SYMCIPHER)));
7.3 Context Management Command Support (Context_spt.c)

7.3.1 Includes

```c
#include "Tpm.h"
#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_ContextSave() and TPM2_ContextLoad() to create the symmetric encryption key and IV.

```c
void ComputeContextProtectionKey(
    TPM2B_CONTEXT *contextBlob, // IN: context blob
    TPM2B_SYM_KEY *symKey, // OUT: the symmetric key
    TPM2B_IV *iv // OUT: the IV.
)
{
    UINT16 symKeyBits; // number of bits in the parent's symmetric key
    TPM2B_PROOF *proof = NULL; // the proof value to use. Is null for everything but a primary object in the Endorsement Hierarchy
    BYTE kdfResult[sizeof(TPMU_HA) * 2]; // Value produced by the KDF
    TPM2B_DATA sequence2B, handle2B;

    // Get proof value
    proof = HierarchyGetProof(contextBlob->hierarchy);

    // Get sequence value in 2B format
    sequence2B.t.size = sizeof(contextBlob->sequence);
    cAssert(sizeof(contextBlob->sequence) <= sizeof(sequence2B.t.buffer));
    MemoryCopy(sequence2B.t.buffer, &contextBlob->sequence, sizeof(contextBlob->sequence));

    // Get handle value in 2B format
    handle2B.t.size = sizeof(contextBlob->savedHandle);
    cAssert(sizeof(contextBlob->savedHandle) <= sizeof(handle2B.t.buffer));
    MemoryCopy(handle2B.t.buffer, &contextBlob->savedHandle, sizeof(contextBlob->savedHandle));

    // Get the symmetric encryption key size
    symKey->t.size = CONTEXT_ENCRYPT_KEY_BYTES;
    symKeyBits = CONTEXT_ENCRYPT_KEY_BITS;

    // Get the size of the IV for the algorithm
    iv->t.size = CryptGetSymmetricBlockSize(CONTEXT_ENCRYPT_ALG, symKeyBits);

    // KDFa to generate symmetric key and IV value
    CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &proof->b, CONTEXT_KEY, &sequence2B.b, &handle2B.b, (symKey->t.size + iv->t.size) * 8, kdfResult, NULL, FALSE);

    // Copy part of the returned value as the key
    pAssert(symKey->t.size <= sizeof(symKey->t.buffer));
    MemoryCopy(symKey->t.buffer, kdfResult, symKey->t.size);
```
// Copy the rest as the IV
pAssert(iv->t.size <= sizeof(iv->t.buffer));
MemoryCopy(iv->t.buffer, &kdfResult[symKey->t.size], iv->t.size);
return;
}

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.

void ComputeContextIntegrity(
    TPMS_CONTEXT *contextBlob, // IN: context blob
    TPM2B_DIGEST *integrity,   // OUT: integrity
)
{
    HMAC_STATE hmacState;
    TPM2B_PROOF *proof;
    UINT16 integritySize;

    // Get proof value
    proof = HierarchyGetProof(contextBlob->hierarchy);

    // Start HMAC
    integrity->t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
                                        &proof->b);

    // Compute integrity size at the beginning of context blob
    integritySize = sizeof(integrity->t.size) + integrity->t.size;

    // Adding total reset counter so that the context cannot be
    // used after a TPM Reset
    CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
                         gp.totalResetCount);

    // If this is a ST_CLEAR object, add the clear count
    // so that this context cannot be loaded after a TPM Restart
    if (contextBlob->savedHandle == 0x80000002)
        CryptDigestUpdateInt(&hmacState.hashState, sizeof(gr.clearCount),
                             gr.clearCount);

    // Adding sequence number to the HMAC to make sure that it doesn't
    // get changed
    CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->sequence),
                         contextBlob->sequence);

    // Protect the handle
    CryptDigestUpdateInt(&hmacState.hashState, sizeof(contextBlob->savedHandle),
                         contextBlob->savedHandle);

    // Adding sensitive contextData, skip the leading integrity area
    CryptDigestUpdate(&hmacState.hashState, contextBlob->contextBlob.t.size - integritySize,
                      contextBlob->contextBlob.t.buffer + integritySize);

    // Complete HMAC
    CryptHmacEnd2B(&hmacState, &integrity->b);
    return;
}
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 outsize of the hash context area gets modified.

```c
void
SequenceDataExport(
    HASH_OBJECT         *object,   // IN: an internal hash object
    HASH_OBJECT_BUFFER  *exportObject  // OUT: a sequence context in a buffer
){
    // If the hash object is not an event, then only one hash context is needed
    int count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
    for(count--; count >= 0; count--)
    {
        HASH_STATE          *hash = &object->state.hashState[count];
        size_t              offset = (BYTE *)hash - (BYTE *)object;
        BYTE                *exportHash = &((BYTE *)exportObject)[offset];
        CryptHashExportState(hash, (EXPORT_HASH_STATE *)exportHash);
    }
}
```

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 outsize of the hash context area gets modified.

```c
void
SequenceDataImport(
    HASH_OBJECT         *object,   // IN/OUT: an internal hash object
    HASH_OBJECT_BUFFER  *exportObject  // IN/OUT: a sequence context in a buffer
){
    // If the hash object is not an event, then only one hash context is needed
    int count = (object->attributes.eventSeq) ? HASH_COUNT : 1;
    for(count--; count >= 0; count--)
    {
        HASH_STATE          *hash = &object->state.hashState[count];
        size_t              offset = (BYTE *)hash - (BYTE *)object;
        BYTE                *importHash = &((BYTE *)exportObject)[offset];
        // CryptHashImportState(hash, (EXPORT_HASH_STATE *)importHash);
    }
}
```
7.4 Policy Command Support (Policy_spt.c)

7.4.1 Includes

```c
#include "Tpm.h"
#include "Policy_spt_fp.h"
#include "PolicySigned_fp.h"
#include "PolicySecret_fp.h"
#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.

```c
TPM_RC
PolicyParameterChecks(
    SESSION *session,
    UINT64 authTimeout,
    TPM2B_DIGEST *cpHashA,
    TPM2B_NONCE *nonce,
    TPM_RC blameNonce,
    TPM_RC blameCpHash,
    TPM_RC blameExpiration
)
{
    // Validate that input nonceTPM is correct if present
    if(nonce != NULL && nonce->t.size != 0) {
        if(!MemoryEqual2B(&nonce->b, &session->nonceTPM.b))
            return TPM_RC_NONCE + blameNonce;
    }
    // If authTimeout is set (expiration != 0...)
    if(authTimeout != 0) {
        // Validate input expiration.
        // Cannot compare time if clock stop advancing. A TPM_RC_NV_UNAVAILABLE
        // or TPM_RC_NV_RATE error may be returned here.
        RETURN_IF_NV_IS_NOT_AVAILABLE;
        // if the time has already passed or the time epoch has changed then the
        // time value is no longer good.
        if((authTimeout < g_time
            || (session->epoch != g_timeEpoch))
            return TPM_RC_EXPIRED + blameExpiration;
    }
    // If the cpHash is present, then check it
    if(cpHashA != NULL && cpHashA->t.size != 0) {
        // The cpHash input has to have the correct size
        if(cpHashA->t.size != session->u2.policyDigest.t.size)
            return TPM_RC_SIZE + blameCpHash;
        // If the cpHash has already been set, then this input value
        // must match the current value.
        if(session->u1.cpHash.b.size != 0
            && !MemoryEqual2B(&cpHashA->b, &session->u1.cpHash.b))
            return TPM_RC_CPHASH;
    }
```
Trusted Platform Module Library

Part 4: Supporting Routines

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.

```c
void PolicyContextUpdate(
    TPM_CC commandCode, // IN: command code
    TPM2B_NAME *name,    // IN: name of entity
    TPM2B_NONCE *ref,    // IN: the reference data
    TPM2B_DIGEST *cpHash, // IN: the cpHash (optional)
    UINT64 policyTimeout, // IN: the timeout value for the policy
    SESSION *session     // IN/OUT: policy session to be updated
)
{
    HASH_STATE hashState;

    // Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    // policyDigest size should always be the digest size of session hash algorithm.
    pAssert(session->u2.policyDigest.t.size == CryptHashGetDigestSize(session->authHashAlg));

    // add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    // add name if applicable
    if(name != NULL)
        CryptDigestUpdate2B(&hashState, &name->b);

    // Complete the digest and get the results
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // If the policy reference is not null, do a second update to the digest.
    if(ref != NULL)
    {
        // Start second hash computation
        CryptHashStart(&hashState, session->authHashAlg);

        // add policyDigest
        CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

        // add policyRef
        CryptDigestUpdate2B(&hashState, &ref->b);

        // Complete second digest
        CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
    }
    // Deal with the cpHash. If the cpHash value is present
    // then it would have already been checked to make sure that
    // it is compatible with the current value so all we need
    // to do here is copy it and set the isCpHashDefined attribute
    if(cpHash != NULL & & cpHash->t.size != 0)
    {
        session->u1.cphash = *cpHash;
        session->attributes.isCpHashDefined = SET;
    }
```

```
            return TPM_RC_SUCCESS;
        }
    }
```

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.

```c
void PolicyContextUpdate(
    TPM_CC commandCode, // IN: command code
    TPM2B_NAME *name,    // IN: name of entity
    TPM2B_NONCE *ref,    // IN: the reference data
    TPM2B_DIGEST *cpHash, // IN: the cpHash (optional)
    UINT64 policyTimeout, // IN: the timeout value for the policy
    SESSION *session     // IN/OUT: policy session to be updated
)
{
    HASH_STATE hashState;

    // Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    // policyDigest size should always be the digest size of session hash algorithm.
    pAssert(session->u2.policyDigest.t.size == CryptHashGetDigestSize(session->authHashAlg));

    // add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    // add name if applicable
    if(name != NULL)
        CryptDigestUpdate2B(&hashState, &name->b);

    // Complete the digest and get the results
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // If the policy reference is not null, do a second update to the digest.
    if(ref != NULL)
    {
        // Start second hash computation
        CryptHashStart(&hashState, session->authHashAlg);

        // add policyDigest
        CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

        // add policyRef
        CryptDigestUpdate2B(&hashState, &ref->b);

        // Complete second digest
        CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
    }
    // Deal with the cpHash. If the cpHash value is present
    // then it would have already been checked to make sure that
    // it is compatible with the current value so all we need
    // to do here is copy it and set the isCpHashDefined attribute
    if(cpHash != NULL & & cpHash->t.size != 0)
    {
        session->u1.cphash = *cpHash;
        session->attributes.isCpHashDefined = SET;
    }
```

```c
            return TPM_RC_SUCCESS;
        }
    }
```

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.

```c
void PolicyContextUpdate(
    TPM_CC commandCode, // IN: command code
    TPM2B_NAME *name,    // IN: name of entity
    TPM2B_NONCE *ref,    // IN: the reference data
    TPM2B_DIGEST *cpHash, // IN: the cpHash (optional)
    UINT64 policyTimeout, // IN: the timeout value for the policy
    SESSION *session     // IN/OUT: policy session to be updated
)
{
    HASH_STATE hashState;

    // Start hash
    CryptHashStart(&hashState, session->authHashAlg);

    // policyDigest size should always be the digest size of session hash algorithm.
    pAssert(session->u2.policyDigest.t.size == CryptHashGetDigestSize(session->authHashAlg));

    // add old digest
    CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

    // add name if applicable
    if(name != NULL)
        CryptDigestUpdate2B(&hashState, &name->b);

    // Complete the digest and get the results
    CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);

    // If the policy reference is not null, do a second update to the digest.
    if(ref != NULL)
    {
        // Start second hash computation
        CryptHashStart(&hashState, session->authHashAlg);

        // add policyDigest
        CryptDigestUpdate2B(&hashState, &session->u2.policyDigest.b);

        // add policyRef
        CryptDigestUpdate2B(&hashState, &ref->b);

        // Complete second digest
        CryptHashEnd2B(&hashState, &session->u2.policyDigest.b);
    }
    // Deal with the cpHash. If the cpHash value is present
    // then it would have already been checked to make sure that
    // it is compatible with the current value so all we need
    // to do here is copy it and set the isCpHashDefined attribute
    if(cpHash != NULL & & cpHash->t.size != 0)
    {
        session->u1.cphash = *cpHash;
        session->attributes.isCpHashDefined = SET;
    }
```
Part 4: Supporting Routines

Trusted Platform Module Library

7.4.2.3 ComputeAuthTimeout()

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

```c
UINT64 ComputeAuthTimeout(
    SESSION *session,          // IN: the session containing the time values
    INT32 expiration,           // IN: either the number of seconds from the start of the session or the time in g_timer;
    TPM2B_NONCE *nonce          // IN: indicator of the time base
)
{
    UINT64 policyTime;
    // If no expiration, policy time is 0
    if(expiration == 0)
        policyTime = 0;
    else
    { /* The input time is absolute Time (not Clock), but it is expressed in seconds. To make sure that we don't time out too early, take the current value of milliseconds in g_time and add that to the input seconds value. */
        policyTime = (((UINT64)expiration) * 1000) + g_time % 1000;
        // The policy timeout is the absolute value of the expiration in seconds
        // added to the start time of the policy.
        policyTime = session->startTime + (((UINT64)expiration) * 1000);
    }
    return policyTime;
}
```

7.4.2.4 PolicyDigestClear()

Function to reset the policyDigest of a session

```c
void PolicyDigestClear(
    SESSION *session
)
{
    session->u2.policyDigest.t.size = CryptHashGetDigestSize(session->authHashAlg);
    MemorySet(session->u2.policyDigest.t.buffer, 0,
               session->u2.policyDigest.t.size);
}
```
BOOL PolicySptCheckCondition(
    TPM EO operation,
    BYTE *opA,
    BYTE *opB,
    UINT16 size
)
{
    // Arithmetic Comparison
    switch(operation)
    {
    case TPM EO_EQ:
        // compare A = B
        return (UnsignedCompareB(size, opA, size, opB) == 0);
        break;
    case TPM EO_NEQ:
        // compare A != B
        return (UnsignedCompareB(size, opA, size, opB) != 0);
        break;
    case TPM EO_SIGNED_GT:
        // compare A > B signed
        return (SignedCompareB(size, opA, size, opB) > 0);
        break;
    case TPM EO_UNSIGNED_GT:
        // compare A > B unsigned
        return (UnsignedCompareB(size, opA, size, opB) > 0);
        break;
    case TPM EO_SIGNED_LT:
        // compare A < B signed
        return (SignedCompareB(size, opA, size, opB) < 0);
        break;
    case TPM EO_UNSIGNED_LT:
        // compare A < B unsigned
        return (UnsignedCompareB(size, opA, size, opB) < 0);
        break;
    case TPM EO_SIGNED_GE:
        // compare A >= B signed
        return (SignedCompareB(size, opA, size, opB) >= 0);
        break;
    case TPM EO_UNSIGNED_GE:
        // compare A >= B unsigned
        return (UnsignedCompareB(size, opA, size, opB) >= 0);
        break;
    case TPM EO_SIGNED_LE:
        // compare A <= B signed
        return (SignedCompareB(size, opA, size, opB) <= 0);
        break;
    case TPM EO_UNSIGNED_LE:
        // compare A <= B unsigned
        return (UnsignedCompareB(size, opA, size, opB) <= 0);
        break;
    case TPM EO_BITSET:
        // All bits SET in B are SET in A. ((A&B)=B)
        {
            UINT32 i;
            for(i = 0; i < size; i++)
                if((opA[i] & opB[i]) != opB[i])
                    return FALSE;
        }
        break;
    case TPM EO_BITCLEAR:
        // All bits SET in B are CLEAR in A. ((A&B)=0)
        {
            UINT32 i;
            for(i = 0; i < size; i++)
                if((opA[i] & opB[i]) != 0)
                    return FALSE;
        }
        break;
    }
return FALSE;
break;
default:
    FAIL(FATAL_ERROR_INTERNAL);
break;
} return TRUE;
7.5 NV Command Support (NV_spt.c)

7.5.1 Includes

```c
#include "Tpm.h"
#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()

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_AUTHORIZATION</td>
<td>authHandle is not allowed to authorize read of the index</td>
</tr>
<tr>
<td>TPM_RC_NV_LOCKED</td>
<td>Read locked</td>
</tr>
<tr>
<td>TPM_RC_NV_UNINITIALIZED</td>
<td>Try to read an uninitialized index</td>
</tr>
</tbody>
</table>

```c
TPM_RC
NvReadAccessChecks(
    TPM_HANDLE authHandle, // IN: the handle that provided the
    //    authorization
    TPM_HANDLE nvHandle,   // IN: the handle of the NV index to be read
    TPMA_NV attributes    // IN: the attributes of 'nvHandle'
)
{
    // If data is read locked, returns an error
    if(IS_ATTRIBUTE(attributes, TPMA_NV, READLOCKED))
        return TPM_RC_NV_LOCKED;
    // If the authorization was provided by the owner or platform, then check
    // that the attributes allow the read. If the authorization handle
    // is the same as the index, then the checks were made when the authorization
    // was checked.
    if(authHandle == TPM_RH_OWNER)
    {
        // If Owner provided authorization then ONWERWRITE must be SET
        if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERREAD))
            return TPM_RC_NV_AUTHORIZATION;
    }
    else if(authHandle == TPM_RH_PLATFORM)
    {
        // If Platform provided authorization then PPWRITE must be SET
        if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPREAD))
            return TPM_RC_NV_AUTHORIZATION;
    }
    // If neither Owner nor Platform provided authorization, make sure that it was
    // provided by this index.
    else if(authHandle != nvHandle)
        return TPM_RC_NV_AUTHORIZATION;
    // If the index has not been written, then the value cannot be read
    // NOTE: This has to come after other access checks to make sure that
    // the proper authorization is given to TPM2_NV_ReadLock()
    if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN))
        return TPM_RC_NV_UNINITIALIZED;
    return TPM_RC_SUCCESS;
}
```
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()

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_AUTHORIZATION</td>
<td>Authorization fails</td>
</tr>
<tr>
<td>TPM_RC_NV_LOCKED</td>
<td>Write locked</td>
</tr>
</tbody>
</table>

```c
TPM_RC
NvWriteAccessChecks(
    TPM_HANDLE authHandle,       // IN: the handle that provided the
    TPM_HANDLE nvHandle,         // IN: the handle of the NV index to be written
    TPMA_NV attributes          // IN: the attributes of 'nvHandle'
)
{
    // If data is write locked, returns an error
    if(IS_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED))
        return TPM_RC_NV_LOCKED;
    // If the authorization was provided by the owner or platform, then check
    // that the attributes allow the write. If the authorization handle
    // is the same as the index, then the checks were made when the authorization
    // was checked..
    if(authHandle == TPM_RH_OWNER)
    {
        // If Owner provided authorization then ONWERWRITE must be SET
        if(!IS_ATTRIBUTE(attributes, TPMA_NV, OWNERWRITE))
            return TPM_RC_NV_AUTHORIZATION;
    } else if(authHandle == TPM_RH_PLATFORM)
    {
        // If Platform provided authorization then PPWRITE must be SET
        if(!IS_ATTRIBUTE(attributes, TPMA_NV, PPWRITE))
            return TPM_RC_NV_AUTHORIZATION;
    } // If neither Owner nor Platform provided authorization, make sure that it was
    // provided by this index.
    else if(authHandle != nvHandle)
        return TPM_RC_NV_AUTHORIZATION;
    return TPM_RC_SUCCESS;
}
```

7.5.2.3 NvClearOrderly()

This function is used to cause gp.orderlyState to be cleared to the non-orderly state.

```c
TPM_RC
NvClearOrderly(
    void
)
{
    if(gp.orderlyState < SU_DA_USED_VALUE)
        RETURN_IF_NV_IS_NOT_AVAILABLE;
    g_clearOrderly = TRUE;
    return TPM_RC_SUCCESS;
}
```
7.5.2.4 NvIsPinPassIndex()

Function to check to see if an NV index is a PIN Pass Index

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>is pin pass</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>is not pin pass</td>
</tr>
</tbody>
</table>

```c
BOOL NvIsPinPassIndex(
    TPM_HANDLE index // IN: Handle to check
)
{
    if (HandleGetType(index) == TPM_HT_NV_INDEX)
    {
        NV_INDEX *nvIndex = NvGetIndexInfo(index, NULL);
        return IsNvPinPassIndex(nvIndex->publicArea.attributes);
    }
    return FALSE;
}
```
7.6 Object Command Support (Object_spt.c)

7.6.1 Includes

```c
#include "Tpm.h"
#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 appended to the start of the sensitive data. It includes both size of size field and size of iv data.

```c
static UINT16 GetIV2BSize(OBJECT *protector) // IN: the protector handle
{

    TPM_ALG_ID symAlg;
    UINT16 keyBits;

    // Determine the symmetric algorithm and size of key
    if (protector == NULL) {
        // Use the context encryption algorithm and key size
        symAlg = CONTEXT_ENCRYPT_ALG;
        keyBits = CONTEXT_ENCRYPT_KEY_BITS;
    } else {
        symAlg = protector->publicArea.parameters.asymDetail.symmetric.algorithm;
        keyBits = protector->publicArea.parameters.asymDetail.symmetric.keyBits.sym;
    }

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

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.

```c
static void ComputeProtectionKeyParms(OBJECT *protector, // IN: the protector object
    TPM_ALG_ID hashAlg, // IN: hash algorithm for KDFa
    TPM2B *name, // IN: name of the object
    TPM2B *seedIn, // IN: optional seed for duplication blob.
    TPM_ALG_ID *symAlg,  // OUT: the symmetric algorithm
    UINT16 *keyBits, // OUT: the symmetric key size in bits
    TPM2B_SYM_KEY *symKey // OUT: the symmetric key
)
```
static void ComputeOuterIntegrity(
    TPM2B *name,       // IN: the name of the object
    OBJECT *protector,  // IN: the object that
                        // provides protection. For an object,
                        // it is a parent. For a credential, it
                        // is the encrypt object. For
                        // a Temporary Object, it is NULL
    TPMI_ALG_HASH hashAlg,  // IN: algorithm to use for integrity
    TPM2B *seedIn,      // IN: an external seed may be provided for
                        // duplication blob. For non duplication
                        // blob, this parameter should be NULL
    UINT32 sensitiveSize, // IN: size of the marshaled sensitive data
    BYTE *sensitiveData, // IN: sensitive area
    TPM2B_DIGEST *integrity // OUT: integrity
)
{
    HMAC_STATE hmacState;
    TPM2B_DIGEST hmacKey;
    const TPM2B *seed = seedIn;

    // Get seed for KDF
    if(seed == NULL)
    
    // Determine the HMAC key bits
    hmacKey.t.size = CryptHashGetDigestSize(hashAlg);

    // KDFa to generate HMAC key
7.6.2.4 ComputeInnerIntegrity()

This function computes the integrity of an inner wrap.

```c
static void
ComputeInnerIntegrity(
    TPM_ALG_ID       hashAlg,         // IN: hash algorithm for inner wrap
    TPM2B           *name,            // IN: the name of the object
    UINT16           dataSize,        // IN: the size of sensitive data
    BYTE            *sensitiveData,   // IN: sensitive data
    TPM2B_DIGEST    *integrity)      // OUT: inner integrity
{
    HASH_STATE      hashState;
    // Start hash and get the size of the digest which will become the integrity
    integrity->t.size = CryptHashStart(&hashState, hashAlg);
    // Adding the marshaled sensitive area to the integrity value
    CryptDigestUpdate(&hashState, dataSize, sensitiveData);
    // Adding name
    CryptDigestUpdate2B(&hashState, name);
    // Compute hash
    CryptHashEnd2B(&hashState, &integrity->b);
    return;
}
```

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.

```c
static UINT16
ProduceInnerIntegrity(
    TPM2B           *name,            // IN: the name of the object
    TPM_ALG_ID       hashAlg,         // IN: hash algorithm for inner wrap
    UINT16           dataSize,        // IN: the size of sensitive data, excluding the
    BYTE            *innerBuffer,     // IN/OUT: inner buffer with sensitive data in
                               // leading integrity buffer size
                               // it. At input, the leading bytes of this
```
//     buffer is reserved for integrity

BYTE *sensitiveData; // pointer to the sensitive data
TPM2B_DIGEST integrity;
UINT16 integritySize;
BYTE *buffer;      // Auxiliary buffer pointer

// sensitiveData points to the beginning of sensitive data in innerBuffer
integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
sensitiveData = innerBuffer + integritySize;

ComputeInnerIntegrity(hashAlg, name, dataSize, sensitiveData, &integrity);

// Add integrity at the beginning of inner buffer
buffer = innerBuffer;
TPM2B_DIGEST_Marshal(&integrity, &buffer, NULL);
return dataSize + integritySize;

7.6.2.6 \textbf{CheckInnerIntegrity()}

This function check integrity of inner blob

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_INTEGRITY</td>
<td>if the outer blob integrity is bad</td>
</tr>
<tr>
<td>unmarshal errors</td>
<td>unmarshal errors while unmarshaling integrity</td>
</tr>
</tbody>
</table>

static TPM_RC

CheckInnerIntegrity(

TPM2B *name,     // IN: the name of the object
TPM_ALG_ID hashAlg, // IN: hash algorithm for inner wrap
UINT16 dataSize,  // IN: the size of sensitive data, including the
                 // leading integrity buffer size
BYTE *innerBuffer // IN/OUT: inner buffer with sensitive data in
                     // it
)

TPM_RC result;
TPM2B_DIGEST integrity;
TPM2B_DIGEST integrityToCompare;
BYTE *buffer;      // Auxiliary buffer pointer
INT32 size;

// Unmarshal integrity
buffer = innerBuffer;
size = (INT32)dataSize;
result = TPM2B_DIGEST_Unmarshal(&integrity, &buffer, &size);
if(result == TPM_RC_SUCCESS)
{
    // Compute integrity to compare
    ComputeInnerIntegrity(hashAlg, name, (UINT16)size, buffer,
                           &integrityToCompare);
    // Compare outer blob integrity
    if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
    {
        result = TPM_RC_INTEGRITY;
    }
}
return result;
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.

```c
BOOL AdjustAuthSize(
    TPM2B_AUTH *auth,       // IN/OUT: value to adjust
    TPMI_ALG_HASH nameAlg   // IN:
)
{
    UINT16 digestSize;
    // If there is no nameAlg, then this is a LoadExternal and the authVale can
    // be any size up to the maximum allowed by the
    digestSize = (nameAlg == TPM_ALG_NULL) ? sizeof(TPMU_HA)
        : CryptHashGetDigestSize(nameAlg);
    if (digestSize < MemoryRemoveTrailingZeros(auth))
        return FALSE;
    else if (digestSize > auth->t.size)
        MemoryPad2B(&auth->t.b, digestSize);
    auth->t.size = digestSize;
    return TRUE;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>properties are those of a parent</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>properties are not those of a parent</td>
</tr>
</tbody>
</table>

```c
BOOL ObjectIsParent(
    OBJECT *parentObject  // IN: parent handle
)
{
    return parentObject->attributes.isParent;
}
```

7.6.3.3 CreateChecks()

Attribute checks that are unique to creation.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>sensitiveDataOrigin is not consistent with the object type</td>
</tr>
<tr>
<td>other</td>
<td>returns from PublicAttributesValidation()</td>
</tr>
</tbody>
</table>

```c
TPM_RC CreateChecks(
```
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.
<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_HASH</td>
<td>non-duplicable storage key and its parent have different name algorithm</td>
</tr>
<tr>
<td>TPM_RC_KDF</td>
<td>incorrect KDF specified for decrypting keyed hash object</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>invalid key size values in an asymmetric key public area</td>
</tr>
<tr>
<td>TPM_RCS_SCHEME</td>
<td>inconsistent attributes decrypt, sign, restricted and key's scheme ID; or hash algorithm is inconsistent with the scheme ID for keyed hash object</td>
</tr>
<tr>
<td>TPM_RC_SYMMETRIC</td>
<td>a storage key with no symmetric algorithm specified; or non-storage key with symmetric algorithm different from ALG_NULL</td>
</tr>
</tbody>
</table>

```
TPM_RC
SchemeChecks(
    OBJECT *parentObject,  // IN: parent (null if primary seed)
    TPM_PUBLIC *publicArea  // IN: public area of the object
)
{
    TPMT_SYM_DEF_OBJECT *symAlgs = NULL;
    TPM_ALG_ID            scheme = TPM_ALG_NULL;
    TPMA_OBJECT           attributes = publicArea->objectAttributes;
    TPMU_PUBLIC_PARMS     *parms = &publicArea->parameters;

    switch(publicArea->type)
    {
    case ALG_SYMCIPHER_VALUE:
        symAlgs = &parms->symDetail.sym;
        // If this is a decrypt key, then only the block cipher modes (not
        // SMAC) are valid. TPM_ALG_NULL is OK too. If this is a 'sign' key,
        // then any mode that got through the unmarshaling is OK.
        if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt)
            && !CryptSymModeIsValid(symAlgs->mode.sym, TRUE))
            return TPM_RCS_SCHEME;
        break;

    case ALG_KEYEDHASH_VALUE:
        scheme = parms->keyedHashDetail.scheme.scheme;
        // if both sign and decrypt
        if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
            == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
        {
            // if both sign and decrypt are set or clear, then need
            // ALG_NULL as scheme
            if(scheme != TPM_ALG_NULL)
                return TPM_RCS_SCHEME;
        }
        else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
            && scheme != TPM_ALG_HMAC)
            return TPM_RCS_SCHEME;
        else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
        {
            if(scheme != TPM_ALG_XOR)
                return TPM_RCS_SCHEME;
            // if this is a derivation parent, then the KDF needs to be
            // SP800-108 for this implementation. This is the only derivation
            // supported by this implementation. Other implementations could
            // support additional schemes. There is no default.
            if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
            {
                if(parms->keyedHashDetail.scheme.details.xor.kdf
                    != TPM_ALG_KDF1_SP800_108)
                    return TPM_RCS_SCHEME;
            // Must select a digest.
```
if(CryptHashGetDigestSize(
    parms->keyedHashDetail.scheme.details.xor.hashAlg) == 0)
    return TPM_RCS_HASH;
}
break;
default: // handling for asymmetric
    scheme = parms->asymDetail.scheme.scheme;
symAlgs = &parms->asymDetail.symmetric;
    // if the key is both sign and decrypt, then the scheme must be
    // ALG_NULL because there is no way to specify both a sign and a
    // decrypt scheme in the key.
    if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
        == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
        { // scheme must be TPM_ALG_NULL
            if(scheme != TPM_ALG_NULL)
                return TPM_RCS_SCHEME;
        } else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
        { // If this is a signing key, see if it has a signing scheme
            if(CryptIsAsymSignScheme(publicArea->type, scheme))
                { // if proper signing scheme then it needs a proper hash
                    if(parms->asymDetail.scheme.details.anySig.hashAlg
                        == TPM_ALG_NULL)
                        return TPM_RCS_SCHEME;
                } else
                    { // signing key that does not have a proper signing scheme.
                        // This is OK if the key is not restricted and its scheme
                        // is TPM_ALG_NULL
                        if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
                            || scheme != TPM_ALG_NULL)
                            return TPM_RCS_SCHEME;
                    }
        } else if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
        { // for a restricted decryption key (a parent), scheme
            // is required to be TPM_ALG_NULL
            if(scheme != TPM_ALG_NULL)
                return TPM_RCS_SCHEME;
        } else
        { // for an unrestricted decryption key, the scheme has to
            // be a valid scheme or TPM_ALG_NULL
            if(scheme != TPM_ALG_NULL &&
                !CryptIsAsymDecryptScheme(publicArea->type, scheme))
                return TPM_RCS_SCHEME;
        }
    } else if(!IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
        { // For an asymmetric key that is not a parent, the symmetric
            // algorithms must be TPM_ALG_NULL
            if(symAlgs->algorithm != TPM_ALG_NULL)
                return TPM_RCS_SYMMETRIC;
        } // Special checks for an ECC key
#if ALG_ECC
  if(publicArea->type == TPM_ALG_ECC)
  {
    TPM_ECC_CURVE curveID;
    const TPMT_ECC_SCHEME *curveScheme;

    curveID = publicArea->parameters.eccDetail.curveID;
    curveScheme = CryptGetCurveSignScheme(curveID);
    // The curveId must be valid or the unmarshaling is busted.
    pAssert(curveScheme != NULL);

    // If the curveID requires a specific scheme, then the key must
    // select the same scheme
    if(curveScheme->scheme != TPM_ALG_NULL)
    {
      TPMS_ECC_PARMS *ecc = &publicArea->parameters.eccDetail;
      if(ecc->scheme != curveScheme->scheme)
      {
        return TPM_RCS_SCHEME;
      }

      // The scheme can allow any hash, or not...
      if(curveScheme->details.anySig.hashAlg != TPM_ALG_NULL
         && (ecc->scheme.details.anySig.hashAlg
             != curveScheme->details.anySig.hashAlg))
      {
        return TPM_RCS_SCHEME;
      }

      // For now, the KDF must be TPM ALG NULL
    }

    if(publicArea->parameters.eccDetail.kdf.scheme != TPM_ALG_NULL)
    {
      return TPM_RCS_KDF;
    }
  }

  #endif

  break;

  // If this is a restricted decryption key with symmetric algorithms, then it
  // is an ordinary parent (not a derivation parent). It needs to specific
  // symmetric algorithms other than TPM_ALG_NULL
  if(symAlgs != NULL
    && IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted)
    && IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
  {
    if(symAlgs->algorithm == TPM_ALG_NULL)
    {
      return TPM_RCS_SYMMETRIC;
    }

    #if 0
    // This next check is under investigation. Need to see if it will break Windows
    // before it is enabled. If it does not, then it should be default because a
    // the mode used with a parent is always CFB and Part 2 indicates as much.
    if(symAlgs->mode.sym != TPM_ALG_CFB)
    {
      return TPM_RCS_MODE;
    }
    #endif

    // If this parent is not duplicable, then the symmetric algorithms
    // (encryption and hash) must match those of its parent
    if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
      && (parentObject != NULL))
    {
      if(publicArea->nameAlg != parentObject->publicArea.nameAlg)
      {
        return TPM_RCS_HASH;
      }

      if(!MemoryEqual(symAlgs, &parentObject->publicArea.parameters,
                       sizeof(TPMT_SYM_DEF_OBJECT)))
      {
        return TPM_RCS_SYMMETRIC;
      }
    } else
    {
      return TPM_RC_SUCCESS;
    }
}
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>fixedTPM, fixedParent, or encryptedDuplication attributes are inconsistent between themselves or with those of the parent object; inconsistent restricted, decrypt and sign attributes; attempt to inject sensitive data for an asymmetric key; attempt to create a symmetric cipher key that is not a decryption key</td>
</tr>
<tr>
<td>TPM_RC_HASH</td>
<td>nameAlg is TPM_ALG_NULL</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>authPolicy size does not match digest size of the name algorithm in publicArea</td>
</tr>
<tr>
<td>other</td>
<td>returns from SchemeChecks()</td>
</tr>
</tbody>
</table>

```
TPM_RC PublicAttributesValidation(
    OBJECT *parentObject, // IN: input parent object
    TPM_PUBLIC *publicArea // IN: public area of the object
)
{
    TPMA_OBJECT attributes = publicArea->objectAttributes;
    TPMA_OBJECT parentAttributes = {0};

    // if(parentObject != NULL)
    parentAttributes = parentObject->publicArea.objectAttributes;

    if(publicArea->nameAlg == TPM_ALG_NULL)
        return TPM_RC_HASH;

    // If there is an authPolicy, it needs to be the size of the digest produced
    // by the nameAlg of the object
    if((publicArea->authPolicy.t.size != 0
        && (publicArea->authPolicy.t.size
            != CryptHashGetDigestSize(publicArea->nameAlg)))))
        return TPM_RC_SIZE;

    // If the parent is fixedTPM (including a Primary Object) the object must have
    // the same value for fixedTPM and fixedParent
    if(parentObject == NULL
        || IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
    {
        if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent)
            != IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
            return TPM_RC_ATTRIBUTES;
    }

    else
    {
        // The parent is not fixedTPM so the object can't be fixedTPM
        if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
            return TPM_RC_ATTRIBUTES;
    }

    // See if sign and decrypt are the same
    if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
        == IS_ATTRIBUTE(attributes, TPMA_OBJECT, decrypt))
    {
        // a restricted key cannot have both SET or both CLEAR
        if(IS_ATTRIBUTE(attributes, TPMA_OBJECT, restricted))
            return TPM_RC_ATTRIBUTES;

        // only a data object may have both sign and decrypt CLEAR
        // BTW, since we know that decrypt==sign, no need to check both
    }
```
if (publicArea->type != TPM_ALG_KEYEDHASH
        && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign))
    return TPM_RC_ATTRIBUTES;
}

// If the object can't be duplicated (directly or indirectly) then there
// is no justification for having encryptedDuplication SET
if (IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
    && IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication))
    return TPM_RCS_ATTRIBUTES;

// If a parent object has fixedTPM CLEAR, the child must have the
// same encryptedDuplication value as its parent.
// Primary objects are considered to have a fixedTPM parent (the seeds).
if (parentObject != NULL
    && !IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
{
    if (IS_ATTRIBUTE(attributes, TPMA_OBJECT, encryptedDuplication)
        != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, encryptedDuplication))
        return TPM_RCS_ATTRIBUTES;
}

// Special checks for derived objects
if ((parentObject != NULL) && (parentObject->attributes.derivation == SET))
{
    // A derived object has the same settings for fixedTPM as its parent
    if (IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM)
        != IS_ATTRIBUTE(parentAttributes, TPMA_OBJECT, fixedTPM))
        return TPM_RCS_ATTRIBUTES;

    // A derived object is required to be fixedParent
    if (!IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedParent))
        return TPM_RCS_ATTRIBUTES;

    return SchemeChecks(parentObject, publicArea);
}

7.6.3.6 FillInCreationData()

Fill in creation data for an object.

void FillInCreationData(
    TPMI_DH_OBJECT parentHandle, // IN: handle of parent
    TPMI_ALG_HASH nameHashAlg, // IN: name hash algorithm
    TPML_PCR_SELECTION *creationPCR, // IN: PCR selection
    TPM2B_DATA *outsideData, // IN: outside data
    TPM2B_CREATION_DATA *outCreation,
    // OUT: creation data for output
    TPM2B_DIGEST *creationDigest // OUT: creation digest
    )
{
    BYTE creationBuffer[sizeof(TPMS_CREATION_DATA)];
    BYTE *buffer;
    HASH_STATE hashState;

    // Fill in TPMS_CREATION_DATA in outCreation

    // Compute PCR digest
    PCRComputeCurrentDigest(nameHashAlg, creationPCR,
        &outCreation->creationData.pcrDigest);

    // Put back PCR selection list
    outCreation->creationData.pcrSelect = *creationPCR;

    // Get locality
    outCreation->creationData.locality
        = LocalityGetAttributes(_plat__LocalityGet());
    outCreation->creationData.parentNameAlg = TPM_ALG_NULL;
// If the parent is either a primary seed or TPM_ALG_NULL, then the Name
// and QN of the parent are the parent's handle.
if (HandleGetType(parentHandle) == TPM_HT_PERMANENT)
{
    buffer = &outCreation->creationData.parentName.t.name[0];
    outCreation->creationData.parentName.t.size =
        TPM_HANDLE_Marshal(&parentHandle, &buffer, NULL);
    // For a primary or temporary object, the parent name (a handle) and the
    // parent's QN are the same
    outCreation->creationData.parentQualifiedName
        = outCreation->creationData.parentName;
}
else // Regular object
{
    OBJECT *parentObject = HandleToObject(parentHandle);
    // Set name algorithm
    outCreation->creationData.parentNameAlg = parentObject->publicArea.nameAlg;
    // Copy parent name
    outCreation->creationData.parentName = parentObject->name;
    // Copy parent qualified name
    outCreation->creationData.parentQualifiedName = parentObject->qualifiedName;
}
// Copy outside information
outCreation->creationData.outsideInfo = *outsideData;

// Marshal creation data to canonical form
buffer = creationBuffer;
outCreation->size = TPMS_CREATION_DATA_Marshal(&outCreation->creationData,
                                                &buffer, NULL);
// Compute hash for creation field in public template
creationDigest.t.size = CryptHashStart(&hashState, nameHashAlg);
CryptDigestUpdate(&hashState, outCreation->size, creationBuffer);
CryptHashEnd2B(&hashState, &creationDigest->b);
return;

7.6.3.7 GetSeedForKDF()

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

const TPM2B *
GetSeedForKDF(
    OBJECT *protector /* IN: the protector handle */
)
{
    // Get seed for encryption key. Use input seed if provided.
    // Otherwise, using protector object's seedValue. TPM_RH_NULL is the only
    // exception that we may not have a loaded object as protector. In such a
    // case, use nullProof as seed.
    if (protector == NULL)
    {
        return &gr.nullProof.b;
    }
    else
    {
        return &protector->sensitive.seedValue.b;
    }
}
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 ProduceOuterWrap(
611     OBJECT *protector, // IN: The handle of the object that provides
612     // protection. For object, it is parent
613     // handle. For credential, it is the handle
614     // of encrypt object.
615     TPM2B *name, // IN: the name of the object
616     TPM_ALG_ID hashAlg, // IN: hash algorithm for outer wrap
617     TPM2B *seed, // IN: an external seed may be provided for
618     // duplication blob. For non duplication
619     // blob, this parameter should be NULL
620     BOOL useIV, // IN: indicate if an IV is used
621     UINT16 dataSize, // IN: the size of sensitive data, excluding the
622     // leading integrity buffer size or the
623     // optional iv size
624     BYTE *outerBuffer // IN/OUT: outer buffer with sensitive data in
625     // it
626 )
627 {
628     TPM_ALG_ID symAlg;
629     UINT16 keyBits;
630     TPM2B_SYM_KEY symKey;
631     TPM2B_IV ivRNG; // IV from RNG
632     TPM2B_IV *iv = NULL;
633     UINT16 ivSize = 0; // size of iv area, including the size field
634     BYTE *sensitiveData; // pointer to the sensitive data
635     TPM2B_DIGEST integrity;
636     UINT16 integritySize;
637     BYTE *buffer; // Auxiliary buffer pointer
638     // Compute the beginning of sensitive data. The outer integrity should
639     // always exist if this function is called to make an outer wrap
640     integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
641     sensitiveData = outerBuffer + integritySize;
642     // If iv is used, adjust the pointer of sensitive data and add iv before it
643     if(useIV)
644     {
645         ivSize = GetIV2BSize(protector);
646         // Generate IV from RNG. The iv data size should be the total IV area
647         // size minus the size of size field
648         ivRNG.t.size = ivSize - sizeof(UINT16);
649         CryptRandomGenerate(ivRNG.t.size, ivRNG.t.buffer);
650         // Marshal IV to buffer
651         buffer = sensitiveData;
652         TPM2B_IV.Marshal(&ivRNG, &buffer, NULL);
653         // adjust sensitive data starting after IV area
654         sensitiveData += ivSize;
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   // return the total size in outer wrap
680   return dataSize + integritySize + ivSize;
681 }

7.6.3.9 UnwrapOuter()

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

a) check integrity of outer blob

b) decrypt outer blob

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RCS_INSUFFICIENT</td>
<td>error during sensitive data unmarshaling</td>
</tr>
<tr>
<td>TPM_RCS_INTEGRITY</td>
<td>sensitive data integrity is broken</td>
</tr>
<tr>
<td>TPM_RCS_SIZE</td>
<td>error during sensitive data unmarshaling</td>
</tr>
<tr>
<td>TPM_RCS_VALUE</td>
<td>IV size for CFB does not match the encryption algorithm block size</td>
</tr>
</tbody>
</table>

683   TPM_RC
684   UnwrapOuter(OBJECT *protector, // IN: The object that provides
685       // protection. For object, it is parent
686       // handle. For credential, it is the
687       // encrypt object.
688       TPM2B *name, // IN: the name of the object
689       TPM_ALG_ID hashAlg, // IN: hash algorithm for outer wrap
690       TPM2B *seed, // IN: an external seed may be provided for
691       // duplication blob. For non duplication
692       // blob, this parameter should be NULL.
693       BOOL useIV, // IN: indicates if an IV is used
694       UINT16 dataSize, // IN: size of sensitive data in outerBuffer,
695       // including the leading integrity buffer
696       // size, and an optional iv area
697       BYTE *outerBuffer // IN/OUT: sensitive data
698   )
699 {
700   TPM_RC result;
701   TPM_ALG_ID symAlg = TPM_ALG_NULL;
702   TPM2B_SYM_KEY symKey;
703   UINT16 keyBits = 0;
704   TPM2B_IV ivIn; // input IV retrieved from input buffer
705   TPM2B_IV *iv = NULL;
706   BYTE *sensitiveData; // pointer to the sensitive data
TPM2B_DIGEST integrityToCompare;
TPM2B_DIGEST integrity;
INT32 size;

// Unmarshal integrity
sensitiveData = outerBuffer;
size = (INT32)dataSize;
result = TPM2B_DIGEST_Unmarshal(&integrity, &sensitiveData, &size);
if(result == TPM_RC_SUCCESS)
{
    // Compute integrity to compare
    ComputeOuterIntegrity(name, protector, hashAlg, seed,
        (UINT16)size, sensitiveData,
        &integrityToCompare);
    // Compare outer blob integrity
    if(!MemoryEqual2B(&integrity.b, &integrityToCompare.b))
        return TPM_RCS_INTEGRITY;
    // Get the symmetric algorithm parameters used for encryption
    ComputeProtectionKeyParms(protector, hashAlg, name, seed,
        &symAlg, &keyBits, &symKey);
    // Retrieve IV if it is used
    if(useIV)
    {  
        result = TPM2B_IV_Unmarshal(&ivIn, &sensitiveData, &size);
        if(result == TPM_RC_SUCCESS)
        {
            // The input iv size for CFB must match the encryption algorithm
            // block size
            if(ivIn.t.size != CryptGetSymmetricBlockSize(symAlg, keyBits))
                result = TPM_RC_VALUE;
            else
                iv = &ivIn;
        }
    }
    // If no errors, decrypt private in place. Since this function uses CFB,
    // CryptSymmetricDecrypt() will not return any errors. It may fail but it will
    // not return an error.
    if(result == TPM_RC_SUCCESS)
        CryptSymmetricDecrypt(sensitiveData, symAlg, keyBits,
            symKey.t.buffer, iv, TPM_ALG_CFB,
            (UINT16)size, sensitiveData);
    return result;
}

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.

static UINT16 MarshalSensitive(OBJECT *parent, // IN: the object parent (optional)
 BYTE *buffer, // OUT: receiving buffer
 TPMT_SENSITIVE *sensitive, // IN: the sensitive area to marshal
 TPMI_ALG_HASH nameAlg // IN:
 )
{
    BYTE *sizeField = buffer; // saved so that size can be
    // marshaled after it is known
    UINT16 retVal;
    // Pad the authValue if needed
MemoryPad2B(&sensitive->authValue.b, CryptHashGetDigestSize(nameAlg));
buffer += 2;

// Marshal the structure
#if ALG_RSA
// If the sensitive size is the special case for a prime in the type
if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) > 0)
{
    UINT16 sizeSave = sensitive->sensitive.rsa.t.size;
    // Turn off the flag that indicates that the sensitive->sensitive contains
    // the CRT form of the exponent.
    sensitive->sensitive.rsa.t.size &= ~(RSA_prime_flag);
    // If the parent isn’t fixedTPM, then truncate the sensitive data to be
    // the size of the prime. Otherwise, leave it at the current size which
    // is the full CRT size.
    if(parent == NULL || !IS_ATTRIBUTE(parent->publicArea.objectAttributes,
        TPMA_OBJECT, fixedTPM))
    {
        sensitive->sensitive.rsa.t.size /= 5;
        // Restore the flag and the size.
        sensitive->sensitive.rsa.t.size = sizeSave;
    }
    else
    #endif
    retVal = TPMT_SENSITIVE_Marshal(sensitive, &buffer, NULL);
    // Marshal the size
    retVal = (UINT16)(retVal + UINT16.Marshal(&retVal, &sizeField, NULL));
}
return retVal;
#endif

7.6.3.11 SensitiveToPrivate()

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

a) marshal TPM2B_SENSITIVE structure into the buffer of TPM2B_PRIVATE
b) apply encryption to the sensitive area.
c) apply outer integrity computation.

void SensitiveToPrivate(
TPMT_SENSITIVE *sensitive, // IN: sensitive structure
TPM2B_NAME *name, // IN: the name of the object
OBJECT *parent, // IN: The parent object
TPM_ALG_ID nameAlg, // IN: hash algorithm in public area. This
                    // parameter is used when parentHandle is
                    // NULL, in which case the object is
                    // temporary.
TPM2B_PRIVATE *outPrivate // OUT: output private structure
)
{
BYTE *sensitiveData; // pointer to the sensitive data
UINT16 dataSize; // data blob size
TPMI_ALG_HASH hashAlg; // hash algorithm for integrity
UINT16 integritySize;
UINT16 ivSize;
// pAssert(name != NULL && name->t.size != 0);
// Find the hash algorithm for integrity computation
if(parent == NULL)
{
    // For Temporary Object, using self name algorithm
    hashAlg = nameAlg;
}
else
{
    // Otherwise, using parent's name algorithm
    hashAlg = parent->publicArea.nameAlg;
}
// Starting of sensitive data without wrappers
sensitiveData = outPrivate->t.buffer;
// Compute the integrity size
integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);
// Reserve space for integrity
sensitiveData += integritySize;
// Get iv size
ivSize = GetIV2BSize(parent);
// Reserve space for iv
sensitiveData += ivSize;
// Marshal the sensitive area including authValue size adjustments.
dataSize = MarshalSensitive(parent, sensitiveData, sensitive, nameAlg);
//Produce outer wrap, including encryption and HMAC
outPrivate->t.size = ProduceOuterWrap(parent, &name->b, hashAlg, NULL,
TRUE, dataSize, outPrivate->t.buffer);
return;
}

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:

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RCS_INTEGRITY</td>
<td>if the private area integrity is bad</td>
</tr>
<tr>
<td>TPM_RCSENSITIVE</td>
<td>unmarshal errors while unmarshaling TPMS_ENCRYPT from input private</td>
</tr>
<tr>
<td>TPM_RCS_SIZE</td>
<td>error during sensitive data unmarshaling</td>
</tr>
<tr>
<td>TPM_RCS_VALUE</td>
<td>outer wrapper does not have an IV of the correct size</td>
</tr>
</tbody>
</table>

TPM_RC

PrivateToSensitive(

TPM2B *inPrivate,   // IN: input private structure
TPM2B *name,       // IN: the name of the object
OBJECT *parent,    // IN: parent object
TPM_ALG_ID nameAlg, // IN: hash algorithm in public area. It is passed separately because we only pass
// name, rather than the whole public area
// of the object. This parameter is used in the following two cases: 1. primary
7.6.3.13 SensitiveToDuplicate()

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

```c
TPMT_SENSITIVE *sensitive  // OUT: sensitive structure
{
    TPM_RC result;
    BYTE *buffer;
    INT32 size;
    BYTE *sensitiveData; // pointer to the sensitive data
    UINT16 dataSize;
    UINT16 dataSizeInput;
    TPMI_ALG_HASH hashAlg;  // hash algorithm for integrity
    UINT16 integritySize;
    UINT16 ivSize;

    // Make sure that name is provided
    pAssert(name != NULL && name->size != 0);

    // Find the hash algorithm for integrity computation
    // For Temporary Object (parent == NULL) use self name algorithm;
    // Otherwise, using parent's name algorithm
    hashAlg = (parent == NULL) ? nameAlg : parent->publicArea.nameAlg;

    // unwarp outer
    result = UnwrapOuter(parent, name, hashAlg, NULL, TRUE,
                          inPrivate->size, inPrivate->buffer);
    if(result != TPM_RC_SUCCESS)
        return result;

    // Compute the inner integrity size.
    integritySize = sizeof(UINT16) + CryptHashGetDigestSize(hashAlg);

    // Get iv size
    ivSize = GetIV2BSize(parent);

    // The starting of sensitive data and data size without outer wrapper
    sensitiveData = inPrivate->buffer + integritySize + ivSize;
    dataSize = inPrivate->size - integritySize - ivSize;

    // Unmarshal input data size
    buffer = sensitiveData;
    size = (INT32)dataSize;
    result = UINT16_Unmarshal(&dataSizeInput, &buffer, &size);
    if(result == TPM_RC_SUCCESS)
    {
        if((dataSizeInput + sizeof(UINT16)) != dataSize)
            result = TPM_RC_SENSITIVE;
        else
            {
                // Unmarshal sensitive buffer to sensitive structure
                result = TPMT_SENSITIVE_Unmarshal(sensitive, &buffer, &size);
                if(result != TPM_RC_SUCCESS || size != 0)
                    result = TPM_RC_SENSITIVE;
            }
    }
    return result;
}
```
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

```c
void
SensitiveToDuplicate(  
    TPMT_SENSITIVE *sensitive, // IN: sensitive structure
    TPM2B *name, // IN: the name of the object
    OBJECT *parent, // IN: The new parent object
    TPM_ALG_ID nameAlg, // IN: hash algorithm in public area. It
    // is passed separately because we
    // only pass name, rather than the
    // whole public area of the object.
    TPM2B *seed, // IN: the external seed. If external
    // seed is provided with size of 0,
    // no outer wrap should be applied
    // to duplication blob.
    TPMT_SYM_DEF_OBJECT *symDef, // IN: Symmetric key definition. If the
    // symmetric key algorithm is NULL,
    // no inner wrap should be applied.
    TPM2B_DATA *innerSymKey, // IN/OUT: a symmetric key may be
    // provided to encrypt the inner
    // wrap of a duplication blob. May
    // be generated here if needed.
    TPM2B_PRIVATE *outPrivate // OUT: output private structure
)
{
    BYTE *sensitiveData; // pointer to the sensitive data
    TPM_ALG_HASH outerHash = TPM_ALG_NULL; // The hash algorithm for outer wrap
    TPM_ALG_HASH innerHash = TPM_ALG_NULL; // The hash algorithm for inner wrap
    UINT16 dataSize; // data blob size
    BOOL doInnerWrap = FALSE;
    BOOL doOuterWrap = FALSE;

    // Make sure that name is provided
    pAssert(name != NULL && name->size != 0);

    // Make sure symDef and innerSymKey are not NULL
    pAssert(symDef != NULL && innerSymKey != NULL);

    // Starting of sensitive data without wrappers
    sensitiveData = outPrivate->t.buffer;

    // Find out if inner wrap is required
    if(symDef->algorithm != TPM_ALG_NULL)
    {
        doInnerWrap = TRUE;

        // Use self nameAlg as inner hash algorithm
        innerHash = nameAlg;

        // Adjust sensitive data pointer
        sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(innerHash);
    }

    // Find out if outer wrap is required
    if(seed->size != 0)
    {
        doOuterWrap = TRUE;

        // Use parent nameAlg as outer hash algorithm
        outerHash = parent->publicArea.nameAlg;

        // Adjust sensitive data pointer
        sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
    }
```

Trusted Platform Module Library

Part 4: Supporting Routines

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_INSUFFICIENT</td>
<td>unmarshaling sensitive data from <code>inPrivate</code> failed</td>
</tr>
<tr>
<td>TPM_RC_INTEGRITY</td>
<td><code>inPrivate</code> data integrity is broken</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>unmarshaling sensitive data from <code>inPrivate</code> failed</td>
</tr>
</tbody>
</table>

```c
TPM_RC
DuplicateToSensitive(
    TPM2B *inPrivate,  // IN: input private structure
    TPM2B *name,      // IN: the name of the object
    OBJECT *parent,   // IN: the parent
    TPM_ALG_ID nameAlg, // IN: hash algorithm in public area.
    TPM2B *seed,      // IN: an external seed may be provided.
    //     If external seed is provided with
    //     size of 0, no outer wrap is
    //     applied
    TPMT_SYM_DEF_OBJECT *symDef,  // IN: Symmetric key definition. If the
    //     symmetric key algorithm is NULL,
    //     no inner wrap is applied
    TPM2B *innerSymKey, // IN: a symmetric key may be provided
    //     to decrypt the inner wrap of a
    //     duplication blob.
    TPMT_SENSITIVE *sensitive // OUT: sensitive structure
) {
    TPM_RC result;
    BYTE *buffer;
    INT32 size;
    BYTE *sensitiveData; // pointer to the sensitive data
    UINT16 dataSize;
    UINT16 dataSizeInput;
    // Make sure that name is provided
    pAssert(name != NULL && name->size != 0);
    // Make sure symDef and innerSymKey are not NULL
    pAssert(symDef != NULL && innerSymKey != NULL);
    // Starting of sensitive data
    sensitiveData = inPrivate->buffer;
    dataSize = inPrivate->size;
    // Find out if outer wrap is applied
    if(seed->size != 0) {
        // Use parent nameAlg as outer hash algorithm
        TPMI_ALG_HASH outerHash = parent->publicArea.nameAlg;
        result = UnwrapOuter(parent, name, outerHash, seed, FALSE,
                              dataSize, sensitiveData);
        if(result != TPM_RC_SUCCESS)
            return result;
        // Adjust sensitive data pointer and size
        sensitiveData += sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
        dataSize -= sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
    }
    // Find out if inner wrap is applied
    if(symDef->algorithm != TPM_ALG_NULL)
```
7.6.3.15 SecretToCredential()

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

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

```c
void SecretToCredential(
    TPM2B_DIGEST        *secret,  // IN: secret information
    TPM2B               *name,   // IN: the name of the object
    TPM2B               *seed,   // IN: an external seed.
    OBJECT              *protector, // IN: the protector
    TPM2B_ID_OBJECT     *outIDObject // OUT: output credential
)
```

```c
    BYTE                *buffer;  // Auxiliary buffer pointer
    BYTE                *sensitiveData; // pointer to the sensitive data
    TPMI_ALG_HASH       outerHash; // The hash algorithm for outer wrap
    UINT16               dataSize; // data blob size
    // pAssert(secret != NULL && outIDObject != NULL);
```
1137 // use protector's name algorithm as outer hash ???
1138 outerHash = protector->publicArea.nameAlg;
1139
1140 // Marshal secret area to credential buffer, leave space for integrity
1141 sensitiveData = outIDObject->t.credential
1142 + sizeof(UINT16) + CryptHashGetDigestSize(outerHash);
1143 // Marshal secret area
1144 buffer = sensitiveData;
1145 dataSize = TPM2B_DIGEST.Marshal(secret, &buffer, NULL);
1147 // Apply outer wrap
1148 outIDObject->t.size = ProduceOuterWrap(protector, name, outerHash, seed, FALSE,
1149 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:
a) check the integrity HMAC of the input credential area
b) decrypt the credential buffer
c) unmarshal TPM2B_DIGEST structure into the buffer of TPM2B_DIGEST

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_INSUFFICIENT</td>
<td>error during credential unmarshaling</td>
</tr>
<tr>
<td>TPM_RC_INTEGRITY</td>
<td>credential integrity is broken</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>error during credential unmarshaling</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>IV size does not match the encryption algorithm block size</td>
</tr>
</tbody>
</table>

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  // 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
// Unmarshal secret buffer to TPM2B_DIGEST structure
buffer = sensitiveData;
size = (INT32)dataSize;
result = TPM2B_DIGEST_Unmarshal(secret, &buffer, &size);

// If there were no other unmarshaling errors, make sure that the
// expected amount of data was recovered
if(result == TPM_RC_SUCCESS && size != 0)
    return TPM_RC_SIZE;
else
    return result;
}]

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.

```c
UINT16 MemoryRemoveTrailingZeros(TPM2B_AUTH *auth) {  // IN/OUT: value to adjust
    while((auth->t.size > 0) && (auth->t.buffer[auth->t.size - 1] == 0))
        auth->t.size--;
    return auth->t.size;
}
```

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.

```c
TPM_RC SetLabelAndContext(TPMS_DERIVE *labelContext, // IN/OUT: the recovered label and
                          TPMS_SENSITIVE_DATA *sensitive // IN: the sensitive data
) {
    TPMS_DERIVE sensitiveValue;
    TPM_RC result;
    INT32 size;
    BYTE *buff;
    // Unmarshal a TPMS_DERIVE from the TPM2B_SENSITIVE_DATA buffer
    // If there is something to unmarshal...
    if(sensitive->t.size != 0)
        {
            size = sensitive->t.size;
            buff = sensitive->t.buffer;
            result = TPMS_DERIVE_Unmarshal(&sensitiveValue, &buff, &size);
            if(result != TPM_RC_SUCCESS)
                return result;
            // If there was a label in the public area leave it there, otherwise, copy
            // the new value
            if(labelContext->label.t.size == 0)
                MemoryCopy2B(&labelContext->label.b, &sensitiveValue.label.b,
                             sizeof(labelContext->label.t.buffer));
            // if there was a context string in publicArea, it overrides
```
if(labelContext->context.t.size == 0)
        MemoryCopy2B(&labelContext->context.b, &sensitiveValue.context.b,
                     sizeof(labelContext->label.t.buffer));
    }
return TPM_RC_SUCCESS;
}

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.

TPM_RC
UnmarshalToPublic(
    TPMT_PUBLIC       *tOut, // OUT: output
    TPM2B_TEMPLATE    *tIn,  // IN:
    BOOL              derivation, // IN: indicates if this is for a derivation
    TPMS_DERIVE       *labelContext// OUT: label and context if derivation
) {
    BYTE                *buffer = tIn->t.buffer;
    INT32                size = tIn->t.size;
    TPM_RC               result;
    // make sure that tOut is zeroed so that there are no remnants from previous
    // uses
    MemorySet(tOut, 0, sizeof(TPM_PUBLIC));
    // Unmarshal the components of the TPMT_PUBLIC up to the unique field
    result = TPMI_ALG_PUBLIC_Unmarshal(&tOut->type, &buffer, &size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMI_ALG_HASH_Unmarshal(&tOut->nameAlg, &buffer, &size, FALSE);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMA_OBJECT_Unmarshal(&tOut->objectAttributes, &buffer, &size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPM2B_DIGEST_Unmarshal(&tOut->authPolicy, &buffer, &size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMU_PUBLIC_PARMS_Unmarshal(&tOut->parameters, &buffer, &size, 
                                          tOut->type);
    if(result != TPM_RC_SUCCESS)
        return result;
    // Now unmarshal a TPMS_DERIVE if this is for derivation
    if(derivation)
        result = TPMS_DERIVE_Unmarshal(labelContext, &buffer, &size);
    else
        // otherwise, unmarshal a TPMU_PUBLIC_ID
        result = TPMU_PUBLIC_ID_Unmarshal(&tOut->unique, &buffer, &size, 
                                           tOut->type);
    // Make sure the template was used up
    if((result == TPM_RC_SUCCESS) && (size != 0))
        result = TPM_RC_SIZE;
    return result;
}

7.6.3.20 ObjectSetExternal()

Set the external attributes for an object.
void ObjectSetExternal(
  OBJET *object
)
{
  object->attributes.external = SET;
}
7.7 Encrypt Decrypt Support (EncryptDecrypt_spt.c)

```c
#include "Tpm.h"
#include "EncryptDecrypt_fp.h"
#include "EncryptDecrypt_spt_fp.h"
#if CC_EncryptDecrypt2

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_KEY</td>
<td>is not a symmetric decryption key with both public and private portions loaded</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>IvIn size is incompatible with the block cipher mode; or inData size is not an even multiple of the block size for CBC or ECB mode</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>keyHandle is restricted and the argument mode does not match the key's mode</td>
</tr>
</tbody>
</table>

TPM_RC

void EncryptDecryptShared(
    TPMI_DH_OBJECT        keyHandleIn,
    TPMI_YES_NO           decryptIn,
    TPMI_ALG_SYM_MODE     modeIn,
    TPM2B_IV              *ivIn,
    TPM2B_MAX_BUFFER      *inData,
    EncryptDecrypt_Out    *out)
{
    OBJECT              *symKey;
    UINT16               keySize;
    UINT16               blockSize;
    BYTE                *key;
    TPM_ALG_ID           alg;
    TPM_ALG_ID           mode;
    TPM_RC               result;
    BOOL                 OK;

    // Input Validation
    symKey = HandleToObject(keyHandleIn);
    mode = symKey->publicArea.parameters.symDetail.sym.mode.sym;

    // The input key should be a symmetric key
    if(symKey->publicArea.type != TPM_ALG_SYM_CIPHER)
        return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;

    // The key must be unrestricted and allow the selected operation
    OK = !IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
                        TPMA_OBJECT, restricted);
    if(YES == decryptIn)
        OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
                                 TPMA_OBJECT, decrypt);
    else
        OK = OK && IS_ATTRIBUTE(symKey->publicArea.objectAttributes,
                                 TPMA_OBJECT, sign);

    if(!OK)
        return TPM_RCS_ATTRIBUTES + RC_EncryptDecrypt_keyHandle;

    // Make sure that key is an encrypt/decrypt key and not SMAC
    if(!CryptSymModeIsValid(mode, TRUE))
        return TPM_RCS_MODE + RC_EncryptDecrypt_keyHandle;

    // If the key mode is not TPM_ALG_NULL...
    if(mode != TPM_ALG_NULL)
    {
        // then the input mode has to be TPM_ALG_NULL or the same as the key
```
if ((modeIn != TPM_ALG_NULL) && (modeIn != mode))
    return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
}
else

    if (modeIn == TPM_ALG_NULL)
        return TPM_RCS_MODE + RC_EncryptDecrypt_mode;
    mode = modeIn;

    if (keyMode == TPM_ALG_NULL) then input can't be null

    if (keyMode == TPM_ALG_NULL)
        return TPM_RCS_MODE + RC_EncryptDecrypt_mode;

    keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
    alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
    blockSize = CryptGetSymmetricBlockSize(alg, keySize);

    // The input iv for ECB mode should be an Empty Buffer. All the other modes
    // should have an iv size same as encryption block size
    keySize = symKey->publicArea.parameters.symDetail.sym.keyBits.sym;
    alg = symKey->publicArea.parameters.symDetail.sym.algorithm;
    blockSize = CryptGetSymmetricBlockSize(alg, keySize);

    // reverify the algorithm. This is mainly to keep static analysis tools happy
    if (blockSize == 0)
        return TPM_RCS_SIZE + RC_EncryptDecrypt_keyHandle;

    // Note: When an algorithm is not supported by a TPM, the TPM_ALG_xxx for that
    // algorithm is not defined. However, it is assumed that the ALG_xxx_VALUE for
    // the algorithm is always defined. Both have the same numeric value.
    // ALG_xxx_VALUE is used here so that the code does not get cluttered with
    // #ifdef's. Having this check does not mean that the algorithm is supported.
    // If it was not supported the unmarshaling code would have rejected it before
    // this function were called. This means that, depending on the implementation,
    // the check could be redundant but it doesn't hurt.
    if (((mode == ALG_ECB_VALUE) && (ivIn->t.size != 0))
        || ((mode != ALG_ECB_VALUE) && (ivIn->t.size != blockSize))
        return TPM_RCS_SIZE + RC_EncryptDecrypt_ivIn;

    // The input data size of CBC mode or ECB mode must be an even multiple of
    // the symmetric algorithm's block size
    if (((mode == ALG_CBC_VALUE) || (mode == ALG_ECB_VALUE))
        && ((inData->t.size % blockSize) != 0))
        return TPM_RCS_SIZE + RC_EncryptDecrypt_inData;

    // Copy IV
    // Note: This is copied here so that the calls to the encrypt/decrypt functions
    // will modify the output buffer, not the input buffer
    out->ivOut = *ivIn;

    // Command Output
    key = symKey->sensitive.sensitive.sym.t.buffer;
    // For symmetric encryption, the cipher data size is the same as plain data
    // size.
    out->outData->t.size = inData->t.size;
    if (decryptIn == YES)
        return TPM_RCS_KEY + RC_EncryptDecrypt_keyHandle;
    }
    else
        return TPM_RCS_SIZE + RC_EncryptDecrypt_keyHandle;

    // Decrypt data to output
    result = CryptSymmetricDecrypt(out->outData->t.buffer, alg, keySize, key,
        &out->ivOut, mode, inData->t.size,
        inData->t.buffer);
}

    else
        return TPM_RCS_SIZE + RC_EncryptDecrypt_keyHandle;

    // Encrypt data to output
    result = CryptSymmetricEncrypt(out->outData->t.buffer, alg, keySize, key,
        &out->ivOut, mode, inData->t.size,
        inData->t.buffer);

    return result;
#endif // CC_E
8 Subsystem

8.1 CommandAudit.c

8.1.1 Introduction

This file contains the functions that support command audit.

8.1.2 Includes

```c
#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.

```c
void CommandAuditPreInstall_Init(
    void
) {
    // Clear all the audit commands
    MemorySet(gp.auditCommands, 0x00, sizeof(gp.auditCommands));

    // TPM_CC_SetCommandCodeAuditStatus always being audited
    CommandAuditSet(TPM_CC_SetCommandCodeAuditStatus);

    // Set initial command audit hash algorithm to be context integrity hash
    // algorithm
    gp.auditHashAlg = CONTEXT_INTEGRITY_HASH_ALG;

    // Set up audit counter to be 0
    gp.auditCounter = 0;

    // Write command audit persistent data to NV
    NV_SYNC_PERSISTENT(auditCommands);
    NV_SYNC_PERSISTENT(auditHashAlg);
    NV_SYNC_PERSISTENT(auditCounter);

    return;
}
```

8.1.3.2 CommandAuditStartup()

This function clears the command audit digest on a TPM Reset.

```c
BOOL CommandAuditStartup(
    STARTUP_TYPE type       // IN: start up type
) {
    if ((type != SU_RESTART) && (type != SU_RESUME))
```
{ // Reset the digest size to initialize the digest
gr.commandAuditDigest.t.size = 0;
}
return TRUE;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>command code audit status was changed</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>command code audit status was not changed</td>
</tr>
</tbody>
</table>

BOOL CommandAuditSet(
    TPM_CC commandCode // IN: command code
)
{
    COMMAND_INDEX commandIndex = CommandCodeToCommandIndex(commandCode);
    // Only SET a bit if the corresponding command is implemented
    if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
    {
        // Can't audit shutdown
        if(commandCode != TPM_CC_Shutdown)
        {
            if(!TEST_BIT(commandIndex, gp.auditCommands))
            {
                // Set bit
                SET_BIT(commandIndex, gp.auditCommands);
                return TRUE;
            }
        }
    }
    // No change
    return FALSE;
}

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.
8.1.3.5 CommandAuditIsRequired()

This function indicates if the audit flag is SET for a command.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>command is audited</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>command is not audited</td>
</tr>
</tbody>
</table>

```c
BOOL CommandAuditIsRequired(
    COMMAND_INDEX commandIndex  // IN: command index
) {
    // Check the bit map. If the bit is SET, command audit is required
    return (TEST_BIT(commandIndex, gp.auditCommands));
}
```

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`. 
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more command code available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available command code has been returned</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO CommandAuditCapGetCCList(
    TPM_CC commandCode, // IN: start command code
    UINT32 count,       // IN: count of returned TPM_CC
    TPML_CC *commandList // OUT: list of TPM_CC
) {
    TPMI_YES_NO more = NO;
    COMMAND_INDEX commandIndex;
    // Initialize output handle list
    commandList->count = 0;
    // The maximum count of command we may return is MAX_CAP_CC
    if(count > MAX_CAP_CC) count = MAX_CAP_CC;
    // Find the implemented command that has a command code that is the same or
    // higher than the input
    // Collect audit commands
    for(commandIndex = GetClosestCommandIndex(commandCode);
        commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
        commandIndex = GetNextCommandIndex(commandIndex))
    {
        if(CommandAuditIsRequired(commandIndex))
        {
            if(commandList->count < count)
            {
                // If we have not filled up the return list, add this command
                // code to its
                TPM_CC cc = GET_ATTRIBUTE(s_ccAttr[commandIndex],
                    TPMA_CC, commandIndex);
                if(IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V))
                    cc += (1 << 29);
                commandList->commandCodes[commandList->count] = cc;
                commandList->count++;
            }
        }
        else
        {
            // If the return list is full but we still have command
            // available, report this and stop iterating
            more = YES;
            break;
        }
    }
    return more;
}
```

### 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.

```c
void CommandAuditGetDigest()
```
TPM2B_DIGEST *digest  // OUT: command digest
{
    TPM_CC commandCode;
    COMMAND_INDEX commandIndex;
    HASH_STATE hashState;

    // Start hash
    digest->t.size = CryptHashStart(&hashState, gp.auditHashAlg);

    // Add command code
    for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
    {
        if(CommandAuditIsRequired(commandIndex))
        {
            commandCode = GetCommandCode(commandIndex);
            CryptDigestUpdateInt(&hashState, sizeof(commandCode), commandCode);
        }
    }

    // Complete hash
    CryptHashEnd2B(&hashState, &digest->b);
    return;
}
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

```c
#define DA_C
#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().

```c
void DAPreInstall_Init()
{
    gp.failedTries = 0;
    gp.maxTries = 3;
    gp.recoveryTime = 1000; // in seconds (~16.67 minutes)
    gp.lockoutRecovery = 1000; // in seconds
    gp.lockOutAuthEnabled = TRUE; // Use of lockoutAuth is enabled
    // Record persistent DA parameter changes to NV
    _plat__TimerWasReset();
    return;
}
```

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.

```c
BOOL DAStartup(
    STARTUP_TYPE     type // IN: startup type
)
{
    NOT_REFERENCED(type);
    #if !ACCUMULATE_SELF_HEAL_TIMER
       _plat__TimerWasReset();
    ```
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.

```c
void DARegisterFailure(
    TPM_HANDLE       handle       // IN: handle for failure
)
```
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.

```c
void DASelfHeal()
{
    // Regular authorization self healing logic
    // If no failed authorization tries, do nothing. Otherwise, try to
    // decrease failedTries
    if(gp.failedTries != 0)
    {
        // if recovery time is 0, DA logic has been disabled. Clear failed tries
        // immediately
        if(gp.recoveryTime == 0)
        {
            gp.failedTries = 0;
            // Update NV record
            NV_SYNC_PERSISTENT(failedTries);
        }
        else
        {
            UINT64 decreaseCount;
            #if 0 // Errata eliminates this code
            // In the unlikely event that failedTries should become larger than
            // maxTries
            if(gp.failedTries > gp.maxTries)
                gp.failedTries = gp.maxTries;
            #endif
            // How much can failedTries be decreased
            // Cast s_selfHealTimer to an int in case it became negative at
            // startup
decreaseCount = ((g_time - (INT64)s_selfHealTimer) / 1000)
            / gp.recoveryTime;
            if(gp.failedTries <= (UINT32)decreaseCount)
                // should not set failedTries below zero
                gp.failedTries = 0;
            else
                gp.failedTries -= (UINT32)decreaseCount;
            // the cast prevents overflow of the product
            s_selfHealTimer += (decreaseCount * (UINT64)gp.recoveryTime) * 1000;
            if(decreaseCount != 0)
                // If there was a change to the failedTries, record the changes
                // to NV
                NV_SYNC_PERSISTENT(failedTries);
        }
    }
}
```
// LockoutAuth self healing logic
// If lockoutAuth is enabled, do nothing. Otherwise, try to see if we
// may enable it
if(!gp.lockOutAuthEnabled)
{
    // if lockout authorization recovery time is 0, a reboot is required to
    // re-enable use of lockout authorization. Self-healing would not
    // apply in this case.
    if(gp.lockoutRecovery != 0)
    {
        if(((g_time - (INT64)s_lockoutTimer) / 1000) >= gp.lockoutRecovery)
        {
            gp.lockOutAuthEnabled = TRUE;
            // Record the changes to NV
            NV_SYNC_PERSISTENT(lockOutAuthEnabled);
        }
    }
    return;
}
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

```c
#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.

```c
void HierarchyPreInstall_Init()
{
    // Allow lockout clear command
    gp.disableClear = FALSE;

    // Initialize Primary Seeds
    gp.EPSeed.t.size = sizeof(gp.EPSeed.t.buffer);
    gp.SPSeed.t.size = sizeof(gp.SPSeed.t.buffer);
    gp.PPSeed.t.size = sizeof(gp.PPSeed.t.buffer);
    #if (defined USE_PLATFORM_EPS) && (USE_PLATFORM_EPS != NO)
        _plat__GetEPS(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
    #else
        CryptRandomGenerate(gp.EPSeed.t.size, gp.EPSeed.t.buffer);
    #endif
    CryptRandomGenerate(gp.SPSeed.t.size, gp.SPSeed.t.buffer);
    CryptRandomGenerate(gp.PPSeed.t.size, gp.PPSeed.t.buffer);

    // Initialize owner, endorsement and lockout authorization
    gp.ownerAuth.t.size = 0;
    gp.endorsementAuth.t.size = 0;
    gp.lockoutAuth.t.size = 0;

    // Initialize owner, endorsement, and lockout policy
    gp.ownerAlg = TPM_ALG_NULL;
    gp.ownerPolicy.t.size = 0;
    gp.endorsementAlg = TPM_ALG_NULL;
    gp.endorsementPolicy.t.size = 0;
    gp.lockoutAlg = TPM_ALG_NULL;
    gp.lockoutPolicy.t.size = 0;

    // Initialize ehProof, shProof and phProof
    gp.phProof.t.size = sizeof(gp.phProof.t.buffer);
    gp.shProof.t.size = sizeof(gp.shProof.t.buffer);
    gp.ehProof.t.size = sizeof(gp.ehProof.t.buffer);
    CryptRandomGenerate(gp.phProof.t.size, gp.phProof.t.buffer);
    CryptRandomGenerate(gp.shProof.t.size, gp.shProof.t.buffer);
    CryptRandomGenerate(gp.ehProof.t.size, gp.ehProof.t.buffer);

    // Write hierarchy data to NV
```
```c
NV_SYNC_PERSISTENT(disableClear);
NV_SYNC_PERSISTENT(EPSeed);
NV_SYNC_PERSISTENT(SPSeed);
NV_SYNC_PERSISTENT(PPSeed);
NV_SYNC_PERSISTENT(ownerAuth);
NV_SYNC_PERSISTENT(endorsementAuth);
NV_SYNC_PERSISTENT(lockoutAuth);
NV_SYNC_PERSISTENT(ownerAlg);
NV_SYNC_PERSISTENT(ownerPolicy);
NV_SYNC_PERSISTENT(endorsementAlg);
NV_SYNC_PERSISTENT(endorsementPolicy);
NV_SYNC_PERSISTENT(lockoutAlg);
NV_SYNC_PERSISTENT(lockoutPolicy);
NV_SYNC_PERSISTENT(phProof);
NV_SYNC_PERSISTENT(shProof);
NV_SYNC_PERSISTENT(ehProof);
return;
}

8.3.3.2 HierarchyStartup()

This function is called at TPM2_Startup() to initialize the hierarchy related values.

``` BOOL
HierarchyStartup(
STARTUP_TYPE type    // IN: start up type
)
{
    // phEnable is SET on any startup
    g_phEnable = TRUE;
    // Reset platformAuth, platformPolicy; enable SH and EH at TPM_RESET and
    // TPM_RESTART
    if(type != SU_RESUME)
    { gc.platformAuth.t.size = 0;
      gc.platformPolicy.t.size = 0;
      gc.platformAlg = TPM_ALG_NULL;
      // enable the storage and endorsement hierarchies and the platformNV
      gc.shEnable = gc.ehEnable = gc.phEnableNV = TRUE;
    }
    // nullProof and nullSeed are updated at every TPM_RESET
    if((type != SU_RESTART) && (type != SU_RESUME))
    { gr.nullProof.t.size = sizeof(gr.nullProof.t.buffer);
      CryptRandomGenerate(gr.nullProof.t.size, gr.nullProof.t.buffer);
      gr.nullSeed.t.size = sizeof(gr.nullSeed.t.buffer);
      CryptRandomGenerate(gr.nullSeed.t.size, gr.nullSeed.t.buffer);
    }
    return TRUE;
}

8.3.3.3 HierarchyGetProof()

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

```c
TPM2B_PROOF *
HierarchyGetProof(
    TPMI_RH_HIERARCHY hierarchy    // IN: hierarchy constant
```
TPM2B_PROOF *proof = NULL;

switch (hierarchy) {
    case TPM_RH_PLATFORM:
        // phProof for TPM_RH_PLATFORM
        proof = &gp.phProof;
        break;
    case TPM_RH_ENDORSEMENT:
        // ehProof for TPM_RH_ENDORSEMENT
        proof = &gp.ehProof;
        break;
    case TPM_RH_OWNER:
        // shProof for TPM_RH_OWNER
        proof = &gp.shProof;
        break;
    default:
        // nullProof for TPM_RH_NULL or anything else
        proof = &gr.nullProof;
        break;
}
return proof;

8.3.3.4 HierarchyGetPrimarySeed()

This function returns the primary seed of a hierarchy.

TPM2B_SEED * HierarchyGetPrimarySeed(
    TPMI_RH_HIERARCHY hierarchy  // IN: hierarchy
)
{
    TPM2B_SEED *seed = NULL;
    switch (hierarchy) {
        case TPM_RH_PLATFORM:
            seed = &gp.PPSeed;
            break;
        case TPM_RH_OWNER:
            seed = &gp.SPSeed;
            break;
        case TPM_RH_ENDORSEMENT:
            seed = &gp.EPSeed;
            break;
        default:
            seed = &gr.nullSeed;
            break;
    }
    return seed;

8.3.3.5 HierarchyIsEnabled()

This function checks to see if a hierarchy is enabled.
NOTE: The TPM_RH_NULL hierarchy is always enabled.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>hierarchy enabled</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>hierarchy disabled</td>
</tr>
</tbody>
</table>

```c
BOOL HierarchyIsEnabled(
    TPMI_RH_HIERARCHY hierarchy  // IN: hierarchy
) {
    BOOL enabled = FALSE;
    switch(hierarchy)
    {
        case TPM_RH_PLATFORM:
            enabled = g_phEnable;
            break;
        case TPM_RH_OWNER:
            enabled = gc.shEnable;
            break;
        case TPM_RH_ENDORSEMENT:
            enabled = gc.ehEnable;
            break;
        case TPM_RH_NULL:
            enabled = TRUE;
            break;
        default:
            enabled = FALSE;
            break;
    }
    return enabled;
}
```
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

```c
#define NV_C
#include "Tpm.h"
#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.

```c
static NV_REF
NvNext(
    NV_REF *iter,      // IN/OUT: the list iterator
    TPM_HANDLE *handle // OUT: the handle of the next item.
)
{
    NV_REF               currentAddr;
    NV_ENTRY_HEADER      header;
    //
```
8.4.3.2 NvNextByType()

This function returns a reference to the next NV entry of the desired type.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>end of list</td>
</tr>
<tr>
<td>!= 0</td>
<td>the next entry of the indicated type</td>
</tr>
</tbody>
</table>

```c
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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>end of list</td>
</tr>
<tr>
<td>!= 0</td>
<td>the next reference</td>
</tr>
</tbody>
</table>

### 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.

```c
#define NvNextEvict(handle, iter) NvNextByType(handle, iter, TPM_HT_PERSISTENT)
```

### 8.4.3.5 NvGetEnd()

Function to find the end of the NV dynamic data list.

```c
static NV_REF
NvGetEnd()
{
    void
	nextRef; iter = NV_REF_INIT;
    NV_REF currentAddr;

    // Scan until the next address is 0
    while((currentAddr = NvNext(&iter, NULL)) != 0);
    return iter;
}
```

### 8.4.3.6 NvGetFreeBytes

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

```c
static UINT32
NvGetFreeBytes()
{
    void

    // This does not have an overflow issue because NvGetEnd() cannot return a value
    // that is larger than s_evictNvEnd. This is because there is always a 'stop'
    // word in the NV memory that terminates the search for the end before the
    // value can go past s_evictNvEnd.

    return s_evictNvEnd - NvGetEnd();
}
```

### 8.4.3.7 NvTestSpace()

This function will test if there is enough space to add a new entity.
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>space available</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>no enough space</td>
</tr>
</tbody>
</table>

```
static BOOL NvTestSpace(
    UINT32 size,    // IN: size of the entity to be added
    BOOL isIndex,  // IN: TRUE if the entity is an index
    BOOL isCounter // IN: TRUE if the index is a counter
)
{
    UINT32 remainBytes = NvGetFreeBytes();
    UINT32 reserved = sizeof(UINT32)    // size of the forward pointer
                     + sizeof(NV_LIST_TERMINATOR);

    // Do a compile time sanity check on the setting for NV_MEMORY_SIZE
    #if NV_MEMORY_SIZE < 1024
    #error "NV_MEMORY_SIZE probably isn't large enough"
    #endif

    // For NV Index, need to make sure that we do not allocate an Index if this
    // would mean that the TPM cannot allocate the minimum number of evict
    // objects.
    if(isIndex)
    {
        // Get the number of persistent objects allocated
        UINT32 persistentNum = NvCapGetPersistentNumber();

        // If we have not allocated the requisite number of evict objects, then we
        // need to reserve space for them.
        // NOTE: some of this is not written as simply as it might seem because
        // the values are all unsigned and subtracting needs to be done carefully
        // so that an underflow doesn't cause problems.
        if(persistentNum < MIN_EVICT_OBJECTS)
        { reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE; }
    }
    // If this is not an index or is not a counter, reserve space for the
    // required number of counter indexes
    if(!isIndex || !isCounter)
    {
        // Get the number of counters
        UINT32 counterNum = NvCapGetCounterNumber();

        // If the required number of counters have not been allocated, reserved
        // space for the extra needed counters
        if(counterNum < MIN_COUNTER_INDICES)
        { reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE; }
    }
    // Check that the requested allocation will fit after making sure that there
    // will be no chance of overflow
    return ((reserved < remainBytes)
             && (size <= remainBytes)
             && (size + reserved <= remainBytes));
}
```

### 8.4.3.8 NvWriteNvListEnd()

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

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.

static TPM_RC
NvAdd(
    UINT32 totalSize, // IN: total size needed for this entity
    UINT32 bufferSize, // IN: size of initial buffer
    TPM_HANDLE handle, // IN: optional handle
    BYTE *entity // IN: initial buffer
) {
    NV_REF newAddr;  // IN: where the new entity will start
    NV_REF nextAddr;
    // RETURN_IF_NV_IS_NOT_AVAILABLE;
    // Get the end of data list
    newAddr = NvGetEnd();
    // Step over the forward pointer
    nextAddr = newAddr + sizeof(UINT32);
    // Optionally write the handle. For indexes, the handle is TPM_RH_UNASSIGNED
    // so that the handle in the nvIndex is used instead of writing this value
    if(handle != TPM_RH_UNASSIGNED)
    {
        NvWrite((UINT32)nextAddr, sizeof(TPM_HANDLE), &handle);
        nextAddr += sizeof(TPM_HANDLE);
    }
    // Write entity data
    NvWrite((UINT32)nextAddr, bufferSize, entity);
    // Advance the pointer by the amount of the total
    nextAddr += totalSize;
    // Finish by writing the link value
    // Write the next offset (relative addressing)
**8.4.3.10 NvDelete()**

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

```c
static TPM_RC
NvDelete(
    NV_REF entityRef // IN: reference to entity to be deleted
)
{
    UINT32 entrySize;
    // adjust entityAddr to back up and point to the forward pointer
    NV_REF entryRef = entityRef - sizeof(UINT32);
    NV_REF endRef = NvGetEnd();
    NV_REF nextAddr; // address of the next entry
    // RETURN_IF_NV_IS_NOT_AVAILABLE;

    // Get the offset of the next entry. That is, back up and point to the size
    // field of the entry
    NvRead(&entrySize, entryRef, sizeof(UINT32));

    // The next entry after the one being deleted is at a relative offset
    // from the current entry
    nextAddr = entryRef + entrySize;

    // If this is not the last entry, move everything up
    if(nextAddr < endRef)
    {
        pAssert(nextAddr > entryRef);
        _plat__NvMemoryMove(nextAddr, entryRef,
            (endRef - nextAddr));
    }
    // The end of the used space is now moved up by the amount of space we just
    // reclaimed
    endRef -= entrySize;

    // Write the end marker, and make the new end equal to the first byte after
    // the just added end value. This will automatically update the NV value for
    // maxCounter.
    // NOTE: This is the call that sets flag to cause NV to be updated
    endRef = NvWriteNvListEnd(endRef);

    // Clear the reclaimed memory
    _plat__NvMemoryClear(endRef, entrySize);

    return TPM_RC_SUCCESS;
}
```

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

```c
static NV_RAM_REF
NvRamNext(
    NV_RAM_REF *iter,         // IN/OUT: the list iterator
    TPM_HANDLE *handle        // OUT: the handle of the next item.
)
{
    NV_RAM_REF           currentAddr;
    NV_RAM_HEADER        header;

    // If iterator is at the beginning of list
    if(*iter == NV_RAM_REF_INIT)
    {
        // Initialize iterator
        *iter = &s_indexOrderlyRam[0];
    }
    else
    {
        // if we are going to return what the iter is currently pointing to...
        currentAddr = *iter;
        // If iterator reaches the end of NV space, then don't advance and return
        // that we are at the end of the list. The end of the list occurs when
        // we don't have space for a size and a handle
        if(currentAddr + sizeof(NV_RAM_HEADER) > RAM_ORDERLY_END)
            return NULL;
        // read the header of the next entry
        MemoryCopy(&header, currentAddr, sizeof(NV_RAM_HEADER));

        // if the size field is zero, then we have hit the end of the list
        if(header.size == 0)
            // leave the *iter pointing at the end of the list
            return NULL;
        // advance the header by the size of the entry
        *iter = currentAddr + header.size;

        if(header.handle != NULL)
            *handle = header.handle;
        else
            return NULL;
    }
}
```

8.4.4.3 NvRamGetEnd()

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

```c
static NV_RAM_REF
NvRamGetEnd()
{
    void
}
```
Part 4: Supporting Routines

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

8.4.4 NvRamTestSpaceIndex()

This function indicates if there is enough RAM space to add a data for a new NV Index.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>space available</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>no enough space</td>
</tr>
</tbody>
</table>

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

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

8.4.6 NvUpdateIndexOrderlyData()

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

323 void NvUpdateIndexOrderlyData( 
324   void 
325 })
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.

```c
static void
NvAddRAM(
    TPMS_NV_PUBLIC  *index, // IN: the index descriptor
)
{
    NV_RAM_HEADER       header;
    NV_RAM_REF          end = NvRamGetEnd();
    // header.size = sizeof(NV_RAM_HEADER) + index->dataSize;
    header.handle = index->nvIndex;
    MemoryCopy(&header.attributes, &index->attributes, sizeof(TPMA_NV));
    pAssert(ORDERLY_RAM_ADDRESS_OK(end, header.size));
    // Copy the header to the memory
    MemoryCopy(end, &header, sizeof(NV_RAM_HEADER));
    // Clear the data area (just in case)
    MemorySet(end + sizeof(NV_RAM_HEADER), 0, index->dataSize);
    // Step over this new entry
    end += header.size;
    // If the end marker will fit, add it
    if (end + sizeof(UINT32) < RAM_ORDERLY_END)
    {
        MemorySet(end, 0, sizeof(UINT32));
        // Write reserved RAM space to NV to reflect the newly added NV Index
        SET_NV_UPDATE(UT_ORDERLY);
    }
    return;
}
```

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.

```c
static void
NvDeleteRAM(
    TPMI_RH_NV_INDEX     handle    // IN: NV handle
)
{
    NV_RAM_REF           nodeAddress;
    NV_RAM_REF           nextNode;
    // Write reserved RAM space to NV
    NvWrite(NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam), s_indexOrderlyRam);
}
```
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.

```c
void NvReadNvIndexInfo(
    NV_REF ref,               // IN: points to NV where index is located
    NV_INDEX *nvIndex         // OUT: place to receive index data
)
{
    pAssert(nvIndex != NULL);
    NvRead(nvIndex, ref, sizeof(NV_INDEX));
    return;
}
```

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.

```c
void NvReadObject(
    NV_REF ref,               // IN: points to NV where index is located
    OBJECT *object            // OUT: place to receive the object data
)
{
    NvRead(object, (ref + sizeof(TPM_HANDLE)), sizeof(OBJECT));
    return;
}
```

8.4.4.11 NvFindEvict()

This function will return the NV offset of an evict object
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>evict object not found</td>
</tr>
<tr>
<td>!= 0</td>
<td>offset of evict object</td>
</tr>
</tbody>
</table>

```c
static NV_REF
NvFindEvict(
    TPM_HANDLE       nvHandle,
    OBJECT          *object
)
{
    NV_REF          found = NvFindHandle(nvHandle);
    // If we found the handle and the request included an object pointer, fill it in
    if(found != 0 && object != NULL)
    
        NvReadObject(found, object);
    
    return found;
}
```

### 8.4.4.12 NvIndexIsDefined()

See if an index is already defined

```c
BOOL
NvIndexIsDefined(
    TPM_HANDLE       nvHandle   // IN: Index to look for
)
{
    return (NvFindHandle(nvHandle) != 0);
}
```

### 8.4.4.13 NvConditionallyWrite()

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

```c
static TPM_RC
NvConditionallyWrite(
    NV_REF           entryAddr,   // IN: stating address
    UINT32           size,        // IN: size of the data to write
    void             *data        // IN: the data to write
)
{
    // If the index data is actually changed, then a write to NV is required
    if(_plat__NvIsDifferent(entryAddr, size, data))
    
        // Write the data if NV is available
        if(g_NvStatus == TPM_RC_SUCCESS)
            
            NvWrite(entryAddr, size, data);
    
        return g_NvStatus;
    
    return TPM_RC_SUCCESS;
}
```
8.4.4.14 NvReadNvIndexAttributes()

This function returns the attributes of an NV Index.

```c
static TPMA_NV
NvReadNvIndexAttributes(
    NV_REF locator // IN: reference to an NV index
)
{
    TPMA_NV attributes;

    NvRead(&attributes, locator + offsetof(NV_INDEX, publicArea.attributes), sizeof(TPMA_NV));
    return attributes;
}
```

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.

```c
static TPMA_NV
NvReadRamIndexAttributes(
    NV_RAM_REF ref // IN: pointer to a NV_RAM_HEADER
)
{
    TPMA_NV attributes;

    MemoryCopy(&attributes, ref + offsetof(NV_RAM_HEADER, attributes), sizeof(TPMA_NV));
    return attributes;
}
```

8.4.4.16 NvWriteNvIndexAttributes()

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting so retry</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
NvWriteNvIndexAttributes(
    NV_REF locator, // IN: location of the index
    TPMA_NV attributes // IN: attributes to write
)
{
    return NvConditionallyWrite(
        locator + offsetof(NV_INDEX, publicArea.attributes),
        sizeof(TPMA_NV),
        &attributes);
}
```

8.4.4.17 NvWriteRamIndexAttributes()

This function is used to write the index attributes into an unaligned structure
static void NvWriteRamIndexAttributes(NV_RAM_REF ref, // IN: address of the header, TPMA_NV attributes // IN: the attributes to write )
{
    MemoryCopy(ref + offsetof(NV_RAM_HEADER, attributes), &attributes, sizeof(TPMA_NV));
    return;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>handle references a platform persistent object and may reference an owner persistent object either</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>handle does not reference platform persistent object</td>
</tr>
</tbody>
</table>

BOOL NvIsPlatformPersistentHandle(TPM_HANDLE handle) // IN: handle
{
    return (handle >= PLATFORM_PERSISTENT && handle <= PERSISTENT_LAST);
}

8.4.5.2 NvIsOwnerPersistentHandle()

This function indicates if a handle references a persistent object in the range belonging to the owner.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>handle is owner persistent handle</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>handle is not owner persistent handle and may not be a persistent handle at all</td>
</tr>
</tbody>
</table>

BOOL NvIsOwnerPersistentHandle(TPM_HANDLE handle) // IN: handle
{
    return (handle >= PERSISTENT_FIRST && handle < PLATFORM_PERSISTENT);
}

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 shEnable is CLEAR, this would include an index created using ownerAuth. If phEnableNV is CLEAR, this would include a platform created using platformAuth.
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.

```c
/* NvIndexIsAccessible */
TPM_RC
NvIndexIsAccessible(
    TPM_RH_NV_INDEX handle  // IN: handle
) {
    NV_INDEX *nvIndex = NvGetIndexInfo(handle, NULL);
    // if index is not found, return TPM_RC_HANDLE
    if (nvIndex == NULL)
        return TPM_RC_HANDLE;
    if (gc.shEnable == FALSE || gc.phEnableNV == FALSE) {
        // if shEnable is CLEAR, an ownerCreate NV Index should not be indicated as present
        if (!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE)) {
            if (gc.shEnable == FALSE)
                return TPM_RC_HANDLE;
        } // if phEnableNV is CLEAR, a platform created Index should not be visible
    } else if (gc.phEnableNV == FALSE)
        return TPM_RC_HANDLE;
    #if 0 // Writelock test for debug
    // If the Index is write locked and this is an NV Write operation...
    if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITELOCKED)
        && IsWriteOperation(commandIndex)) {
        // then return a locked indication unless the command is TPM2_NV_WriteLock
        if(GetCommandCode(commandIndex) != TPM_CC_NV_WriteLock)
            return TPM_RC_NV_LOCKED;
    }
    #endif
    #if 0 // Readlock Test for debug
    // If the Index is read locked and this is an NV Read operation...
    if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, READLOCKED)
        && IsReadOperation(commandIndex)) {
        // then return a locked indication unless the command is TPM2_NV_ReadLock
        if(GetCommandCode(commandIndex) != TPM_CC_NV_ReadLock)
            return TPM_RC_NV_LOCKED;
    }
    #endif
    // NV Index is accessible
    return TPM_RC_SUCCESS;
}
```

8.4.5.4 **NvGetEvictObject()**

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

<table>
<thead>
<tr>
<th>Return</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_HANDLE</td>
<td>the handle does not point to an existing persistent object</td>
</tr>
</tbody>
</table>

8.4.5.5 **NvIndexCacheInit()**

Function to initialize the Index cache

```c
void
NvIndexCacheInit()
{
    s_cachedNvRef = NV_REF_INIT;
    s_cachedNvRamRef = NV_RAM_REF_INIT;
    s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
    return;
}
```

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.

```c
void
NvGetIndexData()
{
    pAssert(nvIndex != NULL);
    TPMA_NV
    nvAttributes;
    *nvIndex,  // IN: the in RAM index descriptor
    nvAttributes = nvIndex->publicArea.attributes;
    return;
}
```
8.4.5.7 NvHashIndexData()

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

```c
void NvHashIndexData(
    HASH_STATE *hashState,  // IN: Initialized hash state
    NV_INDEX *nvIndex,      // IN: Index
    NV_REF locator,         // IN: where the data is located
    UINT32 offset,          // IN: starting offset
    UINT16 size             // IN: amount to hash
)
{
    #define BUFFER_SIZE 64
    BYTE buffer[BUFFER_SIZE];
    if (offset > nvIndex->publicArea.dataSize)
        return;
    // Make sure that we don't try to read off the end.
    if ((offset + size) > nvIndex->publicArea.dataSize)
        size = nvIndex->publicArea.dataSize - (UINT16)offset;
    #if BUFFER_SIZE >= MAX_NV_INDEX_SIZE
    NvGetIndexData(nvIndex, locator, offset, size, buffer);
    CryptDigestUpdate(hashState, size, buffer);
    #else
    INT16 i;
    UINT16 readSize;
    // (i = size; i > 0; offset += readSize, i -= readSize)
    readSize = (i < BUFFER_SIZE) ? i : BUFFER_SIZE;
    NvGetIndexData(nvIndex, locator, offset, readSize, buffer);
    CryptDigestUpdate(hashState, readSize, buffer);
    #endif
    #undef BUFFER_SIZE
}
```

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.

```c
UINT64 NvGetUINT64Data(
    NV_INDEX *nvIndex,    // IN: the in RAM index descriptor
    NV_REF locator      // IN: where index exists in NV
)
{
    UINT64 intVal;
    // Read the value and convert it to internal format
    NvGetIndexData(nvIndex, locator, 0, 8, &intVal);
    return BYTE_ARRAY_TO_UINT64(((BYTE *)&intVal));
}
```

### 8.4.5.9 NvWriteIndexAttributes()

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting so retry</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available</td>
</tr>
</tbody>
</table>

```c
TPM_RC NvWriteIndexAttributes(
    TPM_HANDLE handle,
    NV_REF locator,    // IN: location of the index
    TPMA_NV attributes // IN: attributes to write
)
{
    TPM_RC result;
    //
    if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
    {
        NV_RAM_REF ram = NvRamGetIndex(handle);
        NvWriteRamIndexAttributes(ram, attributes);
        result = TPM_RC_SUCCESS;
    }
    else
    {
        result = NvWriteNvIndexAttributes(locator, attributes);
    }
    return result;
}
```

### 8.4.5.10 NvWriteIndexAuth()

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting so retry</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available</td>
</tr>
</tbody>
</table>

```c
TPM_RC NvWriteIndexAuth(
    NV_REF locator,    // IN: location of the index
    TPM2B_AUTH *authValue // IN: the authValue to write
)
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.

```
NV_INDEX * NvGetIndexInfo(
    TPM_HANDLE       nvHandle,     // IN: the index handle
    NV_REF          *locator      // OUT: location of the index
)
{
    if(s_cachedNvIndex.publicArea.nvIndex != nvHandle)
    {
        s_cachedNvIndex.publicArea.nvIndex = TPM_RH_UNASSIGNED;
        s_cachedNvRamRef = 0;
        s_cachedNvRef = NvFindHandle(nvHandle);
        if(s_cachedNvRef == 0)
            return NULL;
        NvReadNvIndexInfo(s_cachedNvRef, &s_cachedNvIndex);
        if(IS_ATTRIBUTE(s_cachedNvIndex.publicArea.attributes, TPMA_NV, ORDERLY))
            {
                s_cachedNvRamRef = NvRamGetIndex(nvHandle);
                s_cachedNvIndex.publicArea.attributes =
                    NvReadRamIndexAttributes(s_cachedNvRamRef);
            }
    }
    if(locator != NULL)
        *locator = s_cachedNvRef;
    return &s_cachedNvIndex;
}
```

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting so retry</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not available</td>
</tr>
</tbody>
</table>

```c
TPM_RC
NvWriteIndexData(
    NV_INDEX        *nvIndex,  // IN: the description of the index
    UINT32           offset,  // IN: offset of NV data
    UINT32           size,    // IN: size of NV data
    void *data       // IN: data buffer
)
{
    TPM_RC               result = TPM_RC_SUCCESS;
    //
    pAssert(nvIndex != NULL);
    // Make sure that this is dealing with the 'default' index.
    // Note: it is tempting to change the calling sequence so that the 'default' is presumed.
    pAssert(nvIndex->publicArea.nvIndex == s_cachedNvIndex.publicArea.nvIndex);
    // Validate that write falls within range of the index
    pAssert(offset <= nvIndex->publicArea.dataSize
        &&  size <= (nvIndex->publicArea.dataSize - offset));
    // Update TPMA_NV_WRITTEN bit if necessary
    if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
    {
        // Update the in memory version of the attributes
        SET_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN);
        // If this is not orderly, then update the NV version of
        // the attributes
        if(!IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
        {
            result = NvWriteNvIndexAttributes(s_cachedNvRef,
                nvIndex->publicArea.attributes);
            if(result != TPM_RC_SUCCESS)
                return result;
            // If this is a partial write of an ordinary index, clear the whole
            // index.
            if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes)
                && (nvIndex->publicArea.dataSize > size))
                _plat__NvMemoryClear(s_cachedNvRamRef + sizeof(NV_INDEX),
                    nvIndex->publicArea.dataSize);
        }
        else
        {
            // This is orderly so update the RAM version
            MemoryCopy(s_cachedNvRamRef + offsetof(NV_RAM_HEADER, attributes),
                &nvIndex->publicArea.attributes, sizeof(TPMA_NV));
            // If setting WRITTEN for an orderly counter, make sure that the
            // state saved version of the counter is saved
            if(IsNvCounterIndex(nvIndex->publicArea.attributes))
                SET_NV_UPDATE(UT_ORDERLY);
            // If setting the written attribute on an ordinary index, make sure that
            // the data is all cleared out in case there is a partial write. This
            // is only necessary for ordinary indexes because all of the other types
            // are always written in total.
            else if(IsNvOrdinaryIndex(nvIndex->publicArea.attributes))
                MemorySet(s_cachedNvRamRef + sizeof(NV_RAM_HEADER),
        ```
792            0, nvIndex->publicArea.dataSize);
793    }
794    // If this is orderly data, write it to RAM
795    if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
796    {
797        // Note: if this is the first write to a counter, the code above will queue
798        // the write to NV of the RAM data in order to update TPMA_NV_WRITTEN. In
799        // process of doing that write, it will also write the initial counter value
800        // Update RAM
801        MemoryCopy(s_cachedNvRamRef + sizeof(NV_RAM_HEADER) + offset, data, size);
802        // And indicate that the TPM is no longer orderly
803        g_clearOrderly = TRUE;
804    }
805    else
806    {
807        // Offset into the index to the first byte of the data to be written to NV
808        result = NvConditionallyWrite(s_cachedNvRef + sizeof(NV_INDEX) + offset,
809            size, data);
810        return result;
811    }
812
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        TPM2B_NAME *name      // OUT: name of the index
831    )
832    {
833        UINT16      dataSize, digestSize;
834        BYTE        *marshalBuffer[sizeof(TPMS_NV_PUBLIC)];
835        BYTE        *buffer;
836        HASH_STATE  hashState;
838  // Marshal public area
839  buffer = marshalBuffer;
840  dataSize = TPMS_NV_PUBLIC_Marshal(&nvIndex->publicArea, &buffer, NULL);
841  // hash public area
842  digestSize = CryptHashStart(&hashState, nvIndex->publicArea.nameAlg);
843  CryptDigestUpdate(&hashState, dataSize, marshalBuffer);
844  // Complete digest leaving room for the nameAlg
845  CryptHashEnd(&hashState, digestSize, &name->b.buffer[2]);
846
847  // Include the nameAlg
848  UINT16_TO_BYTE_ARRAY(nvIndex->publicArea.nameAlg, name->t.size = digestSize + 2;
849  return name;
850 }

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_SPACE</td>
<td>insufficient NV space</td>
</tr>
</tbody>
</table>

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     // only allocate data space for indexes that are going to be written to NV.
878     // Orderly indexes don't need space.
879     if(!IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
880         entrySize += publicArea->dataSize;
// Check if we have enough space to create the NV Index
// In this implementation, the only resource limitation is the available NV
// space (and possibly RAM space.) Other implementation may have other
// limitation on counter or on NV slots
if(!NvTestSpace(entrySize, TRUE, IsNvCounterIndex(publicArea->attributes)))
    return TPM_RC_NV_SPACE;

// if the index to be defined is RAM backed, check RAM space availability
// as well
if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY)
    && !NvRamTestSpaceIndex(publicArea->dataSize))
    return TPM_RC_NV_SPACE;

// Copy input value to nvBuffer
nvIndex.publicArea = *publicArea;

// Copy the authValue
nvIndex.authValue = *authValue;

// Add index to NV memory
result = NvAdd(entrySize, sizeof(NV_INDEX), TPM_RH_UNASSIGNED,
    (BYTE *)&nvIndex);
if(result == TPM_RC_SUCCESS)
{
    // If the data of NV Index is RAM backed, add the data area in RAM as well
    if(IS_ATTRIBUTE(publicArea->attributes, TPMA_NV, ORDERLY))
        NvAddRAM(publicArea);
}
return result;

8.4.5.17 NvAddEvictObject()

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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_HANDLE</td>
<td>the requested handle is already in use</td>
</tr>
<tr>
<td>TPM_RC_NV_SPACE</td>
<td>insufficient NV space</td>
</tr>
</tbody>
</table>

TPM_RC

NvAddEvictObject(

    TPMI_DH_OBJECT   evictHandle, // IN: new evict handle
    OBJECT          *object      // IN: object to be added

)

{  
    TPM_HANDLE       temp = object->evictHandle;
    TPM_RC           result;

    // Check if we have enough space to add the evict object
    // An evict object needs 8 bytes in index table + sizeof OBJECT
    // In this implementation, the only resource limitation is the available NV
    // space. Other implementation may have other limitation on evict object
    // handle space
    if(!NvTestSpace(sizeof(OBJECT) + sizeof(TPM_HANDLE), FALSE, FALSE))
        return TPM_RC_NV_SPACE;

    // Set evict attribute and handle
    object->attributes.evict = SET;
    object->evictHandle = evictHandle;

    // Now put this in NV
    result = NvAdd(sizeof(OBJECT), sizeof(OBJECT), evictHandle, (BYTE *)object);
```c
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is not accessible</td>
</tr>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is rate limiting</td>
</tr>
</tbody>
</table>

```c
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) && IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, WRITTEN))
954      NvUpdateMaxCount(NvGetUINT64Data(nvIndex, entityAddr));
955      result = NvDelete(entityAddr);
956      if(result != TPM_RC_SUCCESS)
957      return result;
958      // If the NV Index is RAM backed, delete the RAM data as well
959      if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, ORDERLY))
960      NvDeleteRAM(nvIndex->publicArea.nvIndex);
961      NvIndexCacheInit();
962    }
963    return TPM_RC_SUCCESS;
964}

8.4.5.19 NvDeleteEvict()

This function will delete a NV evict object. Will return success if object deleted or if it does not exist

```c
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    if(entityAddr != 0)
974    result = NvDelete(entityAddr);
975    return result;
976}
```
8.4.5.20 NvFlushHierarchy()

This function will delete persistent objects belonging to the indicated hierarchy. If the storage hierarchy is selected, the function will also delete any NV Index defined using ownerAuth.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NV_RATE</td>
<td>NV is unavailable because of rate limit</td>
</tr>
<tr>
<td>TPM_RC_NV_UNAVAILABLE</td>
<td>NV is inaccessible</td>
</tr>
</tbody>
</table>

```
TPM_RC
NvFlushHierarchy(
    TPMI_RH_HIERARCHY    hierarchy    // IN: hierarchy to be flushed.
)
{
    NV_REF           iter = NV_REF_INIT;
    NV_REF           currentAddr;
    TPM_HANDLE       entityHandle;
    TPM_RC           result = TPM_RC_SUCCESS;
    //
    while((currentAddr = NvNext(&iter, &entityHandle)) != 0)
    {
        if(HandleGetType(entityHandle) == TPM_HT_NV_INDEX)
        {
            NV_INDEX        nvIndex;
            // If flush endorsement or platform hierarchy, no NV Index would be
            // flushed
            if((hierarchy == TPM_RH_ENDORSEMENT || hierarchy == TPM_RH_PLATFORM)
                continue;
            // Get the index information
            NvReadNvIndexInfo(currentAddr, &nvIndex);
            // For storage hierarchy, flush OwnerCreated index
            if(!IS_ATTRIBUTE(nvIndex.publicArea.attributes, TPMA_NV,
                PLATFORMCREATE))
            {
                // Delete the index (including RAM for orderly)
                result = NvDeleteIndex(&nvIndex, currentAddr);
                if(result != TPM_RC_SUCCESS)
                    break;
                // Re-iterate from beginning after a delete
                iter = NV_REF_INIT;
            }
        }
        else if(HandleGetType(entityHandle) == TPM_HT_PERSISTENT)
        {
            OBJECT_ATTRIBUTES           attributes;
            // If the evict object belongs to the hierarchy to be flushed...
            if((hierarchy == TPM_RH_PLATFORM && attributes.ppsHierarchy == SET)
                || (hierarchy == TPM_RH_OWNER && attributes.epsHierarchy == SET)
                || (hierarchy == TPM_RH_ENDORSEMENT
                    && attributes.epsHierarchy == SET))
            {
                // ...then delete the evict object
                result = NvDelete(currentAddr);
                if(result != TPM_RC_SUCCESS)
                    break;
            }
        }
    }
}
```
// Re-iterate from beginning after a delete
iter = NV_REF_INIT;

// Check all normal indexes
while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
{
    TPMA_NV attributes = NvReadNvIndexAttributes(currentAddr);
    // See if it should be locked
    if(!IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
       && IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
    {
        SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
        result = NvWriteNvIndexAttributes(currentAddr, attributes);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
}

// Now search all the orderly attributes
while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
{
    // See if it should be locked
    TPMA_NV attributes = NvReadRamIndexAttributes(currentRamAddr);
    if(IS_ATTRIBUTE(attributes, TPMA_NV, GLOBALLOCK))
    {
        SET_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);
        NvWriteRamIndexAttributes(currentRamAddr, attributes);
    }
}
return result;
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

```c
static void InsertSort(
    TPMHANDLE *handleList, // IN/OUT: sorted handle list
    UINT32 count,           // IN: maximum count in the handle list
    TPM_HANDLE entityHandle // IN: handle to be inserted
)
{
    UINT32 i, j;
    UINT32 originalCount;

    // For a corner case that the maximum count is 0, do nothing
    if(count == 0)
        return;

    // For empty list, add the handle at the beginning and return
    if(handleList->count == 0)
    {
        handleList->handle[0] = entityHandle;
        handleList->count++;
        return;
    }

    // Check if the maximum of the list has been reached
    originalCount = handleList->count;
    if(originalCount < count)
        handleList->count++;

    // Insert the handle to the list
    for(i = 0; i < originalCount; i++)
    {
        if(handleList->handle[i] > entityHandle)
        {
            for(j = handleList->count - 1; j > i; j--)
            {
                handleList->handle[j] = handleList->handle[j - 1];
            }
            break;
        }
    }

    // If a slot was found, insert the handle in this position
    if(i < originalCount || handleList->count > originalCount)
        handleList->handle[i] = entityHandle;
    return;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO NvCapGetPersistent(
    TPM_DH_OBJECT handle, // IN: start handle
    UINT32 count,        // IN: maximum number of returned handles
)
```
TPML_HANDLE *handleList  // OUT: list of handle
{
    TPMI_YES_NO more = NO;
    NV_REF iter = NV_REF_INIT;
    NV_REF currentAddr;
    TPM_HANDLE entityHandle;
    //
    pAssert(HandleGetType(handle) == TPM_HT_PERSISTENT);
    // Initialize output handle list
    handleList->count = 0;
    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if (count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
    while((currentAddr = NvNextEvict(&entityHandle, &iter)) != 0)
    {
        // Ignore persistent handles that have values less than the input handle
        if(entityHandle < handle) continue;
        // if the handles in the list have reached the requested count, and there
        // are still handles need to be inserted, indicate that there are more.
        if(handleList->count == count)
            more = YES;
        // A handle with a value larger than start handle is a candidate
        // for return. Insert sort it to the return list. Insert sort algorithm
        // is chosen here for simplicity based on the assumption that the total
        // number of NV indexes is small. For an implementation that may allow
        // large number of NV indexes, a more efficient sorting algorithm may be
        // used here.
        InsertSort(handleList, count, entityHandle);
    }
    return more;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles to report</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been reported</td>
</tr>
</tbody>
</table>

TPMI_YES_NO
NvCapGetIndex(
    TPMI_DH_OBJECT handle,  // IN: start handle
    UINT32 count,           // IN: max number of returned handles
    TPML_HANDLE *handleList // OUT: list of handle
)
{
    TPMI_YES_NO more = NO;
    NV_REF iter = NV_REF_INIT;
    NV_REF currentAddr;
    TPM_HANDLE nvHandle;
    //
    pAssert(HandleGetType(handle) == TPM_HT_NV_INDEX);
    // Initialize output handle list
    handleList->count = 0;
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.

```
UINT32 NvCapGetPersistentAvail(
    void
)
{
    UINT32 availNVSpace;
    UINT32 counterNum = NvCapGetCounterNumber();
    UINT32 reserved = sizeof(NV_LIST_TERMINATOR);
    // Get the available space in NV storage
    availNVSpace = NvGetFreeBytes();
    if(counterNum < MIN_COUNTER_INDICES)
    {
        // Some space has to be reserved for counter objects
        reserved += (MIN_COUNTER_INDICES - counterNum) * NV_INDEX_COUNTER_SIZE;
        if(reserved > availNVSpace)
            availNVSpace = 0;
        else
            availNVSpace -= reserved;
    }
    return availNVSpace / NV_EVICT_OBJECT_SIZE;
}
```

8.4.5.28 NvCapGetCounterNumber()

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

```
UINT32 NvCapGetCounterNumber(
    void
)
{
    NV_REF iter = NV_REF_INIT;
    NV_REF currentAddr;
    UINT32 num = 0;
    //
    while((currentAddr = NvNextIndex(NULL, &iter)) != 0)
    {
        TPMA_NV attributes = NvReadNvIndexAttributes(currentAddr);
        if(IsNvCounterIndex(attributes))
            num++;
    }
    return num;
}
```

8.4.5.29 NvSetStartupAttributes()

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

```
static TPMA_NV NvSetStartupAttributes(
    TPMA_NV attributes,   // IN: attributes to change
    STARTUP_TYPE type     // IN: start up type
)
{
    // Clear read lock
```
CLEAR_ATTRIBUTE(attributes, TPMA_NV, READLOCKED);

// Will change a non counter index to the unwritten state if:
// a) TPMA_NV_CLEAR_STCLEAR is SET
// b) orderly and TPM Reset
if(!IsNvCounterIndex(attributes))
{
    if(IS_ATTRIBUTE(attributes, TPMA_NV, CLEAR_STCLEAR)
        || (IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY)
            && (type == SU_RESET)))
        CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITTEN);

    // Unlock any index that is not written or that does not have
    // TPMA_NV_WRITEDEFINE SET.
    if(!IS_ATTRIBUTE(attributes, TPMA_NV, WRITTEN)
        || !IS_ATTRIBUTE(attributes, TPMA_NV, WRITEDEFINE))
        CLEAR_ATTRIBUTE(attributes, TPMA_NV, WRITELOCKED);

    return attributes;
}

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.

BOOL NvEntityStartup(
    STARTUP_TYPE type        // IN: start up type
)
{
    NV_REF iter = NV_REF_INIT;
    NV_RAM_REF ramIter = NV_RAM_REF_INIT;
    NV_REF currentAddr;    // offset points to the current entity
    NV_RAM_REF currentRamAddr;
    TPM_HANDLE nvHandle;
    TPMA_NV attributes;

    // Restore RAM index data
    NvRead(s_indexOrderlyRam, NV_INDEX_RAM_DATA, sizeof(s_indexOrderlyRam));

    // Initialize the max NV counter value
    NvSetMaxCount(NvGetMaxCount());

    // If recovering from state save, do nothing else
    if(type == SU_RESUME)
        return TRUE;

    // Iterate all the NV Index to clear the locks
    while((currentAddr = NvNextIndex(&nvHandle, &iter)) != 0)
    {
        attributes = NvReadNvIndexAttributes(currentAddr);

        // If this is an orderly index, defer processing until loop below
        if(IS_ATTRIBUTE(attributes, TPMA_NV, ORDERLY))
            continue;

        // Set the attributes appropriate for this startup type
        attributes = NvSetStartupAttributes(attributes, type);
        NvWriteNvIndexAttributes(currentAddr, attributes);
1324    } // Iterate all the orderly indexes to clear the locks and initialize counters
1325    while((currentRamAddr = NvRamNext(&ramIter, NULL)) != 0)
1326    {
1327        attributes = NvReadRamIndexAttributes(currentRamAddr);
1328        attributes = NvSetStartupAttributes(attributes, type);
1329        // update attributes in RAM
1330        NvWriteRamIndexAttributes(currentRamAddr, attributes);
1331        // Set the lower bits in an orderly counter to 1 for a non-orderly startup
1332        if(IsNvCounterIndex(attributes) && (g_prevOrderlyState == SU_NONE_VALUE))
1333        {
1334            UINT64      counter;
1335            // Read the counter value last saved to NV.
1336            counter = BYTE_ARRAY_TO_UINT64(currentRamAddr + sizeof(NV_RAM_HEADER));
1337            // Set the lower bits of counter to 1's
1338            counter |= MAX_ORDERLY_COUNT;
1339            // Write back to RAM
1340            // NOTE: Do not want to force a write to NV here. The counter value will
1341            // stay in RAM until the next shutdown or rollover.
1342            UINT64_TO_BYTE_ARRAY(counter, currentRamAddr + sizeof(NV_RAM_HEADER));
1343        }
1344    }
1345
1346   return TRUE;
1347
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   // Get the available space in NV storage
1365   availNVSpace = NvGetFreeBytes();
1366   if(persistentNum < MIN_EVICT_OBJECTS)
1367   {
1368       // Some space has to be reserved for evict object. Adjust availNVSpace.
1369       reserved += (MIN_EVICT_OBJECTS - persistentNum) * NV_EVICT_OBJECT_SIZE;
1370   if(reserved > availNVSpace)
1371       availNVSpace = 0;
1372   else
1373       availNVSpace -= reserved;
1374   }
1375   // Compute the available space in RAM
1376   availRAMSpace = RAM_ORDERLY_END - NvRamGetEnd();
1377   // Return the min of counter number in NV and in RAM
1378   if(availNVSpace / NV_INDEX_COUNTER_SIZE
8.4.5.32 NvFindHandle()

dis 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.

```c
UINT64 NvFindHandle(
    TPM_HANDLE handle
) {
    NV_REF addr;
    NV_REF iter = NV_REF_INIT;
    TPM_HANDLE nextHandle;
    //
    while((addr = NvNext(&iter, &nextHandle)) != 0)
    {
        if(nextHandle == handle)
            break;
    }
    return addr;
}
```

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.

```c
UINT64 NvReadMaxCount(
    void
) {
    return s_maxCounter;
}
```

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.
void NvUpdateMaxCount(
    UINT64 count
)
{
    if(count > s_maxCounter)
        s_maxCounter = count;
}

8.4.6.4 NvSetMaxCount()

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

void NvSetMaxCount(
    UINT64 value
)
{
    s_maxCounter = value;
}

8.4.6.5 NvGetMaxCount()

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

UINT64 NvGetMaxCount(
    void
)
{
    NV_REF iter = NV_REF_INIT;
    NV_REF currentAddr;
    UINT64 maxCount;
    // Find the end of list marker and initialize the NV Max Counter value.
    while((currentAddr = NvNext(&iter, NULL )) != 0);
    // 'iter' should be pointing at the end of list marker so read in the current
    // value of the s_maxCounter.
    NvRead(&maxCount, iter + sizeof(UINT32), sizeof(maxCount));
    return maxCount;
}
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

```c
#define NV_C
#include "Tpm.h"
```

8.5.3 Functions

8.5.3.1 NvInitStatic()

This function initializes the static variables used in the NV subsystem.

```c
static void
NvInitStatic(
    void
)
{
    // In some implementations, the end of NV is variable and is set at boot time.
    // This value will be the same for each boot, but is not necessarily known
    // at compile time.
    s_evictNvEnd = (NV_REF)NV_MEMORY_SIZE;
    return;
```
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.

```c
void NvCheckState()
{
    int   func_return;
    //
    func_return = _plat__IsNvAvailable();
    if(func_return == 0)
    g_NvStatus = TPM_RC_SUCCESS;
    else if(func_return == 1)
    g_NvStatus = TPM_RC_NV_UNAVAILABLE;
    else
    g_NvStatus = TPM_RC_NV_RATE;
    return;
}
```

8.5.3.3 NvCommit

This is a wrapper for the platform function to commit pending NV writes.

```c
BOOL NvCommit()
{
    return (_plat__NvCommit() == 0);
}
```

8.5.3.4 NvPowerOn()

This function is called at _TPM_Init() to initialize the NV environment.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>all NV was initialized</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>the NV containing saved state had an error and TPM2_Startup(CLEAR) is required</td>
</tr>
</tbody>
</table>

```c
BOOL NvPowerOn()
{
    int   nvError = 0;
    // If power was lost, need to re-establish the RAM data that is loaded from
    // NV and initialize the static variables
    if(g_powerWasLost)
    {
        if((nvError = _plat__NVEnable(0)) < 0)
            FAIL(FATAL_ERROR_NV_UNRECOVERABLE);
    }
```
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.

```c
void
NvManufacture()
{
    void
    {
#if SIMULATION
        // Simulate the NV memory being in the erased state.
        _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
#endif
        // Initialize static variables
        NvInitStatic();
        // Clear the RAM used for Orderly Index data
        MemorySet(s_indexOrderlyRam, 0, RAM_INDEX_SPACE);
        // Write that Orderly Index data to NV
        NvUpdateIndexOrderlyData();
        // Initialize the next offset of the first entry in evict/index list to 0 (the
        // end of list marker) and the initial s_maxCounterValue;
        NvSetMaxCount(0);
        // Put the end of list marker at the end of memory. This contains the MaxCount
        // value as well as the end marker.
        NvWriteNvListEnd(NV_USER_DYNAMIC);
        return;
    }
}
```

8.5.3.6  NvRead()

This function is used to move reserved data from NV memory to RAM.

```c
void
NvRead()
{
    void
    {
        // Input type should be valid
        pAssert(nvOffset + size < NV_MEMORY_SIZE);
        _plat__NvMemoryRead(nvOffset, size, outBuffer);
        return;
    }
}
```

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.

```c
BOOL
NvWrite()
{
    UINT32
    {
        // IN: location in NV to receive data
```
8.5.3.8 NvUpdatePersistent()

This function is used to update a value in the PERSISTENT_DATA structure and commits the value to NV.

```c
void NvUpdatePersistent(
    UINT32 offset,  // IN: location in PERMANENT_DATA to be updated
    UINT32 size,    // IN: size of the value
    void *buffer    // IN: the new data
)
```

8.5.3.9 NvClearPersistent()

This function is used to clear a persistent data entry and commit it to NV.

```c
void NvClearPersistent(
    UINT32 offset,  // IN: the offset in the PERMANENT_DATA structure to be cleared (zeroed)
    UINT32 size     // IN: number of bytes to clear
)
```

8.5.3.10 NvReadPersistent()

This function reads persistent data to the RAM copy of the gp structure.

```c
void NvReadPersistent(
    void
)
```
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

```c
#define OBJECT_C
#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.

```c
void ObjectFlush(
    OBJECT *object
) {
    object->attributes.occupied = CLEAR;
}
```

8.6.3.2 ObjectSetInUse()

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

```c
void ObjectSetInUse(
    OBJECT *object
) {
    object->attributes.occupied = SET;
}
```

8.6.3.3 ObjectStartup()

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

```c
BOOL ObjectStartup(
    void
) {
    UINT32 i;
    // object slots initialization
    for(i = 0; i < MAX_LOADED_OBJECTS; i++) {
        // Set the slot to not occupied
        ObjectFlush(&s_objects[i]);
    }
}```
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().

```c
void ObjectCleanupEvict()
{
    UINT32 i;
    // This has to be iterated because a command may have two handles
    // and they may both be persistent.
    // This could be made to be more efficient so that a search is not needed.
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        // If an object is a temporary evict object, flush it from slot
        OBJECT *object = &s_objects[i];
        if(object->attributes.evict == SET)
            ObjectFlush(object);
    }
    return;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>handle references a loaded object</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>handle is not an object handle, or it does not reference to a loaded object</td>
</tr>
</tbody>
</table>

```c
BOOL IsObjectPresent(TPMI_DH_OBJECT handle) // IN: handle to be checked
{
    UINT32 slotIndex = handle - TRANSIENT_FIRST;
    // Since the handle is just an index into the array that is zero based, any
    // handle value outside of the range of:
    //    TRANSIENT_FIRST -- (TRANSIENT_FIRST + MAX_LOADED_OBJECTS - 1)
    // will now be greater than or equal to MAX_LOADED_OBJECTS
    if(slotIndex >= MAX_LOADED_OBJECTS)
        return FALSE;
    // Indicate if the slot is occupied
    return (s_objects[slotIndex].attributes.occupied == TRUE);
}
```

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>object is an HMAC, hash, or event sequence object</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>object is not an HMAC, hash, or event sequence object</td>
</tr>
</tbody>
</table>

### 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.

```c
OBJECT* HandleToObject(TPMI_DH_OBJECT handle) // IN: handle of the object
{
    UINT32              index;
    // In this implementation, the handle is determined by the slot occupied by the object.
    index = handle - TRANSIENT_FIRST;
    pAssert(index < MAX_LOADED_OBJECTS);
    pAssert(s_objects[index].attributes.occupied);
    return &s_objects[index];
}
```

### 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.

```c
void GetQualifiedName(TPMI_DH_OBJECT handle, TPM2B_NAME *qualifiedName) // IN: handle of the object // OUT: qualified name of the object
{
    OBJECT *object;
    //
    switch(HandleGetType(handle))
    {
      case TPM_HT_PERMANENT:
```
qualifiedName->t.size = sizeof(TPM_HANDLE);
    UINT32_TO_BYTE_ARRAY(handle, qualifiedName->t.name);
    break;
  case TPM_HT_TRANSIENT:
    object = HandleToObject(handle);
    if(object == NULL || object->publicArea.nameAlg == TPM_ALG_NULL)
      qualifiedName->t.size = 0;
    else
      // Copy the name
      *qualifiedName = object->qualifiedName;
      break;
  default:
    FAIL(FATAL_ERROR_INTERNAL);
  }
  return;
}

8.6.3.9 ObjectGetHierarchy()

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

TPMI_RH_HIERARCHY
ObjectGetHierarchy(
  OBJECT          *object    // IN :object
)
{
  if(object->attributes.spsHierarchy)
    return TPM_RH_OWNER;
  else if(object->attributes.epsHierarchy)
    return TPM_RH_ENDORSEMENT;
  else if(object->attributes.ppsHierarchy)
    return TPM_RH_PLATFORM;
  else
    return TPM_RH_NULL;
}

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.

TPMI_RH_HIERARCHY
GetHierarchy(
  TPMI_DH_OBJECT   handle    // IN :object handle
)
{
  OBJECT          *object = HandleToObject(handle);
  //
  return ObjectGetHierarchy(object);
}
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>no open slot found</td>
</tr>
<tr>
<td>!= NULL</td>
<td>pointer to available slot</td>
</tr>
</tbody>
</table>

```c
OBJECT * FindEmptyObjectSlot(
    TPMI_DH_OBJECT *handle  // OUT: (optional)
) {
    UINT32 i;
    OBJECT *object;

    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
        object = &s_objects[i];
    if(object->attributes.occupied == CLEAR)
        if(handle)
            *handle = i + TRANSIENT_FIRST;
        // Initialize the object attributes
        MemorySet(object->attributes, 0, sizeof(OBJECT_ATTRIBUTES));
    return object;
}
```

8.6.3.12 ObjectAllocateSlot()

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

```c
OBJECT * ObjectAllocateSlot(
    TPMI_DH_OBJECT *handle  // OUT: handle of allocated object
) {
    OBJECT *object = FindEmptyObjectSlot(handle);
    if(object != NULL)
        ObjectSetInUse(object);
    return object;
}
```

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.

```c
void ObjectSetLoadedAttributes(
    OBJECT *object,          // IN: object attributes to finalize
    TPM_HANDLE parentHandle  // IN: the parent handle
)`
OBJECT *parent = HandleToObject(parentHandle);

TPMA_OBJECT objectAttributes = object->publicArea.objectAttributes;

// Copy the stClear attribute from the public area. This could be overwritten
// if the parent has stClear SET
object->attributes.stClear =
    IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear);

if (parent == NULL)
{
    object->attributes.primary = SET;
    switch(parentHandle)
    {
        case TPM_RH_ENDORSEMENT:
            object->attributes.epsHierarchy = SET;
            break;
        case TPM_RH_OWNER:
            object->attributes.spsHierarchy = SET;
            break;
        case TPM_RH_PLATFORM:
            object->attributes.ppsHierarchy = SET;
            break;
        default:
            // Treat the temporary attribute as a hierarchy
            object->attributes.temporary = SET;
            object->attributes.primary = CLEAR;
            break;
    }
    else
    {
        // is this a stClear object
        object->attributes.stClear =
            (IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, stClear)
             || (parent->attributes.stClear == SET));
        object->attributes.epsHierarchy = parent->attributes.epsHierarchy;
        object->attributes.spsHierarchy = parent->attributes.spsHierarchy;
        object->attributes.ppsHierarchy = parent->attributes.ppsHierarchy;
        // An object is temporary if its parent is temporary or if the object
        // is external
        object->attributes.temporary = parent->attributes.temporary
            || object->attributes.external;
    }
}
// If this is an external object, set the QN == name but don't SET other
// key properties ('parent' or 'derived')
if(object->attributes.external)
    object->qualifiedName = object->name;
else
{
    // check attributes for different types of parents
    if(IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, restricted)
        && !object->attributes.publicOnly
        && IS_ATTRIBUTE(objectAttributes, TPMA_OBJECT, decrypt)
        && object->publicArea.nameAlg != TPM_ALG_NULL)
    {
        // This is a parent. If it is not a KEYEDHASH, it is an ordinary parent.
        // Otherwise, it is a derivation parent.
        if(object->publicArea.type == TPM_ALG_KEYEDHASH)
            object->attributes.derivation = SET;
        else
            object->attributes.isParent = SET;
    }
    ComputeQualifiedName(parentHandle, object->publicArea.nameAlg,
        &object->name, &object->qualifiedName);
258     } // Set slot occupied
259     ObjectSetInUse(object);
260     return;
261 }

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.

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

#else

// If this is an RSA key, then expand the private exponent.
// Note: ObjectLoad() is only called by TPM2_Import() if the parent is fixedTPM.
// For any key that does not have a fixedTPM parent, the exponent is computed
// whenever it is loaded
if((publicArea->type == TPM_ALG_RSA) && (sensitive != NULL))
{
    result = CryptRsaLoadPrivateExponent(publicArea, sensitive);
    if(result != TPM_RC_SUCCESS)
        return result;

#endif // ALG_RSA

// See if there is an object to populate
if((result == TPM_RC_SUCCESS) && (object != NULL))
{
    // Initialize public
    object->publicArea = *publicArea;
    // Copy sensitive if there is one
    if(sensitive != NULL)
        object->attributes.publicOnly = SET;
    else
        object->sensitive = *sensitive;
    // Set the name, if one was provided
    if(name != NULL)
        object->name = *name;
    else
        object->name.t.size = 0;

    return result;

}

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.

static HASH_OBJECT *
AllocateSequenceSlot(
    TPM_HANDLE      *newHandle,  // OUT: receives the allocated handle
    TPM2B_AUTH      *auth        // IN: the authValue for the slot
)
{
    HASH_OBJECT      *object = (HASH_OBJECT *)ObjectAllocateSlot(newHandle);

    // Validate that the proper location of the hash state data relative to the
    // object state data. It would be good if this could have been done at compile
    // time but it can't so do it in something that can be removed after debug.
    cAssert(offsetof(HASH_OBJECT, auth) == offsetof(OBJECT, publicArea.authPolicy));

    if(object != NULL)
    {
        // Set the common values that a sequence object shares with an ordinary object
        // First, clear all attributes
        MemorySet(&object->objectAttributes, 0, sizeof(TPMA_OBJECT));

        // The type is TPM_ALG_NULL
        object->type = TPM_ALG_NULL;

        // This has no name algorithm and the name is the Empty Buffer
        object->nameAlg = TPM_ALG_NULL;

        return result;
    }
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>if there is no free slot for an object</td>
</tr>
</tbody>
</table>

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     // Try to allocate a slot for new object
398     hmacObject = AllocateSequenceSlot(newHandle, auth);
399     if(hmacObject == NULL)
400         return TPM_RC_OBJECT_MEMORY;
401     // Set HMAC sequence bit
402     hmacObject->attributes.hmacSeq = SET;
403
404     #if !SMAC_IMPLEMENTED
405         if(CryptHmacStart(&hmacObject->state.hmacState, hashAlg,
406             keyObject->sensitive.sensitive.bits.b.size,
407             keyObject->sensitive.sensitive.bits.b.buffer) == 0)
408             else
409                 if(CryptMacStart(&hmacObject->state.hmacState,
410                     &keyObject->publicArea.parameters,
411                     hashAlg, &keyObject->sensitive.sensitive.any.b) == 0)
412                     // SMAC_IMPLEMENTED
413                         return TPM_RC_FAILURE;
414             return TPM_RC_SUCCESS;
415     #endif // SMAC_IMPLEMENTED
416 }
417 #endif

8.6.3.17  **ObjectCreateHashSequence()**

This function creates a hash sequence object.
8.6.3.18 ObjectCreateEventSequence()

This function creates an event sequence object.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>if there is no free slot for an object</td>
</tr>
</tbody>
</table>

```c
TPM_RC
ObjectCreateEventSequence(
    TPM2B_AUTH      *auth,   // IN: authValue
    TPMI_DH_OBJECT  *newHandle // OUT: sequence object handle
)
{
    HASH_OBJECT         *hashObject = AllocateSequenceSlot(newHandle, auth);
    UINT32               count;
    TPM_ALG_ID           hash;
    // See if slot allocated
    if(hashObject == NULL)
        return TPM_RC_OBJECT_MEMORY;
    // Set the event sequence attribute
    hashObject->attributes.eventSeq = SET;
    // Initialize hash states for each implemented PCR algorithms
    for(count = 0; (hash = CryptHashGetAlgByIndex(count)) != TPM_ALG_NULL; count++)
        CryptHashStart(&hashObject->state.hashState[count], hash);
    return TPM_RC_SUCCESS;
}
```

8.6.3.19 ObjectTerminateEvent()

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

```c
void
ObjectTerminateEvent(
    void
)
```

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>if there is no free slot for an object</td>
</tr>
</tbody>
</table>

```c
void
ObjectTerminateEvent(
    void
)
{
}
```
8.6.3.20 ObjectContextLoad()

This function loads an object from a saved object context.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>if there is no free slot for an object</td>
</tr>
<tr>
<td>!= NULL</td>
<td>points to the loaded object</td>
</tr>
</tbody>
</table>

```c
87 OBJECT * ObjectContextLoad(
88   ANY_OBJECT_BUFFER   *object, // IN: pointer to object structure in saved context
89   TPMI_DH_OBJECT      *handle  // OUT: object handle
90 )
91 {
92   OBJECT      *newObject = ObjectAllocateSlot(handle);
93   // Try to allocate a slot for new object
94   if(newObject != NULL)
95   {
96     // Copy the first part of the object
97     MemoryCopy(newObject, object, offsetof(HASH_OBJECT, state));
98     // See if this is a sequence object
99     if(ObjectIsSequence(newObject))
100    {
101       // If this is a sequence object, import the data
102       SequenceDataImport((HASH_OBJECT *)newObject,
103                            (HASH_OBJECT_BUFFER *)object);
104    }
105    else
106    {
107      // Copy input object data to internal structure
108      MemoryCopy(newObject, object, sizeof(OBJECT));
109    }
110   }
111   return newObject;
112 }
```
8.6.3.21 FlushObject()

This function frees an object slot.
This function requires that the object is loaded.

```c
void FlushObject(
    TPMI_DH_OBJECT   handle   // IN: handle to be freed
)
{
    UINT32   index = handle - TRANSIENT_FIRST;
    // pAssert(index < MAX_LOADED_OBJECTS);
    // Clear all the object attributes
    MemorySet((BYTE*)&(s_objects[index].attributes),
                0, sizeof(OBJECT_ATTRIBUTES));
    return;
}
```

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.

```c
void ObjectFlushHierarchy(
    TPMI_RH_HIERARCHY    hierarchy   // IN: hierarchy to be flush
)
{
    UINT16   i;
    // iterate object slots
    for(i = 0; i < MAX_LOADED_OBJECTS; i++)
    {
        if(s_objects[i].attributes.occupied)    // If found an occupied slot
            switch(hierarchy)
            {
            case TPM_RH_PLATFORM:
                if(s_objects[i].attributes.ppsHierarchy == SET)
                    s_objects[i].attributes.occupied = FALSE;
                break;
            case TPM_RH_OWNER:
                if(s_objects[i].attributes.spsHierarchy == SET)
                    s_objects[i].attributes.occupied = FALSE;
                break;
            case TPM_RH_ENDORSEMENT:
                if(s_objects[i].attributes.epsHierarchy == SET)
                    s_objects[i].attributes.occupied = FALSE;
                break;
            default:
                FAIL(FATAL_ERROR_INTERNAL);
                break;
            }
    }
    return;
}
```
8.6.3.23  ObjectLoadEvict()

This function loads a persistent object into a transient object slot.

This function requires that \textit{handle} is associated with a persistent object.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_HANDLE</td>
<td>the persistent object does not exist or the associated hierarchy is disabled.</td>
</tr>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>no object slot</td>
</tr>
</tbody>
</table>

```c
564  TPM_RC  ObjectLoadEvict(
565   TPM_HANDLE      *handle,       // IN:OUT: evict object handle. If success, it
566   COMMAND_INDEX   commandIndex   // IN: the command being processed
567   )
568  {
569   TPM_RC  result;
570   TPM_HANDLE  evictHandle = *handle;   // Save the evict handle
571   OBJECT  *object;
572   // If this is an index that references a persistent object created by
573   // the platform, then return TPM_RC_HANDLE if the phEnable is FALSE
574   if(*handle >= PLATFORM_PERSISTENT)
575     {
576       // belongs to platform
577       if(g_phEnable == CLEAR)
578         return TPM_RC_HANDLE;
579     }  // belongs to owner
580   else if(gc.shEnable == CLEAR)
581     return TPM_RC_HANDLE;
582   // Try to allocate a slot for an object
583   object = ObjectAllocateSlot(handle);
584   if(object == NULL)
585     return TPM_RC_OBJECT_MEMORY;
586   // Copy persistent object to transient object slot. A TPM_RC_HANDLE
587   // may be returned at this point. This will mark the slot as containing
588   // a transient object so that it will be flushed at the end of the
589   // command
590   result = NvGetEvictObject(evictHandle, object);
591   // Bail out if this failed
592   if(result != TPM_RC_SUCCESS)
593     return result;
594   // check the object to see if it is in the endorsement hierarchy
595   // if it is and this is not a TPM2_EvictControl() command, indicate
596   // that the hierarchy is disabled.
597   if(ObjectGetHierarchy(object) == TPM_RH_ENDORSEMENT
598     && gc.ehEnable == CLEAR
599     && GetCommandCode(commandIndex) != TPM_CC_EvictControl)
600     return TPM_RC_HANDLE;
601   return result;
602  }
```

8.6.3.24  ObjectComputeName()

This does the name computation from a public area (can be marshaled or not).
TPM2B_NAME *
ObjectComputeName(
    UINT32 size,     // IN: the size of the area to digest
    BYTE *publicArea, // IN: the public area to digest
    TPM_ALG_ID nameAlg, // IN: the hash algorithm to use
    TPM2B_NAME *name   // OUT: Computed name
)
{
    // Hash the publicArea into the name buffer leaving room for the nameAlg
    name->t.size = CryptHashBlock(nameAlg, size, publicArea,
        sizeof(name->t.name) - 2,
        &name->t.name[2]);
    // set the nameAlg
    UINT16_TO_BYTE_ARRAY(nameAlg, name->t.name);
    name->t.size += 2;
    return name;
}

8.6.3.25 PublicMarshalAndComputeName()
This function computes the Name of an object from its public area.

TPM2B_NAME *
PublicMarshalAndComputeName(
    TPMT_PUBLIC *publicArea, // IN: public area of an object
    TPM2B_NAME *name       // OUT: name of the object
)
{
    // Will marshal a public area into a template. This is because the internal
    // format for a TPM2B_PUBLIC is a structure and not a simple BYTE buffer.
    TPM2B_TEMPLATE marshaled;    // this is big enough to hold a
    BYTE *buffer = (BYTE *)&marshaled.t.buffer;
    // if the nameAlg is NULL then there is no name.
    if(publicArea->nameAlg == TPM_ALG_NULL)
        name->t.size = 0;
    else
    {
        // Marshal the public area into its canonical form
        marshaled.t.size = TPMT_PUBLIC_Marshal(publicArea, &buffer, NULL);
        // and compute the name
        ObjectComputeName(marshaled.t.size, marshaled.t.buffer,
            publicArea->nameAlg, name);
        return name;
    }
}

8.6.3.26 ComputeQualifiedName()
This function computes the qualified name of an object.

void
ComputeQualifiedName(
    TPM_HANDLE parentHandle, // IN: parent’s handle
    TPM_ALG_ID nameAlg,     // IN: name hash
    TPM2B_NAME *name,       // IN: name of the object
    TPM2B_NAME *qualifiedName // OUT: qualified name of the object
)
{
    HASH_STATE hashState;    // hash state
    TPM2B_NAME parentName;
    //
if (parentHandle == TPM_RH_UNASSIGNED)
{
    MemoryCopy2B(&qualifiedName->b, &name->b, sizeof(qualifiedName->t.name));
    *qualifiedName = *name;
}
else
{
    GetQualifiedName(parentHandle, &parentName);
    // QN_A = hash_A (QN of parent || NAME_A)
    // Start hash
    qualifiedName->t.size = CryptHashStart(&hashState, nameAlg);
    // Add parent’s qualified name
    CryptDigestUpdate2B(&hashState, &parentName.b);
    // Add self name
    CryptDigestUpdate2B(&hashState, &name->b);
    // Complete hash leaving room for the name algorithm
    CryptHashEnd(&hashState, qualifiedName->t.size,
                 &qualifiedName->t.name[2]);
    UINT16_TO_BYTE_ARRAY(nameAlg, qualifiedName->t.name);
    qualifiedName->t.size += 2;
}
return;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>object is a storage key</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>object is not a storage key</td>
</tr>
</tbody>
</table>

BOOL ObjectIsStorage(TPMI_DH_OBJECT handle // IN: object handle)
{
    OBJECT *object = HandleToObject(handle);
    TPMT_PUBLIC *publicArea = ((object != NULL) ? &object->publicArea : NULL);
    //
    return (publicArea != NULL
            && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
            && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
            && !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
            && (object->publicArea.type == ALG_RSA_VALUE
                || object->publicArea.type == ALG_ECC_VALUE));
}

8.6.3.28 ObjectCapGetLoaded()

This function returns a 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

<table>
<thead>
<tr>
<th>YES</th>
<th>if there are more handles available</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO

ObjectCapGetLoaded(
    TPMI_DH_OBJECT handle,  // IN: start handle
    UINT32 count,          // IN: count of returned handles
    TPML_HANDLE *handleList // OUT: list of handle
)
{
    TPMI_YES_NO more = NO;
    UINT32 i;
    // pAssert(HandleGetType(handle) == TPM_HT_TRANSIENT);
    // Initialize output handle list
    handleList->count = 0;
    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
    // Iterate object slots to get loaded object handles
    for(i = handle - TRANSIENT_FIRST; i < MAX_LOADED_OBJECTS; i++)
    {
        if(s_objects[i].attributes.occupied == TRUE)
        {
            // A valid transient object can not be the copy of a persistent object
            pAssert(s_objects[i].attributes.evict == CLEAR);
            if(handleList->count < count)
            {
                // If we have not filled up the return list, add this object
                // handle to it
                handleList->handle[handleList->count] = i + TRANSIENT_FIRST;
                handleList->count++;
            }
            else
            {
                // If the return list is full but we still have loaded object
                // available, report this and stop iterating
                more = YES;
                break;
            }
        }
    }
    return more;
}
```

### 8.6.3.29 ObjectCapGetTransientAvail()

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

```c
UINT32
ObjectCapGetTransientAvail(
    void
)
{
    UINT32 num = 0;
```
// Iterate object slot to get the number of unoccupied slots
for (i = 0; i < MAX_LOADED_OBJECTS; i++)
{
    if (s_objects[i].attributes.occupied == FALSE) num++;
}
return num;

8.6.3.30 ObjectGetPublicAttributes()

Returns the attributes associated with an object handles.

TPMA_OBJECT
ObjectGetPublicAttributes(
    TPM_HANDLE handle
) {
    return HandleToObject(handle)->publicArea.objectAttributes;
}

OBJECT_ATTRIBUTES
ObjectGetProperties(
    TPM_HANDLE handle
) {
    return HandleToObject(handle)->attributes;
}
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 \texttt{g\_hashPcrMap} is a bit array that indicates which of the PCR are implemented. The \texttt{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

\begin{verbatim}
#define PCR_C
#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.

\begin{verbatim}
static const PCR_Attributes s_initAttributes[] =
{
  // PCR 0 - 15, static RTM
  {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
  {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
  {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F}, {1, 0, 0x1F},
  {0, 0x0F, 0x1F}, // PCR 16, Debug
  {0, 0x10, 0x1C}, // PCR 17, Locality 4
  {0, 0x10, 0x1C}, // PCR 18, Locality 3
  {0, 0x10, 0x0C}, // PCR 19, Locality 2
  {0, 0x14, 0x0E}, // PCR 20, Locality 1
  {0, 0x14, 0x04}, // PCR 21, Dynamic OS
  {0, 0x14, 0x04}, // PCR 22, Dynamic OS
  {0, 0x0F, 0x1F}, // PCR 23, Application specific
  {0, 0x0F, 0x1F} // PCR 24, testing policy
};
\end{verbatim}

8.7.3 Functions

8.7.3.1 PCRBelongsAuthGroup()

This function indicates if a PCR belongs to a group that requires an \texttt{authValue} in order to modify the PCR. If it does, \texttt{groupId} is set to value of the group index. This feature of PCR is decided by the platform specification.

\begin{verbatim}
BOOL PCRBelongsAuthGroup(
  TPMI_DH_PCR handle,     // IN: handle of PCR
  UINT32 *groupId           // OUT: group index if PCR belongs a
)
\end{verbatim}
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 #if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0
31 // Platform specification determines to which authorization group a PCR belongs
32 // (if any). In this implementation, we assume there is only
33 // one authorization group which contains PCR[20-22]. If the platform
34 // specification requires differently, the implementation should be changed
35 // accordingly
36 if(handle >= 20 && handle <= 22)
37 {
38   *groupIndex = 0;
39   return TRUE;
40 }
41 #endif
42 return FALSE;
43 }

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>PCR belongs a policy group</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>PCR does not belong a policy group</td>
</tr>
</tbody>
</table>

４６
４７ BOOL
４８ PCRBelongsPolicyGroup(  
４９   TPMI_DH_PCR      handle,     // IN: handle of PCR  
５０   UINT32          *groupIndex  // OUT: group index if PCR belongs a group that  
５１   // allows policy. If PCR does not belong to  
５２   // a policy group, the value in this  
５３   // parameter is invalid  
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 PCR[20-22]. 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

```c
static BOOL PCRBelongsTCBGroup(
    TPMI_DH_PCR handle // IN: handle of PCR
) {
    #if ENABLE_PCR_NO_INCREMENT == YES
        // Platform specification decides if a PCR belongs to a TCB group. In this
        // implementation, we assume PCR[20-22] belong to TCB group. If the platform
        // specification requires differently, the implementation should be
        // changed accordingly
        if(handle >= 20 && handle <= 22)
            return TRUE;
    #endif
    return FALSE;
}
```

### 8.7.3.4 PCRPolicyIsAvailable()

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>the PCR should be authorized by policy</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>the PCR does not allow policy</td>
</tr>
</tbody>
</table>

```c
BOOL PCRPolicyIsAvailable(
    TPMI_DH_PCR handle // IN: PCR handle
) {
    UINT32 groupIndex;
    return PCRBelongsPolicyGroup(handle, &groupIndex);
}
```

### 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.

```c
TPM2B_AUTH * PCRGetAuthValue(
    TPMI_DH_PCR handle // IN: PCR handle
) {
    UINT32 groupIndex;
    if(PCRBelongsAuthGroup(handle, &groupIndex))
        return &gc.pcrAuthValues.auth[groupIndex];
    else
        return NULL;
}
```
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.

```c
TPMI_ALG_HASH
PCRGetAuthPolicy(
    TPMI_DH_PCR  handle,  // IN: PCR handle
    TPM2B_DIGEST *policy   // OUT: policy of PCR
)
{
    UINT32           groupIndex;
    if(PCRBelongsPolicyGroup(handle, &groupIndex))
    {
        *policy = gp.pcrPolicies.policy[groupIndex];
        return gp.pcrPolicies.hashAlg[groupIndex];
    }
    else
    {
        policy->t.size = 0;
        return TPM_ALG_NULL;
    }
}
```

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.

```c
void
PCRSimStart(
    void
)
{
    UINT32  i;
    // We need to give an initial configuration on allocated PCR before
    // receiving any TPM2_PCR_Allocate command to change this configuration
    for(i = 0; i < NUM_POLICY_PCR_GROUP; i++)
    {
        gp.pcrPolicies.hashAlg[i] = TPM_ALG_NULL;
        gp.pcrPolicies.policy[i].t.size = 0;
    }
    // When the simulation environment starts, we allocate all the PCRs
    for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
    {
        gc.pcrAuthValues.auth[i].t.size = 0;
    }
    gp.pcrAllocated.count = 0; gp.pcrAllocated.count < HASH_COUNT;
    for(i = 0; i < NUM_AUTHVALUE_PCR_GROUP; i++)
    {
        gp.pcrAllocated.pcrSelections[i].hash
            = CryptHashGetAlgByIndex(gp.pcrAllocated.count);
        gp.pcrAllocated.count++;
    }
    gp.pcrAllocated.pcrSelections[i].sizeofSelect
```
157     = PCR_SELECT_MAX;
158     for (i = 0; i < PCR_SELECT_MAX; i++)
159         gp.pcrAllocated.pcrSelections[gp.pcrAllocated.count].pcrSelect[i] = 0xFF;
160     }
161 }
162 // Store the initial configuration to NV
163 NV_SYNC_PERSISTENT(pcrPolicies);
164 NV_SYNC_PERSISTENT(pcrAllocated);
165 return;
166 }

8.7.3.8 GetSavedPcrPointer()

This function returns the address of an array of state saved PCR based on the hash algorithm.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>no such algorithm</td>
</tr>
<tr>
<td>!NULL</td>
<td>pointer to the 0th byte of the 0th PCR</td>
</tr>
</tbody>
</table>

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 #if ALG_SHA512
194         case ALG_SHA512_VALUE:
195             retVal = gc.pcrSave.sha512[pcrIndex];
196             break;
197 #endif
198 #if ALG_SM3_256
199         case ALG_SM3_256_VALUE:
200             retVal = gc.pcrSave.sm3_256[pcrIndex];
201             break;
202 #endif
203     default:
204         FAIL(FATAL_ERROR_INTERNAL);
205     }
206     return retVal;
207 }
8.7.3.9 PcrIsAllocated()

This function indicates if a PCR number for the particular hash algorithm is allocated.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>PCR is allocated</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>PCR is not allocated</td>
</tr>
</tbody>
</table>

```c
BOOL PcrIsAllocated(
    UINT32 pcr,  // IN: The number of the PCR
    TPMI_ALG_HASH hashAlg  // IN: The PCR algorithm
)
{
    UINT32 i;
    BOOL allocated = FALSE;
    if(pcr < IMPLEMENTATION_PCR)
    {
        for(i = 0; i < gp.pcrAllocated.count; i++)
        {
            if(gp.pcrAllocated.pcrSelections[i].hash == hashAlg)
            {
                if(((gp.pcrAllocated.pcrSelections[i].pcrSelect[pcr / 8]) & (1 << (pcr % 8))) != 0)
                allocated = TRUE;
                else
                allocated = FALSE;
                break;
            }
        }
        return allocated;
    }
}
```

8.7.3.10 GetPcrPointer()

This function returns the address of an array of PCR based on the hash algorithm.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>no such algorithm</td>
</tr>
<tr>
<td>!= NULL</td>
<td>pointer to the 0th byte of the 0th PCR</td>
</tr>
</tbody>
</table>

```c
static BYTE * GetPcrPointer(
    TPM_ALG_ID alg,      // IN: algorithm for bank
    UINT32 pcrNumber     // IN: PCR number
)
{
    static BYTE *pcr = NULL;
    if(!PcrIsAllocated(pcrNumber, alg))
        return NULL;
    switch(alg)
    {
        #if ALG_SHA1
        case ALG_SHA1_VALUE:
            pcr = s_pcrs[pcrNumber].sha1Pcr;
            break;
        #endif
    }
    return pcr;
}
```
8.7.3.11 IsPcrSelected()

This function indicates if an indicated PCR number is selected by the bit map in selection.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>PCR is selected</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>PCR is not selected</td>
</tr>
</tbody>
</table>

static BOOL IsPcrSelected(
UINT32 pcr,  // IN: The number of the PCR
TPMS_PCR_SELECTION *selection  // IN: The selection structure
)
{
    BOOL selected;
    selected = (pcr < IMPLEMENTATION_PCR
                && ((selection->pcrSelect[pcr / 8]) & (1 << (pcr % 8))) != 0);
    return selected;
}

8.7.3.12 FilterPcr()

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

static void FilterPcr(
    TPMS_PCR_SELECTION *selection  // IN: input PCR selection
    )
{
    UINT32 i;
    TPMS_PCR_SELECTION *allocated = NULL;
    // If size of select is less than PCR_SELECT_MAX, zero the unspecified PCR
for (i = selection->sizeofSelect; i < PCR_SELECT_MAX; i++)
    selection->pcrSelect[i] = 0;

// Find the internal configuration for the bank
for (i = 0; i < gp.pcrAllocated.count; i++)
{
    if (gp.pcrAllocated.pcrSelections[i].hash == selection->hash)
    {
        allocated = &gp.pcrAllocated.pcrSelections[i];
        break;
    }
}
for (i = 0; i < selection->sizeofSelect; i++)
{
    if (allocated == NULL)
    {
        // If the required bank does not exist, clear input selection
        selection->pcrSelect[i] = 0;
    }

    selection->pcrSelect[i] &= allocated->pcrSelect[i];
}
return;

void
PcrDrtm(
    const TPMI_DH_PCR        pcrHandle,    // IN: the index of the PCR to be
    const TPMI_ALG_HASH      hash,        // IN: the bank identifier
    const TPM2B_DIGEST      *digest       // IN: the digest to modify the PCR
)
{
    BYTE        *pcrData = GetPcrPointer(hash, pcrHandle);

    if (pcrData != NULL)
    {
        // Rest the PCR to zeros
        MemorySet(pcrData, 0, digest->t.size);

        // if the TPM has not started, then set the PCR to 0...04 and then extend
        if (!TPMIsStarted())
        {
            pcrData[digest->t.size - 1] = 4;
        }

        // Now, extend the value
        PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
    }
}

void
PCR_ClearAuth(
}

8.7.3.13   PcrDrtm()

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

void
PcrDrtm(
    const TPMI_DH_PCR        pcrHandle,    // IN: the index of the PCR to be
    const TPMI_ALG_HASH      hash,        // IN: the bank identifier
    const TPM2B_DIGEST      *digest       // IN: the digest to modify the PCR
)
{
    BYTE        *pcrData = GetPcrPointer(hash, pcrHandle);

    if (pcrData != NULL)
    {
        // Rest the PCR to zeros
        MemorySet(pcrData, 0, digest->t.size);

        // if the TPM has not started, then set the PCR to 0...04 and then extend
        if (!TPMIsStarted())
        {
            pcrData[digest->t.size - 1] = 4;
        }

        // Now, extend the value
        PCRExtend(pcrHandle, hash, digest->t.size, (BYTE *)digest->t.buffer);
    }
}

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().

void
PCR_ClearAuth(
}
8.7.3.15 PCRStartup()

This function initializes the PCR subsystem at TPM2_Startup().

```c
BOOL
PCRStartup(
STARTUP_TYPE type,     // IN: startup type
BYTE locality       // IN: startup locality
)
{
  UINT32 pcr, j;
  UINT32 saveIndex = 0;

  g_pcrReConfig = FALSE;

  // Don’t test for SU_RESET because that should be the default when nothing
  // else is selected
  if(type != SU_RESUME && type != SU_RESTART)
  {
    // PCR generation counter is cleared at TPM_RESET
    gr.pcrCounter = 0;
  }

  // Initialize/Restore PCR values
  for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
  {
    // On resume, need to know if this PCR had its state saved or not
    UINT32 stateSaved;

    if(type == SU_RESUME
     && _s_initAttributes[pcr].stateSave == SET)
    {
      stateSaved = 1;
    }
    else
    {
      stateSaved = 0;
      PCRChanged(pcr);
    }

    // If this is the H-CRTM PCR and we are not doing a resume and we
    // had an H-CRTM event, then we don’t change this PCR
    if(pcr == HCRTM_PCR && type != SU_RESUME && g_DrtmPreStartup == TRUE)
      continue;

    // Iterate each hash algorithm bank
    for(j = 0; j < gp.pcrAllocated.count; j++)
    {
      TPMI_ALG_HASH hash = gp.pcrAllocated.pcrSelections[j].hash;
      BYTE *pcrData = GetPcrPointer(hash, pcr);
      UINT16 pcrSize = CryptHashGetDigestSize(hash);
```
if(pcrData != NULL)
{
    // if state was saved
    if(stateSaved == 1)
    {
        // Restore saved PCR value
        BYTE *pcrSavedData;
        pcrSavedData = GetSavedPcrPointer(
            gp.pcrAllocated.pcrSelections[j].hash,
            saveIndex);
        if(pcrSavedData == NULL)
            return FALSE;
        MemoryCopy(pcrData, pcrSavedData, pcrSize);
    }
    else
    // PCR was not restored by state save
    {
        // If the reset locality of the PCR is 4, then
        // the reset value is all one’s, otherwise it is
        // all zero.
        if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
            MemorySet(pcrData, 0xFF, pcrSize);
        else
        {
            MemorySet(pcrData, 0, pcrSize);
            if(pcr == HCRTM_PCR)
                pcrData[pcrSize - 1] = locality;
        }
    }
    saveIndex += stateSaved;
}
// Reset authValues on TPM2_Startup(CLEAR)
if(type != SU_RESUME)
    PCR_ClearAuth();
return TRUE;

8.7.3.16 PCRStateSave()

This function is used to save the PCR values that will be restored on TPM Resume.

void
PCRStateSave(
    TPM_SU type  // IN: startup type
)
{
    UINT32 pcr, j;
    UINT32 saveIndex = 0;

    // if state save CLEAR, nothing to be done. Return here
    if(type == TPM_SU_CLEAR)
        return;

    // Copy PCR values to the structure that should be saved to NV
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        UINT32 stateSaved = (s_initAttributes[pcr].stateSave == SET) ? 1 : 0;

        // Iterate each hash algorithm bank
        for(j = 0; j < gp.pcrAllocated.count; j++)
        {
            BYTE *pcrData;
UINT32 pcrSize;

pcrData = GetPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, pcr);

if(pcrData != NULL)
{
    pcrSize = CryptHashGetDigestSize(gp.pcrAllocated.pcrSelections[j].hash);
    if(stateSaved == 1)
    {
        // Restore saved PCR value
        BYTE *pcrSavedData;
        pcrSavedData = GetSavedPcrPointer(gp.pcrAllocated.pcrSelections[j].hash, saveIndex);
        MemoryCopy(pcrSavedData, pcrData, pcrSize);
    }
    saveIndex += stateSaved;
}

return;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>PCR is state saved</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>PCR is not state saved</td>
</tr>
</tbody>
</table>

BOOL PCRIsStateSaved(

    TPMI_DH_PCR handle   // IN: PCR handle to be extended
)
{
    UINT32 pcr = handle - PCR_FIRST;
    if(s_initAttributes[pcr].stateSave == SET)
        return TRUE;
    else
        return FALSE;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>TPM2_PCR_Reset() is allowed</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>TPM2_PCR_Reset() is not allowed</td>
</tr>
</tbody>
</table>

BOOL PCRIsResetAllowed(
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().

```c
void
PCRChanged(
    TPM_HANDLE       pcrHandle   // IN: the handle of the PCR that changed.
)
{
    // For the reference implementation, the only change that does not cause
    // increment is a change to a PCR in the TCB group.
    if((pcrHandle == 0) || !PCRBelongsTCBGroup(pcrHandle))
    {
        gr.pcrCounter++;
        if(gr.pcrCounter == 0)
            FAIL(FATAL_ERROR COUNTER_OVERFLOW);
    }
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>extend is allowed</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>extend is not allowed</td>
</tr>
</tbody>
</table>

```c
BOOL
PCRIsExtendAllowed(
    TPMI_DH_PCR      handle       // IN: PCR handle to be extended
)
{
    UINT8               commandLocality;
```
void PCRExtend()
{
    BYTE    *pcrData;
    HASH_STATE           hashState;
    UINT16               pcrSize;

    pcrData = GetPcrPointer(hash, handle - PCR_FIRST);

    if(pcrData != NULL)
    {
        pcrSize = CryptHashGetDigestSize(hash);
        CryptHashStart(&hashState, hash);
        CryptDigestUpdate(&hashState, pcrSize, pcrData);
        CryptDigestUpdate(&hashState, size, data);
        CryptHashEnd(&hashState, pcrSize, pcrData);

        // PCR has changed so update the pcrCounter if necessary
        PCRChanged(handle);
    }

    return;
}

void PCRCopyCurrentDigest()
{
    TPMI_ALG_HASH    hashAlg, // IN: hash algorithm to compute digest
    TPMI_PCR_SELECTION *selection, // IN/OUT: PCR selection (filtered on
    TPM2B_DIGEST       *digest, // OUT: digest
    TPMI_DH_PCR      handle,     // IN: PCR handle to be extended
    TPML_PCR_SELECTION  *selection,
    TPM2B_DIGEST       *digest,
    TPMI_DH_PCR      handle,     // IN: PCR handle to be extended
    TPML_PCR_SELECTION  *selection,
UINT32 pcrSize;
UINT32 pcr;
UINT32 i;

// Initialize the hash
digest->t.size = CryptHashStart(&hashState, hashAlg);
pAssert(digest->t.size > 0 && digest->t.size < UINT16_MAX);

// Iterate through the list of PCR selection structures
for(i = 0; i < selection->count; i++)
{
    // Point to the current selection
    select = &selection->pcrSelections[i]; // Point to the current selection
    FilterPcr(select); // Clear out the bits for unimplemented PCR

    // Need the size of each digest
    pcrSize = CryptHashGetDigestSize(selection->pcrSelections[i].hash);

    // Iterate through the selection
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        if(IsPcrSelected(pcr, select)) // Is this PCR selected
        {
            // Get pointer to the digest data for the bank
            pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
            pAssert(pcrData != NULL);
            CryptDigestUpdate(&hashState, pcrSize, pcrData); // add to digest
        }
    }
}

// Complete hash stack
CryptHashEnd2B(&hashState, &digest->b);
return;

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.

void PCRRead(
    TPML_PCR_SELECTION *selection, // IN/OUT: PCR selection (filtered on output)
    TPML_DIGEST *digest, // OUT: digest
    UINT32 *pcrCounter // OUT: the current value of PCR generation number
)
{
    TPMS_PCR_SELECTION *select;
    BYTE *pcrData; // will point to a digest
    UINT32 pcr;
    UINT32 i;

digest->count = 0;

    // Iterate through the list of PCR selection structures
    for(i = 0; i < selection->count; i++)
    {
        // Point to the current selection
        select = &selection->pcrSelections[i]; // Point to the current selection
        FilterPcr(select); // Clear out the bits for unimplemented PCR
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     // Get pointer to the digest data for the bank
683     pcrData = GetPcrPointer(selection->pcrSelections[i].hash, pcr);
684     pAssert(pcrData != NULL);
685     // Add to the data to digest
686     MemoryCopy(digest->digests[digest->count].t.buffer,
687        pcrData,
688        digest->digests[digest->count].t.size);
689     digest->count++;
690  }
691  // If we exit inner loop because we have exceed the output upper bound
692  if(digest->count > 7 && pcr < IMPLEMENTATION_PCR)
693  {
694    // Clear rest of the selection
695    while(i < selection->count)
696     {
697       MemorySet(selection->pcrSelections[i].pcrSelect, 0,
698                  selection->pcrSelections[i].sizeofSelect);
699        i++;
700     }
701     // exit outer loop
702     break;
703  }
704  
705  }
706  
707  
708  *pcrCounter = gr.pcrCounter;
709  return;
710 }

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      )
```c
{  UINT32         pcr = handle - PCR_FIRST;
  BYTE      *pcrData;

  // Copy value to the PCR if it is allocated
  pcrData = GetPcrPointer(hash, pcr);
  if(pcrData != NULL)
  {
    MemoryCopy(pcrData, digest->t.buffer, digest->t.size);
  }
  return;
}

8.7.3.25  PCRAllocate()

This function is used to change the PCR allocation.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>allocate failed</td>
</tr>
<tr>
<td>TPM_RC_PCR</td>
<td>improper allocation</td>
</tr>
</tbody>
</table>

TPM_RC

PCRAllocate(
  TPML_PCR_SELECTION  *allocate,  // IN: required allocation
  UINT32              *maxPCR,    // OUT: Maximum number of PCR
  UINT32              *sizeNeeded, // OUT: required space
  UINT32              *sizeAvailable // OUT: available space
)
{
  UINT32                  i, j, k;
  TPML_PCR_SELECTION      newAllocate;  // Initialize the flags to indicate if HCRTM PCR and DRTM PCR are allocated.
  BOOL                    pcrHcrtm = FALSE;
  BOOL                    pcrDrtm = FALSE;

  // Create the expected new PCR allocation based on the existing allocation
  // and the new input:
  // 1. if a PCR bank does not appear in the new allocation, the existing
  //    allocation of this PCR bank will be preserved.
  // 2. if a PCR bank appears multiple times in the new allocation, only the
  //    last one will be in effect.
  newAllocate = gp.pcrAllocated;
  for(i = 0; i < allocate->count; i++)
  {
    for(j = 0; j < newAllocate.count; j++)
    {
      // If hash matches, the new allocation covers the old allocation
      // for this particular bank
      // The assumption is the initial PCR allocation (from manufacture)
      // has all the supported hash algorithms with an assigned bank
      // (possibly empty). So there must be a match for any new bank
      // allocation from the input.
      if(newAllocate.pcrSelections[j].hash ==
         allocate->pcrSelections[i].hash)
      {
        newAllocate.pcrSelections[j] = allocate->pcrSelections[i];
        break;
      }
    }
    // The j loop must exit with a match.
  }
  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
790             // if DRTM PCR is not required, indicate that the allocation is OK
791             pcrDrtm = TRUE;
792         #endif
793
794         #if defined(HCRTM_PCR)
795             // and one HCRTM PCR (since this is usually PCR 0...)
796             pcrHcrtm = pcrHcrtm || TestBit(HCRTM_PCR, 
797                 newAllocate.pcrSelections[i].pcrSelect, 
798                 newAllocate.pcrSelections[i].sizeofSelect);
799         #else
800             pcrHcrtm = TRUE;
801         #endif
802
803         for(j = 0; j < newAllocate.pcrSelections[i].sizeofSelect; j++)
804         {
805             BYTE        mask = 1;
806             for(k = 0; k < 8; k++)
807             {
808                 if((newAllocate.pcrSelections[i].pcrSelect[j] & mask) != 0)
809                     *sizeNeeded += digestSize;
810                 mask = mask << 1;
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 sizeNeeded.
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.
```c
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 }

870 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     return;
900 }

8.7.3.28 PCRCapGetAllocation()

This function is used to get the current allocation of PCR banks.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if the return count is 0</td>
</tr>
<tr>
<td>NO</td>
<td>if the return count is not 0</td>
</tr>
</tbody>
</table>

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.
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>the property type is implemented</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>the property type is not implemented</td>
</tr>
</tbody>
</table>

```c
static BOOL
PCRGetProperty(
    TPM_PT_PCR  property,
    TPMS_TAGGED_PCR_SELECT  *select
)
{
    UINT32  pcr;
   UINT32  groupIndex;

    select->tag = property;
    // Always set the bitmap to be the size of all PCR
    select->sizeofSelect = (IMPLEMENTATION_PCR + 7) / 8;

    // Initialize bitmap
    MemorySet(select->pcrSelect, 0, select->sizeofSelect);

    // Collecting properties
    for(pcr = 0; pcr < IMPLEMENTATION_PCR; pcr++)
    {
        switch(property)
        {
        case TPM_PT_PCR_SAVE:
            if(s_initAttributes[pcr].stateSave == SET)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_EXTEND_L0:
            if((s_initAttributes[pcr].extendLocality & 0x01) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_RESET_L0:
            if((s_initAttributes[pcr].resetLocality & 0x01) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_EXTEND_L1:
            if((s_initAttributes[pcr].extendLocality & 0x02) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_RESET_L1:
            if((s_initAttributes[pcr].resetLocality & 0x02) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_EXTEND_L2:
            if((s_initAttributes[pcr].extendLocality & 0x04) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_RESET_L2:
            if((s_initAttributes[pcr].resetLocality & 0x04) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_EXTEND_L3:
            if((s_initAttributes[pcr].extendLocality & 0x08) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_RESET_L3:
            if((s_initAttributes[pcr].resetLocality & 0x08) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        case TPM_PT_PCR_EXTEND_L4:
            if((s_initAttributes[pcr].extendLocality & 0x10) != 0)
                PCRSetSelectBit(pcr, select->pcrSelect);
            break;
        }
    }
}
```
PCRSetSelectBit(pcr, select->pcrSelect);
break;

case TPM_PT_PCR_RESET_L4:
if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
PCRSetSelectBit(pcr, select->pcrSelect);
break;

case TPM_PT_PCR_DRTM_RESET:
// DRTM reset PCRs are the PCR reset by locality 4
if((s_initAttributes[pcr].resetLocality & 0x10) != 0)
PCRSetSelectBit(pcr, select->pcrSelect);
break;

#if defined NUM_POLICY_PCR_GROUP && NUM_POLICY_PCR_GROUP > 0

case TPM_PT_PCR_POLICY:
if(PCRBelongsPolicyGroup(pcr + PCR_FIRST, &groupIndex))
PCRSetSelectBit(pcr, select->pcrSelect);
bREAK;
#endif

#if defined NUM_AUTHVALUE_PCR_GROUP && NUM_AUTHVALUE_PCR_GROUP > 0

case TPM_PT_PCR_AUTH:
if(PCRBelongsAuthGroup(pcr + PCR_FIRST, &groupIndex))
PCRSetSelectBit(pcr, select->pcrSelect);
bREAK;
#endif

#if ENABLE_PCR_NO_INCREMENT == YES

case TPM_PT_PCR_NO_INCREMENT:
if(PCRBelongsTCBGroup(pcr + PCR_FIRST))
PCRSetSelectBit(pcr, select->pcrSelect);
bREAK;
#endif

default:
// If property is not supported, stop scanning PCR attributes
// and return.
return FALSE;
bREAK;
}

return TRUE;

8.7.3.31 PCRCapGetProperties()

This function returns a list of PCR properties starting at property.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if no more property is available</td>
</tr>
<tr>
<td>NO</td>
<td>if there are more properties not reported</td>
</tr>
</tbody>
</table>

TPM_BOOL

PCRCapGetProperties(
TPM_PT_PCR property, // IN: the starting PCR property
UINT32 count, // IN: count of returned properties

TPML_TAGGED_PCR_PROPERTY *select // OUT: PCR select
)

{  
TPM_BOOL more = NO;
UINT32 i;

// Initialize output property list
select->count = 0;

// The maximum count of properties we may return is MAX_PCR_PROPERTIES
if(count > MAX_PCR_PROPERTIES) count = MAX_PCR_PROPERTIES;
1040 // TPM_PT_PCR_FIRST is defined as 0 in spec. It ensures that property
1041 // value would never be less than TPM_PT_PCR_FIRST
1042 cAssert(TPM_PT_PCR_FIRST == 0);
1043
1044 // Iterate PCR properties. TPM_PT_PCR_LAST is the index of the last property
1045 // implemented on the TPM.
1046 for(i = property; i <= TPM_PT_PCR_LAST; i++)
1047 {
1048     if(select->count < count)
1049     {
1050         // If we have not filled up the return list, add more properties to it
1051         if(PCRGetProperty(i, &select->pcrProperty[select->count]))
1052             // only increment if the property is implemented
1053             select->count++;
1054     }
1055     else
1056     {
1057         // If the return list is full but we still have properties
1058         // available, report this and stop iterating.
1059         more = YES;
1060         break;
1061     }
1062 }
1063 return more;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

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     pAssert(HandleGetType(handle) == TPM_HT_PCR);
1076     // Initialize output handle list
1077     handleList->count = 0;
1078     // The maximum count of handles we may return is MAX_CAP_HANDLES
1079     if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
1080     // Iterate PCR handle range
1081     for(i = handle & HR_HANDLE_MASK; i <= PCR_LAST; i++)
1082     {
1083         if(handleList->count < count)
1084         {
1085             // If we have not filled up the return list, add this PCR
1086             // handle to it
1087             handleList->handle[handleList->count] = i + PCR_FIRST;
1088         }
handleList->count++;
    }
  }
  
  } // If the return list is full but we still have PCR handle
  
  // available, report this and stop iterating
  more = YES;
  break;
}

return more;
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

```c
#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.

```c
void PhysicalPresencePreInstall_Init()
{
    COMMAND_INDEX commandIndex;
    // Clear all the PP commands
    MemorySet(&gp.ppList, 0, sizeof(gp.ppList));
    // Any command that is PP_REQUIRED should be SET
    for(commandIndex = 0; commandIndex < COMMAND_COUNT; commandIndex++)
    {
        if(s_commandAttributes[commandIndex] & IS_IMPLEMENTED
           && s_commandAttributes[commandIndex] & PP_REQUIRED)
            SET_BIT(commandIndex, gp.ppList);
    }
    // Write PP list to NV
    NV_SYNC_PERSISTENT(ppList);
    return;
}
```

8.8.3.2 PhysicalPresenceCommandSet()

This function is used to set the indicator that a command requires PP confirmation.

```c
void PhysicalPresenceCommandSet(
    TPM_CC commandCode  // IN: command code
)
{
    COMMAND_INDEX commandIndex = CommandCodeToCommandIndex(commandCode);
    // if the command isn't implemented, the do nothing
    if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)
        return;
    // only set the bit if this is a command for which PP is allowed
    if(s_commandAttributes[commandIndex] & PP_COMMAND)
```
8.8.3.3 PhysicalPresenceCommandClear()

This function is used to clear the indicator that a command requires PP confirmation.

```c
void PhysicalPresenceCommandClear(  
    TPM_CC  commandCode  // IN: command code
) {
    COMMAND_INDEX  commandIndex = CommandCodeToCommandIndex(commandCode);
    // If the command isn't implemented, then don't do anything
    if(commandIndex == UNIMPLEMENTED_COMMAND_INDEX)  
        return;
    // Only clear the bit if the command does not require PP
    if((s_commandAttributes[commandIndex] & PP_REQUIRED) == 0)  
        CLEAR_BIT(commandIndex, gp.ppList);
    return;
}
```

8.8.3.4 PhysicalPresenceIsRequired()

This function indicates if PP confirmation is required for a command.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>physical presence is required</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>physical presence is not required</td>
</tr>
</tbody>
</table>

```c
BOOL PhysicalPresenceIsRequired(  
    COMMAND_INDEX  commandIndex   // IN: command index
) {
    // Check the bit map. If the bit is SET, PP authorization is required
    return (TEST_BIT(commandIndex, gp.ppList));
}
```

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`.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more command codes available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available command codes have been returned</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO PhysicalPresenceCapGetCCList(  
    TPM_CC  commandCode,  // IN: start command code
    UINT32  count,        // IN: count of returned TPM_CC
    TPML_CC  *commandList  // OUT: list of TPM_CC
) {
    ...  // implementation...
    return;
}
```
{  
  TPMI_YES_NO more = NO;
  COMMAND_INDEX commandIndex;

  // Initialize output handle list
  commandList->count = 0;

  // The maximum count of command we may return is MAX_CAP_CC
  if(count > MAX_CAP_CC) count = MAX_CAP_CC;

  // Collect PP commands
  for(commandIndex = GetClosestCommandIndex(commandCode);
      commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
      commandIndex = GetNextCommandIndex(commandIndex))
  {
    if(PhysicalPresenceIsRequired(commandIndex))
      {
        if(commandList->count < count)
          {
            // If we have not filled up the return list, add this command
            // code to it
            commandList->commandCodes[commandList->count] = GetCommandCode(commandIndex);
            commandList->count++;
          }
        else
          {
            // If the return list is full but we still have PP command
            // available, report this and stop iterating
            more = YES;
            break;
          }
      }
  }

  return more;
}
### 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

```c
#define SESSION_C
#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.

```c
static void ContextIdSetOldest(
    void
) {
    CONTEXT_SLOT lowBits;
    CONTEXT_SLOT entry;
    CONTEXT_SLOT smallest = ((CONTEXT_SLOT)~0);
    UINT32 i;

    // Set oldestSaveContext to a value indicating none assigned
    s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;

    lowBits = (CONTEXT_SLOT)gr.contextCounter;
    for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
    {
        entry = gr.contextArray[i];

        // only look at entries that are saved contexts
        if(entry > MAX_LOADED_SESSIONS)
        {
            // Use a less than or equal in case the oldest
            // is brand new (= lowBits-1) and equal to our initial
            // value for smallest.
            if(((CONTEXT_SLOT)(entry - lowBits)) <= smallest)
            {
                smallest = (entry - lowBits);
                s_oldestSavedSession = i;
            }
        }
    }
    // When we finish, either the s_oldestSavedSession still has its initial
    // value, or it has the index of the oldest saved context.
```
8.9.4 Startup Function -- SessionStartup()

This function initializes the session subsystem on TPM2_Startup().

```c
BOOL SessionStartup(
    STARTUP_TYPE type
)
{
    UINT32 i;
    // Initialize session slots. At startup, all the in-memory session slots
    // are cleared and marked as not occupied
    for(i = 0; i < MAX_LOADED_SESSIONS; i++)
        s_sessions[i].occupied = FALSE; // session slot is not occupied
    // The free session slots the number of maximum allowed loaded sessions
    s_freeSessionSlots = MAX_LOADED_SESSIONS;
    // Initialize context ID data. On a ST_SAVE or hibernate sequence, it will
    // scan the saved array of session context counts, and clear any entry that
    // references a session that was in memory during the state save since that
    // memory was not preserved over the ST_SAVE.
    if(type == SU_RESUME || type == SU_RESTART)
        { // On ST_SAVE we preserve the contexts that were saved but not the ones
            // in memory
            for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
            { // If the array value is unused or references a loaded session then
                // that loaded session context is lost and the array entry is
                // reclaimed.
                    if(gr.contextArray[i] <= MAX_LOADED_SESSIONS)
                        gr.contextArray[i] = 0;
            } // Find the oldest session in context ID data and set it in
            // s_oldestSavedSession
            ContextIdSetOldest();
        } else
        { // For STARTUP_CLEAR, clear out the contextArray
            for(i = 0; i < MAX_ACTIVE_SESSIONS; i++)
                gr.contextArray[i] = 0;
            // reset the context counter
            gr.contextCounter = MAX_LOADED_SESSIONS + 1;
            // Initialize oldest saved session
            s_oldestSavedSession = MAX_ACTIVE_SESSIONS + 1;
        } return TRUE;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>session is loaded</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>session is not loaded</td>
</tr>
</tbody>
</table>

```
BOOL SessionIsLoaded(
    TPM_HANDLE handle       // IN: session handle
){
    pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION 
        || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
    handle = handle & HR_HANDLE_MASK;
    // if out of range of possible active session, or not assigned to a loaded 
    // session return false
    if(handle >= MAX_ACTIVE_SESSIONS 
        || gr.contextArray[handle] == 0 
        || gr.contextArray[handle] > MAX_LOADED_SESSIONS)
        return FALSE;
    return TRUE;
}
```

### 8.9.5.2 SessionIsSaved()

This function tests 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>session is saved</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>session is not saved</td>
</tr>
</tbody>
</table>

```
BOOL SessionIsSaved(
    TPM_HANDLE handle       // IN: session handle
){
    pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION 
        || HandleGetType(handle) == TPM_HT_HMAC_SESSION);
    handle = handle & HR_HANDLE_MASK;
    // if out of range of possible active session, or not assigned, or 
    // assigned to a loaded session, return false
    if(handle >= MAX_ACTIVE_SESSIONS 
        || gr.contextArray[handle] == 0 
        || gr.contextArray[handle] <= MAX_LOADED_SESSIONS)
        return FALSE;
    return TRUE;
}```
8.9.5.3 SequenceNumberForSavedContextIsValid()

This function validates that the sequence number and handle value within a saved context are valid.

```c
BOOL SequenceNumberForSavedContextIsValid(
    TPMS_CONTEXT context, // IN: pointer to a context structure to be validated
)
{
    #define MAX_CONTEXT_GAP ((UINT64)((CONTEXT_SLOT)~0) + 1)
    TPM_HANDLE handle = context->savedHandle & HR_HANDLE_MASK;

    if(strcmp((context->sequence), (CONTEXT_SLOT)context->sequence) <= MAX_LOADED_SESSIONS) // the array entry must agree with the sequence number
    { return TRUE; }

    return FALSE;
}
```

8.9.5.4 SessionPCRValueIsCurrent()

This function is used to check if PCR values have been updated since the last time they were checked in a policy session.

This function requires the session is loaded.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>PCR value is current</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>PCR value is not current</td>
</tr>
</tbody>
</table>

```c
BOOL SessionPCRValueIsCurrent(
    SESSION *session // IN: session structure
)
{
    if(session->pcrCounter != 0 && session->pcrCounter != gr.pcrCounter)
    { return FALSE; }
    else
    { return TRUE; }
}
```

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.

```c
SESSION *
```
```c
SessionGet(
    TPM_HANDLE       handle // IN: session handle
 )
{
    size_t           slotIndex;
    CONTEXT_SLOT    sessionIndex;
    pAssert(HandleGetType(handle) == TPM_HT_POLICY_SESSION
             || HandleGetType(handle) == TPM_HT_HMAC_SESSION
         );
    slotIndex = handle & HR_HANDLE_MASK;
    pAssert(slotIndex < MAX_ACTIVE_SESSIONS);
    // get the contents of the session array. Because session is loaded, we
    // should always get a valid sessionIndex
    sessionIndex = gr.contextArray[slotIndex] - 1;
    pAssert(sessionIndex < MAX_LOADED_SESSIONS);
    return &s_sessions[sessionIndex].session;
}
```

### 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CONTEXT_GAP</td>
<td>can't assign a new <code>contextID</code> until the oldest saved session context is recycled</td>
</tr>
<tr>
<td>TPM_RC_SESSION_HANDLE</td>
<td>there is no slot available in the context array for tracking of this session context</td>
</tr>
</tbody>
</table>

```c
static TPM_RC ContextIdSessionCreate(
    TPM_HANDLE      *handle, // OUT: receives the assigned handle. This will
    // be an index that must be adjusted by the
    // caller according to the type of the
    // session created
    UINT32           sessionIndex // IN: The session context array entry that will
    // be occupied by the created session
 )
{
    pAssert(sessionIndex < MAX_LOADED_SESSIONS);
    // check to see if creating the context is safe
    // Is this going to be an assignment for the last session context
    // array entry? If so, then there will be no room to recycle the
    // oldest context if needed. If the gap is not at maximum, then
    // it will be possible to save a context if it becomes necessary.
    if(s_oldestSavedSession < MAX_ACTIVE_SESSIONS
        && s_freeSessionSlots == 1)
    {
        // See if the gap is at maximum
```
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CONTEXT_GAP</td>
<td>need to recycle sessions</td>
</tr>
<tr>
<td>TPM_RC_SESSION_HANDLE</td>
<td>active session space is full</td>
</tr>
<tr>
<td>TPM_RC_SESSION_MEMORY</td>
<td>loaded session space is full</td>
</tr>
</tbody>
</table>

```c
TPM_RC SessionCreate(
    TPM_SE           sessionType, // IN: the session type
    TPMI_ALG_HASH    authHash,   // IN: the hash algorithm
    TPM2B_NONCE     *nonceCaller, // IN: initial nonceCaller
    TPMI_DH_ENTITY   bind,       // IN: the bind object
    TPM2B_DATA      *seed,       // IN: seed data
    TPM_HANDLE      *sessionHandle, // OUT: the session handle
    TPM2B_NONCE     *nonceTpm    // OUT: the session nonce
)
    
    { TPM_RC result = TPM_RC_SUCCESS;
      CONTEXT SLOT slotIndex;
      SESSION *session = NULL;
      
      pAssert(sessionType == TPM_SE_HMAC
          | | sessionType == TPM_SE_POLICY
          | | sessionType == TPM_SE_TRIAL);
      
      // If there are no open spots in the session array, then no point in searching
      if(s_freeSessionSlots == 0)
          return TPM_RC_SESSION_MEMORY;
      
      // Find a space for loading a session
      for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
```
{ // Is this available?
    if(s_sessions[slotIndex].occupied == FALSE)
    {
        session = &s_sessions[slotIndex].session;
        break;
    }
}
// if no spot found, then this is an internal error
if(slotIndex >= MAX_LOADED_SESSIONS)
    FAIL(FATAL_ERROR_INTERNAL);

// Call context ID function to get a handle. TPM_RC_SESSION_HANDLE may be
// returned from ContextIdHandleAssign()
result = ContextIdSessionCreate(sessionHandle, slotIndex);
if(result != TPM_RC_SUCCESS)
    return result;

//*** Only return from this point on is TPM_RC_SUCCESS

// Can now indicate that the session array entry is occupied.
s_freeSessionSlots--;
s_sessions[slotIndex].occupied = TRUE;

// Initialize the session data
MemorySet(session, 0, sizeof(SESSION));

// Initialize internal session data
session->authHashAlg = authHash;
// Initialize session type
if(sessionType == TPM_SE_HMAC)
{
    *sessionHandle += HMAC_SESSION_FIRST;
}
else
{
    *sessionHandle += POLICY_SESSION_FIRST;
}
// For TPM_SE_POLICY or TPM_SE_TRIAL
session->attributes.isPolicy = SET;
if(sessionType == TPM_SE_TRIAL)
    session->attributes.isTrialPolicy = SET;
SessionSetStartTime(session);

// Initialize policyDigest. policyDigest is initialized with a string of 0
// of session algorithm digest size. Since the session is already clear.
// Just need to set the size
session->u2.policyDigest.t.size =
    CryptHashGetDigestSize(session->authHashAlg);

// Create initial session nonce
session->nonceTPM.t.size = nonceCaller->t.size;
CryptRandomGenerate(session->nonceTPM.t.size, session->nonceTPM.t.buffer);
MemoryCopy2B(&nonceTpm->b, &session->nonceTPM.b, sizeof(nonceTpm->t.buffer));

// Set up session parameter encryption algorithm
session->symmetric = symmetric;
// If there is a bind object or a session secret, then need to compute
// a sessionKey.
if(bind != TPM_RH_NULL || seed->t.size != 0)
{
    // sessionKey = KDFa(hash, (authValue || seed), "ATH", nonceTPM,
    // nonceCaller, bits)
// The HMAC key for generating the sessionSecret can be the concatenation
// of an authorization value and a seed value
TPM2B_TYPE(KEY, (sizeof(TPMT_HA) + sizeof(seed->t.buffer)));
TPM2B_KEY key;

// Get hash size, which is also the length of sessionKey
session->sessionKey.t.size = CryptHashGetDigestSize(session->authHashAlg);

// Get authValue of associated entity
EntityGetAuthValue(bind, (TPM2B_AUTH *)&key);
pAssert(key.t.size + seed->t.size <= sizeof(key.t.buffer));

// Concatenate authValue and seed
MemoryConcat2B(&key.b, &seed->b, sizeof(key.t.buffer));

// Compute the session key
CryptKDFa(session->authHashAlg, &key.b, SESSION_KEY, &session->nonceTPM.b,
&nonceCaller->b,
session->sessionKey.t.size * 8, session->sessionKey.t.buffer,
NULL, FALSE);
}

// Copy the name of the entity that the HMAC session is bound to
// Policy session is not bound to an entity
if(bind != TPM_RH_NULL && sessionType == TPM_SE_HMAC)
{
    session->attributes.isBound = SET;
    SessionComputeBoundEntity(bind, &session->u1.boundEntity);
}

// If there is a bind object and it is subject to DA, then use of this session
// is subject to DA regardless of how it is used.
session->attributes.isDaBound = (bind != TPM_RH_NULL)
&& (IsDAExempted(bind) == FALSE);

// If the session is bound, then check to see if it is bound to lockoutAuth
session->attributes.isLockoutBound
= (session->attributes.isDaBound == SET)
&& (bind == TPM_RH_LOCKOUT);
return TPM_RC_SUCCESS;

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CONTEXT_GAP</td>
<td>a contextID could not be assigned.</td>
</tr>
<tr>
<td>TPM_RC_TOOMANYCONTEXTS</td>
<td>the counter maxed out</td>
</tr>
</tbody>
</table>

TPM_RC
SessionContextSave(
TPM_HANDLE           handle, // IN: session handle
CONTEXT_COUNTER     *contextID // OUT: assigned contextID
)
{
    UINT32 contextIndex;
    CONTEXT_SLOT slotIndex;

    pAssert(SessionIsLoaded(handle));
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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SESSION_MEMORY</td>
<td>no free session slots</td>
</tr>
<tr>
<td>TPM_RC_CONTEXT_GAP</td>
<td>the gap count is maximum and this is not the oldest saved context</td>
</tr>
</tbody>
</table>

```c
TPM_RC
SessionContextLoad(
    SESSION_BUF *session, // IN: session structure from saved context
    TPM_HANDLE *handle   // IN/OUT: session handle
)
{
    UINT32 contextIndex;
    CONTEXT_SLOT slotIndex;

    pAssert(HandleGetType(*handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(*handle) == TPM_HT_HMAC_SESSION);

    // Don't bother looking if no openings
    if (s_freeSessionSlots == 0)
        return TPM_RC_SESSION_MEMORY;

    // Find a free session slot to load the session
    for(slotIndex = 0; slotIndex < MAX_LOADED_SESSIONS; slotIndex++)
        if (s_sessions[slotIndex].occupied == FALSE) break;

    // if no spot found, then this is an internal error
    pAssert(slotIndex < MAX_LOADED_SESSIONS);
    contextIndex = *handle & HR_HANDLE_MASK; // extract the index

    // If there is only one slot left, and the gap is at maximum, the only session
    // context that we can safely load is the oldest one.
    if (s_oldestSavedSession < MAX_ACTIVE_SESSIONS
        && s_freeSessionSlots == 1
        && (CONTEXT_SLOT)gr.contextCounter == gr.contextArray[s_oldestSavedSession]
        && contextIndex != s_oldestSavedSession)
        return TPM_RC_CONTEXT_GAP;

    pAssert(contextIndex < MAX_ACTIVE_SESSIONS);

    // set the contextArray value to point to the session slot where
    // the context is loaded
    gr.contextArray[contextIndex] = slotIndex + 1;

    // if this was the oldest context, find the new oldest
    if (contextIndex == s_oldestSavedSession)
        ContextIdSetOldest();

    // Copy session data to session slot
    MemoryCopy(&s_sessions[slotIndex].session, session, sizeof(SESSION));

    // Set session slot as occupied
    s_sessions[slotIndex].occupied = TRUE;

    // Reduce the number of open spots
    s_freeSessionSlots--;

    return TPM_RC_SUCCESS;
}
```
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.

```c
void SessionFlush(TPM_HANDLE handle) // IN: loaded or saved session handle
{
    CONTEXT_SLOT slotIndex;
    UINT32 contextIndex; // Index into contextArray
    pAssert((HandleGetType(handle) == TPM_HT_POLICY_SESSION
            || HandleGetType(handle) == TPM_HT_HMAC_SESSION
            )
            && (SessionIsLoaded(handle) || SessionIsSaved(handle))
    );
    // Flush context ID of this session
    // Convert handle to an index into the contextArray
    contextIndex = handle & HR_HANDLE_MASK;
    pAssert(contextIndex < sizeof(gr.contextArray) / sizeof(gr.contextArray[0]));
    // Get the current contents of the array
    slotIndex = gr.contextArray[contextIndex];
    // Mark context array entry as available
    gr.contextArray[contextIndex] = 0;
    // Is this a saved session being flushed
    if(slotIndex > MAX_LOADED_SESSIONS)
    {
        // Flushing the oldest session?
        if(slotIndex == s_oldestSavedSession)
            // If so, find a new value for oldest.
            ContextIdSetOldest();
    }
    else
    {
        // Adjust slot index to point to session array index
        slotIndex -= 1;
        // Free session array index
        s_sessions[slotIndex].occupied = FALSE;
        s_freeSessionSlots++;
    }
    return;
}
```

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, the will be overlapped by XORing bytes. If XOR is required, the bind value will be full.

```c
void SessionComputeBoundEntity()
```
TPMI_DH_ENTITY entityHandle,  // IN: handle of entity
TPM2B_NAME *bind     // OUT: binding value
}

TPM2B_AUTH auth;
BYTE *pAuth = auth.t.buffer;
UINT16 i;

// Get name
EntityGetName(entityHandle, bind);

// The bound value of a reserved handle is the handle itself
if(bind-t.size == sizeof(TPM_HANDLE)) return;

// For all the other entities, concatenate the authorization value to the name.
// Get a local copy of the authorization value because some overlapping
// may be necessary.
EntityGetAuthValue(entityHandle, &auth);

// Make sure that the extra space is zeroed
MemorySet(&bind-t.name[bind-t.size], 0, sizeof(bind-t.name) - bind-t.size);
// XOR the authValue at the end of the name
for(i = sizeof(bind-t.name) - auth.t.size; i < sizeof(bind-t.name); i++)
    bind-t.name[i] ^= *pAuth++;

// Set the bind value to the maximum size
bind-t.size = sizeof(bind-t.name);
return;

8.9.6.7 SessionSetStartTime()

This function is used to initialize the session timing

void SessionSetStartTime(
    SESSION *session     // IN: the session to update
)
{
    session->startTime = g_time;
    session->epoch = g_timeEpoch;
    session->timeout = 0;
}

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.

void SessionResetPolicyData(
    SESSION *session     // IN: the session to reset
)
{
    SESSION_ATTRIBUTES oldAttributes;
pAssert(session != NULL);
    // Will need later
    oldAttributes = session->attributes;
    // No command
    session->commandCode = 0;
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

```
TPMI_YES_NO
SessionCapGetLoaded(
    TPMI_SH_POLICY   handle, // IN: start handle
    UINT32           count, // IN: count of returned handles
    TPM_HANDLE     *handleList // OUT: list of handle
)
{
    TPMI_YES_NO     more = NO;
    UINT32          i;

    pAssert(HandleGetType(handle) == TPM_HT_LOADED_SESSION);

    // Initialize output handle list
    handleList->count = 0;

    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;

    // Iterate session context ID slots to get loaded session handles
    for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
    {
        // If session is active
        if(gr.contextArray[i] != 0)
        {
            // If session is loaded
```
if (gr.contextArray[i] <= MAX_LOADED_SESSIONS)
{
    if (handleList->count < count)
    {
        SESSION *session;

        // If we have not filled up the return list, add this
        // session handle to it
        // assume that this is going to be an HMAC session
        handle = i + HMAC_SESSION_FIRST;
        session = SessionGet(handle);
        if (session->attributes.isPolicy)
            handle = i + POLICY_SESSION_FIRST;
        handleList->handle[handleList->count] = handle;
        handleList->count++;
    }
    else
    {
        // If the return list is full but we still have loaded object
        // available, report this and stop iterating
        more = YES;
        break;
    }
}
return more;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

 TPMI_YES_NO
SessionCapGetSaved(
    TPMI_SH_HMAC handle, // IN: start handle
    UINT32 count, // IN: count of returned handles
    TPML_HANDLE *handleList // OUT: list of handle
)
{
    pAssert(HandleGetType(handle) == TPM_HT_SAVED_SESSION);
    // Initialize output handle list
    handleList->count = 0;
    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if (count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
    // Iterate session context ID slots to get loaded session handles

    // Initialize output handle list
    handleList->count = 0;
    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if (count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
    // Iterate session context ID slots to get loaded session handles
for(i = handle & HR_HANDLE_MASK; i < MAX_ACTIVE_SESSIONS; i++)
{
    // If session is active
    if(gr.contextArray[i] != 0)
    {
        // If session is saved
        if(gr.contextArray[i] > MAX_LOADED_SESSIONS)
        {
            if(handleList->count < count)
            {
                // If we have not filled up the return list, add this
                // session handle to it
                handleList->handle[handleList->count] = i + HMAC_SESSION_FIRST;
                handleList->count++;
            } else
            {
                // If the return list is full but we still have loaded object
                // available, report this and stop iterating
                more = YES;
                break;
            }
        } else
        {
            return more;
        }
}

8.9.6.11 SessionCapGetLoadedNumber()

This function return the number of authorization sessions currently loaded into TPM RAM.

UINT32 SessionCapGetLoadedNumber()
{
    return MAX_LOADED_SESSIONS - s_freeSessionSlots;
}

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.

UINT32 SessionCapGetLoadedAvail()
{
    return s_freeSessionSlots;
}

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

```c
#include "Tpm.h"
#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.

```c
void
TimePowerOn(
    void
)
{
    g_time = _plat__TimerRead();
}
```

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.

```c
static void
TimeNewEpoch(
    void
)
{
    //if CLOCK_STOPS
    CryptRandomGenerate(sizeof(CLOCK_NONCE), (BYTE *)&g_timeEpoch);
    #else
        // if the epoch is kept in NV, update it.
        gp.timeEpoch++;
        NV_SYNC_PERSISTENT(timeEpoch);
    #endif
    // Clean out any lingering state
    _plat__TimerWasStopped();
}
```

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.

```c
BOOL
TimeStartup(
    STARTUP_TYPE     type     // IN: start up type
)
{
    NOT_REFERENCED(type);
    // If the previous cycle is orderly shut down, the value of the safe bit
    // the same as previously saved. Otherwise, it is not safe.
    if(!NV_IS_ORDERLY)
        go.clockSafe = NO;
    return TRUE;
}
```

### 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.

```c
void
TimeClockUpdate(
    UINT64           newTime   // IN: New time value in mS.
)
{
    #define CLOCK_UPDATE_MASK  ((1ULL << NV_CLOCK_UPDATE_INTERVAL) - 1)
    // Check to see if the update will cause a need for an nvClock update
    if((newTime | CLOCK_UPDATE_MASK) > (go.clock | CLOCK_UPDATE_MASK))
    {
        pAssert(g_NvStatus == TPM_RC_SUCCESS);
        // Going to update the NV time state so SET the safe flag
        go.clockSafe = YES;
        // update the time
        go.clock = newTime;
        NvWrite(NV_ORDERLY_DATA, sizeof(go), &go);
    }
    else
        // No NV update needed so just update
        go.clock = newTime;
}
```

### 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.

```c
void
TimeUpdate(
```
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.

void TimeUpdateToCurrent()
{
    // Can't update time during the dark interval or when rate limiting so don't
    // make any modifications to the internal clock value. Also, defer any clock
    // processing until TPM has run TPM2_Startup()
    if(!NV_IS_AVAILABLE || !TPMIsStarted())
        return;
    TimeUpdate();
}

8.10.3.7 TimeSetAdjustRate()

This function is used to perform rate adjustment on Time and Clock.

void TimeSetAdjustRate(
    TPM_CLOCK_ADJUST adjust   // IN: adjust constant
)
{
    switch(adjust)
    {
        // IN: adjust constant
case TPM_CLOCK_COARSE_SLOWER:
    _plat_ClockAdjustRate(CLOCK_ADJUST_COARSE);
    break;

case TPM_CLOCK_COARSE_FASTER:
    _plat_ClockAdjustRate(-CLOCK_ADJUST_COARSE);
    break;

case TPM_CLOCK_MEDIUM_SLOWER:
    _plat_ClockAdjustRate(CLOCK_ADJUST_MEDIUM);
    break;

case TPM_CLOCK_MEDIUM_FASTER:
    _plat_ClockAdjustRate(-CLOCK_ADJUST_MEDIUM);
    break;

case TPM_CLOCK_FINE_SLOWER:
    _plat_ClockAdjustRate(CLOCK_ADJUST_FINE);
    break;

case TPM_CLOCK_FINE_FASTER:
    _plat_ClockAdjustRate(-CLOCK_ADJUST_FINE);
    break;

case TPM_CLOCK_NO_CHANGE:
    break;

default:
    FAIL(FATAL_ERROR_INTERNAL);
    break;
}

return;

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

UINT16
TimeGetMarshaled(
    TIME_INFO       *dataBuffer  // OUT: result buffer
)
{
    TPMS_TIME_INFO      timeInfo;

    // Fill TPMS_TIME_INFO structure
    timeInfo.time = g_time;
    TimeFillInfo(&timeInfo.clockInfo);

    // Marshal TPMS_TIME_INFO to canonical form
    return TPMS_TIME_INFO_Marshal(&timeInfo, (BYTE **)&dataBuffer, NULL);
}

8.10.3.9 TimeFillInfo

This function gathers information to fill in a TPMS_CLOCK_INFO structure.

void
TimeFillInfo(
    TPMS_CLOCK_INFO     *clockInfo
)
{
    clockInfo->clock = go.clock;
    clockInfo->resetCount = gp.resetCount;
    clockInfo->restartCount = gr.restartCount;

    // If NV is not available, clock stopped advancing and the value reported is
    // not "safe".
if(NV_IS_AVAILABLE)
    clockInfo->safe = go.clockSafe;
else
    clockInfo->safe = NO;
return;
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

```c
#include "Tpm.h"
typedef struct
{
    TPM_ALG_ID algID;
    TPMA_ALGORITHM attributes;
} ALGORITHM;
static const ALGORITHM s_algorithms[] =
{
    // The entries in this table need to be in ascending order but the table doesn't
    // need to be full (gaps are allowed). One day, a tool might exist to fill in the
    // table from the TPM_ALG description
    #if ALG_RSA
    {TPM_ALG_RSA, TPMA_ALGORITHM_INITIALIZER(1, 0, 1, 0, 0, 0, 0, 0, 0)},
    #endif
    #if ALG_TDES
    {TPM_ALG_TDES, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
    #endif
    #if ALG_SHA1
    {TPM_ALG_SHA1, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
    #endif
    {TPM_ALG_HMAC, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 1, 0, 0, 0)},
    #if ALG_AES
    {TPM_ALG_AES, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
    #endif
    #if ALG_MGF1
    {TPM_ALG_MGF1, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 1, 0, 0)},
    #endif
    {TPM_ALG_KEYEDHASH, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 1, 0, 0, 1, 0, 0)},
    #if ALG_XOR
    {TPM_ALG_XOR, TPMA_ALGORITHM_INITIALIZER(0, 1, 1, 0, 0, 0, 0, 0, 0)},
    #endif
    #if ALG_SHA256
    {TPM_ALG_SHA256, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
    #endif
    #if ALG_SHA384
    {TPM_ALG_SHA384, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
    #endif
    #if ALG_SHA512
    {TPM_ALG_SHA512, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
    #endif
    #if ALG_SM3_256
    {TPM_ALG_SM3_256, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)},
    #endif
    #if ALG_SM4
    {TPM_ALG_SM4, TPMA_ALGORITHM_INITIALIZER(0, 1, 0, 0, 0, 0, 0, 0, 0)},
    #endif
```
#if ALG_RSASSA
  {TPM_ALG_RSASSA, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 1, 0, 0, 0, 0)}
#endif

#if ALG_RSAES
  {TPM_ALG_RSAES, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 1, 0, 0, 0, 0)}
#endif

#if ALG_RSAPSS
  {TPM_ALG_RSAPSS, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 1, 0, 0, 0, 0)}
#endif

#if ALG_OAEP
  {TPM_ALG_OAEP, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 1, 0, 0, 0, 0)}
#endif

#if ALG_ECDSA
  {TPM_ALG_ECDSA, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 1, 0, 1, 0, 0)}
#endif

#if ALG_ECDH
  {TPM_ALG_ECDH, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 0, 1, 0, 0, 0)}
#endif

#if ALG_ECDA
  {TPM_ALG_ECDA, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 1, 0, 0, 0, 0)}
#endif

#if ALG_SM2
  {TPM_ALG_SM2, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 1, 0, 0, 0, 0)}
#endif

#if ALG_ECSCHNORR
  {TPM_ALG_ECSCHNORR, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 1, 0, 0, 0, 0)}
#endif

#if ALG_ECMQV
  {TPM_ALG_ECMQV, TPMA_ALGORITHM_INITIALIZER(1, 0, 0, 0, 1, 0, 0, 0, 0)}
#endif

#if ALG_KDF1_SP800_56A
  {TPM_ALG_KDF1_SP800_56A, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_KDF2
  {TPM_ALG_KDF2, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_KDF1_SP800_108
  {TPM_ALG_KDF1_SP800_108, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_ECC
  {TPM_ALG_ECC, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_CAMELLIA
  {TPM_ALG_CAMELLIA, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)}
#endif

#if ALG_CMAC
  {TPM_ALG_CMAC, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 0, 0)}
#endif

#if ALG_CTR
  {TPM_ALG_CTR, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_OFB
  {TPM_ALG_OFB, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_CBC
  {TPM_ALG_CBC, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_CFB
  {TPM_ALG_CFB, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_ECB
  {TPM_ALG_ECB, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_KDF1_SP800_108
  {TPM_ALG_KDF1_SP800_108, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif

#if ALG_ECB
  {TPM_ALG_ECB, TPMA_ALGORITHM_INITIALIZER(0, 0, 1, 0, 0, 0, 0, 1, 0)}
#endif
9.1.3 AlgorithmCapGetImplemented()

This function is used by TPM2_GetCapability() to return a list of the implemented algorithms.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>more algorithms to report</td>
</tr>
<tr>
<td>NO</td>
<td>no more algorithms to report</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO AlgorithmCapGetImplemented(
    TPM_ALG_ID algID,  // IN: the starting algorithm ID
    UINT32 count,      // IN: count of returned algorithms
    TPM_ALG_PROPERTY *algList // OUT: algorithm list
)
{
    TPMI_YES_NO more = NO;
    UINT32 i;
    UINT32 algNum;

    // initialize output algorithm list
    algList->count = 0;

    // The maximum count of algorithms we may return is MAX_CAP_ALGS.
    if(count > MAX_CAP_ALGS)
        count = MAX_CAP_ALGS;

    // Compute how many algorithms are defined in s_algorithms array.
    algNum = sizeof(s_algorithms) / sizeof(s_algorithms[0]);

    // Scan the implemented algorithm list to see if there is a match to 'algID'.
    for(i = 0; i < algNum; i++)
    {
        // If algID is less than the starting algorithm ID, skip it
        if(s_algorithms[i].algID < algID)
            continue;
        if(algList->count < count)
        {
            // If we have not filled up the return list, add more algorithms
            // to it
            algList->algProperties[algList->count].alg = s_algorithms[i].algID;
            algList->algProperties[algList->count].algProperties =
                s_algorithms[i].attributes;
            algList->count++;
        }
        else
        {
            // If the return list is full but we still have algorithms
            // available, report this and stop scanning.
            more = YES;
            break;
        }
    }

    return more;
}
```

9.1.4 AlgorithmGetImplementedVector()

This function returns the bit vector of the implemented algorithms.
LIB_EXPORT
void AlgorithmGetImplementedVector(
    ALGORITHM_VECTOR *implemented // OUT: the implemented bits are SET
)
{
    int index;

    // Nothing implemented until we say it is
    MemorySet(implemented, 0, sizeof(ALGORITHM_VECTOR));

    for(index = (sizeof(s_algorithms) / sizeof(s_algorithms[0])) - 1;
        index >= 0;
        index--)
    SET_BIT(s_algorithms[index].algID, *implemented);

    return;
}
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

```c
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>bit is set</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>bit is not set</td>
</tr>
</tbody>
</table>

```c
bool TestBit(unsigned int bitNum, // IN: number of the bit in 'bArray'
              BYTE *bArray,     // IN: array containing the bits
              unsigned int bytesInArray // IN: size in bytes of 'bArray'
            )
{
    pAssert(bytesInArray > (bitNum >> 3));
    return ((bArray[bitNum >> 3] & (1 << (bitNum & 7))) != 0);
}
```

9.2.3.2 SetBit()

This function will set the indicated bit in bArray.

```c
void SetBit(unsigned int bitNum, // IN: number of the bit in 'bArray'
            BYTE *bArray,     // IN: array containing the bits
            unsigned int bytesInArray // IN: size in bytes of 'bArray'
        )
{
    pAssert(bytesInArray > (bitNum >> 3));
    bArray[bitNum >> 3] |= (1 << (bitNum & 7));
}
```

9.2.3.3 ClearBit()

This function will clear the indicated bit in bArray.

```c
void ClearBit(unsigned int bitNum, // IN: number of the bit in 'bArray'
              BYTE *bArray,     // IN: array containing the bits
              unsigned int bytesInArray // IN: size in bytes of 'bArray'
            )
{
    pAssert(bytesInArray > (bitNum >> 3));
    bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
}
```
ClearBit(
    unsigned int bitNum,  // IN: number of the bit in 'bArray'.
    BYTE *bArray,        // IN: array containing the bits
    unsigned int bytesInArray  // IN: size in bytes of 'bArray'
)
{
    pAssert(bytesInArray > (bitNum >> 3));
    bArray[bitNum >> 3] &= ~(1 << (bitNum & 7));
}
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIMPLEMENTED_COMMAND_INDEX</td>
<td>command is not implemented</td>
</tr>
<tr>
<td>other</td>
<td>index of the command</td>
</tr>
</tbody>
</table>

9 #if !COMPRRESSED_LISTS
10 static COMMAND_INDEX
11 NextImplementedIndex(commandIndex)
12 )
13 {
14      COMMAND_INDEX commandIndex
15      for (; commandIndex < COMMAND_COUNT; commandIndex++)
16         { if (s_commandAttributes[commandIndex] & IS_IMPLEMENTED)
17            Return commandIndex;
18         }
19         return UNIMPLEMENTED_COMMAND_INDEX;
20    }
21 #else
22 #define NextImplementedIndex(x) (x)
23 #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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIMPLEMENTED_COMMAND_INDEX</td>
<td>command is not implemented</td>
</tr>
<tr>
<td>other</td>
<td>index of a command</td>
</tr>
</tbody>
</table>

```
COMMAND_INDEX
GetClosestCommandIndex(
    TPM_CC    commandCode     // IN: the command code to start at
)
{
    BOOL    vendor = (commandCode & CC_VEND) != 0;
    COMMAND_INDEX searchIndex = (COMMAND_INDEX)commandCode;
    // The commandCode is a UINT32 and the search index is UINT16. We are going to
    // search for a match but need to make sure that the commandCode value is not
    // out of range. To do this, need to clear the vendor bit of the commandCode
    // (if set) and compare the result to the 16-bit searchIndex value. If it is
    // out of range, indicate that the command is not implemented
    if((commandCode & ~CC_VEND) != searchIndex)
        return UNIMPLEMENTED_COMMAND_INDEX;

    // if there is at least one vendor command, the last entry in the array will
    // have the v bit set. If the input commandCode is larger than the last
    // vendor-command, then it is out of range.
    if(vendor)
    {
        #if VENDOR_COMMAND_ARRAY_SIZE > 0
            COMMAND_INDEX commandIndex;
            COMMAND_INDEX min;
            COMMAND_INDEX max;
            int diff;
        #if LIBRARY_COMMAND_ARRAY_SIZE == COMMAND_COUNT
            #error "Constants are not consistent."
        #endif
            // Check to see if the value is equal to or below the minimum
            // entry.
            // Note: Put this check first so that the typical case of only one vendor-
            // specific command doesn't waste any more time.
            if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE], TPMA_CC,
                commandIndex) >= searchIndex)
                { // the vendor array is always assumed to be packed so there is
                    // no need to check to see if the command is implemented
                    return LIBRARY_COMMAND_ARRAY_SIZE;
                }
            // See if this is out of range on the top
            if(GET_ATTRIBUTE(s_ccAttr[COMMAND_COUNT - 1], TPMA_CC, commandIndex)
                < searchIndex)
                { // Needs initialization to keep
                    return UNIMPLEMENTED_COMMAND_INDEX;
                }
            commandIndex = UNIMPLEMENTED_COMMAND_INDEX; // Needs initialization to keep
            // compiler happy
            min = LIBRARY_COMMAND_ARRAY_SIZE; // first vendor command
            max = COMMAND_COUNT - 1; // last vendor command
            diff = 1; // needs initialization to keep
            // compiler happy
            while(min <= max)
            {
```
commandIndex = (min + max + 1) / 2;

diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex) - searchIndex;

if(diff == 0)
    return commandIndex;
else
    if(diff > 0)
        max = commandIndex - 1;
    else
        min = commandIndex + 1;

// didn't find and exact match. commandIndex will be pointing at the last
// item tested. If 'diff' is positive, then the last item tested was
// larger index of the command code so it is the smallest value
// larger than the requested value.
if(diff > 0)
    return commandIndex;
else
    return commandIndex + 1;
#endif

// Get here if the V-Bit was not set in 'commandCode'

if(GET_ATTRIBUTE(s_ccAttr[LIBRARY_COMMAND_ARRAY_SIZE - 1], TPMA_CC, commandIndex) < searchIndex)
{
    // requested index is out of the range to the top

    #if VENDOR_COMMAND_ARRAY_SIZE > 0
    // If there are vendor commands, then the first vendor command
    // is the next value greater than the commandCode.
    // NOTE: we got here if the starting index did not have the V bit but we
    // reached the end of the array of library commands (non-vendor). Since
    // there is at least one vendor command, and vendor commands are always
    // in a compressed list that starts after the library list, the next
    // index value contains a valid vendor command.
    return LIBRARY_COMMAND_ARRAY_SIZE;
    #else
    // if there are no vendor commands, then this is out of range
    return UNIMPLEMENTED_COMMAND_INDEX;
    #endif

    // If the request is lower than any value in the array, then return
    // the lowest value (needs to be an index for an implemented command
    if(GET_ATTRIBUTE(s_ccAttr[0], TPMA_CC, commandIndex) >= searchIndex)
    {
        return NextImplementedIndex(0);
    }
    else
    {
        #if COMPRESSED LISTS
        COMMAND_INDEX commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
        COMMAND_INDEX min = 0;
        COMMAND_INDEX max = LIBRARY_COMMAND_ARRAY_SIZE - 1;
        int diff = 1;
        #if LIBRARY_COMMAND_ARRAY_SIZE == 0
        #error "Something is terribly wrong"
        #endif
        // The s_ccAttr array contains an extra entry at the end (a zero value).
        // Don't count this as an array entry. This means that max should start
        // out pointing to the last valid entry in the array which is - 2

        // if there are vendor commands, then the first vendor command
        // is the next value greater than the commandCode.
        // The s_ccAttr array contains an extra entry at the end (a zero value).
        // Don't count this as an array entry. This means that max should start
        // out pointing to the last valid entry in the array which is - 2
pAssert(max == (sizeof(s_ccAttr) / sizeof(TPMA_CC) - VENDOR_COMMAND_ARRAY_SIZE - 2));

while(min <= max)
{
    commandIndex = (min + max + 1) / 2;
    diff = GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex) - searchIndex;
    if(diff == 0)
        return commandIndex;
    if(diff > 0)
        max = commandIndex - 1;
    else
        min = commandIndex + 1;
}

// didn't find an exact match. commandIndex will be pointing at the
// last item tested. If diff is positive, then the last item tested was
// larger index of the command code so it is the smallest value
// larger than the requested value.
if(diff > 0)
    return commandIndex;

// if diff is negative, then the value tested was smaller than
// the command code index and the next higher value is the correct one.
// Note: this will necessarily be in range because of the earlier check
// that the index was within range.
return commandIndex + 1;

#endif
}
}

9.3.3.3 CommandCodeToCommandIndex()

This function returns the index in the various attributes arrays of the command.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIMPLEMENTED_COMMAND_INDEX</td>
<td>command is not implemented</td>
</tr>
<tr>
<td>other</td>
<td>index of the command</td>
</tr>
</tbody>
</table>

COMMAND_INDEX

CommandCodeToCommandIndex(
    TPM_CC       commandCode   // IN: the command code to look up
)
{
    // Extract the low 16-bits of the command code to get the starting search index
    COMMAND_INDEX searchIndex = (COMMAND_INDEX)commandCode;
    BOOL        vendor = (commandCode & CC_VEND) != 0;
    COMMAND_INDEX commandIndex;

    #if !COMPRESSED_LISTS
    if(!vendor)
    {
        commandIndex = searchIndex - (COMMAND_INDEX)s_ccAttr[0].commandIndex;
        // Check for out of range or unimplemented.
        // Note, since a COMMAND_INDEX is unsigned, if searchIndex is smaller than
        // the lowest value of command, it will become a 'negative' number making
        // it look like a large unsigned number, this will cause it to fail
        // the unsigned check below.
        if(commandIndex >= LIBRARY_COMMAND_ARRAY_SIZE
            return LIBRARY_COMMAND_ARRAY_SIZE;
        
        // If the index is in range, check vendor bit to determine
        // which index value to look up.
        if(vendor)
            commandIndex += VendorCommandCount;
        else
            commandIndex += VendorCommandCount;
    }

    return commandIndex;
}
198  || (s_commandAttributes[commandIndex] & IS_IMPLEMENTED) == 0)
199     return UNIMPLEMENTED_COMMAND_INDEX;
200  
201   return commandIndex;
202 
203 endif
204
205 // Need this code for any vendor code lookup or for compressed lists
206 commandIndex = GetClosestCommandIndex(commandCode);
207
208 // Look at the returned value from get closest. If it isn't the one that was
209 // requested, then the command is not implemented.
210 if(commandIndex != UNIMPLEMENTED_COMMAND_INDEX)
211 {
212   if((GET_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, commandIndex)
213    != searchIndex)
214     || (IS_ATTRIBUTE(s_ccAttr[commandIndex], TPMA_CC, V)) != vendor)
215     commandIndex = UNIMPLEMENTED_COMMAND_INDEX;
216 }
217 
218 return commandIndex;
219
9.3.3.4 GetNextCommandIndex()

This function returns the index of the next implemented command.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIMPLEMENTED_COMMAND_INDEX</td>
<td>no more implemented commands</td>
</tr>
<tr>
<td>other</td>
<td>the index of the next implemented command</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTH_NONE</td>
<td>no authorization is required</td>
</tr>
<tr>
<td>AUTH_USER</td>
<td>user role authorization is required</td>
</tr>
<tr>
<td>AUTH_ADMIN</td>
<td>admin role authorization is required</td>
</tr>
<tr>
<td>AUTH_DUP</td>
<td>duplication role authorization is required</td>
</tr>
</tbody>
</table>

```
AUTH_ROLE

CommandAuthRole(commandIndex, handleIndex)
{
    if (0 == handleIndex)
    {
        // Any authorization role set?
        COMMAND_ATTRIBUTES properties = s_commandAttributes[commandIndex];
        if (properties & HANDLE_1_USER)
            return AUTH_USER;
        if (properties & HANDLE_1_ADMIN)
            return AUTH_ADMIN;
        if (properties & HANDLE_1_DUP)
            return AUTH_DUP;
    }
    else if (1 == handleIndex)
    {
        if (s_commandAttributes[commandIndex] & HANDLE_2_USER)
            return AUTH_USER;
    }
    return AUTH_NONE;
}
```

9.3.3.7 EncryptSize()

This function returns the size of the decrypt size field. This function returns 0 if encryption is not allowed.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>encryption not allowed</td>
</tr>
<tr>
<td>2</td>
<td>size field is two bytes</td>
</tr>
<tr>
<td>4</td>
<td>size field is four bytes</td>
</tr>
</tbody>
</table>

```
int

EncryptSize(commandIndex)
{
    return ((s_commandAttributes[commandIndex] & ENCRYPT_2) ? 2 :
             (s_commandAttributes[commandIndex] & ENCRYPT_4) ? 4 : 0);}
```
9.3.3.8 DecryptSize()

This function returns the size of the decrypt size field. This function returns 0 if decryption is not allowed.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>encryption not allowed</td>
</tr>
<tr>
<td>2</td>
<td>size field is two bytes</td>
</tr>
<tr>
<td>4</td>
<td>size field is four bytes</td>
</tr>
</tbody>
</table>

```c
int DecryptSize(
    COMMAND_INDEX commandIndex // IN: command index
)
{
    return ((s_commandAttributes[commandIndex] & DECRYPT_2) ? 2 :
             (s_commandAttributes[commandIndex] & DECRYPT_4) ? 4 : 0);
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>session is allowed with this command</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>session is not allowed with this command</td>
</tr>
</tbody>
</table>

```c
BOOL IsSessionAllowed(
    COMMAND_INDEX commandIndex // IN: the command to be checked
)
{
    return ((s_commandAttributes[commandIndex] & NO_SESSIONS) == 0);
}
```

9.3.3.10 IsHandleInResponse()

This function determines if a command has a handle in the response.

```c
BOOL IsHandleInResponse(
    COMMAND_INDEX commandIndex
)
{
    return ((s_commandAttributes[commandIndex] & R_HANDLE) != 0);
}
```

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.

```c
BOOL IsWriteOperation(
    COMMAND_INDEX commandIndex // IN: Command to check
)
```
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     return FALSE;
9.3.3.13 CommandCapGetCCList()

This function returns a list of implemented commands and command attributes starting from the command in \texttt{commandCode}.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>more command attributes are available</td>
</tr>
<tr>
<td>NO</td>
<td>no more command attributes are available</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO CommandCapGetCCList(
    TPM_CC commandCode,   // IN: start command code
    UINT32 count,         // IN: maximum count for number of entries in 'commandList'
    TPML_CCA *commandList // OUT: list of TPMA_CC
)
{
    TPMI_YES_NO more = NO;
    COMMAND_INDEX commandIndex;

    // initialize output handle list count
    commandList->count = 0;

    for(commandIndex = GetClosestCommandIndex(commandCode);
        commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
        commandIndex = GetNextCommandIndex(commandIndex))
    {
        #if !COMPRESSED_LISTS
            // this check isn't needed for compressed lists.
            if(!(s_commandAttributes[commandIndex] & IS_IMPLEMENTED))
                continue;
        #endif

        if(commandList->count < count)
        {
            // If the list is not full, add the attributes for this command.
            commandList->commandAttributes[commandList->count] = s_ccAttr[commandIndex];
            commandList->count++;
        }
        else
        {
            // If the list is full but there are more commands to report,
            // indicate this and return.
            more = YES;
            break;
        }
    }
    return more;
}
```

9.3.3.14 IsVendorCommand()

Function indicates if a command index references a vendor command.
### Part 4: Supporting Routines

#### Trusted Platform Module Library

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>command is a vendor command</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>command is not a vendor command</td>
</tr>
</tbody>
</table>

```c
402  BOOL  IsVendorCommand(
403         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

```c
#include "Tpm.h"
```

9.4.3 Functions

9.4.3.1 EntityGetLoadStatus()

This function will check that all the handles access loaded entities.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_HANDLE</td>
<td>handle type does not match</td>
</tr>
<tr>
<td>TPM_RC_REFERENCE_Hx</td>
<td>entity is not present</td>
</tr>
<tr>
<td>TPM_RC_HIERARCHY</td>
<td>entity belongs to a disabled hierarchy</td>
</tr>
<tr>
<td>TPM_RC_OBJECT_MEMORY</td>
<td>handle is an evict object but there is no space to load it to RAM</td>
</tr>
</tbody>
</table>

```c
TPM_RC
EntityGetLoadStatus(
    COMMAND *command, // IN/OUT: command parsing structure
)
{
    UINT32 i;
    TPM_RC result = TPM_RC_SUCCESS;
    //
    for(i = 0; i < command->handleNum; i++)
    {
        TPM_HANDLE handle = command->handles[i];
        switch(HandleGetType(handle))
        {
            // For handles associated with hierarchies, the entity is present
            // only if the associated enable is SET.
            case TPM_HT_PERMANENT:
                switch(handle)
                {
                    case TPM_RH_OWNER:
                        if(!gc.shEnable)
                            result = TPM_RC_HIERARCHY;
                        break;

                    #ifdef VENDOR_PERMANENT
                    case VENDOR_PERMANENT:
                        break;
                    #endif
                    case TPM_RH_ENDORSEMENT:
                        if(!gc.ehEnable)
                            result = TPM_RC_HIERARCHY;
                        break;
                    case TPM_RH_PLATFORM:
                        if(!g_phEnable)
                            result = TPM_RC_HIERARCHY;
                        break;
```

// null handle, PW session handle and lockout
// handle are always available
    case TPM_RH_NULL:
    case TPM_RS_PW:
        // Need to be careful for lockout. Lockout is always available
        // for policy checks but not always available when authValue
        // is being checked.
        case TPM_RH_LOCKOUT:
        break;
    default:
        // handling of the manufacture_specific handles
        if(((TPM_RH)handle >= TPM_RH_AUTH_00)
            && ((TPM_RH)handle <= TPM_RH_AUTH_FF))
            // use the value that would have been returned from
            // unmarshaling if it did the handle filtering
            result = TPM_RC_VALUE;
        else
            result = TPM_RC_REFERENCE_H0;
        break;
    case TPM_HT_TRANSIENT:
        // For a transient object, check if the handle is associated
        // with a loaded object.
        if(!IsObjectPresent(handle))
            result = TPM_RC_REFERENCE_H0;
        break;
    case TPM_HT_PERSISTENT:
        // Persistent object
        // Copy the persistent object to RAM and replace the handle with the
        // handle of the assigned slot. A TPM_RC_OBJECT_MEMORY,
        // TPM_RC_HIERARCHY or TPM_RC_REFERENCE_H0 error may be returned by
        // ObjectLoadEvict()
        result = ObjectLoadEvict(&command->handles[i], command->index);
        break;
    case TPM_HT_HMAC_SESSION:
        // For an HMAC session, see if the session is loaded
        // and if the session in the session slot is actually
        // an HMAC session.
        if(SessionIsLoaded(handle))
        {  
            SESSION *session;
            session = SessionGet(handle);
            // Check if the session is a HMAC session
            if(session->attributes.isPolicy == SET)
                result = TPM_RC_HANDLE;
        } else
            result = TPM_RC_REFERENCE_H0;
        break;
    case TPM_HT_POLICY_SESSION:
        // For a policy session, see if the session is loaded
        // and if the session in the session slot is actually
        // a policy session.
        if(SessionIsLoaded(handle))
        {  
            SESSION *session;
            session = SessionGet(handle);
            // Check if the session is a policy session
            if(session->attributes.isPolicy == CLEAR)
                result = TPM_RC_HANDLE;
        } else
            result = TPM_RC_REFERENCE_H0;
        break;
    case TPM_HT_NV_INDEX:
// For an NV Index, use the TPM-specific routine
to search the IN Index space.
result = NvIndexIsAccessible(handle);
break;
case TPM_HT_PCR:
    // Any PCR handle that is unmarshaled successfully referenced
    // a PCR that is defined.
    break;
#if CC_AC_Send
    case TPM_HT_AC:
        // Use the TPM-specific routine to search for the AC
        result = AcIsAccessible(handle);
        break;
#endif
default:
    // Any other handle type is a defect in the unmarshaling code.
    FAIL(FATAL_ERROR_INTERNAL);
    break;
}

if(result != TPM_RC_SUCCESS)
{
    if(result == TPM_RC_REFERENCE_H0)
        result = result + i;
    else
        result = RcSafeAddToResult(result, TPM_RC_H + g_rcIndex[i]);
    break;
}
return result;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>number of bytes in the authValue with 0's stripped</td>
</tr>
</tbody>
</table>

UINT16 EntityGetAuthValue(
    TPMI_DH_ENTITY handle,       // IN: handle of entity
    TPM2B_AUTH    *auth        // OUT: authValue of the entity
)
{
    pAuth = NULL;
    auth->t.size = 0;
    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
        {
            switch(handle)
            {
                case TPM_RH_OWNER:
                    // ownerAuth for TPM_RH_OWNER
                    pAuth = &gp.ownerAuth;
                    break;
                break;
            }
        }
case TPM_RH_ENDORSEMENT:
  // endorsementAuth for TPM_RH_ENDORSEMENT
  pAuth = &gp.endorsementAuth;
  break;

case TPM_RH_PLATFORM:
  // platformAuth for TPM_RH_PLATFORM
  pAuth = &gc.platformAuth;
  break;

case TPM_RH_LOCKOUT:
  // lockoutAuth for TPM_RH_LOCKOUT
  pAuth = &gp.lockoutAuth;
  break;

case TPM_RH_NULL:
  // nullAuth for TPM_RH_NULL. Return 0 directly here
  return 0;
  break;

#ifdef VENDOR_PERMANENT
  case VENDOR_PERMANENT:
    // vendor authorization value
    pAuth = &g_platformUniqueDetails;
#endif
default:
  // If any other permanent handle is present it is
  // a code defect.
  FAIL(FATAL_ERROR_INTERNAL);
  break;
}
}
break;

case TPM_HT_TRANSIENT:
  // authValue for an object
  // A persistent object would have been copied into RAM
  // and would have an transient object handle here.
{
  OBJECT *object;

  object = HandleToObject(handle);
  // special handling if this is a sequence object
  if(ObjectIsSequence(object))
  {
    pAuth = &((HASH_OBJECT *)object)->auth;
  }
  else
  {
    // Authorization is available only when the private portion of
    // the object is loaded. The check should be made before
    // this function is called
    pAssert(object->attributes.publicOnly == CLEAR);
    pAuth = &object->sensitive.authValue;
  }
}
break;

case TPM_HT_NV_INDEX:
  // authValue for an NV index
{
  NV_INDEX *nvIndex = NvGetIndexInfo(handle, NULL);
  pAssert(nvIndex != NULL);
  pAuth = &nvIndex->authValue;
}
break;

case TPM_HT_PCR:
  // authValue for PCR
  pAuth = PCRGetAuthValue(handle);
  break;

default:
  // If any other handle type is present here, then there is a defect
// in the unmarshaling code.
FAIL(FATAL_ERROR_INTERNAL);
break;
}
// Copy the authValue
MemoryCopy2B(&auth->b, &pAuth->b, sizeof(auth->t.buffer));
MemoryRemoveTrailingZeros(auth);
return auth->t.size;
}

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.

TPMI_ALG_HASH EntityGetAuthPolicy(
TPMI_DH_ENTITY handle, // IN: handle of entity
TPM2B_DIGEST *authPolicy // OUT: authPolicy of the entity
)
{
TPMI_ALG_HASH hashAlg = TPM_ALG_NULL;
authPolicy->t.size = 0;
switch(HandleGetType(handle))
{
  case TPM_HT_PERMANENT:
    switch(handle)
    {
      case TPM_RH_OWNER:
        // ownerPolicy for TPM_RH_OWNER
        *authPolicy = gp.ownerPolicy;
        hashAlg = gp.ownerAlg;
        break;
      case TPM_RH_ENDORSEMENT:
        // endorsementPolicy for TPM_RH_ENDORSEMENT
        *authPolicy = gp.endorsementPolicy;
        hashAlg = gp.endorsementAlg;
        break;
      case TPM_RH_PLATFORM:
        // platformPolicy for TPM_RH_PLATFORM
        *authPolicy = gc.platformPolicy;
        hashAlg = gc.platformAlg;
        break;
      case TPM_RH_LOCKOUT:
        // lockOutPolicy for TPM_RH_LOCKOUT
        *authPolicy = gp.lockoutPolicy;
        hashAlg = gp.lockoutAlg;
        break;
      default:
        return TPM_ALG_ERROR;
    }
    break;
  case TPM_HT_TRANSIENT:
    // authPolicy for an object
    {
      OBJECT *object = HandleToObject(handle);
      ...
*authPolicy = object->publicArea.authPolicy;
hashAlg = object->publicArea.nameAlg;
}
break;
case TPM_HT_NV_INDEX:
    // authPolicy for a NV index
    {
        NV_INDEX   *nvIndex = NvGetIndexInfo(handle, NULL);
pAssert(nvIndex != 0);
    *authPolicy = nvIndex->publicArea.authPolicy;
hashAlg = nvIndex->publicArea.nameAlg;
    }
break;
case TPM_HT_PCR:
    // authPolicy for a PCR
    hashAlg = PCRGetAuthPolicy(handle, authPolicy);
    break;
default:
    // If any other handle type is present it is a code defect.
    FAIL(FATAL_ERROR_INTERNAL);
    break;
    }
return hashAlg;
}

9.4.3.4 EntityGetName()

This function returns the Name associated with a handle.

TPM2B_NAME *
EntityGetName(
TPMI_DH_ENTITY   handle,   // IN: handle of entity
TPM2B_NAME      *name      // OUT: name of entity
)
{
switch(HandleGetType(handle))
{
  case TPM_HT_TRANSIENT:
    {
      // Name for an object
      OBJECT      *object = HandleToObject(handle);
      // an object with no nameAlg has no name
      if(object->publicArea.nameAlg == TPM_ALG_NULL)
        name->b.size = 0;
      else
          *name = object->name;
      break;
    }
  case TPM_HT_NV_INDEX:
    // Name for a NV index
    NvGetNameByIndexHandle(handle, name);
    break;
  default:
    // For all other types, the handle is the Name
    name->t.size = sizeof(TPM_HANDLE);
    UINT32_TO_BYTE_ARRAY(handle, name->t.name);
    break;
}
return name;
}
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.

```
TPMI_RH_HIERARCHY
EntityGetHierarchy(
    TPMI_DH_ENTITY   handle   // IN :handle of entity
) {
    TPMI_RH_HIERARCHY hierarchy = TPM_RH_NULL;
    switch(HandleGetType(handle))
    {
        case TPM_HT_PERMANENT:
            // hierarchy for a permanent handle
            switch(handle)
            {
                case TPM_RH_PLATFORM:
                case TPM_RH_ENDORSEMENT:
                    hierarchy = handle;
                    break;
                // all other permanent handles are associated with the owner
                // hierarchy. (should only be TPM_RH_OWNER and TPM_RH_LOCKOUT)
                default:
                    hierarchy = TPM_RH_OWNER;
                    break;
            }
            break;
        case TPM_HT_NV_INDEX:
            // hierarchy for NV index
            {
                NV_INDEX        *nvIndex = NvGetIndexInfo(handle, NULL);
                pAssert(nvIndex != NULL);
                // If only the platform can delete the index, then it is
                // considered to be in the platform hierarchy, otherwise it
                // is in the owner hierarchy.
                if(IS_ATTRIBUTE(nvIndex->publicArea.attributes, TPMA_NV, PLATFORMCREATE))
                    hierarchy = TPM_RH_PLATFORM;
                else
                    hierarchy = TPM_RH_OWNER;
            }
            break;
        case TPM_HT_TRANSIENT:
            // hierarchy for an object
            {
                OBJECT          *object;
                object = HandleToObject(handle);
                if(object->attributes.ppsHierarchy)
                {
                    hierarchy = TPM_RH_PLATFORM;
                }
                else if(object->attributes.epsHierarchy)
                {
                    hierarchy = TPM_RH_ENDORSEMENT;
                }
                else if(object->attributes.spsHierarchy)
```
```c
    {    hierarchy = TPM_RH_OWNER;
    }
    break;
    case TPM_HT_PCR:
    hierarchy = TPM_RH_OWNER;
    break;
    default:
    FAIL(FATAL_ERROR_INTERNAL);
    break;
} // this is unreachable but it provides a return value for the default
// case which makes the compiler happy
return hierarchy;
```
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 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

```c
#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.

```c
TPM_HT
HandleGetType(
    TPM_HANDLE handle,      // IN: a handle to be checked
)
{
    // return the upper bytes of input data
    return (TPM_HT)((handle & HR_RANGE_MASK) >> HR_SHIFT);
}
```

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:

```c
TPM_HANDLE
NextPermanentHandle(
    TPM_HANDLE inHandle,  // IN: the handle to check
)
{
    // If inHandle is below the start of the range of permanent handles
    // set it to the start and scan from there
    if(inHandle < TPM_RH_FIRST)
        inHandle = TPM_RH_FIRST;
    // scan from input value until we find an implemented permanent handle
    // or go out of range
    for(; inHandle <= TPM_RH_LAST; inHandle++)
    {
        switch(inHandle)
        {
            case TPM_RH_OWNER:
            case TPM_RH_NULL:
            case TPM_RS_PW:
            case TPM_RH_LOCKOUT:
            case TPM_RH_ENDORSEMENT:
            case TPM_RH_PLATFORM:
            case TPM_RH_PLATFORM_NV:
                #ifdef VENDOR_PERMANENT
                case VENDOR_PERMANENT
                #endif
                return inHandle;
            break;
            default:
                break;
        }
    }
    return 0;
}
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if there are more handles available</td>
</tr>
<tr>
<td>NO</td>
<td>all the available handles has been returned</td>
</tr>
</tbody>
</table>

```c
TPMI_YES_NO
PermanentCapGetHandles(
    TPM_HANDLE       handle,       // IN: start handle
    UINT32           count,        // IN: count of returned handles
    TPM_HANDLE     *handleList     // OUT: list of handle
)
{
    TPMI_YES_NO more = NO;
    UINT32 i;
    pAssert(HandleGetType(handle) == TPM_HT_PERMANENT);
    // Initialize output handle list
    handleList->count = 0;
    // The maximum count of handles we may return is MAX_CAP_HANDLES
    if(count > MAX_CAP_HANDLES) count = MAX_CAP_HANDLES;
    // Iterate permanent handle range
    for(i = Next PermanentHandle(handle);
        i != 0; i = Next PermanentHandle(i + 1))
    {
        if(handleList->count < count)
        {
            // If we have not filled up the return list, add this permanent handle
            handleList->handle[handleList->count] = i;
            handleList->count++;
        }
        else
        {
            // If the return list is full but we still have permanent handle
            // available, report this and stop iterating
            more = YES;
            break;
        }
    }
    return more;
}
```

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

```c
TPMI_YES_NO
PermanentHandleGetPolicy(
    TPM_HANDLE   handle,       // IN: start handle
    UINT32       count,        // IN: max count of returned handles
    TPML_TAGGED_POLICY *policyList  // OUT: list of handle
)
{
    TPMI_YES_NO more = NO;
    pAssert(HandGetType(handle) == TPM_HT_PERMANENT);

    // Initialize output handle list
    policyList->count = 0;

    // The maximum count of policies we may return is MAX_TAGGED_POLICIES
    if(count > MAX_TAGGED_POLICIES)
        count = MAX_TAGGED_POLICIES;

    // Iterate permanent handle range
    for(handle = NextPermanentHandle(handle);
        handle != 0;
        handle = NextPermanentHandle(handle + 1))
    {
        TPM2B_DIGEST    policyDigest;
        TPM_ALG_ID      policyAlg;
        // Check to see if this permanent handle has a policy
        policyAlg = EntityGetAuthPolicy(handle, &policyDigest);
        if(policyAlg == TPM_ALG_ERROR)
            continue;
        if(policyList->count < count)
            { // If we have not filled up the return list, add this
                // policy to the list;
                policyList->policies[policyList->count].handle = handle;
                policyList->policies[policyList->count].policyHash.hashAlg = policyAlg;
                MemoryCopy(&policyList->policies[policyList->count].policyHash.digest,
                    policyDigest.t.buffer, policyDigest.t.size);
                policyList->count++;
            }
        else
            { // If the return list is full but we still have permanent handle
                more = YES;
                break;
            }
    }
    return more;
}```
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.

```c
#define IO_BUFFER_C
#include "Tpm.h"
#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.

```c
void MemoryIoBufferAllocationReset()
{
    s_actionIoAllocation = 0;
}
```

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.

```c
void MemoryIoBufferZero()
{
    memset(s_actionIoBuffer, 0, s_actionIoAllocation);
}
```

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.

```c
BYTE * MemoryGetInBuffer(
    UINT32 size // Size, in bytes, required for the input
)
{
    pAssert(size <= sizeof(s_actionIoBuffer));
    // In this implementation, a static buffer is set aside for the command action
    // buffers. The buffer is shared between input and output. This is because
    // there is no need to allocate for the worst case input and worst case output
```
// at the same time.
// Round size up
#define UoM (sizeof(s_actionIoBuffer[0]))
size = (size + (UoM - 1)) & (UINT32_MAX - (UoM - 1));
memset(s_actionIoBuffer, 0, size);
s_actionIoAllocation = size;
return (BYTE *)&s_actionIoBuffer[0];
}

### 9.7.2.4 MemoryGetOutBuffer()

This function returns the address of the buffer into which the command action code places its output values.

```c
BYTE *
MemoryGetOutBuffer(
UINT32 size  // required size of the buffer
)
{
BYTE    *retVal = (BYTE *)(&s_actionIoBuffer[s_actionIoAllocation / UoM]);
pAssert((size + s_actionIoAllocation) < (sizeof(s_actionIoBuffer)));
// In this implementation, a static buffer is set aside for the command action
// output buffer.
memset(retVal, 0, size);
s_actionIoAllocation += size;
return retVal;
}
```

### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>string is null terminated</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>string is not null terminated</td>
</tr>
</tbody>
</table>

```c
BOOL
IsLabelProperlyFormatted(
TPM2B *x
)
{
return ((x)->size == 0) || ((x)->buffer[(x)->size - 1] == 0));
}
```
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     MemorySet(&locality_attributes, 0, sizeof(TPMA_LOCALITY));
10    switch(locality)
11    {
12        case 0:
13            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ZERO);
14            break;
15        case 1:
16            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_ONE);
17            break;
18        case 2:
19            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_TWO);
20            break;
21        case 3:
22            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_THREE);
23            break;
24        case 4:
25            SET_ATTRIBUTE(locality_attributes, TPMA_LOCALITY, TPM_LOC_FOUR);
26            break;
27        default:
28            pAssert(locality > 31);
29            *localityAsByte = locality;
30            break;
31    }
32    return locality_attributes;
33 }

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

```c
#define MANUFACTURE_C
#include "Tpm.h"
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>1</td>
<td>manufacturing process previously performed</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT int
TPM_Manufacture(int firstTime, // IN: indicates if this is the first call from
                 // main())
{
    TPM_SU orderlyShutdown;

    #if RUNTIME_SIZE_CHECKS
    // Call the function to verify the sizes of values that result from different
    // compile options.
    TpmSizeChecks();
    #endif

    // If TPM has been manufactured, return indication.
    if(!firstTime && g_manufactured)
        return 1;

    // Do power on initializations of the cryptographic libraries.
    CryptInit();

    s_DAPendingOnNV = FALSE;

    // initialize NV
    NvManufacture();

    // Clear the magic value in the DRBG state
    go.drbgState.magic = 0;

    CryptStartup(SU_RESET);

    // default configuration for PCR
    PCRSimStart();
```
37     // initialize pre-installed hierarchy data
38     // This should happen after NV is initialized because hierarchy data is
39     // stored in NV.
40     HierarchyPreInstall_Init();
41
42     // initialize dictionary attack parameters
43     DAPreInstall_Init();
44
45     // initialize PP list
46     PhysicalPresencePreInstall_Init();
47
48     // initialize command audit list
49     CommandAuditPreInstall_Init();
50
51     // first start up is required to be Startup(CLEAR)
52     orderlyShutdown = TPM_SU_CLEAR;
53     NV_WRITE_PERSISTENT(orderlyState, orderlyShutdown);
54
55     // initialize the firmware version
56     gp.firmwareV1 = FIRMWARE_V1;
57     #ifdef FIRMWARE_V2
58     gp.firmwareV2 = FIRMWARE_V2;
59     #else
60     gp.firmwareV2 = 0;
61     #endif
62     NV_SYNC_PERSISTENT(firmwareV1);
63     NV_SYNC_PERSISTENT(firmwareV2);
64
65     // initialize the total reset counter to 0
66     gp.totalResetCount = 0;
67     NV_SYNC_PERSISTENT(totalResetCount);
68
69     // initialize the clock stuff
70     go.clock = 0;
71     go.clockSafe = YES;
72     NvWrite(NV_ORDERLY_DATA, sizeof(ORDERLY_DATA), &go);
73
74     // Commit NV writes. Manufacture process is an artificial process existing
75     // only in simulator environment and it is not defined in the specification
76     // that what should be the expected behavior if the NV write fails at this
77     // point. Therefore, it is assumed the NV write here is always success and
78     // no return code of this function is checked.
79     NvCommit();
80
81     g_manufactured = TRUE;
82
83     return 0;
84     }
85
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 needs 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.
## 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.

```c
LIB_EXPORT void
TpmEndSimulation()
{
    #if SIMULATION
        HashLibSimulationEnd();
        SymLibSimulationEnd();
        MathLibSimulationEnd();
        #if ALG_RSA
            RsaSimulationEnd();
        #endif
        #if ALG_ECC
            EccSimulationEnd();
        #endif
    #endif
    // SIMULATION
}
```

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>1</td>
<td>TPM not previously manufactured</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>Values</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>[TRANSIENT_FIRST:TRANSIENT_LAST]</td>
<td>allowed range for transient objects</td>
</tr>
<tr>
<td>[PERSISTENT_FIRST:PERSISTENT_LAST]</td>
<td>allowed range for persistent objects</td>
</tr>
<tr>
<td>+TPM_RH_NULL</td>
<td>the null handle</td>
</tr>
<tr>
<td>#TPM_RC_VALUE</td>
<td></td>
</tr>
</tbody>
</table>

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     {
9       if(*target == TPM_RH_NULL)
10       {
11         if(flag)
12         return TPM_RC_SUCCESS;
13       }
14       else
15         return TPM_RC_VALUE;
16     }
17     if(((*target < TRANSIENT_FIRST) || (*target > TRANSIENT_LAST))
18       &&((*target < PERSISTENT_FIRST) || (*target > PERSISTENT_LAST)))
19     return TPM_RC_VALUE;
20     return TPM_RC_SUCCESS;
21 }
```

and the following marshaling code:

```
1   UINT16
2   TPMI_DH_OBJECT_Marshal(TPMI_DH_OBJECT *source, BYTE **buffer, INT32 *size)
```

NOTE: The marshaling code does not do parameter checking, as the TPM is the source of the marshaling data.
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_OBJECTMarshal() that do nothing but call another function and replaces the function with a #define.

```c
#define TPMI_DH_OBJECTMarshal(source, buffer, size) \  UINT32.Marshal((UINT32 *)source, buffer, size)
```

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

### 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>**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Selector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyedHash</td>
<td>TPMS_KEYEDHASH_PARMS</td>
<td>TPM_ALG_KEYEDHASH</td>
<td>sign</td>
</tr>
<tr>
<td>symDetail</td>
<td>TPMT_SYM_DEF_OBJECT</td>
<td>TPM_ALG_SYMCIPHER</td>
<td>a symmetric block cipher</td>
</tr>
<tr>
<td>rsaDetail</td>
<td>TPMS_RSA_PARMS</td>
<td>TPM_ALG_RSA</td>
<td>decrypt + sign</td>
</tr>
<tr>
<td>eccDetail</td>
<td>TPMS_ECC_PARMS</td>
<td>TPM_ALG_ECC</td>
<td>decrypt + sign</td>
</tr>
<tr>
<td>asymDetail</td>
<td>TPMS_ASYM_PARMS</td>
<td></td>
<td>common scheme structure for RSA and ECC keys</td>
</tr>
</tbody>
</table>

**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.

```c
TPM_RC TPMU_PUBLIC_PARMS_Unmarshal(TPMU_PUBLIC_PARMS *target, BYTE **buffer, INT32 *size, UINT32 selector) {
    switch(selector) {
    #if ALG_KEYEDHASH
        case TPM_ALG_KEYEDHASH:
            return TPMS_KEYEDHASH_PARMS_Unmarshal((TPMS_KEYEDHASH_PARMS *)&(target->keyedHash), buffer, size);
    #endif
    #if ALG_SYMCIPHER
        case TPM_ALG_SYMCIPHER:
            return TPMT_SYM_DEF_OBJECT_Unmarshal((TPMT_SYM_DEF_OBJECT *)&(target->symDetail), buffer, size, FALSE);
    #endif
    #if ALG_RSA
        case TPM_ALG_RSA:
            return TPMS_RSA_PARMS_Unmarshal((TPMS_RSA_PARMS *)&(target->rsaDetail), buffer, size);
    #endif
    #if ALG_ECC
        case TPM_ALG_ECC:
            return TPMS_ECC_PARMS_Unmarshal((TPMS_ECC_PARMS *)&(target->eccDetail), buffer, size);
    #endif
    }
```
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>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>TPMI_ALG_PUBLIC</td>
<td>&quot;algorithm&quot; associated with this object</td>
</tr>
<tr>
<td>nameAlg</td>
<td>+TPMI_ALG_HASH</td>
<td>algorithm used for computing the Name of the object</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOTE The &quot;+&quot; indicates that the instance of a TPMT_PUBLIC may have a &quot;+&quot; to indicate that the nameAlg may be TPM_ALG_NULL.</td>
</tr>
<tr>
<td>objectAttributes</td>
<td>TPMA_OBJECT</td>
<td>attributes that, along with type, determine the manipulations of this object</td>
</tr>
<tr>
<td>authPolicy</td>
<td>TPM2B_DIGEST</td>
<td>optional policy for using this key</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The policy is computed using the nameAlg of the object.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>NOTE shall be the Empty Buffer if no authorization policy is present</td>
</tr>
<tr>
<td>[type]parameters</td>
<td>TPMU_PUBLIC_PARMS</td>
<td>the algorithm or structure details</td>
</tr>
<tr>
<td>[type]unique</td>
<td>TPMU_PUBLIC_ID</td>
<td>the unique identifier of the structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>For an asymmetric key, this would be the public key.</td>
</tr>
</tbody>
</table>

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:

```c
TPM_RC
TPM_PUBLIC_Unmarshal(TPMT_PUBLIC *target, BYTE **buffer, INT32 *size, BOOL flag)
{
    TPM_RC    result;
    result = TPMI_ALG_PUBLIC_Unmarshal((TPMI_ALG_PUBLIC *)target->type,
                                     buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMI_ALG_HASH_Unmarshal((TPMI_ALG_HASH *)target->nameAlg,
                                      buffer, size, flag);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMA_OBJECT_Unmarshal((TPMA_OBJECT *)target->objectAttributes,
                                    buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPM2B_DIGEST_Unmarshal((TPM2B_DIGEST *)target->authPolicy,
                                     buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMU_PUBLIC_PARMS_Unmarshal((TPMU_PUBLIC_PARMS *)target->parameters,
                                          buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;
    result = TPMU_PUBLIC_ID_Unmarshal((TPMU_PUBLIC_ID *)target->unique,
                                       buffer, size, (UINT32)target->type);
    if(result != TPM_RC_SUCCESS)
        return result;
    return TPM_RC_SUCCESS;
}
```
The marshaling code for the TPMT_PUBLIC structure is:

```c
UINT16
TPMT_PUBLIC_Marshal(TPMT_PUBLIC *source, BYTE **buffer, INT32 *size)
{
    UINT16 result = 0;
    result = (UINT16)(result + TPMI_ALG_PUBLIC_Marshal(
                   (TPMI_ALG_PUBLIC *)&(source->type), buffer, size));
    result = (UINT16)(result + TPMI_ALG_HASH_Marshal(
                   (TPMI_ALG_HASH *)&(source->nameAlg), buffer, size));
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
                   (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
    result = (UINT16)(result + TPM2B_DIGEST_Marshal(
                   (TPM2B_DIGEST *)&(source->authPolicy), buffer, size));
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
                   (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
                   (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
                   (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
    result = (UINT16)(result + TPMA_OBJECT_Marshal(
                   (TPMA_OBJECT *)&(source->objectAttributes), buffer, size));
    return result;
}
```

9.10.5 Unmarshal and Marshal an Array

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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>count {2:}</td>
<td>UINT32</td>
<td>number of digests in the list, minimum is two</td>
</tr>
<tr>
<td>digests[count][8]</td>
<td>TPM2B_DIGEST</td>
<td>a list of digests</td>
</tr>
<tr>
<td>#TPM_RC_SIZE</td>
<td></td>
<td>response code when count is not at least two or is greater than 8</td>
</tr>
</tbody>
</table>

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

```c
TPM_RC
TPML_DIGEST_Unmarshal(TPML_DIGEST *target, BYTE **buffer, INT32 *size)
{
    TPM_RC result;
    result = UINT32_Unmarshal((UINT32 *)&(target->count), buffer, size);
    if(result != TPM_RC_SUCCESS)
        return result;
    if( (target->count < 2) ) // This check is triggered by the {2:} notation
        return TPM_RC_SIZE;   // on 'count'
    if((target->count > 8) ) // This check is triggered by the {:8} notation
```
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:

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

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

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

The marshaling code for the array is:

```c
TPM_RC
TPM2B_DIGEST_Array_Unmarshal(TPM2B_DIGEST *target, BYTE **buffer, INT32 *size,
    INT32 count)
{
    TPM_RC    result;
    INT32 i;
    for(i = 0; i < count; i++) {
        result = TPM2B_DIGEST_Unmarshal(&target[i], buffer, size);
        if(result != TPM_RC_SUCCESS)
            return result;
    }
    return TPM_RC_SUCCESS;
}
```
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>
<thead>
<tr>
<th>Parameter</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>size</td>
<td>UINT16</td>
<td>Size of the operand</td>
</tr>
<tr>
<td>buffer [size] {:1024}</td>
<td>BYTE</td>
<td>The operand</td>
</tr>
</tbody>
</table>

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

using these structure definitions:

```c
typedef union {
    struct {
        UINT16    size;
        BYTE      buffer[1024];
    } t;
    TPM2B      b;
} 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.

```c
#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).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if (a &gt; b)</td>
</tr>
<tr>
<td>0</td>
<td>if (a = b)</td>
</tr>
<tr>
<td>-1</td>
<td>if (a &lt; b)</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT int UnsignedCompareB(
    UINT32 aSize,     // IN: size of a
    const BYTE *a,     // IN: a
    UINT32 bSize,     // IN: size of b
    const BYTE *b     // IN: b
)
{
    UINT32 i;
    if(aSize > bSize)
        return 1;
    else if(aSize < bSize)
        return -1;
    else
    {
        for(i = 0; i < aSize; i++)
            if(a[i] != b[i])
                return (a[i] > b[i]) ? 1 : -1;
        return 0;
    }
}
```

9.11.2.2 SignedCompareB()

Compare two signed integers:
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>if ( a &gt; b )</td>
</tr>
<tr>
<td>0</td>
<td>if ( a = b )</td>
</tr>
<tr>
<td>-1</td>
<td>if ( a &lt; b )</td>
</tr>
</tbody>
</table>

```c
int SignedCompareB(
    const UINT32 aSize, // IN: size of a
    const BYTE *a,     // IN: a buffer
    const UINT32 bSize, // IN: size of b
    const BYTE *b      // IN: b buffer
)
{
    int signA, signB;    // sign of a and b

    // For positive or 0, sign_a is 1
    // for negative, sign_a is 0
    signA = ((a[0] & 0x80) == 0) ? 1 : 0;

    // For positive or 0, sign_b is 1
    // for negative, sign_b is 0
    signB = ((b[0] & 0x80) == 0) ? 1 : 0;

    if(signA != signB)
    {
        return signA - signB;
    }
    if(signA == 1)
    {
        // do unsigned compare function
        return UnsignedCompareB(aSize, a, bSize, b);
    }
    else
    {
        // do unsigned compare the other way
        return 0 - UnsignedCompareB(aSize, a, bSize, b);
    }
}
```

### 9.11.2.3 ModExpB

This function is used to do modular exponentiation in support of RSA. The most typical uses are: \( c = m^e \mod n \) (RSA encrypt) and \( m = c^d \mod n \) (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 \( n \).

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIZE</td>
<td>number to exponentiate is larger than the modulus</td>
</tr>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>result will not fit into the provided buffer</td>
</tr>
</tbody>
</table>

```c
TPM_RC ModExpB(
    UINT32 cSize,  // IN: the size of the output buffer. It will
    BYTE *c,       // OUT: the buffer to receive the results
    const UINT32 mSize,
    const BYTE *m, // IN: number to exponentiate
)
```
```
63  const UINT32  eSize,
64  const BYTE   *e,       // IN: power
65  const UINT32  nSize,    // IN: modulus
66  const BYTE   *n
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>q or r is too small to receive the result</td>
</tr>
</tbody>
</table>

LIB_EXPORT TPM_RC DivideB(
92     const TPM2B   *n,       // IN: numerator
93     const TPM2B   *d,       // IN: denominator
94     TPM2B          *q,       // OUT: quotient
95     TPM2B          *r       // OUT: remainder
96     )
97     { 
98         BN_MAX_INITIALIZED(bnN, n);
99         BN_MAX_INITIALIZED(bnD, d);
100        BN_MAX(bnQ);
101        BN_MAX(bnR);
102        // Do divide with converted values
103        BnDiv(bnQ, bnR, bnN, bnD);
104        // Convert the BIGNUM result back to 2B format using the size of the original number
105        if(q != NULL)
106            if(!BnTo2B(bnQ, q, q->size))
107                return TPM_RC_NO_RESULT;
108        if(r != NULL)
109            if(!BnTo2B(bnR, r, r->size))
110                return TPM_RC_NO_RESULT;
111        return TPM_RC_SUCCESS;
112     }
```
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.

```c
UINT16 AdjustNumberB(  
    TPM2B *num,  
    UINT16 requestedSize
)  
{  
    BYTE *from;  
    UINT16 i;  
    // See if number is already the requested size  
    if(num->size == requestedSize)  
        return requestedSize;  
    from = num->buffer;  
    if (num->size > requestedSize)  
    {  
        // This is a request to shift the number to the left (remove leading zeros)  
        // Find the first non-zero byte. Don't look past the point where removing  
        // more zeros would make the number smaller than requested, and don't throw  
        // away any significant digits.  
        for(i = num->size; *from == 0 && i > requestedSize; from++, i--);  
        if(i < num->size)  
            {  
                num->size = i;  
                MemoryCopy(num->buffer, from, i);  
            }  
    }  
    else  
    {  
        // This is a request to shift the number to the right (add leading zeros)  
        MemoryCopy(&num->buffer[requestedSize - num->size], num->buffer, num->size);  
        MemorySet(num->buffer, 0, requestedSize - num->size);  
        num->size = requestedSize;  
    }  
    return num->size;  
}
```

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.

```c
TPM2B * ShiftLeft(  
    TPM2B *value  // IN/OUT: value to shift and shifted value out
)  
{  
    UINT16 count = value->size;  
    BYTE *buffer = value->buffer;  
    if(count > 0)  
    {  
        for(count -= 1; count > 0; buffer++, count--)  
        {  
            buffer[0] = (buffer[0] << 1) + ((buffer[1] & 0x80) ? 1 : 0);  
        }  
        *buffer <<= 1;
    }
    return value;
}
```
9.11.2.7 IsNumeric()

Verifies that all the characters are simple numeric (0-9)

```c
BOOL IsNumeric(TPM2B *value)
{
    UINT16 i;
    for(i = 0; i < value->size; i++)
    {
        if(value->buffer[i] < '0' || value->buffer[i] > '9')
            return FALSE;
    }
    return TRUE;
}
```
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

```c
#include "Tpm.h"
#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.

```c
void MemoryCopy(
    void *dest,
    const void *src,
    int sSize
) {
    if(dest != src)
        memmove(dest, src, sSize);
}
```

9.12.3.2 MemoryEqual()

This function indicates if two buffers have the same values in the indicated number of bytes.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>all octets are the same</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>all octets are not the same</td>
</tr>
</tbody>
</table>

```c
BOOL MemoryEqual(
    const void *buffer1,  // IN: compare buffer1
    const void *buffer2,  // IN: compare buffer2
    unsigned int size     // IN: size of bytes being compared
) {
    BYTE equal = 0;
    const BYTE *b1 = (BYTE *)buffer1;
    const BYTE *b2 = (BYTE *)buffer2;
    // Compare all bytes so that there is no leakage of information
    // due to timing differences.
    for(; size > 0; size--)
```
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.

```c
LIB_EXPORT INT16 MemoryCopy2B(
    const TPM2B *source, // IN: source TPM2B
    unsigned int dSize, // IN: size of the receiving buffer
)
```

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 | b)).

```c
void MemoryConcat2B(
    TPM2B *aInOut, // IN/OUT: destination 2B
    TPM2B *bIn, // IN: second 2B
    unsigned int aMaxSize // IN: The size of aInOut.buffer (max values for
    //     aInOut.size)
)
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>size and buffer contents are the same</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>size or buffer contents are not the same</td>
</tr>
</tbody>
</table>

```c
BOOL MemoryEqual2B(
)
```
const TPM2B *aIn, // IN: compare value
const TPM2B *bIn // IN: compare value

if(aIn->size != bIn->size)
    return FALSE;
return MemoryEqual(aIn->buffer, bIn->buffer, aIn->size);

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.

void MemorySet(
    void *dest,
    int value,
    size_t size
)
{
    memset(dest, value, size);
}

9.12.3.7 MemoryPad2B()

Function to pad a TPM2B with zeros and adjust the size.

void MemoryPad2B(
    TPM2B *b,
    UINT16 newSize
)
{
    MemorySet(&b->buffer[b->size], 0, newSize - b->size);
    b->size = newSize;
}

9.12.3.8 Uint16ToByteArray()

Function to write an integer to a byte array

void Uint16ToByteArray(
    UINT16 i,
    BYTE *a
)
{
    a[1] = (BYTE)(i); i >>= 8;
    a[0] = (BYTE)(i);
}

9.12.3.9 Uint32ToByteArray()

Function to write an integer to a byte array

void
9.12.3.10 Uint64ToByteArray()

Function to write an integer to a byte array

```c
void Uint64ToByteArray(
    UINT64 i,
    BYTE *a
)
{
    a[7] = (BYTE)(i); i >>= 8;
    a[6] = (BYTE)(i); i >>= 8;
    a[5] = (BYTE)(i); i >>= 8;
    a[4] = (BYTE)(i); i >>= 8;
    a[3] = (BYTE)(i); i >>= 8;
    a[2] = (BYTE)(i); i >>= 8;
    a[1] = (BYTE)(i); i >>= 8;
    a[0] = (BYTE)(i);
}
```

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

```c
UINT8 ByteArrayToUint8(
    BYTE *a
)
{
    return *a;
}
```

9.12.3.12 ByteArrayToUint16()

Function to write an integer to a byte array

```c
UINT16 ByteArrayToUint16(
    BYTE *a
)
{
    return ((UINT16)a[0] << 8) + a[1];
}
```

9.12.3.13 ByteArrayToUint32()

Function to write an integer to a byte array

```c
Uint32ToByteArray(
    UINT32 i,
    BYTE *a
)
{
    a[3] = (BYTE)(i); i >>= 8;
    a[2] = (BYTE)(i); i >>= 8;
    a[1] = (BYTE)(i); i >>= 8;
    a[0] = (BYTE)(i);
}
```
9.12.3.14 **ByteArrayToUint64()**

Function to write an integer to a byte array

```c
UINT64
ByteArrayToUint64(
    BYTE               *a
)
{
    return (((UINT64)BYTE_ARRAY_TO_UINT32(a)) << 32) + BYTE_ARRAY_TO_UINT32(&a[4]);
}
```
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

```c
#define POWER_C
#include "Tpm.h"
```

9.13.3 Functions

9.13.3.1 TPMInit()

This function is used to process a power on event.

```c
void TPMInit()
{
    void
    {
        // Set state as not initialized. This means that Startup is required
        g_initialized = FALSE;
        return;
    }
}
```

9.13.3.2 TPMRegisterStartup()

This function registers the fact that the TPM has been initialized (a TPM2_Startup() has completed successfully).

```c
BOOL TPMRegisterStartup()
{
    void
    {
        g_initialized = TRUE;
        return TRUE;
    }
}
```

9.13.3.3 TPMIsStarted()

Indicates if the TPM has been initialized (a TPM2_Startup() has completed successfully after a _TPM_Init()).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>TPM has been initialized</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>TPM has not been initialized</td>
</tr>
</tbody>
</table>

```c
BOOL TPMIsStarted()
{
    void
    {
        }
    ```
```
25     return g_initialized;
26 }

9.14 PropertyCap.c

9.14.1 Description

This file contains the functions that are used for accessing the TPM_CAP_TPMPROPERTY values.

9.14.2 Includes

```
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>referenced property exists and value set</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>referenced property does not exist</td>
</tr>
</tbody>
</table>

```
static BOOL TPMPropertyIsDefined(
    TPM_PT property,   // IN: property
    UINT32 *value      // OUT: property value
)
{
    switch(property)
    {
    case TPM_PT_FAMILY_INDICATOR:
        // from the title page of the specification
        // For this specification, the value is "2.0".
        *value = TPM_SPEC_FAMILY;
        break;
    case TPM_PT_LEVEL:
        // from the title page of the specification
        *value = TPM_SPEC_LEVEL;
        break;
    case TPM_PT_REVISION:
        // from the title page of the specification
        *value = TPM_SPEC_VERSION;
        break;
    case TPM_PT_DAY_OF_YEAR:
        // computed from the date value on the title page of the specification
        *value = TPM_SPEC_DAY_OF_YEAR;
        break;
    case TPM_PT_YEAR:
        // from the title page of the specification
        *value = TPM_SPEC_YEAR;
        break;
    case TPM_PT_MANUFACTURER:
        // vendor ID unique to each TPM manufacturer
        *value = BYTE_ARRAY_TO_UINT32(MANUFACTURER);
        break;
    case TPM_PT_VENDOR_STRING_1:
        // first four characters of the vendor ID string
        *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_1);
    }`
```
break;
  case TPM_PT_VENDOR_STRING_2:
    // second four characters of the vendor ID string
    #ifdef VENDOR_STRING_2
      *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_2);
    #else
      *value = 0;
    #endif
    break;
  case TPM_PT_VENDOR_STRING_3:
    // third four characters of the vendor ID string
    #ifdef VENDOR_STRING_3
      *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_3);
    #else
      *value = 0;
    #endif
    break;
  case TPM_PT_VENDOR_STRING_4:
    // fourth four characters of the vendor ID string
    #ifdef VENDOR_STRING_4
      *value = BYTE_ARRAY_TO_UINT32(VENDOR_STRING_4);
    #else
      *value = 0;
    #endif
    break;
  case TPM_PT_VENDOR_TPM_TYPE:
    // vendor-defined value indicating the TPM model
    *value = 1;
    break;
  case TPM_PT_FIRMWARE_VERSION_1:
    // more significant 32-bits of a vendor-specific value
    *value = gp.firmwareV1;
    break;
  case TPM_PT_FIRMWARE_VERSION_2:
    // less significant 32-bits of a vendor-specific value
    *value = gp.firmwareV2;
    break;
  case TPM_PT_INPUT_BUFFER:
    // maximum size of TPM2B_MAX_BUFFER
    *value = MAX_DIGEST_BUFFER;
    break;
  case TPM_PT_HR_TRANSIENT_MIN:
    // minimum number of transient objects that can be held in TPM
    // RAM
    *value = MAX_LOADED_OBJECTS;
    break;
  case TPM_PT_HR_PERSISTENT_MIN:
    // minimum number of persistent objects that can be held in
    // TPM NV memory
    // In this implementation, there is no minimum number of
    // persistent objects.
    *value = MIN_EVICT_OBJECTS;
    break;
  case TPM_PT_HR_LOADED_MIN:
    // minimum number of authorization sessions that can be held in
    // TPM RAM
    *value = MAX_LOADED_SESSIONS;
    break;
  case TPM_PT_ACTIVE_SESSIONS_MAX:
    // number of authorization sessions that may be active at a time
    *value = MAX_ACTIVE_SESSIONS;
    break;
  case TPM_PT_PCR_COUNT:
    // number of PCR implemented
    *value = IMPLEMENTATION_PCR;
    break;
case TPM_PT_PCR_SELECT_MIN:
    // minimum number of bytes in a TPMS_PCR_SELECT.sizeOfSelect
    *value = PCR_SELECT_MIN;
    break;

case TPM_PT_CONTEXT_GAP_MAX:
    // maximum allowed difference (unsigned) between the contextID
    // values of two saved session contexts
    *value = ((UINT32)1 << (sizeof(CONTEXT_SLOT) * 8)) - 1;
    break;

case TPM_PT_NV_COUNTERS_MAX:
    // maximum number of NV indexes that are allowed to have the
    // TPMA_NV_COUNTER attribute SET
    // In this implementation, there is no limitation on the number
    // of counters, except for the size of the NV Index memory.
    *value = 0;
    break;

case TPM_PT_NV_INDEX_MAX:
    // maximum size of an NV index data area
    *value = MAX_NV_INDEX_SIZE;
    break;

case TPM_PT_MEMORY:
    // a TPMA_MEMORY indicating the memory management method for the TPM
    {
        union
        {
            TPMA_MEMORY     att;
            UINT32          u32;
        } attributes = {{0}};
        SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, sharedNV);
        SET_ATTRIBUTE(attributes.att, TPMA_MEMORY, objectCopiedToRam);

        // Note: For a LSb0 machine, the bits in a bit field are in the correct
        // order even if the machine is MSb0. For a MSb0 machine, a TPMA will
        // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
        // be NO) so the bits are manipulate correctly.
        *value = attributes.u32;
        break;
    }

case TPM_PT_CLOCK_UPDATE:
    // interval, In seconds, between updates to the copy of
    // TPMS_TIME_INFO .clock in NV
    *value = (1 << NV_CLOCK_UPDATE_INTERVAL);
    break;

case TPM_PT_CONTEXT_HASH:
    // algorithm used for the integrity hash on saved contexts and
    // for digesting the fuData of TPM2_FirmwareRead()
    *value = CONTEXT_INTEGRITY_HASH_ALG;
    break;

case TPM_PT_CONTEXT_SYM:
    // algorithm used for encryption of saved contexts
    *value = CONTEXT_ENCRYPT_ALG;
    break;

case TPM_PT_CONTEXT_SYM_SIZE:
    // size of the key used for encryption of saved contexts
    *value = CONTEXT_ENCRYPT_KEY_BITS;
    break;

case TPM_PT_ORDERLY_COUNT:
    // maximum difference between the volatile and non-volatile
    // versions of TPMA_NV_COUNTER that have TPMA_NV_ORDERLY SET
    *value = MAX_ORDERLY_COUNT;
    break;

case TPM_PT_MAX_COMMAND_SIZE:
    // maximum value for 'commandSize'
    *value = MAX_COMMAND_SIZE;
    break;

case TPM_PT_MAX_RESPONSE_SIZE:
170     
171     // maximum value for 'responseSize'
172     *value = MAX_RESPONSE_SIZE;
173     break;
174   
175   case TPM_PT_MAX_DIGEST:
176     // maximum size of a digest that can be produced by the TPM
177     *value = sizeof(TPMU_HA);
178     break;
179   
180   case TPM_PT_MAX_OBJECT_CONTEXT:
181     // Header has 'sequence', 'handle' and 'hierarchy'
182   
183   #define SIZE_OF_CONTEXT_HEADER
184     sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) + sizeof(TPMI_RH_HIERARCHY)
185   
186   #define SIZE_OF_CONTEXT_INTEGRITY (sizeof(UINT16) + CONTEXT_INTEGRITY_HASH_SIZE)
187   
188   #define SIZE_OF_FINGERPRINT sizeof(UINT64)
189   
190   #define SIZE_OF_CONTEXT_BLOB_OVERHEAD
191     (sizeof(UINT16) + SIZE_OF_CONTEXT_INTEGRITY + SIZE_OF_FINGERPRINT)
192   
193   #define SIZE_OF_CONTEXT_OVERHEAD
194     (SIZE_OF_CONTEXT_HEADER + SIZE_OF_CONTEXT_BLOB_OVERHEAD)
195   
196   #if 0
197     // maximum size of a TPMS_CONTEXT that will be returned by
198     // TPM2_ContextSave for object context
199     *value = 0;
200     // adding sequence, saved handle and hierarchy
201     *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
202     sizeof(TPMI_RH_HIERARCHY);
203     // add size field in TPM2B_CONTEXT
204     *value += sizeof(UINT16);
205     // add integrity hash size
206     *value += sizeof(UINT16) +
207     CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
208     // Add fingerprint size, which is the same as sequence size
209     *value += sizeof(UINT64);
210     // Add OBJECT structure size
211     *value += sizeof(OBJECT);
212     #else
213     // the maximum size of a TPMS_CONTEXT that will be returned by
214     // TPM2_ContextSave for object context
215     *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(OBJECT);
216     #endif
217     break;
218   
219   case TPM_PT_MAX_SESSION_CONTEXT:
220     break;
221   
222   #if 0
223     // the maximum size of a TPMS_CONTEXT that will be returned by
224     // TPM2_ContextSave for object context
225     *value = 0;
226     // adding sequence, saved handle and hierarchy
227     *value += sizeof(UINT64) + sizeof(TPMI_DH_CONTEXT) +
228     sizeof(TPMI_RH_HIERARCHY);
229     // Add size field in TPM2B_CONTEXT
230     *value += sizeof(UINT16);
231     // Add integrity hash size
232     *value += sizeof(UINT16) +
233     CryptHashGetDigestSize(CONTEXT_INTEGRITY_HASH_ALG);
234     // Add fingerprint size, which is the same as sequence size
235     *value += sizeof(UINT64);
236     // Add SESSION structure size
237     *value += sizeof(SESSION);
238     #else
239     // the maximum size of a TPMS_CONTEXT that will be returned by
240     // TPM2_ContextSave for object context
241     *value = SIZE_OF_CONTEXT_OVERHEAD + sizeof(SESSION);
242     #endif
243     break;
244   
245   case TPM_PT_PS_FAMILY_INDICATOR:
246     // platform specific values for the TPM_PT_PS parameters from
247     // the relevant platform-specific specification
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 /*value = 0;
273 #if
274 COMMAND_INDEX commandIndex;
275 for(commandIndex = GetClosestCommandIndex(0);
276 commandIndex != UNIMPLEMENTED_COMMAND_INDEX;
277 commandIndex = GetNextCommandIndex(commandIndex))
278 {
279 (*value)++;
280 }
281 #endif
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 *value = 0;
292 #if
293 COMMAND_INDEX commandIndex;
294 for(commandIndex = GetClosestCommandIndex(0);
295 commandIndex < LIBRARY_COMMAND_ARRAY_SIZE;
296 commandIndex = GetNextCommandIndex(commandIndex))
297 {
298 (*value)++;
299 }
300 #endif
301 }
case TPM_PT_VENDOR_COMMANDS:
    // number of vendor commands that are implemented
    *value = VENDOR_COMMAND_ARRAY_SIZE;
    break;

case TPM_PT_NV_BUFFER_MAX:
    // Maximum data size in an NV write command
    *value = MAX_NV_BUFFER_SIZE;
    break;

case TPM_PT_MODES:
    
    #if FIPS_COMPLIANT
        *value = 1;
    #else
        *value = 0;
    #endif
    break;

case TPM_PT_MAX_CAP_BUFFER:
    *value = MAX_CAP_BUFFER;
    break;

    // Start of variable commands

    case TPM_PT_PERMANENT:
        // TPMA_PERMANENT
        {
            union {
                TPMA_PERMANENT   attr;
                UINT32           u32;
            } flags = {{0}};

            if(gp.ownerAuth.t.size != 0)
                SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, ownerAuthSet);
            if(gp.endorsementAuth.t.size != 0)
                SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, endorsementAuthSet);
            if(gp.lockoutAuth.t.size != 0)
                SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, lockoutAuthSet);
            if(gp.disableClear)
                SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, disableClear);
            if(gp.failedTries >= gp.maxTries)
                SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, inLockout);

            // In this implementation, EPS is always generated by TPM
            SET_ATTRIBUTE(flags.attr, TPMA_PERMANENT, tpmGeneratedEPS);

            // Note: For a Lsb0 machine, the bits in a bit field are in the correct
            // order even if the machine is MSb0. For a MSb0 machine, a TPMA will
            // be an integer manipulated by masking (USE_BIT_FIELD_STRUCTURES will
            // be NO) so the bits are manipulate correctly.
            *value = flags.u32;
            break;
        }

    case TPM_PT_STARTUP_CLEAR:
        // TPMA_STARTUP_CLEAR
        {
            union {
                TPMA_STARTUP_CLEAR attr;
                UINT32           u32;
            } flags = {{0}};

            if(g_phEnable)
                SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnable);
            if(gc.shEnable)
                SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, shEnable);
            if(gc.ehEnable)
                SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, ehEnable);
            if(gc.phEnableNV)
                SET_ATTRIBUTE(flags.attr, TPMA_STARTUP_CLEAR, phEnableNV);
            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;
#else
  # ALG_ECC
  *value = 0;
#endif

break;

case TPM_PT_LOCKOUT_COUNTER:
  // current value of the lockout counter
  *value = gp.failedTries;
  break;

case TPM_PT_MAX_AUTH_FAIL:
  // number of authorization failures before DA lockout is invoked
  *value = gp.maxTries;
  break;

case TPM_PT_LOCKOUT_INTERVAL:
  // number of seconds before the value reported by
  // TPM_PT_LOCKOUT_COUNTER is decremented
  *value = gp.recoveryTime;
  break;

case TPM_PT_LOCKOUT_RECOVERY:
  // number of seconds after a lockoutAuth failure before use of
  // lockoutAuth may be attempted again
  *value = gp.lockoutRecovery;
  break;

case TPM_PT_NV_WRITE_RECOVERY:
  // number of milliseconds before the TPM will accept another command
  // that will modify NV.
  // This should make a call to the platform code that is doing rate
  // limiting of NV. Rate limiting is not implemented in the reference
  // code so no call is made.
  *value = 0;
  break;

case TPM_PT_AUDIT_COUNTER_0:
  // high-order 32 bits of the command audit counter
  *value = (UINT32)(gp.auditCounter >> 32);
  break;

case TPM_PT_AUDIT_COUNTER_1:
  // low-order 32 bits of the command audit counter
  *value = (UINT32)(gp.auditCounter);
  break;

default:
  // property is not defined
  return FALSE;

break;

}

return TRUE;

}


```c
}  

TPM_STATE more = NO;
UINT32    i;
UINT32    nextGroup;

// initialize output property list
propertyList->count = 0;

// maximum count of properties we may return is MAX_PCR_PROPERTIES
if(count > MAX_TPM_PROPERTIES) count = MAX_TPM_PROPERTIES;

// if property is less than PT_FIXED, start from PT_FIXED
if(property < PT_FIXED)
    property = PT_FIXED;
// There is only the fixed and variable groups with the variable group coming last
if(property >= (PT_VAR + PT_GROUP))
    return more;

// Don't read past the end of the selected group
nextGroup = ((property / PT_GROUP) * PT_GROUP) + PT_GROUP;

// Scan through the TPM properties of the requested group.
for(i = property; i < nextGroup; i++)
{
    UINT32          value;
    // if we have hit the end of the group, quit
    if(i != property && ((i % PT_GROUP) == 0))
        break;
    if(TPMPropertyIsDefined((TPM_PT)i, &value))
    {
        if(propertyList->count < count)
            {  
                // If the list is not full, add this property
                propertyList->tpmProperty[propertyList->count].property = (TPM_PT)i;
                propertyList->tpmProperty[propertyList->count].value = value;
                propertyList->count++;
            }
        else
            {  
                // If the return list is full but there are more properties available, set the indication and exit the loop.
                more = YES;
                break;
            }
    }
    return more;
```
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

```c
#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.

```c
void BuildResponseHeader(
    COMMAND *command, // IN: main control structure
    BYTE *buffer, // OUT: the output buffer
    TPM_RC result // IN: the response code
) {
    TPM_ST tag;
    UINT32 size;

    if (result != TPM_RC_SUCCESS) {
        tag = TPM_ST_NO_SESSIONS;
        size = 10;
    } else {
        tag = command->tag;
        // Compute the overall size of the response
        size = STD_RESPONSE_HEADER + command->handleNum * sizeof(TPM_HANDLE);
        size += command->parameterSize;
        size += (command->tag == TPM_ST_SESSIONS) ?
            command->authSize + sizeof(UINT32) : 0;
    }
    TPM_ST_Marshal(&tag, &buffer, NULL);
    UINT32_Marshal(&size, &buffer, NULL);
    TPM_RC_Marshal(&result, &buffer, NULL);
    if (result == TPM_RC_SUCCESS) {
        if (command->handleNum > 0)
            TPM_HANDLE_Marshal(&command->handles[0], &buffer, NULL);
        if (tag == TPM_ST_SESSIONS)
            UINT32_Marshal((UINT32 *)&command->parameterSize, &buffer, NULL);
    }
    command->parameterSize = size;
}
```
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

```
#define TPM_FAIL_C
#include "Tpm.h"
#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.

```
#include "TpmTypes.h"
```

9.17.2  Typedefs

These defines are used primarily for sizing of the local response buffer.

```
typedef struct {
    TPM_ST          tag;
    UINT32          size;
    TPM_RC          code;
} HEADER;
```

```
typedef struct {
    BYTE            tag[sizeof(TPM_ST)];
    BYTE            size[sizeof(UINT32)];
    BYTE            code[sizeof(TPM_RC)];
} PACKED_HEADER;
```

```
typedef struct {
    BYTE             size[sizeof(UINT16)];
    struct {
        BYTE         function[sizeof(UINT32)];
        BYTE         line[sizeof(UINT32)];
        BYTE         code[sizeof(UINT32)];
    } values;
    BYTE             returnCode[sizeof(TPM_RC)];
} GET_TEST_RESULT_PARAMETERS;
```

```
typedef struct {
    BYTE         moreData[sizeof(TPML_YES_NO)];
    BYTE         capability[sizeof(TPM_CAP)]; // Always TPM_CAP_TPM_PROPERTIES
    BYTE         tpmProperty[sizeof(TPML_TAGGED_TPM_PROPERTY)];
} GET_CAPABILITY_PARAMETERS;
```

```
typedef struct {
    BYTE             header[sizeof(PACKED_HEADER)];
    BYTE             getTestResult[sizeof(GET_TEST_RESULT_PARAMETERS)];
} TEST_RESPONSE;
```

```
typedef struct {
    BYTE             header[sizeof(PACKED_HEADER)];
    BYTE             getCap[sizeof(GET_CAPABILITY_PARAMETERS)];
} CAPABILITY_RESPONSE;
```

```
typedef union {
```
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.

9.17.3 Local Functions

9.17.3.1 MarshalUint16()

Function to marshal a 16 bit value to the output buffer.

```
static INT32 MarshalUint16(
    UINT16 integer,
    BYTE **buffer
)
{
    UINT16_TO_BYTE_ARRAY(integer, *buffer);
    *buffer += 2;
    return 2;
}
```

9.17.3.2 MarshalUint32()

Function to marshal a 32 bit value to the output buffer.

```
static INT32 MarshalUint32(
    UINT32 integer,
    BYTE **buffer
)
{
    UINT32_TO_BYTE_ARRAY(integer, *buffer);
    *buffer += 4;
    return 4;
}
```

9.17.3.3 Unmarshal32()

```
static BOOL Unmarshal32(
    UINT32 *target,
    BYTE **buffer,
    INT32 *size
)
{
    if(*size -= 4) < 0)
        return FALSE;
    *target = BYTE_ARRAY_TO_UINT32(*buffer);
    *buffer += 4;
    return TRUE;
```
9.17.3.4 Unmarshal16()

```c
static BOOL Unmarshal16(
    UINT16 *target,
    BYTE **buffer,
    INT32 *size
) {
    if (*((size -= 2) < 0)
        return FALSE;
    *target = BYTE_ARRAY_TO_UINT16(*buffer);
    *buffer += 2;
    return TRUE;
}
```

9.17.4 Public Functions

9.17.4.1 SetForceFailureMode()

This function is called by the simulator to enable failure mode testing.

```c
#if SIMULATION
LIB_EXPORT void SetForceFailureMode(
    void
) {
    g_forceFailureMode = TRUE;
    return;
}
#endif
```

9.17.4.2 TpmLogFailure()

This function saves the failure values when the code will continue to operate. It if similar to TpmFail() but returns to the caller. The assumption is that the caller will propagate a failure back up the stack.

```c
void TpmLogFailure(
    #if FAIL_TRACE
    const char *function,
    int line,
    #endif
    int code
) {
    // Save the values that indicate where the error occurred.
    // On a 64-bit machine, this may truncate the address of the string
    // of the function name where the error occurred.
    #if FAIL_TRACE
    s_failFunction = (UINT32)function;
    s_failLine = line;
    #else
    s_failFunction = 0;
    s_failLine = 0;
    #endif
    s_failCode = code;
    // We are in failure mode
```
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().

128     g_inFailureMode = TRUE;
129
130     return;
131 }

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  case TPM_CC_GetTestResult:
199     // make sure that the command size is correct
200     if (header.size != 10)
201        goto FailureModeReturn;
202     buffer = &response[10];
203     marshalSize = MarshalUint16(3 * sizeof(UINT32), &buffer);
204     marshalSize += MarshalUint32(s_failFunction, &buffer);
205     marshalSize += MarshalUint32(s_failLine, &buffer);
206     marshalSize += MarshalUint32(s_failCode, &buffer);
207     if (s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
208        marshalSize += MarshalUint32(TPM_RC_NV_UNINITIALIZED, &buffer);
209     else
210        marshalSize += MarshalUint32(TPM_RC_FAILURE, &buffer);
211     break;
212  case TPM_CC_GetCapability:
213     // make sure that the size of the command is exactly the size
214     // returned for the capability, property, and count
215     if (header.size != (10 + (3 * sizeof(UINT32))
216        || !Unmarshal32(&capability, &buffer, &size)
217        || !Unmarshal32(&pt, &buffer, &size)
218        || !Unmarshal32(&count, &buffer, &size))
219        goto FailureModeReturn;
220     // If in failure mode because of an unrecoverable read error, and the
221     // property is 0 and the count is 0, then this is an indication to
222     // re-manufacture the TPM. Do the re-manufacture but stay in failure
223     // mode until the TPM is reset.
224     // Note: this behavior is not required by the specification and it is
225     // OK to leave the TPM permanently bricked due to an unrecoverable NV
226     // error.
227     if (count == 0 && pt == 0 && s_failCode == FATAL_ERROR_NV_UNRECOVERABLE)
228        {
229         g_manufactured = FALSE;
230         TPM_Manufacture(0);
231        }
232     if (count > 0)
233        count = 1;
234     else if (pt > TPM_PT_FIRMWARE_VERSION_2)
235        count = 0;
236     if (pt < TPM_PT_MANUFACTURER)
237        pt = TPM_PT_MANUFACTURER;
238     // set up for return
239     buffer = &response[10];
240     // if the request was for a PT less than the last one
241     // then we indicate more, otherwise, not.
242     if (pt < TPM_PT_FIRMWARE_VERSION_2)
*buffer++ = YES;
else
*buffer++ = NO;
marshalSize = 1;

// indicate the capability type
marshalSize += MarshalUint32(capability, &buffer);
// indicate the number of values that are being returned (0 or 1)
marshalSize += MarshalUint32(count, &buffer);
// indicate the property
marshalSize += MarshalUint32(pt, &buffer);

if(count > 0)
switch(pt)
{
case TPM_PT_MANUFACTURER:
    // the vendor ID unique to each TPM manufacturer
    #ifdef MANUFACTURER
    pt = *(UINT32*)MANUFACTURER;
    #else
    pt = 0;
    #endif
    break;
case TPM_PT_VENDOR_STRING_1:
    // the first four characters of the vendor ID string
    #ifdef VENDOR_STRING_1
    pt = *(UINT32*)VENDOR_STRING_1;
    #else
    pt = 0;
    #endif
    break;
case TPM_PT_VENDOR_STRING_2:
    // the second four characters of the vendor ID string
    #ifdef VENDOR_STRING_2
    pt = *(UINT32*)VENDOR_STRING_2;
    #else
    pt = 0;
    #endif
    break;
case TPM_PT_VENDOR_STRING_3:
    // the third four characters of the vendor ID string
    #ifdef VENDOR_STRING_3
    pt = *(UINT32*)VENDOR_STRING_3;
    #else
    pt = 0;
    #endif
    break;
case TPM_PT_VENDOR_STRING_4:
    // the fourth four characters of the vendor ID string
    #ifdef VENDOR_STRING_4
    pt = *(UINT32*)VENDOR_STRING_4;
    #else
    pt = 0;
    #endif
    break;
case TPM_PT_VENDOR_TPM_TYPE:
    // vendor-defined value indicating the TPM model
    // We just make up a number here
    pt = 1;
    break;
case TPM_PT_FIRMWARE_VERSION_1:
    // the more significant 32-bits of a vendor-specific value
    // indicating the version of the firmware
    #ifdef FIRMWARE_V1
    pt = FIRMWARE_V1;
    #else
    pt = 0;
    #endif
    break;
}
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...

```c
void UnmarshalFail(
    void *type,
    BYTE **buffer,
    INT32 *size
)
```

```c
    NOT_REFERENCED(type);
    NOT_REFERENCED(buffer);
    NOT_REFERENCED(size);
    FAIL(FATAL_ERROR_INTERNAL);
```
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 \(\text{crypt}_u\text{word}_t\). 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 \(\text{used}-1\) should never be zero. All words between \(d[\text{size}]\) and \(d[\text{allocated}-1]\) 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) BIG_ENDIAN_BYTES_TO_UINT64(e, f, g, h), BIG_ENDIAN_BYTES_TO_UINT32(a, b, c, d),
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. #define TO_CRYPT_WORD_64(a, b, c, d) TO_CRYPT_WORD_64(0, 0, 0, 0, a, b, c, d)
19. #elif RADIX_BITS == 32
20. # define TO_CRYPT_WORD_32(a, b, c, d) TO_CRYPT_WORD_32(0, 0, 0, 0, a, b, c, d)
21. #endif
22. #define MAX_CRYPT_UWORD (~((crypt_uword_t)0))
23. #define MAX_CRYPT_WORD (MAX_CRYPT_UWORD >> 1)
24. #define MIN_CRYPT_WORD (~MAX_CRYPT_WORD)
25. #define LARGEST_NUMBER (MAX((ALG_RSA * MAX_RSA_KEY_BYTES), MAX((ALG_ECC * MAX_ECC_KEY_BYTES), MAX_DIGEST_SIZE)))
26. #define LARGEST_NUMBER_BITS (LARGEST_NUMBER * 8)
These are the basic big number formats. This is convertible to the library-specific format without too much difficulty. For the math performed using these numbers, the value is always positive.

```c
#define BN_STRUCT_DEF(count) struct {
  crypt_uword_t allocated;
  crypt_uword_t size;
  crypt_uword_t d[count];
} 

typedef BN_STRUCT_DEF(1) bignum_t;
#endif

extern const bignum_t BnConstZero;
```

The Functions to access the properties of a big number. Get number of allocated words

```c
#define BnGetAllocated(x)   ((unsigned)((x)->allocated)
```

Get number of words used

```c
#define BnGetSize(x)        ((x)->size)
```

Get a pointer to the data array

```c
#define BnGetArray(x)       ((crypt_uword_t *)&((x)->d[0]))
```

Get the nth word of a BIGNUM (zero-based)

```c
#define BnGetWord(x, i)     (crypt_uword_t)((x)->d[i])
```

Some things that are done often. Test to see if a bignum_t is equal to zero

```c
#define BnEqualZero(bn)   (BnGetSize(bn) == 0)
```

Test to see if a bignum_t is equal to a word type

```c
#define BnEqualWord(bn, word)  
  ((BnGetSize(bn) == 1) && (BnGetWord(bn, 0) == (crypt_uword_t)word))
```

Determine if a BIGNUM is even. A zero is even. Although the indication that a number is zero is that it's size is zero, all words of the number are 0 so this test works on zero.

```c
#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.

```c
#define BN_INIT(name)  
  (bigNum)BnInit((bigNum)&(name),(BYTES_TOCRYPT_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_

```c
#define BN_ADDRESS(name) (bigNum)&name##_  
#define BN_STRUCT_ALLOCATION(bits) (BITS_TOCRYPT_WORDS(sizeof(name.d)))
```
Create a structure of the correct size.

```c
#define BN_STRUCT(bits) \
  BN_STRUCT_DEF(BN_STRUCT_ALLOCATION(bits))
```

Define a BIGNUM type with a specific allocation

```c
#define BN_TYPE(name, bits) \
  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.

```c
#define BN_INITIALIZED(name, bits, initializer) \
  BN_STRUCT(bits) name##_; \
  bigNum name = BnFrom2B(BN_INIT(name##_), \ 
                     (const TPM2B *)initializer)
```

Create a local variable that can hold a number with `bits`

```c
#define BN_VAR(name, bits) \
  BN_STRUCT(bits) ##name; \
  bigNum name = BN_INIT(##name)
```

Create a type that can hold the largest number defined by the implementation.

```c
#define BN_MAX(name) BN_VAR(name, LARGEST_NUMBER_BITS) \
#define BN_MAX_INITIALIZED(name, initializer) \
  BN_INITIALIZED(name, LARGEST_NUMBER_BITS, initializer)
```

A word size value is useful

```c
#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.

```c
#define BN_WORD_INITIALIZED(name, initial) \
  BN_INITIALIZED(name, 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

```c
typedef struct _bn_point_t \
{ \
  bigNum x; \
  bigNum y; \
  bigNum z; \
} bn_point_t; \
typedef bn_point_t *bigPoint; \
typedef const bn_point_t *pointConst; \
typedef struct constant_point_t \
{ \
  bigConst x; \
  bigConst y; \
  bigConst z; \
} constant_point_t; \
#define ECC_BITS (MAX_ECC_KEY_BYTES * 8)
```
BN_TYPE(ecc, ECC_BITS);
#define ECC_NUM(name) BN_VAR(name, ECC_BITS)
#define ECC_INITIALIZED(name, initializer)
#define POINT_INSTANCE(name, bits)
BN_STRUCT (bits) name##_x = {BITS_TO_CRYPT_WORDS (bits), 0, {0}};
BN_STRUCT (bits) name##_y = {BITS_TO_CRYPT_WORDS (bits), 0, {0}};
BN_STRUCT (bits) name##_z = {BITS_TO_CRYPT_WORDS (bits), 0, {0}};
bn_point_t name##_;
#define POINT_INITIALIZER(name)
BnInitializePoint(&name##_, (bigNum)&name##_x,
(bigNum)&name##_y, (bigNum)&name##_z);
#define POINT_INITIALIZED(name, initValue)
POINT_INSTANCE(name, MAX_ECC_KEY_BITS);
bigPoint name = BnPointFrom2B(
POINT_INITIALIZER(name),
initValue);
#define POINT_VAR(name, bits)
POINT_INSTANCE (name, bits);
bigPoint name = POINT_INITIALIZER(name)
#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

typedef struct
{
    bigConst prime;  // a prime number
    bigConst order;  // the order of the curve
    bigConst h;      // cofactor
    bigConst a;      // linear coefficient
    bigConst b;      // constant term
    constant_point_t base;  // base point
} 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 noting.

#define CurveGetPrime(C) ((C)->prime)
#define CurveGetOrder(C) ((C)->order)
#define CurveGetCofactor(C) ((C)->h)
#define CurveGet_a(C) ((C)->a)
#define CurveGet_b(C) ((C)->b)
#define CurveGetG(C) ((pointConst)&((C)->base))
#define CurveGetGx(C) ((C)->base.x)
#define CurveGetGy(C) ((C)->base.y)

Convert bytes in initializers according to the endianess of the system. This is used for CryptEccData.c.

#define BIG_ENDIAN_BITS_TO_UINT32(a, b, c, d)
( ((UINT32)(a) << 24) + ((UINT32)(b) << 16) + ((UINT32)(c) << 8) + ((UINT32)(d)) )
#define BIG_ENDIAN_BITS_TO_UINT64(a, b, c, d, e, f, g, h)
( ((UINT64)(a) << 56) + ((UINT64)(b) << 48) + ((UINT64)(c) << 40) + ((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.

```c
#ifndef _CRYPT_ECC_H
#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 to much/not enough initialization data in the curve. The macro is used because not all versions of the CryptEccData.c need the curve name.

```c
#ifdef NAMED_CURVES
#define CURVE_NAME(a) , a
#define CURVE_NAME_DEF const char *name;
#else
#define CURVE_NAME(a)
#define CURVE_NAME_DEF
#endif

typedef struct ECC_CURVE {
    const TPM_ECC_CURVE curveId;
    const UINT16 keySizeBits;
    const TPMT_KDF_SCHEME kdf;
    const TPMT_ECC_SCHEME sign;
    const ECC_CURVE_DATA *curveData; // the address of the curve data
    const BYTE *OID;
    CURVE_NAME_DEF
} ECC_CURVE;

extern const ECC_CURVE eccCurves[ECC_CURVE_COUNT];
```
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.

```c
#ifndef _CRYPT_HASH_H
#define _CRYPT_HASH_H

union SMAC_STATES;

typedef void(* SMAC_DATA_METHOD)(
    union SMAC_STATES *state,
    UINT32 size,
    const BYTE *buffer
);

typedef UINT16(* SMAC_END_METHOD)(
    union SMAC_STATES *state,
    UINT32 size,
    BYTE *buffer
);

typedef struct sequenceMethods {
    SMAC_DATA_METHOD data;
    SMAC_END_METHOD end;
} SMAC_METHODS;

#define SMAC_IMPLEMENTED (CC_MAC || CC_MAC_Start)

These definitions are here because the SMAC state is in the union of hash states.

typedef struct tpmCmacState {
    TPM_ALG_ID symAlg;
    UINT16 keySizeBits;
    INT16 bcount; // current count of bytes accumulated in IV
    TPM2B_IV iv; // IV buffer
    TPM2B_SYM_KEY symKey;
} tpmCmacState_t;

typedef union SMAC_STATES {
    #if ALG_CMAC
    tpmCmacState_t mac;
    #endif
    UINT64 pad;
} SMAC_STATES;

typedef struct SMAC_STATE {
    SMAC_METHODS smacMethods;
    SMAC_STATES state;
} SMAC_STATE;

typedef union {
    #if ALG_SHA1
    tpmHashStateSHA1_t Shal;
    #endif
    #if ALG_SHA256
    tpmHashStateSHA256_t Sha256;
    #endif
```

These definitions are here because the SMAC state is in the union of hash states.
#if ALG_SHA384
  tpmHashStateSHA384_t   Sha384;
#endif

#if ALG_SHA512
  tpmHashStateSHA512_t   Sha512;
#endif

// Additions for symmetric block cipher MAC
#if SMAC_IMPLEMENTED
  SMAC_STATE             smac;
#endif

// to force structure alignment to be no worse than HASH_ALIGNMENT
#if HASH_ALIGNMENT == 4
  uint32_t             align;
#else
  uint64_t             align;
#endif

} ANY_HASH_STATE;

typedef ANY_HASH_STATE *PANY_HASH_STATE;

typedef const ANY_HASH_STATE    *PCANY_HASH_STATE;

#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

#define MAX_HASH_STATE_SIZE ((2 * MAX_HASH_BLOCK_SIZE) + 16)
#define MAX_HASH_STATE_SIZE_ALIGNED ALIGNED_SIZE(MAX_HASH_STATE_SIZE, HASH_ALIGNMENT)

This is an aligned byte array that will hold any of the hash contexts.

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

#ifndef HASH_START_METHOD_DEF
#   define HASH_START_METHOD_DEF void (HASH_START_METHOD)(void)
#endif

#ifndef HASH_DATA_METHOD_DEF
#   define HASH_DATA_METHOD_DEF void (HASH_DATA_METHOD)(void)
#endif

#ifndef HASH_END_METHOD_DEF
#   define HASH_END_METHOD_DEF void (HASH_END_METHOD)(void)
#endif

#ifndef HASH_STATE_COPY_METHOD_DEF
#   define HASH_STATE_COPY_METHOD_DEF void (HASH_STATE_COPY_METHOD)(void)
#endif

#ifndef HASH_STATE_EXPORT_METHOD_DEF
#   define HASH_STATE_EXPORT_METHOD_DEF void (HASH_STATE_EXPORT_METHOD)(void)
#endif

#ifndef HASH_STATE_IMPORT_METHOD_DEF
#   define HASH_STATE_IMPORT_METHOD_DEF void (HASH_STATE_IMPORT_METHOD)(void)
#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.

typedef HASH_START_METHOD_DEF;
typedef HASH_DATA_METHOD_DEF;
typedef HASH_END_METHOD_DEF;
typedef HASH_STATE_COPY_METHOD_DEF;
typedef HASH_STATE_EXPORT_METHOD_DEF;
typedef HASH_STATE_IMPORT_METHOD_DEF;
typedef struct _HASH_METHODS
{
    HASH_START_METHOD           *start;
    HASH_DATA_METHOD            *data;
    HASH_END_METHOD             *end;
    HASH_STATE_COPY_METHOD      *copy;   // Copy a hash block
    HASH_STATE_EXPORT_METHOD    *copyOut; // Copy a hash block from a hash context
    HASH_STATE_IMPORT_METHOD    *copyIn;  // Copy a hash block to a proper hash context
}

#define PKCS1_HASH_REF(const BYTE  *PKCS1;
#define PKCS1_OID(NAME), OID_PKCS1_##NAME
#else
#define PKCS1_HASH_REF
#define PKCS1_OID(NAME)
#endif

#define ECDSA_HASH_REF(const BYTE  *ECDSA;
#define ECDSA_OID(NAME), OID_ECDSA_##NAME
#else
#define ECDSA_HASH_REF
#define ECDSA_OID(NAME)
#endif

typedef const struct HASH_DEF
{
    HASH_METHODS         method;
    uint16_t             blockSize;
    uint16_t             digestSize;
    uint16_t             contextSize;
    uint16_t             hashAlg;
    const BYTE          *OID;
    PKCS1_HASH_REF      // PKCS1 OID
    ECDSA_HASH_REF      // ECDSA OID
}

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.

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.
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.

```
#define HASH_DEF_TEMPLATE(HASH, Hash)

    HASH_DEF Hash##_Def = {
        (HASH_START_METHOD *)&tpmHashStart_##HASH,
        (HASH_DATA_METHOD *)&tpmHashData_##HASH,
        (HASH_END_METHOD *)&tpmHashEnd_##HASH,
        (HASH_STATE_COPY_METHOD *)&tpmHashStateCopy_##HASH,
        (HASH_STATE_EXPORT_METHOD *)&tpmHashStateExport_##HASH,
        (HASH_STATE_IMPORT_METHOD *)&tpmHashStateImport_##HASH,
        HASH##_BLOCK_SIZE, /*block size*/
        HASH##_DIGEST_SIZE, /*data size*/
        sizeof(tpmHashState##HASH##_t),
        TPM_ALG_##HASH, OID_##HASH,
        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.

```
typedef BYTE HASH_STATE_TYPE;
#define HASH_STATE_EMPTY ((HASH_STATE_TYPE) 0)
#define HASH_STATE_HASH   ((HASH_STATE_TYPE) 1)
#define HASH_STATE_HMAC   ((HASH_STATE_TYPE) 2)
```

```
#if CC_MAC || CC_MAC_Start
#define HASH_STATE_SMAC ((HASH_STATE_TYPE) 3)
#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.

```
typedef struct _HASH_STATE
{
    HASH_STATE_TYPE type;        // type of the context
    TPM_ALG_ID hashAlg;
    PHASH_DEF def;
    ANY_HASH_STATE state;
} HASH_STATE, *PHASH_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.

```
typedef struct hmacState
{
    HASH_STATE hashState;        // the hash state
    TPM2B_HASH_BLOCK hmacKey;    // the HMAC key
} 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 is true.

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.

```c
#ifndef _CRYPT_RAND_H
#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

#define DRBG_KEY_SIZE_BITS AES_MAX_KEY_SIZE_BITS
#define DRBG_IV_SIZE_BITS (AES_MAX_BLOCK_SIZE * 8)
#define DRBG_ALGORITHM TPM_ALG_AES
typedef tpmKeyScheduleAES DRBG_KEY_SCHEDULE;
#define DRBG_ENCRYPT_SETUP(key, keySizeInBits, schedule)
   TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule)
#define DRBG_ENCRYPT(keySchedule, in, out)
   TpmCryptEncryptAES(SWIZZLE(keySchedule, in, out))
#if ((DRBG_KEY_SIZE_BITS % RADIX_BITS) != 0) 
   || ((DRBG_IV_SIZE_BITS % RADIX_BITS) != 0)
#error "Key size and IV for DRBG must be even multiples of the radix"
#endif
#if (DRBG_KEY_SIZE_BITS % DRBG_IV_SIZE_BITS) != 0
#error "Key size for DRBG must be even multiple of the cypher block size"
#endif

Derived values

#define DRBG_MAX_REQUESTS_PER_RESEED (1 << 48)
#define DRBG_MAX_REQUEST_SIZE (1 << 32)
#define pDRBG_KEY(seed)   ((DRBG_KEY *)&(((BYTE *)(seed))[0]))
#define pDRBG_IV(seed)    ((DRBG_IV *)&(((BYTE *)(seed))[DRBG_KEY_SIZE_BYTES]))
#define DRBG_KEY_SIZE_WORDS     (BITS_TO_CRYPT_WORDS(DRBG_KEY_SIZE_BITS))
#define DRBG_KEY_SIZE_BYTES     (DRBG_KEY_SIZE_WORDS * RADIX_BYTES)
#define DRBG_IV_SIZE_WORDS      (BITS_TO_CRYPT_WORDS(DRBG_IV_SIZE_BITS))
#define DRBG_IV_SIZE_BYTES      (DRBG_IV_SIZE_WORDS * RADIX_BYTES)
#define DRBG_SEED_SIZE_WORDS    (DRBG_KEY_SIZE_WORDS + DRBG_IV_SIZE_WORDS)
#define DRBG_SEED_SIZE_BYTES    (DRBG_KEY_SIZE_BYTES + DRBG_IV_SIZE_BYTES)

tyedef union
{
   BYTE bytes[DRBG_KEY_SIZE_BYTES];
crypt_uword_t words[DRBG_KEY_SIZE_WORDS];
} DRBG_KEY;
tyedef union
{
   BYTE bytes[DRBG_IV_SIZE_BYTES];
crypt_uword_t words[DRBG_IV_SIZE_WORDS];
} DRBG_IV;
```
#define CTR_DRBG_MAX_REQUESTS_PER_RESEED ((UINT64)1 << 20)
#define CTR_DRBG_MAX_BYTES_PER_REQUEST (1 << 16)
#define CTR_DRBG_MAX_ENTROPY_INPUT_LENGTH DRBG_SEED_SIZE_BYTES
#define CTR_DRBG_MAX_ADDITIONAL_INPUT_LENGTH DRBG_SEED_SIZE_BYTES
#define TESTING (1 << 0)
#define ENTROPY (1 << 1)
#define TESTED (1 << 2)
#define IsTestStateSet(BIT) ((g_cryptoSelfTestState.rng & BIT) != 0)
#define SetTestStateBit(BIT) (g_cryptoSelfTestState.rng |= BIT)
#define ClearTestStateBit(BIT) (g_cryptoSelfTestState.rng &= ~BIT)
#define IsEntropyBad() IsTestStateSet(ENTROPY)
#define SetEntropyBad() SetTestStateBit(ENTROPY)
#define ClearEntropyBad() ClearTestStateBit(ENTROPY)
#define IsDrbgTested() IsTestStateSet(TESTED)
#define SetDrbgTested() SetTestStateBit(TESTED)
#define ClearDrbgTested() ClearTestStateBit(TESTED)

typedef struct
{
    UINT64 reseedCounter;
    UINT32 magic;
    DRBG_SEED seed; // contains the key and IV for the counter mode DRBG
    UINT32 lastValue[4]; // used when the TPM does continuous self-test
} DRBG_STATE, *pDRBG_STATE;

#define DRBG_MAGIC ((UINT32) 0x47425244) // "DRBG" backwards so that it displays

typedef struct
{
    UINT64 counter;
    UINT32 magic;
    UINT32 limit;
    TPM2B *seed;
    const TPM2B *label;
    TPM2B *context;
    TPM_ALG_ID hash;
    TPM_ALG_ID kdf;
    UINT16 digestSize;
    TPM2B_DIGEST residual;
} KDF_STATE, *pKDR_STATE;

#define KDF_MAGIC ((UINT32) 0x4048444a) // "KDF " backwards so that it displays

Make sure that any other structures added to this union start with a 64-bit counter and a 32-bit magic number

typedef union
{
    DRBG_STATE drbg;
    KDF_STATE kdf;
} 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.

extern RAND_STATE *s_random;

When instrumenting RSA key sieve
#if RSA_INSTRUMENT
#define PRIME_INDEX(x)   ((x) == 512 ? 0 : (x) == 1024 ? 1 : 2)
#define INSTRUMENT_SET(a, b) ((a) = (b))
#define INSTRUMENT_ADD(a, b) (a) = (a) + (b)
#define INSTRUMENT_INC(a)    (a) = (a) + 1
extern UINT32  PrimeIndex;
extern UINT32  failedAtIteration[10];
extern UINT32  PrimeCounts[3];
extern UINT32  MillerRabinTrials[3];
extern UINT32  totalFieldsSieved[3];
extern UINT32  bitsInFieldAfterSieve[3];
extern UINT32  emptyFieldsSieved[3];
extern UINT32  noPrimeFields[3];
extern UINT32  primesChecked[3];
extern UINT16  lastSievePrime;
#else
#define INSTRUMENT_SET(a, b)
#define INSTRUMENT_ADD(a, b)
#define INSTRUMENT_INC(a)
#endif
#endif // _CRYPT_RAND_H
10.1.5 CryptRsa.h

This file contains the RSA-related structures and defines.

```c
#ifndef _CRYPT_RSA_H
#define _CRYPT_RSA_H

These values are used in the bigNum representation of various RSA values.

BN_TYPE(rsa, MAX_RSA_KEY_BITS);
#define BN_RSA(name) BN_VAR(name, MAX_RSA_KEY_BITS)
#define BN_RSA_INITIALIZED(name, initializer)
BN_INITIALIZED(name, MAX_RSA_KEY_BITS, initializer)
#define BN_PRIME(name) BN_VAR(name, (MAX_RSA_KEY_BITS / 2))
#define BN_PRIME_INITIALIZED(name, initializer)
BN_INITIALIZED(name, MAX_RSA_KEY_BITS / 2, initializer)
#if !CRT_FORMAT_RSA
#error This verson only works with CRT formatted data
#else
typedef struct privateExponent
{
    bigNum P;
    bigNum Q;
    bigNum dP;
    bigNum dQ;
    bigNum qInv;
    bn_prime_t entries[5];
} privateExponent;
#define NEW_PRIVATE_EXPONENT(X)
privateExponent _##X;
privateExponent *X = RsaInitializeExponent(&(_##X))
#endif // _CRYPT_RSA_H
```

DRAFT
10.1.6 CryptTest.h

This file contains constant definitions used for self-test.

```c
#ifndef _CRYPT_TEST_H
#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.

#define ALGORITHM_VECTOR_BYTES ((ALG_LAST_VALUE + 8) / 8)
typedef BYTE ALGORITHM_VECTOR[ALGORITHM_VECTOR_BYTES];
#ifdef TEST_SELF_TEST
LIB_EXPORT extern ALGORITHM_VECTOR LibToTest;
#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.

typedef struct {
UINT32 rng;
UINT32 hash;
UINT32 sym;
#if ALG_RSA
UINT32 rsa;
#endif
#if ALG_ECC
UINT32 ecc;
#endif
} CRYPTO_SELF_TEST_STATE;
#endif // _CRYPT_TEST_H
```
10.1.7 HashTestData.h

Hash Test Vectors

```c
1 #if ALG_SHA256 == YES
2 TPM2B_TYPE(HASH_TEST_DATA, 256); // Twice the largest block size
3 TPM2B_HASH_TEST_DATA    c_hashTestData = {{256, { // Twice the largest digest size
4 0xa0,0x9e,0xb5,0x9a,0xd2,0xa4,0x21,0x40,0x1a,0xd0,0x81,0x47,0x39,0x63,0xf9,0x50,
5 0xdc,0x59,0xf7,0x11,0x40,0x13,0x99,0x92,0xc0,0x72,0xa4,0xf0,0xe2,0x33,0xe4,0x63,
6 0x9b,0xb6,0x76,0xc3,0xe1,0x6f,0x13,0x6e,0xc0,0x99,0xf1,0x61,0x73,0x05,0xd8,0x58,0x7f,0x60,0x61,0x84,0x36,
7 0xcf,0xdb,0x66,0x70,0x05,0xe3,0x54,0x12,0x25,0xf4,0xe0,0x1b,0x23,0x35,0xe3,0x70,
8 0x7d,0x19,0x5f,0x00,0xe4,0xf1,0x61,0x73,0x05,0xd8,0x58,0x7f,0x60,0x61,0x84,0x36,
9 0xe8,0x2b,0xe6,0xa9,0x90,0xe0,0xf0,0x9a,0xe6,0xe3,0x26,0x73,0xd4,0x17,0x5b,0x33,
10 0x41,0x44,0x9d,0x90,0x9e,0x6b,0x7d,0x48,0x99,0x25,0x93,0x29,0x14,0x2b,0xce,
11 0x93,0x8d,0x8c,0xaf,0x31,0x0e,0x9c,0x57,0xd8,0x5b,0x57,0x20,0x1b,0xf9,0x2d,0xa5
12 })};
13 #endif
```

```c
18 // Twice the largest block size
19 TPM2B_TYPE(HASH_TEST_DATA, 256); // Twice the largest digest size
20 TPM2B_HASH_TEST_DATA c_hashTestData = {{256, { // Twice the largest digest size
21 0x88,0x3a,0xc3,0xe5,0x5f,0x66,0x9d,0x18,0x80,0xc9,0x7a,0xc9,0xa4,0x08,0x90,0x98,
22 0x0f,0x3a,0x53,0x92,0x4c,0x67,0x4e,0xb7,0x37,0xec,0x67,0xa7,0x66,0xbe,0x10,0xca,
23 0x11,0x5b,0x4a,0xb0,0x45,0xc3,0x32,0x68,0x48,0x69,0xce,0x25,0x1b,0x8c,0xa4,0x44,
24 0x79,0x22,0x83,0xc8,0xfb,0xe2,0x63,0x94,0xa2,0xc3,0x59,0x3e,0xe6,0x64,0xc2c,
25 0x1f,0x8c,0x11,0x93,0x24,0xa3,0x17,0xc5,0x2f,0x37,0xcf,0x95,0x97,0xe8,0x63,0x39,
26 0x68,0x5d,0xca,0xba,0x18,0x37,0x69,0xe5,0x4f,0x19,0xfd,0x8a,0xc0,0x8d,0x87,0xa3a,
27 0x9c,0x31,0xe2,0x04,0x05,0xef,0xb5,0x02,0xe0,0x1e,0x92,0x4b,0x7b,0x73,0x2c,0x8c,
28 0xeb,0x23,0x13,0x81,0x34,0xb9,0xb5,0xc1,0x17,0x37,0x39,0xf8,0x3e,0xe4,0x4c,0x66,
29 0xa8,0x81,0x52,0x2f,0xef,0xc9,0x9c,0x69,0x89,0xb6,0x85,0xc9,0x30,0x16,0x02,0xcca,
30 0xe3,0x61,0xd4,0x0f,0xed,0x34,0xb1,0xca,0xc1,0x1b,0xd1,0xfa,0xc1,0x2a,0xe0,0xdf,
31 0x52,0xf2,0x0b,0x4b,0x9f,0xe0,0x45,0x54,0xb9,0x17,0xb6,0xaf,0xd6,0xd5,0xca,0x90,
32 0x29,0x57,0x7b,0x70,0x50,0x94,0xc5,0xe6,0xf6,0xe4,0x21,0xb8,0x6c,0xb8,0xb6,0xc6b,
33 0xb9,0x64,0xd4,0xd3,0xfb,0x68,0xdb,0xb6,0xe8,0xac,0xd8,0xda,0xb5,0x6d,0xcod,0x93e,
34 0x28,0xa4,0xe2,0x5c,0x44,0xef,0xf0,0xe1,0x6f,0x38,0x1a,0x3c,0xe6,0xef,0x2a,0x9d,
35 0xb9,0x8a,0x05,0xa2,0x95,0xec,0x5f,0xdb,0xb0,0x25,0x67,0x9c,0x86,0xe7a,0xe8,0xea,
36 0x51,0xcc,0xc3,0x3d,0xff,0xe6,0xf0,0xed,0xa3,0xae,0xf9,0x5d,0x33,0x70,0xf2,0x11
37 }};```
10.1.8  KdfTestData.h

Hash Test Vectors

```c
#define TEST_KDF_KEY_SIZE   20
TPM2B_TYPE(KDF_TEST_KEY, TEST_KDF_KEY_SIZE);
TPM2B_KDF_TEST_KEY    c_kdfTestKeyIn = {{TEST_KDF_KEY_SIZE, {
  0x27, 0x1F, 0xA0, 0xB0, 0xC5, 0x06, 0x0E, 0xC3, 0xDF, 
  0xA9, 0x28, 0xFF, 0x9B, 0x73, 0x12, 0x3A, 0x12, 0xDA, 0x0C }}};

TPM2B_TYPE(KDF_TEST_LABEL, 17);
TPM2B_KDF_TEST_LABEL    c_kdfTestLabel = {{17, {
  0x4B, 0x44, 0x46, 0x53, 0x45, 0x4C, 0x46, 0x54, 
  0x45, 0x53, 0x54, 0x4C, 0x41, 0x42, 0x45, 0x4C, 0x00 }}};

TPM2B_TYPE(KDF_TEST_CONTEXT, 8);
TPM2B_KDF_TEST_CONTEXT  c_kdfTestContextU = {{8, {
  0xCE, 0x24, 0x39, 0x5D, 0xCA, 0x73, 0x91 }}};

#define KDF_TEST_ALG    TPM_ALG_SHA512
#else
  TPM2B_KDF_TEST_CONTEXT  c_kdfTestContextV = {{8, {
    0xDA, 0x50, 0x31, 0xDD, 0xF1, 0x2E, 0x83 }}};
#endif
```

```c
TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
  0x8b, 0xe2, 0xc1, 0xb8, 0x5b, 0x78, 0x56, 0x9b, 0x9f, 0xa7,
  0x59, 0xf5, 0x85, 0x7c, 0x56, 0xda, 0x84, 0x81, 0x0f, 0xda }}};

#define KDF_TEST_ALG    TPM_ALG_SHA384
#elif
  TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
    0x1d, 0xce, 0x70, 0xc9, 0x11, 0x3e, 0xb2, 0xdb, 0xa4, 0x7b,
    0xda, 0xc0, 0xc7, 0xda, 0x45, 0x0b, 0x93, 0x12 }}};
#endif
```

```c
TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
  0xbb, 0x02, 0x59, 0xe1, 0xc8, 0xba, 0x60, 0x7e, 0x6a, 0x2c,
  0xda, 0x04, 0x6b, 0x9a, 0x90, 0xe2, 0x9a, 0xc4, 0x84, 0xc4 }}};

#define KDF_TEST_ALG    TPM_ALG_SHA256
#elif
  TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
    0xa9, 0x18, 0x4a, 0xa0, 0x74, 0x23, 0xc4, 0x7d,
    0xaee, 0x76, 0x6c, 0x26, 0xa2, 0x37, 0x7d, 0x7c, 0xf8, 0x51 }}};
#endif
```

```c
TPM2B_KDF_TEST_KEY  c_kdfTestKeyOut = {{20, {
  0x55, 0xb5, 0xa7, 0x18, 0x4a, 0xa0, 0x74, 0x23, 0xc4, 0x7d,
  0xa5, 0x26, 0xa2, 0x37, 0x7d, 0x7c, 0xf8, 0x51 }}};
```
Part 4: Supporting Routines

10.1.9

Trusted Platform Module Library

RsaTestData.h

RSA Test Vectors
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61

#define RSA_TEST_KEY_SIZE
256
typedef struct
{
UINT16
size;
BYTE
buffer[RSA_TEST_KEY_SIZE];
} TPM2B_RSA_TEST_KEY;
typedef TPM2B_RSA_TEST_KEY TPM2B_RSA_TEST_VALUE;
typedef struct
{
UINT16
size;
BYTE
buffer[RSA_TEST_KEY_SIZE / 2];
} TPM2B_RSA_TEST_PRIME;
const TPM2B_RSA_TEST_KEY
c_rsaPublicModulus = {256, {
0x91,0x12,0xf5,0x07,0x9d,0x5f,0x6b,0x1c,0x90,0xf6,0xcc,0x87,0xde,0x3a,0x7a,0x15,
0xdc,0x54,0x07,0x6c,0x26,0x8f,0x25,0xef,0x7e,0x66,0xc0,0xe3,0x82,0x12,0x2f,0xab,
0x52,0x82,0x1e,0x85,0xbc,0x53,0xba,0x2b,0x01,0xad,0x01,0xc7,0x8d,0x46,0x4f,0x7d,
0xdd,0x7e,0xdc,0xb0,0xad,0xf6,0x0c,0xa1,0x62,0x92,0x97,0x8a,0x3e,0x6f,0x7e,0x3e,
0xf6,0x9a,0xcc,0xf9,0xa9,0x86,0x77,0xb6,0x85,0x43,0x42,0x04,0x13,0x65,0xe2,0xad,
0x36,0xc9,0xbf,0xc1,0x97,0x84,0x6f,0xee,0x7c,0xda,0x58,0xd2,0xae,0x07,0x00,0xaf,
0xc5,0x5f,0x4d,0x3a,0x98,0xb0,0xed,0x27,0x7c,0xc2,0xce,0x26,0x5d,0x87,0xe1,0xe3,
0xa9,0x69,0x88,0x4f,0x8c,0x08,0x31,0x18,0xae,0x93,0x16,0xe3,0x74,0xde,0xd3,0xf6,
0x16,0xaf,0xa3,0xac,0x37,0x91,0x8d,0x10,0xc6,0x6b,0x64,0x14,0x3a,0xd9,0xfc,0xe4,
0xa0,0xf2,0xd1,0x01,0x37,0x4f,0x4a,0xeb,0xe5,0xec,0x98,0xc5,0xd9,0x4b,0x30,0xd2,
0x80,0x2a,0x5a,0x18,0x5a,0x7d,0xd4,0x3d,0xb7,0x62,0x98,0xce,0x6d,0xa2,0x02,0x6e,
0x45,0xaa,0x95,0x73,0xe0,0xaa,0x75,0x57,0xb1,0x3d,0x1b,0x05,0x75,0x23,0x6b,0x20,
0x69,0x9e,0x14,0xb0,0x7f,0xac,0xae,0xd2,0xc7,0x48,0x3b,0xe4,0x56,0x11,0x34,0x1e,
0x05,0x1a,0x30,0x20,0xef,0x68,0x93,0x6b,0x9d,0x7e,0xdd,0xba,0x96,0x50,0xcc,0x1c,
0x81,0xb4,0x59,0xb9,0x74,0x36,0xd9,0x97,0xdc,0x8f,0x17,0x82,0x72,0xb3,0x59,0xf6,
0x23,0xfa,0x84,0xf7,0x6d,0xf2,0x05,0xff,0xf1,0xb9,0xcc,0xe9,0xa2,0x82,0x01,0xfb}};
const TPM2B_RSA_TEST_PRIME
c_rsaPrivatePrime = {RSA_TEST_KEY_SIZE / 2, {
0xb7,0xa0,0x90,0xc7,0x92,0x09,0xde,0x71,0x03,0x37,0x4a,0xb5,0x2f,0xda,0x61,0xb8,
0x09,0x1b,0xba,0x99,0x70,0x45,0xc1,0x0b,0x15,0x12,0x71,0x8a,0xb3,0x2a,0x4d,0x5a,
0x41,0x9b,0x73,0x89,0x80,0x0a,0x8f,0x18,0x4c,0x8b,0xa2,0x5b,0xda,0xbd,0x43,0xbe,
0xdc,0x76,0x4d,0x71,0x0f,0xb9,0xfc,0x7a,0x09,0xfe,0x4f,0xac,0x63,0xd9,0x2e,0x50,
0x3a,0xa1,0x37,0xc6,0xf2,0xa1,0x89,0x12,0xe7,0x72,0x64,0x2b,0xba,0xc1,0x1f,0xca,
0x9d,0xb7,0xaa,0x3a,0xa9,0xd3,0xa6,0x6f,0x73,0x02,0xbb,0x85,0x5d,0x9a,0xb9,0x5c,
0x08,0x83,0x22,0x20,0x49,0x91,0x5f,0x4b,0x86,0xbc,0x3f,0x76,0x43,0x08,0x97,0xbf,
0x82,0x55,0x36,0x2d,0x8b,0x6e,0x9e,0xfb,0xc1,0x67,0x6a,0x43,0xa2,0x46,0x81,0x71}};
const BYTE
c_RsaTestValue[RSA_TEST_KEY_SIZE] = {
0x2a,0x24,0x3a,0xbb,0x50,0x1d,0xd4,0x2a,0xf9,0x18,0x32,0x34,0xa2,0x0f,0xea,0x5c,
0x91,0x77,0xe9,0xe1,0x09,0x83,0xdc,0x5f,0x71,0x64,0x5b,0xeb,0x57,0x79,0xa0,0x41,
0xc9,0xe4,0x5a,0x0b,0xf4,0x9f,0xdb,0x84,0x04,0xa6,0x48,0x24,0xf6,0x3f,0x66,0x1f,
0xa8,0x04,0x5c,0xf0,0x7a,0x6b,0x4a,0x9c,0x7e,0x21,0xb6,0xda,0x6b,0x65,0x9c,0x3a,
0x68,0x50,0x13,0x1e,0xa4,0xb7,0xca,0xec,0xd3,0xcc,0xb2,0x9b,0x8c,0x87,0xa4,0x6a,
0xba,0xc2,0x06,0x3f,0x40,0x48,0x7b,0xa8,0xb8,0x2c,0x03,0x14,0x33,0xf3,0x1d,0xe9,
0xbd,0x6f,0x54,0x66,0xb4,0x69,0x5e,0xbc,0x80,0x7c,0xe9,0x6a,0x43,0x7f,0xb8,0x6a,
0xa0,0x5f,0x5d,0x7a,0x20,0xfd,0x7a,0x39,0xe1,0xea,0x0e,0x94,0x91,0x28,0x63,0x7a,
0xac,0xc9,0xa5,0x3a,0x6d,0x31,0x7b,0x7c,0x54,0x56,0x99,0x56,0xbb,0xb7,0xa1,0x2d,
0xd2,0x5c,0x91,0x5f,0x1c,0xd3,0x06,0x7f,0x34,0x53,0x2f,0x4c,0xd1,0x8b,0xd2,0x9e,
0xdc,0xc3,0x94,0x0a,0xe1,0x0f,0xa5,0x15,0x46,0x2a,0x8e,0x10,0xc2,0xfe,0xb7,0x5e,
0x2d,0x0d,0xd1,0x25,0xfc,0xe4,0xf7,0x02,0x19,0xfe,0xb6,0xe4,0x95,0x9c,0x17,0x4a,
0x9b,0xdb,0xab,0xc7,0x79,0xe3,0x5e,0x40,0xd0,0x56,0x6d,0x25,0x0a,0x72,0x65,0x80,
0x92,0x9a,0xa8,0x07,0x70,0x32,0x14,0xfb,0xfe,0x08,0xeb,0x13,0xb4,0x07,0x68,0xb4,
0x58,0x39,0xbe,0x8e,0x78,0x3a,0x59,0x3f,0x9c,0x4c,0xe9,0xa8,0x64,0x68,0xf7,0xb9,
0x6e,0x20,0xf5,0xcb,0xca,0x47,0xf2,0x17,0xaa,0x8b,0xbc,0x13,0x14,0x84,0xf6,0xab};
const TPM2B_RSA_TEST_VALUE
c_RsaepKvt = {RSA_TEST_KEY_SIZE, {
0x73,0xbd,0x65,0x49,0xda,0x7b,0xb8,0x50,0x9e,0x87,0xf0,0x0a,0x8a,0x9a,0x07,0xb6,
0x00,0x82,0x10,0x14,0x60,0xd8,0x01,0xfc,0xc5,0x18,0xea,0x49,0x5f,0x13,0xcf,0x65,
0x66,0x30,0x6c,0x60,0x3f,0x24,0x3c,0xfb,0xe2,0x31,0x16,0x99,0x7e,0x31,0x98,0xab,
0x93,0xb8,0x07,0x53,0xcc,0xdb,0x7f,0x44,0xd9,0xee,0x5d,0xe8,0x5f,0x97,0x5f,0xe8,
0x1f,0x88,0x52,0x24,0x7b,0xac,0x62,0x95,0xb7,0x7d,0xf5,0xf8,0x9f,0x5a,0xa8,0x24,

Family “2.0”
Level 00 Revision 01.55

TCG PUBLIC REVIEW
Copyright © TCG 2006-2019

Page 483
April 15, 2019


Part 4: Supporting Routines

```c
#define OAP_TEST_LABEL "OAP Test Value"

#if ALG_SHA1_VALUE == DEFAULT_TEST_HASH
const TPM2B_RSA_TEST_VALUE c_RsassaKvt = {RSA_TEST_KEY_SIZE, {
    0x1e, 0xad, 0xbc, 0xbf, 0x33, 0xca, 0xe1, 0x92, 0x8f, 0x2a, 0x89, 0x6c, 0x45, 0x24, 0xd1,
    0x5f, 0x9a, 0x97, 0xcf, 0xb1, 0x97, 0x9a, 0x85,
    0x6b, 0x9b, 0xf0, 0x7a, 0xa4, 0x69, 0x52,
    0x6c, 0x45, 0x24, 0xd1, 0x5f, 0x9a, 0x97, 0xcf, 0xb1, 0x97, 0x9a, 0x85
}};
#endif
```
const TPM2B_RSA_TEST_VALUE c_RsapssKvt = {RSA_TEST_KEY_SIZE, {
    0xd1, 0x91, 0x3c, 0x20, 0x0e, 0x58, 0x2b, 0x31, 0xf8, 0x8b, 0xee, 0xbc, 0x1f, 0x95, 0x35, 0x58, 0xf0, 0x2e, 0xad, 0xd2, 0xda, 0xaf, 0xff, 0xb6, 0x92, 0x83, 0x5b, 0x8a, 0x06, 0x2d, 0x0c, 0x32, 0x6c, 0x56, 0x38, 0x24, 0x55, 0x92, 0x84, 0x9b, 0x1b, 0x8b, 0x04, 0xcf, 0x24, 0x14, 0x24, 0x13, 0x2f, 0xbd, 0x4d, 0x69, 0x39, 0x5e, 0x85, 0xe2, 0xef, 0x86, 0xa, 0xc6, 0x6b, 0x7e, 0x46, 0x69, 0x67, 0x46, 0xc6, 0x37, 0xdb, 0xea, 0xfb, 0xe, 0xd4, 0x2e, 0xa9, 0x86, 0xf, 0x00, 0x2b, 0x7b, 0x0c, 0x09, 0x64, 0x6a, 0x8f, 0xc0, 0xe0, 0x09, 0x14, 0x36, 0xda, 0x74, 0x31, 0x18, 0x5b, 0x18, 0xe, 0x83, 0xe0a, 0x7d, 0x66, 0x86, 0x7d, 0x43, 0x06, 0x0f, 0x99, 0x60, 0xc6, 0x65, 0x08, 0xf6));
};
#endif // SHA1

family "2.0"

tcg public revision 0.55

copyright © tcg 2006-2019

april 15, 2019
const TPM2B_RSA_TEST_VALUE c_RsaKvt = {RSA_TEST_KEY_SIZE, {
    0x8a, 0xb1, 0xa5, 0xe4, 0x02, 0x7f, 0xd2, 0x45, 0x2a, 0xc0, 0x2b, 0x6b, 0x8c, 0xe0, 0x9a,
    0x92, 0x4f, 0x9b, 0xc5, 0xe4, 0x8b, 0x82, 0xb9, 0xb0, 0xd9, 0x87, 0x8c, 0xc0, 0xf0, 0xb0, 0x59,
    0xa5, 0x92, 0x21, 0xa0, 0x16, 0x51, 0xc5, 0xed, 0xa8, 0xe6, 0xe2, 0x22, 0x29, 0x46, 0xc7, 0x86, 0x37,
    0x4b, 0x1b, 0x1e, 0x94, 0x93, 0xc8, 0x4c, 0x17, 0x7a, 0xae, 0x59, 0x91, 0xf8, 0x83, 0xc4, 0xc2,
    0x83, 0xc8, 0x3c, 0x22, 0xe5, 0x0e, 0x7e, 0x50, 0x67, 0x7e, 0x76, 0x7e, 0x33, 0x03, 0xef, 0xf6, 0x0d, 0x9c,
    0x2f, 0x0a, 0x80, 0x28, 0xd3, 0xc5, 0x7d, 0x2d, 0xa1, 0x0b, 0x96, 0xd6, 0xe5, 0x98, 0x05, 0x8c, 0xa3,
    0x4d, 0xa0, 0x1f, 0x8c, 0xb6, 0xfb, 0xb1, 0xc0, 0xe9, 0xcb, 0x38, 0x27, 0x60, 0x64, 0x17, 0xc0a,
    0xf4, 0x8b, 0x61, 0x67, 0x20, 0xb1, 0x7d, 0x90, 0x40, 0x2a, 0x1c, 0xf0, 0x55, 0x40, 0x4b, 0x95,
    0x39, 0x52, 0x18, 0x3b, 0xe4, 0xe8, 0x83, 0x4b, 0x7c, 0x47, 0xfb, 0xed, 0x06, 0xc9, 0xc6d,
    0x4f, 0xba, 0x81, 0xd6, 0xb7, 0x31, 0xc0f, 0x5c, 0x23, 0xf8, 0x25, 0xa5, 0x9b, 0x77, 0x0a, 0xf8,
    0x46, 0xf0, 0xfb, 0x59, 0xb0, 0x04, 0xd7, 0x1e, 0xfb, 0x15, 0xa1, 0x6a, 0x26, 0x9b, 0xae, 0xf4,
    0x0f, 0x5f, 0x84, 0x6f, 0x3c, 0xed, 0xfb, 0x24, 0x0b, 0x43, 0x1d, 0xba, 0x74, 0x69, 0xe4, 0x39,
    0x0e, 0xab, 0xa5, 0x16, 0xa5, 0x28, 0xee, 0x96, 0x84, 0x43e, 0x1e, 0x6d, 0xf5, 0x4a, 0xe0, 0x7d,
    0x94, 0x9e, 0x1b, 0x8e, 0x9a, 0x00, 0xc9, 0x49, 0x02, 0x04, 0xc3, 0xe6, 0x62, 0xff, 0xe0, 0xa2,
    0x20, 0x33, 0x3e, 0x86, 0xdd, 0xda, 0xe3, 0x15, 0x2d, 0xb7, 0xccd, 0xda, 0xff, 0xb1, 0xb0, 0x45, 0x7b,
    0xdf, 0xa0, 0x42, 0x29, 0xba, 0x97, 0x33, 0x9e, 0x9a, 0x4d, 0x89, 0x83, 0xac, 0xa1, 0x98, 0x2c, 0x2dj}
};
#endif // SHA256

#if ALG_SHA384_VALUE == DEFAULT_NULL_HASH
const TPM2B_RSA_TEST_VALUE c_RsaKvt = {RSA_TEST_KEY_SIZE, {
    0x0f, 0x3c, 0x42, 0x0a, 0x81, 0x09, 0x05, 0x3c, 0xfd, 0x59, 0x3b, 0x7f, 0x29, 0x0bc, 0x03,
    0x67, 0x0c, 0xff, 0x74, 0xe0, 0x79, 0x05, 0x13, 0x4b, 0x0e, 0x13, 0x0d, 0xc9, 0x86, 0x94, 0xfe,
    0xed, 0xa6, 0xe8, 0x3a, 0xcb, 0x89, 0xed, 0x86, 0x63, 0x4c, 0xdb, 0xf1, 0x95, 0xee, 0xc1l,
    0x46, 0xc5, 0x3b, 0xd8, 0xfb, 0x82, 0x41, 0x6a, 0x60, 0x8b, 0x9e, 0x5f, 0x7f, 0x20, 0x16, 0xe3,
    0x69, 0xb6, 0x2d, 0x92, 0xfc, 0x60, 0xa2, 0x74, 0x88, 0xd5, 0xc7, 0x7a, 0x66, 0x1d, 0xf3, 0xe3, 0x45,
    0x02, 0x51, 0x39, 0xd9, 0xf3, 0x56, 0xb0, 0x91, 0x80, 0xe0, 0x6c, 0xa8, 0xc3, 0x78, 0xef, 0x34,
    0x2c, 0x88, 0x85, 0xfb, 0x47, 0x98, 0x5d, 0x57, 0x8e, 0x3a, 0xb9, 0xff, 0x92, 0x04, 0xc7, 0xc2,
    0x6e, 0xfa, 0x14, 0xc1, 0xb9, 0x68, 0x15, 0xc5, 0x12, 0xe8, 0xa8, 0xbe, 0xea, 0xe8, 0x8d, 0x9b,
    0x48, 0x28, 0x35, 0xdb, 0x4b, 0x52, 0xc1, 0x2d, 0x85, 0x47, 0x83, 0xd0, 0xe9, 0xa0, 0x9e, 0x6e,
    0x65, 0xd4, 0x34, 0x7f, 0x81, 0xe6, 0xc9, 0xf0, 0x96, 0x62, 0xf7, 0x0ec, 0x41, 0xd5, 0xc2, 0xe3,
    0x4b, 0xba, 0x9c, 0x8a, 0x02, 0xe0, 0xcf, 0xd0, 0x5d, 0x14, 0xf7, 0x09, 0x42, 0xe8, 0xa4, 0x27, 0xfe,
    0x3e, 0x66, 0x42, 0x99, 0x03, 0xe1, 0x69, 0xcdb, 0xb7, 0x90, 0x70, 0x0ebe, 0xe4, 0xc9, 0xac,
    0x45, 0x67, 0x91, 0x9f, 0x75, 0x10, 0xc6, 0xc0, 0x14, 0xe1, 0x28, 0xc1, 0xe0, 0xe0, 0xe7, 0xc0,
    0x5c, 0x1d, 0xe8, 0xe8, 0xff, 0x45, 0x79, 0x51, 0x86, 0x08, 0xe6, 0x39, 0xc3, 0xb5, 0xfd, 0xdb,
    0xf1, 0xdd, 0xe2, 0xf4, 0xb2, 0x1a, 0x69, 0xd8, 0xb3, 0x91, 0x4d, 0x3d, 0x83, 0x84, 0xad, 0xb7y};
#endif // SHA256

/*
Copyright © TCG 2006
TCG PUBLIC REVIEW
Part 4: Supporting Routines
*/

Routines
/*
  * Part 4: Supporting Routines
  * Trusted Platform Module Library
  */

const TPM2B_RSA_TEST_VALUE c_ReassaKvt = { TPM2B_RSA_TEST_VALUE }

const TPM2B_RSA_TEST_VALUE c_ReassaKvt = { TPM2B_RSA_TEST_VALUE }
const TPM2B_rsa_test_value c_RsassaKvt = {
    0x05, 0x55, 0x00, 0x62, 0x01, 0xc6, 0x04, 0x31, 0x55, 0x73, 0x3f, 0x2a, 0xf9, 0xd4, 0x0f, 0xc1,
    0x2b, 0xeb, 0xd8, 0xc8, 0xdb, 0xb2, 0xab, 0x6c, 0x26, 0xde, 0x2d, 0x89, 0xc2, 0x2d, 0x36, 0x62,
    0xc8, 0x22, 0x5d, 0x58, 0x03, 0xb1, 0x46, 0x14, 0xa5, 0xd4, 0x2b, 0x25, 0x6b, 0x7f, 0x8f, 0x14,
    0x7e, 0x03, 0x2f, 0x3d, 0xb8, 0x39, 0xa5, 0x79, 0x13, 0x7e, 0x22, 0x2a, 0xb9, 0x3e, 0x8f, 0x0a,
    0x01, 0x7c, 0x03, 0x12, 0x21, 0x6c, 0x2a, 0xb4, 0x39, 0x98, 0x6d, 0xff, 0x08, 0x6c, 0x59, 0x2d,
    0xdc, 0xc6, 0xf1, 0x77, 0x62, 0x10, 0xa6, 0xcc, 0xe2, 0x71, 0x8e, 0x97, 0x00, 0x87, 0x5b, 0x0e,
    0x20, 0x00, 0x3f, 0x18, 0x63, 0x83, 0xf0, 0xe4, 0x0a, 0xd4, 0x8c, 0xe9, 0x8c, 0x91, 0x7e, 0x89,
    0x04, 0x64, 0x0c, 0xb2, 0x41, 0xc8, 0xac, 0xf6, 0x5a, 0x75, 0xe6, 0xa5, 0x76, 0x43, 0xcb, 0xa5,
    0x33, 0x88, 0x07, 0xc9, 0x73, 0x0f, 0x45, 0xa4, 0xc3, 0xac, 0xc1, 0xc3, 0xe6, 0x7e, 0x21, 0x66,
    0x1c, 0xb3, 0x0b, 0x0b, 0xe2, 0x3e, 0x39, 0xf9, 0xb2, 0xe2, 0xff, 0xe0, 0xb4, 0x85, 0x89, 0x33,
    0x2a, 0xc0, 0xc8, 0x5d, 0x58, 0xe1, 0x89, 0x12, 0xe9, 0x12, 0x4d, 0xb3, 0x1f, 0x59, 0x0c, 0x3e,
    0xd8, 0xb2, 0xeb, 0xf5, 0x88, 0xfb, 0xe1, 0x4b, 0x8e, 0xdc, 0x33, 0xa8, 0xda, 0xbe, 0x04, 0x45,
    0xbf, 0xc6, 0x54, 0x70, 0x00, 0xb8, 0x66, 0x4d, 0x3a, 0xe1, 0xb6, 0xe4, 0x1a, 0xa0, 0x0a,
    0x0b, 0x33, 0x9a, 0x9a, 0x52, 0xda, 0x60, 0x69, 0xb7, 0xef, 0x93, 0x47, 0x38, 0xab, 0x1a, 0x0a,
    0x22, 0x6e, 0x76, 0x06, 0xb6, 0x74, 0xaf, 0x74, 0x8f, 0x51, 0xc0, 0x89, 0x5a, 0x4b, 0xb6, 0x6a,
    0x91, 0x18, 0x25, 0x7d, 0xa6, 0x77, 0xe6, 0xfd, 0xc2, 0x62, 0x36, 0x07, 0xc6, 0xef, 0x79, 0xc9
};
#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.

```c
#if AES_128
  const BYTE key_AES128 [] = {
    0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
    0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0xf4, 0x3c;
  const BYTE dataIn_AES128 [] = {
    0x5b, 0xc1, 0xbe, 0xe2, 0xe4, 0x40, 0x9f, 0x96,
    0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
    0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
    0xe9, 0xb7, 0x6f, 0xac, 0x45, 0xf4, 0x8e, 0x51};
  const BYTE dataOut_AES128_ECB [] = {
    0x3a, 0xd7, 0x7b, 0xb4, 0x0d, 0x7a, 0x36, 0x60,
    0xa8, 0x9e, 0xca, 0xf3, 0x24, 0x66, 0xf4, 0x97,
    0xf5, 0xd3, 0xd5, 0x85, 0x03, 0xb9, 0x69, 0x9d,
    0xe7, 0x85, 0x89, 0x5a, 0x96, 0xfd, 0xba, 0xaf};
  const BYTE dataOut_AES128_CBC [] = {
    0x76, 0x49, 0xab, 0xac, 0x81, 0x19, 0xb2, 0x46,
    0xce, 0xe9, 0x8e, 0x9b, 0x12, 0xe9, 0x19, 0x7d,
    0x50, 0x86, 0xcb, 0x9b, 0x50, 0x72, 0x19, 0xe6,
    0x95, 0xdb, 0x11, 0x3a, 0x91, 0x76, 0x78, 0xb2};
  const BYTE dataOut_AES128_OFB [] = {
    0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
    0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
    0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
    0x62, 0xf8, 0xe6, 0x9f, 0x1c, 0xe5, 0x8b};
#endif
#endif
#if AES_192
  const BYTE key_AES192 [] = {
    0x8e, 0x73, 0xb0, 0xf7, 0xda, 0xe2, 0x40, 0x9f,
    0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
    0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
    0xe9, 0xb7, 0x6f, 0xac, 0x45, 0xf4, 0x8e, 0x51};
  const BYTE dataOut_AES192_ECB [] = {
    0x3b, 0x3f, 0xd9, 0x2e, 0xb7, 0x2d, 0xad, 0x20,
    0x33, 0x34, 0x49, 0xf8, 0xe8, 0x3c, 0xfb, 0x4a,
    0xc8, 0x10, 0xf3, 0x2b, 0x80, 0x90, 0x79, 0xe5,
    0x62, 0xf8, 0xe6, 0x9f, 0x1c, 0xe5, 0x8b};
#endif
#endif
```

This code provides a set of vectors for testing encryption and decryption using the AES-128 and AES-192 algorithms. The vectors include the key, input, and output data for each scheme (ECB, CBC, CFB, OFB, CTR). Each scheme is tested with at least two blocks to ensure proper chaining value handling.
const BYTE dataOut_AES192_OFB[] = {
    0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
    0x67, 0xce, 0x7f, 0x7f, 0x81, 0x17, 0x36, 0x21,
    0x96, 0x1a, 0x2b, 0x70, 0x17, 0x1d, 0x3d, 0x7a};

const BYTE dataOut_AES192_CTR[] = {
    0xcd, 0xc8, 0x0d, 0x6f, 0xdd, 0xf1, 0x8c, 0xab,
    0x34, 0xc2, 0x59, 0x09, 0xc9, 0x9a, 0x41, 0x74,
    0xfc, 0xc2, 0x8b, 0x8d, 0x4c, 0x63, 0x83, 0x7c,
    0x69, 0xe8, 0x17, 0x00, 0xc1, 0x10, 0x04, 0x01};

#endif

#if AES_256
const BYTE key_AES256[] = {
    0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0xbe,
    0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81,
    0x1f, 0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7,
    0x2d, 0x98, 0x10, 0xa3, 0x09, 0x14, 0xdf, 0xf4};

const BYTE dataIn_AES256[] = {
    0x6b, 0xc1, 0xbe, 0xe2, 0x2e, 0x40, 0x9f, 0x96,
    0xe9, 0x3d, 0x7e, 0x11, 0x73, 0x93, 0x17, 0x2a,
    0xae, 0x2d, 0x8a, 0x57, 0x1e, 0x03, 0xac, 0x9c,
    0x9e, 0x03, 0x39, 0xec, 0x0a, 0xa6, 0xfa, 0xef,
    0xd5, 0xcc, 0xc2, 0xc6, 0xf4, 0xce, 0x8e, 0x94};

const BYTE dataOut_AES256_ECB[] = {
    0xf3, 0xee, 0xd1, 0xbd, 0xb5, 0xd2, 0xa0, 0x3c,
    0x06, 0x4b, 0x5a, 0x7e, 0x3d, 0xb1, 0x81, 0xf8,
    0x59, 0x1c, 0xcb, 0xe3, 0xa0, 0x10, 0xda, 0x7d,
    0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x16, 0x4b};

const BYTE dataOut_AES256_CBC[] = {
    0xf5, 0x8c, 0x4c, 0x04, 0xd6, 0xe5, 0xf1, 0x9a,
    0x77, 0x9e, 0xab, 0xb7, 0x0d, 0x8b, 0x6d, 0x86,
    0x9c, 0xf0, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d,
    0x67, 0x9f, 0x77, 0x7b, 0xc6, 0x70, 0x2c, 0x7d};

const BYTE dataOut_AES256_CFB[] = {
    0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
    0x7e, 0xc0, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
    0x3f, 0xf0, 0x4e, 0x96, 0x7e, 0xdb, 0x80, 0x8d,
    0x32, 0x11, 0x3c, 0x63, 0x31, 0xe5, 0x40, 0x7b};

const BYTE dataOut_AES256_OFB[] = {
    0xdc, 0x7e, 0x84, 0xbf, 0xda, 0x79, 0x16, 0x4b,
    0x7e, 0xc0, 0x84, 0x86, 0x98, 0x5d, 0x38, 0x60,
    0x4f, 0xe0, 0xdc, 0x67, 0x40, 0xd2, 0x0b, 0x3a,
    0xc8, 0x8f, 0x6a, 0xd8, 0x2a, 0x4f, 0xb0, 0x8d};

const BYTE dataOut_AES256_CTR[] = {
    0x60, 0x1e, 0xc3, 0x13, 0x77, 0x57, 0x89, 0x5a,
    0x7b, 0x43, 0xe0, 0x3a, 0x8c, 0x4d, 0x62, 0xb5,
    0x9a, 0x0c, 0xe9, 0x90, 0xca, 0xca, 0xf5, 0xc5};
#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.

```c
#ifndef SELF_TEST_DATA
#error "This file may only be included in AlgorithmTests.c"
#endif

#ifndef _SYMMETRIC_TEST_H
#define _SYMMETRIC_TEST_H

#include "SymmetricTestData.h"
#endif // _SYMMETRIC_TEST_H

10.1.11.2 Symmetric Test Structures

```c
const SYMMETRIC_TEST_VECTOR c_symTestValues[NUM_SYMS + 1] = {
    #if ALG_AES && AES_128
    {ALG_AES_VALUE, 128, key_AES128, 16, sizeof(dataIn_AES128), dataIn_AES128, 
     {dataOut_AES128_CTR, dataOut_AES128_OFB, dataOut_AES128_CBC, 
     dataOut_AES128_CFB, dataOut_AES128_ECB}},
    #endif

    #if ALG_AES && AES_192
    {ALG_AES_VALUE, 192, key_AES192, 16, sizeof(dataIn_AES192), dataIn_AES192, 
     {dataOut_AES192_CTR, dataOut_AES192_OFB, dataOut_AES192_CBC, 
     dataOut_AES192_CFB, dataOut_AES192_ECB}},
    #endif

    #if ALG_AES && AES_256
    {ALG_AES_VALUE, 256, key_AES256, 16, sizeof(dataIn_AES256), dataIn_AES256, 
     {dataOut_AES256_CTR, dataOut_AES256_OFB, dataOut_AES256_CBC, 
     dataOut_AES256_CFB, dataOut_AES256_ECB}},
    #endif

    #if ALG_SM4 && SM4_128
    {ALG_SM4_VALUE, 128, key_SM4128, 16, sizeof(dataIn_SM4128), dataIn_SM4128, 
     {dataOut_SM4128_CTR, dataOut_SM4128_OFB, dataOut_SM4128_CBC, 
     dataOut_SM4128_CFB, dataOut_AES128_ECB}},
    #endif

    {0}
};
#endif // _SYMMETRIC_TEST_H
This file contains the parameter data for ECC testing.

```c
#ifdef SELF_TEST_DATA
TPM2B_TYPE(EC_TEST, 32);
const TPM_ECC_CURVE c_testCurve = 00003;

The static key
const TPM2B_EC_TEST c_ecTestKey_ds = {{32, {
0xdf,0x8d,0xa4,0xa3,0x88,0xf6,0x76,0x96,0x89,0xfc,0x2f,0x0d,0xa1,0xb4,0x39,0x7a,
0x78,0xc4,0x7f,0x71,0x8c,0xa6,0x91,0x85,0xc0,0xbf,0xf3,0x54,0x20,0x91,0x2f,0x73}});
const TPM2B_EC_TEST c_ecTestKey_QsX = {{32, {
0x17,0xad,0x2f,0xcb,0x18,0xd4,0xdb,0x3f,0x2c,0x53,0x13,0x82,0x42,0x97,0xff,0x8d,
0x99,0x50,0x16,0x02,0x35,0xa7,0x06,0xae,0x1f,0xda,0xe2,0x9c,0x12,0x77,0xc0,0xf9}});
const TPM2B_EC_TEST c_ecTestKey_QsY = {{32, {
0xa6,0xca,0xf2,0x18,0x45,0x96,0x6e,0x58,0xe6,0x72,0x34,0x12,0x89,0xcd,0xaa,0xad,
0xcb,0x68,0xb2,0x51,0xdc,0x5e,0xd1,0x6d,0x38,0x20,0x35,0x57,0xb2,0xfd,0xc7,0x52}});
const TPM2B_EC_TEST c_ecTestKey_de = {{32, {
0xb6,0xb5,0x33,0x5c,0xd1,0xee,0x52,0x07,0x99,0xea,0x2e,0x8f,0x8b,0x19,0x18,0x07,
0xc1,0x8f,0xdf,0xd9,0x8b,0x77,0x00,0xc7,0xd6,0x53,0x21,0xed,0x02,0x53,0xe6,0xac}});
const TPM2B_EC_TEST c_ecTestKey_QeX = {{32, {
0xa5,0x1e,0x80,0xdb,0x92,0x78,0xd8,0x33,0x52,0xed,0x3b,0xfa,0x3b,0x74,0xa3,0x3d,0x2c,
0x2f,0x9c,0x59,0x03,0x07,0xf8,0x22,0x90,0xed,0xe3,0x45,0xf8,0x2a,0x0a,0xd8,0x1d}});
const TPM2B_EC_TEST c_ecTestKey_QeY = {{32, {
0x58,0x94,0x05,0x82,0xbe,0x5f,0x33,0x02,0x25,0x90,0x3a,0x33,0x90,0xe3,0xe5,0x10,0x4a,0xbc,
0x78,0xa5,0xc5,0x07,0x64,0xaf,0x91,0xbc,0xe0,0xef,0x85,0x11,0x40}});
const TPM2B_TYPE(TEST_VALUE, 64);
const TPM2B_TEST_VALUE c_ecTestValue = {{64, {
0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
0xf4,0x45,0xf7,0x13,0x8d,0x13,0x81,0xb0,0xe5,0x76,0xbe,0xaa,0xb6,0xf3,0x8d,0x4d,
0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xc9,0xc0,0xc7,0x11,0x8d,0xc3,0xff,
0x62,0x69,0xa2,0xe6,0x46,0x90,0xe7,0xd1,0x80,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}});
#endif // SHA1

The ephemeral key
const TPM2B_EC_TEST c_ecTestEcdh_X = {{32, {
0x64,0x02,0x68,0x92,0x78,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
0x2f,0x9c,0x59,0x03,0x07,0xf8,0x22,0x90,0xed,0xe3,0x45,0xf8,0x2a,0x0a,0xd8,0x1d}});
const TPM2B_EC_TEST c_ecTestEcdh_Y = {{32, {
0x58,0x94,0x05,0x82,0xbe,0x5f,0x33,0x02,0x25,0x90,0x3a,0x33,0x90,0xe3,0xe5,0x10,0x4a,0xbc,
0x78,0xa5,0xc5,0x07,0x64,0xaf,0x91,0xbc,0xe0,0xef,0x85,0x11,0x40}});
const TPM2B_TYPE(TEST_VALUE, 64);
const TPM2B_TEST_VALUE c_ecTestValue = {{64, {
0x78,0xd5,0xd4,0x56,0x43,0x61,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
0xf4,0x45,0xf7,0x13,0x8d,0x13,0x81,0xb0,0xe5,0x76,0xbe,0xaa,0xb6,0xf3,0x8d,0x4d,
0x23,0x65,0xcc,0xa7,0xc9,0x19,0x10,0xce,0x69,0xc9,0xc0,0xc7,0x11,0x8d,0xc3,0xff,
0x62,0x69,0xa2,0xe6,0x46,0x90,0xe7,0xd1,0x80,0x77,0x94,0x65,0x1c,0x3e,0xc1,0x3e}});
#endif // SHA1

ECDH test results
const TPM2B_EC_TEST c_ecTestEcdh_X = {{32, {
0x64,0x02,0x68,0x92,0x78,0xdb,0x97,0xa4,0x32,0xc4,0x0b,0x06,0xa9,0xa8,0xa0,
0x2f,0x9c,0x59,0x03,0x07,0xf8,0x22,0x90,0xed,0xe3,0x45,0xf8,0x2a,0x0a,0xd8,0x1d}});
const TPM2B_EC_TEST c_ecTestEcdh_Y = {{32, {
0x58,0x94,0x05,0x82,0xbe,0x5f,0x33,0x02,0x25,0x90,0x3a,0x33,0x90,0xe3,0xe5,0x10,0x4a,0xbc,
0x78,0xa5,0xc5,0x07,0x64,0xaf,0x91,0xbc,0xe0,0xef,0x85,0x11,0x40}});
```

10.1.12 EccTestData.h

This file contains the parameter data for ECC testing.
0xf8,0xf2,0xf4,0xa5,0xe0,0x1d,0x3c,0xa2,0x39,0x31,0xe4,0xe7,0x36,0x3b,0xb5,0x5f}\};

const TPM2B_EC_TEST c_TestEcDsa_s = {{32, {
0x8f,0xd0,0x12,0xd9,0x24,0x75,0xf6,0xc4,0x3b,0xb5,0x46,0x75,0x95,0xa5,0x39,0x14,0x49,0x3d,0x2f}};

const TPM2B_EC_TEST c_TestEcSchnorr_r = {{32, {
0xf7,0xb9,0x15,0x4c,0x34,0xf6,0x41,0xa3,0xd2,0xf1,0xbd,0xf4,0x13,0x6a,0x4f,
0x63,0xb8,0x4d,0xb5,0xc8,0xdc,0xe2,0xe0,0x85,0x95,0xa5,0x39,0x14,0x49,0x3d,0x2f}};

const TPM2B_EC_TEST c_TestEcSchnorr_s = {{32, {
0xfe,0xbe,0x17,0xaa,0x31,0x22,0x9f,0xd0,0xd2,0xf5,0x25,0x04,0x92,0xb0,0xaa,0x4e,
0xcc,0x1c,0xb6,0x79,0x6d,0x42,0xb3,0xe3,0x3f,0xbb,0xe0,0x5f,0xd0,0x0d,0x8b,0xc3}};

#endif // SHA256

#if ALG_SHA384_VALUE == DEFAULT_TEST_HASH
const TPM2B_EC_TEST c_TestEcDsa_r = {{32, {
0xf5,0x74,0x6d,0xc6,0x56,0x86,0xbb,0xba,0x1c,0x06,0x75,0x65,0xe6,0x64,0x31,
0x0c,0x04,0x03,0x9f,0x24,0x3f,0xfb,0xe0,0xc5,0xb9,0x0e,0x95,0x0f,0xd5,0x9d,0xfb}};

const TPM2B_EC_TEST c_TestEcDsa_s = {{32, {
0xc2,0x4f,0x32,0xa1,0x06,0x0c,0x08,0x4f,0xc6,0x6d,0x31,0x66,0x91,0x9f,0x79,0x6c,
0x5b,0xe5,0x7b,0x94,0xa1,0x91,0x38,0x4d,0x20,0xa2,0x10,0x0f,0xd5,0x9d,0xfb}};

const TPM2B_EC_TEST c_TestEcSchnorr_r = {{32, {
0x1e,0x1f,0x1f,0x1f,0x36,0x19,0x1c,0x30,0x84,0x85,0x1b,0xe6,0x74,0x35,0x66}};

const TPM2B_EC_TEST c_TestEcSchnorr_s = {{32, {
0xb9,0xe6,0x3e,0xc6,0xb9,0xe0,0xf1,0xc0,0xf4,0x48,0x4a,0xda,0xc8,0x8d,0x77,
0x87,0xb4,0xba,0x40,0xfe,0x5b,0x68,0x11,0x14,0xcf,0xa0,0xe0,0x85,0x46,0x99,0x01}};

#endif // SHA384

#if ALG_SHA512_VALUE == DEFAULT_TEST_HASH
const TPM2B_EC_TEST c_TestEcDsa_r = {{32, {
0xc9,0x71,0xa6,0xb4,0x4a,0x4f,0x26,0x8c,0x27,0x00,0x06,0x3b,0x00,0x0f,0xa3,0x17,
0x72,0x48,0x40,0x49,0x4d,0x51,0xf9,0xa4,0xc6,0x7e,0x86,0xe9,0xe7,0xb4,0x79,0xb2}};

const TPM2B_EC_TEST c_TestEcDsa_s = {{32, {
0x1e,0xb8,0xe1,0xbf,0xa1,0x9e,0x39,0xe1,0xe5,0xa2,0xe6,0x59,0xd0,0x1a,0x6a,0x03,
0x6a,0x1f,0x1c,0x4f,0x36,0x19,0xc1,0xc0,0x84,0x85,0x1b,0xe6,0x74,0x35,0x66}};

const TPM2B_EC_TEST c_TestEcSchnorr_r = {{32, {
0xcc,0x07,0xad,0x65,0x91,0xdd,0xa0,0x10,0x23,0xae,0x53,0xc6,0xdf,0xf1,0x50,0x90,
0x16,0x96,0xf4,0x45,0x09,0x73,0x9c,0x84,0xb5,0x5f,0x0e,0x0f,0xb3,0xe6,0x2c,0xb6,0x05,0xbd,0x14,0x10}};

#endif // SHA512

#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

```c
#ifndef CRYPT_SYM_H
#define CRYPT_SYM_H
typed union tpmCryptKeySchedule_t {
#if ALG_AES
    tpmKeyScheduleAES AES;
#endif
#if ALG_SM4
    tpmKeyScheduleSM4 SM4;
#endif
#if ALG_CAMELLIA
    tpmKeyScheduleCAMELLIA CAMELLIA;
#endif
#if ALG_TDES
    tpmKeyScheduleTDES TDES[3];
#endif
#if SYMMETRIC_ALIGNMENT == 8
    uint64_t            alignment;
#else
    uint32_t            alignment;
#endif
} 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.

```c
#define ENCRYPT(keySchedule, in, out)   
  encrypt(SWIZZLE(keySchedule, in, out))
#define DECRYPT(keySchedule, in, out)   
  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.

```c
#define ENCRYPT_CASE(ALG)   
  case TPM_ALG_##ALG:  
    TpmCryptSetEncryptKey##ALG(key, keySizeInBits, &keySchedule.ALG);  
    encrypt = (TpmCryptSetSymKeyCall_t)TpmCryptEncrypt##ALG;  
    break;
#define DECRYPT_CASE(ALG)   
  case TPM_ALG_##ALG:  
    TpmCryptSetDecryptKey##ALG(key, keySizeInBits, &keySchedule.ALG);  
    decrypt = (TpmCryptSetSymKeyCall_t)TpmCryptDecrypt##ALG;  
    break;
#if ALG_AES
#define ENCRYPT_CASE_AES    ENCRYPT_CASE(AES)
```
For each algorithm the case will either be defined or null.

```c
#define SELECT(direction) 
   switch(algorithm) { 
   direction##_CASE_AES 
   direction##_CASE_SM4 
   direction##_CASE_CAMELLIA 
   direction##_CASE_TDES 
   default: 
      FAIL(FATAL_ERROR_INTERNAL); 
   }
#endif // CRYPT_SYM_H
```
10.1.14 OIDS.h

```c
#ifndef _OIDS_H_
#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).

#define MAKE_OID(NAME) \
    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.

#ifndef ALG_SHA1
    # define ALG_SHA1 NO
#ifndef ALG_SHA1
    # define SHA1_OID(NAME) MAKE_OID(NAME##SHA1)
#endif
#endif
#ifndef ALG_SHA256
    # define ALG_SHA256 NO
#endif
#ifndef ALG_SHA256
    # define SHA256_OID(NAME) MAKE_OID(NAME##SHA256)
#endif
#endif
#ifndef ALG_SHA384
    # define ALG_SHA384 NO
#endif
#ifndef ALG_SHA384
    # define SHA384_OID(NAME) MAKE_OID(NAME##SHA384)
#endif
#endif
#ifndef ALG_SHA512
    # define ALG_SHA512 NO
#endif
#ifndef ALG_SHA512
    # define SHA512_OID(NAME) MAKE_OID(NAME##SHA512)
#endif
#endif
#ifndef ALG_SM3_256
    # define ALG_SM3_256 NO
#endif
#ifndef ALG_SM3_256
    # define SM3_256_OID(NAME) MAKE_OID(NAME##SM3_256)
#endif
#endif
```
#else
#define SHA3_256_OID(NAME) MAKE_OID(NAME##SHA3_256)
#endif

#ifndef ALG_SHA3_384
#define ALG_SHA3_384 NO
#endif
#if ALG_SHA3_384
#define SHA3_384_OID(NAME) MAKE_OID(NAME##SHA3_384)
#else
#define SHA3_384_OID(NAME)
#endif

#ifndef ALG_SHA3_512
#define ALG_SHA3_512 NO
#endif
#if ALG_SHA3_512
#define SSHA3_512_OID(NAME) MAKE_OID(NAME##SHA3_512)
#else
#define SHA3_512_OID(NAME)
#endif

These are encoded to take one additional byte of algorithm selector

#define NIST_HASH 0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 2
#define NIST_SIG 0x06, 0x09, 0x60, 0x86, 0x48, 1, 101, 3, 4, 3

These hash OIDs used in a lot of places.

#define OID_SHA1_VALUE 0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A

// Expands to
//      MAKE_OID(_SHA1)
// which expands to:
//      extern BYTE OID_SHA1[]
// or
//      const BYTE OID_SHA1[] = {OID_SHA1_VALUE}
// which is:
//      const BYTE OID_SHA1[] = {0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A}
#define OID_SHA256_VALUE NIST_HASH, 1
#define OID_SHA384_VALUE NIST_HASH, 2
#define OID_SHA512_VALUE NIST_HASH, 3
#define OID_SM3_256_VALUE 0x06, 0x08, 0x2A, 0x81, 0xCF, 0x55, 0x01,
   0x83, 0x11
#define OID_SHA3_256_VALUE NIST_HASH, 8
#define OID_SHA3_384_VALUE NIST_HASH, 9
#define OID_SHA3_512_VALUE NIST_HASH, 10

These are used for RSA-PSS

#if ALG_RSA
#define OID_MGF1_VALUE 0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \ 0x01, 0x01, 0x08
#define OID_RSAPSS_VALUE 0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \ 0x01, 0x01, 0x0A
#endif

This is the OID to designate the public part of an RSA key.
#define OID_PKCS1_PUB_VALUE         0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, \ 
0x01, 0x01, 0x01
MAKE_OID(_PKCS1_PUB);

These are used for RSA PKCS1 signature Algorithms

#define OID_PKCS1_SHA1_VALUE         0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x0D, 0x01, 0x01, 0x05
SHA1_OID(_PKCS1_);
// (1.2.840.113549.1.1.5)

#define OID_PKCS1_SHA256_VALUE       0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x0D, 0x01, 0x01, 0x0B
SHA256_OID(_PKCS1_);
// (1.2.840.113549.1.1.11)

#define OID_PKCS1_SHA384_VALUE       0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x0D, 0x01, 0x01, 0x0C
SHA384_OID(_PKCS1_);
// (1.2.840.113549.1.1.12)

#define OID_PKCS1_SHA512_VALUE       0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x0D, 0x01, 0x01, 0x0D
SHA512_OID(_PKCS1_);
// (1.2.840.113549.1.1.13)

#define OID_PKCS1_SM3_256_VALUE      0x06, 0x08, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x0D, 0x01, 0x01, 0x0C
SHA3_256_OID(_PKCS1_);
// 1.2.156.10197.1.504

#define OID_PKCS1_SHA3_256_VALUE     NIST_SIG, 14
SHA3_256_OID(_PKCS1_);
#define OID_PKCS1_SHA3_384_VALUE     NIST_SIG, 15
SHA3_384_OID(_PKCS1_);
#define OID_PKCS1_SHA3_512_VALUE     NIST_SIG, 16
SHA3_512_OID(_PKCS1_);
#endif // ALG_RSA

#if ALG_ECDSA
#define OID_ECDSA_SHA1_VALUE         0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x01
SHA1_OID(_ECDSA_);
// (1.2.840.10045.4.1) SHA1 digest signed by an ECDSA key.

#define OID_ECDSA_SHA256_VALUE       0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x03, 0x02
SHA256_OID(_ECDSA_);
// (1.2.840.10045.4.3.2) SHA256 digest signed by an ECDSA key.

#define OID_ECDSA_SHA384_VALUE       0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x03, 0x03
SHA384_OID(_ECDSA_);
// (1.2.840.10045.4.3.3) SHA384 digest signed by an ECDSA key.

#define OID_ECDSA_SHA512_VALUE       0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x03, 0x04
SHA512_OID(_ECDSA_);
// (1.2.840.10045.4.3.4) SHA512 digest signed by an ECDSA key.

#define OID_ECDSA_SM3_256_VALUE      0x00
SM3_256_OID(_ECDSA_);
#define OID_ECDSA_SHA3_256_VALUE     NIST_SIG, 10
SHA3_256_OID(_ECDSA_);
#define OID_ECDSA_SHA3_384_VALUE     NIST_SIG, 11
SHA3_384_OID(_ECDSA_);
#define OID_ECDSA_SHA3_512_VALUE     NIST_SIG, 12
SHA3_512_OID(_ECDSA_);
#endif // ALG_ECDSA

#if ALG_ECC
#define OID_ECC_PUBLIC_VALUE         0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x01
MAKE_OID(_ECC_PUBLIC);
#define OID_ECC_NIST_P192_VALUE      0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x01, 0x01
MAKE_OID(_ECC_NIST_P192);
#endif // ALG_ECC

#if ECC_NIST_P192
#define OID_ECC_NIST_P192_VALUE      0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x01, 0x01
MAKE_OID(_ECC_NIST_P192);
#endif // ECC_NIST_P192

#define OID_ECC_NIST_P224_VALUE      0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x01, 0x01
MAKE_OID(_ECC_NIST_P224);
#define OID_ECC_NIST_P256_VALUE      0x06, 0x09, 0x2A, 0x86, 0x48, 0x86, 0xF7, \ 
0x01, 0x01
MAKE_OID(_ECC_NIST_P256);}
#if ECC_NIST_P256
MAKE_OID(_ECC_NIST_P256); // (1.2.840.10045.3.1.7) 'nistP256'
#endif // ECC_NIST_P256

#define OID_ECC_NIST_P384_VALUE 0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x22
#if ECC_NIST_P384
MAKE_OID(_ECC_NIST_P384); // (1.3.132.0.34) 'nistP384'
#endif // ECC_NIST_P384

#define OID_ECC_NIST_P521_VALUE 0x06, 0x05, 0x2B, 0x81, 0x04, 0x00, 0x23
#if ECC_NIST_P521
MAKE_OID(_ECC_NIST_P521); // (1.3.132.0.35) 'nistP521'
#endif // ECC_NIST_P521

No OIDs defined for these anonymous curves

#define OID_ECC_BN_P256_VALUE 0x00
#if ECC_BN_P256
MAKE_OID(_ECC_BN_P256);
#endif // ECC_BN_P256

#define OID_ECC_BN_P638_VALUE 0x00
#if ECC_BN_P638
MAKE_OID(_ECC_BN_P638);
#endif // ECC_BN_P638

#define OID_ECC_SM2_P256_VALUE 0x06, 0x08, 0x2A, 0x81, 0x1C, 0xCF, 0x55, 0x01, 0x82, 0x2D
#if ECC_SM2_P256
MAKE_OID(_ECC_SM2_P256); // Don’t know where I found this OID. It needs checking
#endif // ECC_SM2_P256

#if ECC_BN_P256
#define OID_ECC_BN_P256 NULL
#endif // ECC_BN_P256

#undef MAKE_OID

#define OID_SIZE(OID) (OID[1] + 2)
#endif // !_OIDS_H_
Trusted Platform Module Library

Part 4: Supporting Routines

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 0x19
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

```c
#define CHECK_SIZE(context, length) \  
   VERIFY(   (((length) + (context)->offset) >= (context)->offset) \  
         && (((length) + (context)->offset) <= (context)->size))
```

```c
#define NEXT_OCTET(context) ((context)->buffer[(context)->offset++])
#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

```c
#define CHECK_SPACE(context, length) VERIFY((context)->offset > length)
```

### 10.1.15.5 Structures

```c
typedef struct ASN1UnmarshalContext {
    BYTE *buffer; // pointer to the buffer
    INT16 size; // size of the buffer (a negative number indicates
    INT16 offset; // a parsing failure)
    INT16 offset; // current offset into the buffer (a negative number
    BYTE tag; // indicates a parsing failure). Not used
    BYTE tag; // The last unmarshaled tag
    } ASN1UnmarshalContext;
```

```c
typedef struct ASN1MarshalContext {
    BYTE *buffer; // pointer to the start of the buffer
    INT16 offset; // place on the top where the last entry was added
    INT16 end; // the end offset of the current value
    INT16 depth; // how many pushed end values.
    INT16 ends[MAX_DEPTH];
    } ASN1MarshalContext;
```

```c
#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 TBSertificate 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) \ const BYTE OID##NAME[] = {OID##NAME##_VALUE}
22 #else
23     #define MAKE_OID(NAME) \ extern const BYTE OID##NAME[]
24 #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;
10.1.16.5 Global X509 Constants

These values are instanced 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.

```
#define OID_KEY_USAGE_EXTENSION_VALUE 0x06, 0x03, 0x55, 0x1D, 0x0F
MAKE_OID(_KEY_USAGE_EXTENSION);
```

This is the DER-encoded value for the TCG-defined TPMA_OBJECT OID (2.23.133.10.1.1.1)

```
#define OID_TCG_TPMA_OBJECT_VALUE       0x06, 0x07, 0x67, 0x81, 0x05, 0x0a, 0x01, 0x01, 0x01
MAKE_OID(_TCG_TPMA_OBJECT);
```

```c
#ifdef _X509_SPT_
const x509KeyUsageUnion keyUsageSign = TPMA_X509_KEY_USAGE_INITIALIZER(
    /* digitalsignature */ 1,
    /* nonrepudiation */ 0,
    /* keyencipherment */ 0,
    /* dataencipherment */ 0,
    /* keyagreement */ 1,
    /* keycertsign */ 1,
    /* crlsign */ 1,
    /* decipheronly */ 0,
    /* encipheronly */ 0,
    /* bits_at_9 */ 0);
const x509KeyUsageUnion keyUsageDecrypt = TPMA_X509_KEY_USAGE_INITIALIZER(
    /* digitalsignature */ 0,
    /* nonrepudiation */ 0,
    /* keyencipherment */ 1,
    /* dataencipherment */ 1,
    /* keyagreement */ 1,
    /* keycertsign */ 0,
    /* crlsign */ 0,
    /* decipheronly */ 1,
    /* encipheronly */ 1,
    /* bits_at_9 */ 0);
#else
extern x509KeyUsageUnion keyUsageSign;
extern x509KeyUsageUnion keyUsageDecrypt;
#endif
#undef MAKE_OID
#endif // _X509_H_
```
10.1.17 MinMax.h

```c
#ifndef _MIN_MAX_H_
#define _MIN_MAX_H_

#ifndef MAX
#define MAX(a, b) ((a) > (b) ? (a) : (b))
#endif

#ifndef MIN
#define MIN(a, b) ((a) < (b) ? (a) : (b))
#endif

#endif // _MIN_MAX_H_
```
10.1.18  TpmAlgorithmDefines.h

This file contains the algorithm values from the TCG Algorithm Registry.

```c
#ifndef _TPM_ALGORITHM_DEFINES_H_
#define _TPM_ALGORITHM_DEFINES_H_

Table 2:3 - Definition of Base Types Base Types are in BaseTypes.h

#define ECC_CURVES
(TPM_ECC_BN_P256, TPM_ECC_BN_P638, TPM_ECC_NIST_P192, 
TPM_ECC_NIST_P224, TPM_ECC_NIST_P256, TPM_ECC_NIST_P384, 
TPM_ECC_NIST_P521, TPM_ECC_SM2_P256)

#define ECC_CURVE_COUNT
(ECC_BN_P256 + ECC_BN_P638 + ECC_NIST_P192 + ECC_NIST_P224 + 
ECC_NIST_P256 + ECC_NIST_P384 + ECC_NIST_P521 + ECC_SM2_P256)

#define MAX_ECC_KEY_BITS
MAX(ECC_BN_P256 * 256, MAX(ECC_BN_P638 * 638, 
MAX(ECC_NIST_P192 * 192, MAX(ECC_NIST_P224 * 224, 
MAX(ECC_NIST_P256 * 256, MAX(ECC_NIST_P384 * 384, 
MAX(ECC_NIST_P521 * 521, MAX(ECC_SM2_P256 * 256, 0))))))))

#define MAX_ECC_KEY_BYTES
BITS_TO_BYTES(MAX_ECC_KEY_BITS)

Table 0:6 - Defines for PLATFORM Values

#define PLATFORM_FAMILY         TPM_SPEC_FAMILY
#define PLATFORM_LEVEL          TPM_SPEC_LEVEL
#define PLATFORM_VERSION        TPM_SPEC_VERSION
#define PLATFORM_YEAR           TPM_SPEC_YEAR
#define PLATFORM_DAY_OF_YEAR    TPM_SPEC_DAY_OF_YEAR

Table 1:12 - Defines for SHA1 Hash Values

#define SHA1_DIGEST_SIZE  20
#define SHA1_BLOCK_SIZE   64

Table 1:13 - Defines for SHA256 Hash Values

#define SHA256_DIGEST_SIZE  32
#define SHA256_BLOCK_SIZE   64

Table 1:14 - Defines for SHA384 Hash Values

#define SHA384_DIGEST_SIZE  48
#define SHA384_BLOCK_SIZE   128

Table 1:15 - Defines for SHA512 Hash Values

#define SHA512_DIGEST_SIZE  64
#define SHA512_BLOCK_SIZE   128

Table 1:16 - Defines for SM3_256 Hash Values

#define SM3_256_DIGEST_SIZE     32
#define SM3_256_BLOCK_SIZE      64

Table 1:16 - Defines for SHA3_256 Hash Values

#define SHA3_256_DIGEST_SIZE    32
```c
#define SHA3_256_BLOCK_SIZE     136

Table 1:16 - Defines for SHA3_384 Hash Values
#define SHA3_384_DIGEST_SIZE    48
#define SHA3_384_BLOCK_SIZE     104

Table 1:16 - Defines for SHA3_512 Hash Values
#define SHA3_512_DIGEST_SIZE    64
#define SHA3_512_BLOCK_SIZE     72

Table 1:00 - Defines for RSA Asymmetric Cipher Algorithm Constants
#define RSA_KEY_SIZES_BITS     \
     (1024 * RSA_1024), (2048 * RSA_2048), (3072 * RSA_3072), (4096 * RSA_4096)
#if RSA_4096
    # define RSA_MAX_KEY_SIZE_BITS    4096
#elif RSA_3072
    # define RSA_MAX_KEY_SIZE_BITS    3072
#elif RSA_2048
    # define RSA_MAX_KEY_SIZE_BITS    2048
#elif RSA_1024
    # define RSA_MAX_KEY_SIZE_BITS    1024
#else
    # define RSA_MAX_KEY_SIZE_BITS    0
#endif
#define MAX_RSA_KEY_BITS       RSA_MAX_KEY_SIZE_BITS
#define MAX_RSA_KEY_BYTES      ((RSA_MAX_KEY_SIZE_BITS + 7) / 8)

Table 1:17 - Defines for AES Symmetric Cipher Algorithm Constants
#define AES_KEY_SIZES_BITS     \
     (128 * AES_128), (192 * AES_192), (256 * AES_256)
#if AES_256
    # define AES_MAX_KEY_SIZE_BITS    256
#elif AES_192
    # define AES_MAX_KEY_SIZE_BITS    192
#elif AES_128
    # define AES_MAX_KEY_SIZE_BITS    128
#else
    # define AES_MAX_KEY_SIZE_BITS    0
#endif
#define MAX_AES_KEY_BITS       AES_MAX_KEY_SIZE_BITS
#define MAX_AES_KEY_BYTES      ((AES_MAX_KEY_SIZE_BITS + 7) / 8)
#define AES_128_BLOCK_SIZE_BYTES    (AES_128 * 16)
#define AES_192_BLOCK_SIZE_BYTES    (AES_192 * 16)
#define AES_256_BLOCK_SIZE_BYTES    (AES_256 * 16)
#define AES_BLOCK_SIZES     \
     AES_128_BLOCK_SIZE_BYTES, AES_192_BLOCK_SIZE_BYTES, AES_256_BLOCK_SIZE_BYTES
#if ALG_AES
    # define AES_MAX_BLOCK_SIZE       16
#else
    # define AES_MAX_BLOCK_SIZE       0
#endif
#define MAX_AES_BLOCK_SIZE_BYTES   AES_MAX_BLOCK_SIZE

Table 1:18 - Defines for SM4 Symmetric Cipher Algorithm Constants
#define SM4_KEY_SIZES_BITS     (128 * SM4_128)
#if SM4_128
```

# define SM4_MAX_KEY_SIZE_BITS 128
#else
# define SM4_MAX_KEY_SIZE_BITS 0
#endif
#define MAX_SM4_KEY_BITS SM4_MAX_KEY_SIZE_BITS
#define MAX_SM4_KEY_BYTES ((SM4_MAX_KEY_SIZE_BITS + 7) / 8)
#define SM4_128_BLOCK_SIZE_BYTES (SM4_128 * 16)
#define SM4_BLOCK_SIZES SM4_128_BLOCK_SIZE_BYTES
#if ALG_SM4
# define SM4_MAX_BLOCK_SIZE 16
#else
# define SM4_MAX_BLOCK_SIZE 0
#endif
#define MAX_SM4_BLOCK_SIZE_BYTES SM4_MAX_BLOCK_SIZE

Table 1:19 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants
#define CAMELLIA_KEY_SIZES_BITS \( (128 \times \text{CAMELLIA}_{128}), (192 \times \text{CAMELLIA}_{192}), (256 \times \text{CAMELLIA}_{256}) \)
#if CAMELLIA_256
# define CAMELLIA_MAX_KEY_SIZE_BITS 256
#else CAMELLIA_192
# define CAMELLIA_MAX_KEY_SIZE_BITS 192
#else CAMELLIA_128
# define CAMELLIA_MAX_KEY_SIZE_BITS 128
#else
# define CAMELLIA_MAX_KEY_SIZE_BITS 0
#endif
#define MAX_CAMELLIA_KEY_BITS CAMELLIA_MAX_KEY_SIZE_BITS
#define MAX_CAMELLIA_KEY_BYTES ((CAMELLIA_MAX_KEY_SIZE_BITS + 7) / 8)
#define CAMELLIA_128_BLOCK_SIZE_BYTES (CAMELLIA_128 * 16)
#define CAMELLIA_192_BLOCK_SIZE_BYTES (CAMELLIA_192 * 16)
#define CAMELLIA_256_BLOCK_SIZE_BYTES (CAMELLIA_256 * 16)
#define CAMELLIA_BLOCK_SIZES \( \text{CAMELLIA}_{128} \text{BLOCK_SIZE_BYTES, \ CAMELLIA}_{192} \text{BLOCK_SIZE_BYTES,} \ CAMELLIA_{256} \text{BLOCK_SIZE_BYTES} \)
#if ALG_CAMELLIA
# define CAMELLIA_MAX_BLOCK_SIZE 16
#else
# define CAMELLIA_MAX_BLOCK_SIZE 0
#endif
#define MAX_CAMELLIA_BLOCK_SIZE_BYTES CAMELLIA_MAX_BLOCK_SIZE

Table 1:17 - Defines for TDES Symmetric Cipher Algorithm Constants
#define TDES_KEY_SIZES_BITS \( (128 \times \text{TDES}_{128}), (192 \times \text{TDES}_{192}) \)
#if TDES_192
# define TDES_MAX_KEY_SIZE_BITS 192
#else TDES_128
# define TDES_MAX_KEY_SIZE_BITS 128
#else
# define TDES_MAX_KEY_SIZE_BITS 0
#endif
#define MAX_TDES_KEY_BITS TDES_MAX_KEY_SIZE_BITS
#define MAX_TDES_KEY_BYTES ((TDES_MAX_KEY_SIZE_BITS + 7) / 8)
#define TDES_128_BLOCK_SIZE_BYTES (TDES_128 * 8)
#define TDES_192_BLOCK_SIZE_BYTES (TDES_192 * 8)
#define TDES_BLOCK_SIZES \( \text{TDES}_{128} \text{BLOCK_SIZE_BYTES, TDES}_{192} \text{BLOCK_SIZE_BYTES} \)
#if ALG_TDES
# define TDES_MAX_BLOCK_SIZE 8
#else
# define TDES_MAX_BLOCK_SIZE 0
#endif
#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.UndefineSpace) /* 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.UndefineSpace) /* 0x00000134 */
170 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000135 */
171 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000136 */
172 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000137 */
173 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000138 */
174 + (ADD_FILL) | CC_DictionaryAttackLockReset) /* 0x00000139 */
175 + (ADD_FILL) | CC_DictionaryAttackParameters) /* 0x0000013A */
176 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000013B */
177 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000013C */
178 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000013D */
179 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000013E */
180 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000013F */
181 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000140 */
182 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000141 */
183 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000142 */
184 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000143 */
185 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000144 */
186 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000145 */
187 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000146 */
188 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000147 */
189 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000148 */
190 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000149 */
191 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000014A */
192 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000014B */
193 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000014C */
194 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000014D */
195 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000014E */
196 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x0000014F */
197 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000150 */
198 + (ADD_FILL) | CC_NV.UndefineSpace) /* 0x00000151 */
Part 4: Supporting Routines

+ (ADD_FILL || CC_Rewrap) /* 0x00000191 */
+ (ADD_FILL || CC_Create) /* 0x00000193 */
+ (ADD_FILL || CC_EC_DH_ZGen) /* 0x00000194 */
+ (ADD_FILL || CC_HMAC || CC_MAC) /* 0x00000195 */
+ (ADD_FILL || CC_Import) /* 0x00000196 */
+ (ADD_FILL || CC_Load) /* 0x00000197 */
+ (ADD_FILL || CC_Quote) /* 0x00000198 */
+ (ADD_FILL || CC_RSA_Decrypt) /* 0x00000199 */
+ ADD_FILL
+ (ADD_FILL || CC_HMAC_Start || CC_MAC_Start) /* 0x0000019B */
+ (ADD_FILL || CC_SequanceUpdate) /* 0x0000019C */
+ (ADD_FILL || CC_Sign) /* 0x0000019D */
+ (ADD_FILL || CC_Unseal) /* 0x0000019E */
+ ADD_FILL /* 0x0000019F */
+ (ADD_FILL || CC_PolicySigned) /* 0x000001A0 */
+ (ADD_FILL || CC_ContextLoad) /* 0x000001A1 */
+ (ADD_FILL || CC_ContextSave) /* 0x000001A2 */
+ (ADD_FILL || CC_EC_DH_KeyGen) /* 0x000001A3 */
+ (ADD_FILL || CC_EncryptDecrypt) /* 0x000001A4 */
+ (ADD_FILL || CC_FlushContext) /* 0x000001A5 */
+ (ADD_FILL || CC_LoadExternal) /* 0x000001A6 */
+ (ADD_FILL || CC_MakeCredential) /* 0x000001A7 */
+ (ADD_FILL || CC_NV_ReadPublic) /* 0x000001A8 */
+ (ADD_FILL || CC_PolicyAuthorize) /* 0x000001A9 */
+ (ADD_FILL || CC_PolicyAuthValue) /* 0x000001AA */
+ (ADD_FILL || CC_PolicyCommandCode) /* 0x000001AB */
+ (ADD_FILL || CC_PolicyCounterTimer) /* 0x000001AC */
+ (ADD_FILL || CC_PolicyCpHash) /* 0x000001AD */
+ (ADD_FILL || CC_PolicyLocality) /* 0x000001AE */
+ (ADD_FILL || CC_PolicyNameHash) /* 0x000001AF */
+ (ADD_FILL || CC_PolicyOR) /* 0x000001B0 */
+ (ADD_FILL || CC_PolicyTicket) /* 0x000001B1 */
+ (ADD_FILL || CC_ReadPublic) /* 0x000001B2 */
+ (ADD_FILL || CC_RSA_Encrypt) /* 0x000001B3 */
+ ADD_FILL /* 0x000001B4 */
+ (ADD_FILL || CC_StartAuthSession) /* 0x000001B5 */
+ (ADD_FILL || CC_VerifySignature) /* 0x000001B6 */
+ (ADD_FILL || CC_ECC_Parameters) /* 0x000001B7 */
+ (ADD_FILL || CC_FirmwareRead) /* 0x000001B8 */
+ (ADD_FILL || CC_GetCapability) /* 0x000001B9 */
+ (ADD_FILL || CC_GetRandom) /* 0x000001BA */
+ (ADD_FILL || CC_GetTestResult) /* 0x000001BB */
+ (ADD_FILL || CC_GetHash) /* 0x000001BC */
+ (ADD_FILL || CC_PICTURE) /* 0x000001BD */
+ (ADD_FILL || CC_PolicyRead) /* 0x000001BE */
+ (ADD_FILL || CC_PolicyPCR) /* 0x000001BF */
+ (ADD_FILL || CC_PolicyRestart) /* 0x000001C0 */
+ (ADD_FILL || CC_ReadClock) /* 0x000001C1 */
+ (ADD_FILL || CC_PolicyCmd) /* 0x000001C2 */
+ (ADD_FILL || CC_PolicyExt) /* 0x000001C3 */
+ (ADD_FILL || CC_PolicyExtValue) /* 0x000001C4 */
+ (ADD_FILL || CC_NV_Certify) /* 0x000001C5 */
+ (ADD_FILL || CC_EventSequenceComplete) /* 0x000001C6 */
+ (ADD_FILL || CC_HashSequenceStart) /* 0x000001C7 */
+ (ADD_FILL || CC_PolicyPhysicalPresence) /* 0x000001C8 */
+ (ADD_FILL || CC_PolicyDuplicationSelect) /* 0x000001C9 */
+ (ADD_FILL || CC_PolicyGetDigest) /* 0x000001CA */
+ ADD_FILL /* 0x000001CB */
+ (ADD_FILL || CC_PolicyPassword) /* 0x000001CC */
+ (ADD_FILL || CC_ZGen_2Phase) /* 0x000001CD */
+ (ADD_FILL || CC_EC_Ephemeral) /* 0x000001CE */
+ (ADD_FILL || CC_PolicyNVW) /* 0x000001CF */
+ (ADD_FILL || CC_PolicyTemplate) /* 0x000001D0 */
+ (ADD_FILL || CC_PolicyRead) /* 0x000001D1 */
+ (ADD_FILL || CC_CreateLoaded) /* 0x000001D2 */
+ (ADD_FILL || CC_PolicyAuthorizeNV) /* 0x000001D3 */
+ (ADD_FILL || CC_EncryptDecrypt2) /* 0x000001D4 */
Define the 2B structure that would hold any hash block

```c
TPM2B_TYPE(MAX_HASH_BLOCK, MAX_HASH_BLOCK_SIZE);
```

Following typedef is for some old code

```c
typedef TPM2B_MAX_HASH_BLOCK TPM2B_HASH_BLOCK;
```

Additional symmetric constants

```c
#define MAX_SYM_KEY_BITS (MAX(AES_MAX_KEY_SIZE_BITS,      MAX(CAMELLIA_MAX_KEY_SIZE_BITS,      MAX(SM4_MAX_KEY_SIZE_BITS,      MAX(TDES_MAX_KEY_SIZE_BITS, 0)))))))
#define MAX_SYM_KEY_BYTES ((MAX_SYM_KEY_BITS + 7) / 8)
```

```c
# if MAX_SYM_KEY_BITS == 0 || MAX_SYM_BLOCK_SIZE == 0
#error "Bad size for MAX_SYM_KEY_BITS or MAX_SYM_BLOCK"
#endif
```

```c
#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

```c
#include "Tpm.h"
#define SELF_TEST_DATA
#if SELF_TEST

These includes pull in the data structures. They contain data definitions for the various tests.

```c
#include "SelfTest.h"
#include "SymmetricTest.h"
#include "RsaTestData.h"
#include "EccTestData.h"
#include "HashTestData.h"
#include "KdfTestData.h"

#define TEST_DEFAULT_TEST_HASH(vector) \ 
if (TEST_BIT(DEFAULT_TEST_HASH, g_toTest)) \ 
  TestHash(DEFAULT_TEST_HASH, vector);

Make sure that the algorithm has been tested
```

```c
#define CLEAR_BOTH(alg) { \ 
  CLEAR_BIT(alg, *toTest); \ 
  if (toTest != &g_toTest) \ 
    CLEAR_BIT(alg, g_toTest); }
#define SET_BOTH(alg) { \ 
  SET_BIT(alg, *toTest); \ 
  if (toTest != &g_toTest) \ 
    SET_BIT(alg, g_toTest); }
#define TEST_BOTH(alg) \ 
  if (((toTest != &g_toTest) \ 
    ? TEST_BIT(alg, *toTest) || TEST_BIT(alg, g_toTest) \ 
    : TEST_BIT(alg, *toTest))

Can only cancel if doing a list.
```

```c
#define CHECK_CANCELED \ 
if (_plat__IsCanceled() && toTest != &g_toTest) \ 
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.
static TPM_RC TestHash(
    TPM_ALG_ID hashAlg,
    ALGORITHM_VECTOR *toTest
) {
    static TPM2B_DIGEST computed; // value computed
    static HMAC_STATE state;
    UINT16 digestSize;
    const TPM2B *testDigest = NULL;
    // TPM2B_TYPE(HMAC_BLOCK, DEFAULT_TEST_HASH_BLOCK_SIZE);
    pAssert(hashAlg != ALG_NULL_VALUE);
    switch(hashAlg) {
        #if ALG_SHA1
            case ALG_SHA1_VALUE:
                testDigest = &c_SHA1_digest.b;
                break;
        #endif
        #if ALG_SHA256
            case ALG_SHA256_VALUE:
                testDigest = &c_SHA256_digest.b;
                break;
        #endif
        #if ALG_SHA384
            case ALG_SHA384_VALUE:
                testDigest = &c_SHA384_digest.b;
                break;
        #endif
        #if ALG_SHA512
            case ALG_SHA512_VALUE:
                testDigest = &c_SHA512_digest.b;
                break;
        #endif
        #if ALG_SM3_256
            case ALG_SM3_256_VALUE:
                testDigest = &c_SM3_256_digest.b;
                break;
        #endif
        default:
            FAIL(FATAL_ERROR_INTERNAL);
    }
    // Clear the to-test bits
    CLEAR_BOTH(hashAlg);
    // Set the HMAC key to twice the digest size
    digestSize = CryptHashGetDigestSize(hashAlg);
    CryptHmacStart(&state, hashAlg, digestSize * 2,
                    (BYTE *)c_hashTestKey.t.buffer);
    CryptDigestUpdate(&state.hashState, 2 * CryptHashGetBlockSize(hashAlg),
                      (BYTE *)c_hashTestData.t.buffer);
    computed.t.size = digestSize;
    CryptHmacEnd(&state, digestSize, computed.t.buffer);
    if((testDigest->size != computed.t.size)
        || (memcmp(testDigest->buffer, computed.t.buffer, computed.b.size) != 0))
        SELF_TEST_FAILURE;
    return TPM_RC_SUCCESS;
}
10.2.1.4 Symmetric Test Functions

10.2.1.4.1 MakeIv()

Internal function to make the appropriate IV depending on the mode.

```
static UINT32
MakeIv(
    TPM_ALG_ID mode,    // IN: symmetric mode
    UINT32 size,        // IN: block size of the algorithm
    BYTE *iv            // OUT: IV to fill in
)
{
    BYTE i;
    if(mode == ALG_ECB_VALUE)
        return 0;
    if(mode == ALG_CTR_VALUE)
    {
        // The test uses an IV that has 0xff in the last byte
        for(i = 1; i <= size; i++)
            *iv++ = 0xff - (BYTE)(size - i);
    }
    else
    {
        for(i = 0; i < size; i++)
            *iv++ = i;
    }
    return size;
}
```

10.2.1.4.2 TestSymmetricAlgorithm()

Function to test a specific algorithm, key size, and mode.

```
static void
TestSymmetricAlgorithm(
    const SYMMETRIC_TEST_VECTOR *test,      // IN: test values
    TPM_ALG_ID mode                        // IN: symmetric mode
)
{
    static BYTE encrypted[MAX_SYM_BLOCK_SIZE * 2];
    static BYTE decrypted[MAX_SYM_BLOCK_SIZE * 2];
    static TPM2B_IV iv;
    // Get the appropriate IV
    iv.t.size = (UINT16)MakeIv(mode, test->ivSize, iv.t.buffer);
    // Encrypt known data
    CryptSymmetricEncrypt(encrypted, test->alg, test->keyBits, test->key, &iv,
        mode, test->dataInOutSize, test->dataIn);
    // Check that it matches the expected value
    if(!MemoryEqual(encrypted, test->dataOut[mode - ALG_CTR_VALUE],
        test->dataInOutSize))
        SELF_TEST_FAILURE;
    // Reinitialize the IV for decryption
    MakeIv(mode, test->ivSize, iv.t.buffer);
    CryptSymmetricDecrypt(decrypted, test->alg, test->keyBits, test->key, &iv,
        mode, test->dataInOutSize,
        test->dataOut[mode - ALG_CTR_VALUE]);
    // Make sure that it matches what we started with
    if(!MemoryEqual(decrypted, test->dataIn, test->dataInOutSize))
        SELF_TEST_FAILURE;
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>all symmetric algorithms tested</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>not all symmetric algorithms tested</td>
</tr>
</tbody>
</table>

```c
static BOOL AllSymsAreDone(  
    ALGORITHM_VECTOR *toTest  
)  
{
    return (!TEST_BOTH(ALG_AES_VALUE) && !TEST_BOTH(ALG_SM4_VALUE));
}
```

10.2.1.4.4 AllModesAreDone()

Checks if all the modes have been tested.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>all modes tested</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>all modes not tested</td>
</tr>
</tbody>
</table>

```c
static BOOL AllModesAreDone(  
    ALGORITHM_VECTOR *toTest  
)  
{
    TPM_ALG_ID alg;
    for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
        if(TEST_BOTH(alg))
            return FALSE;
    return TRUE;
}
```

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.

```c
static TPM_RC TestSymmetric(  
    TPM_ALG_ID alg,  
    ALGORITHM_VECTOR *toTest  
)  
{
    TPM_ALG_ID mode;
    //
    if(!TEST_BIT(alg, *toTest))
        return TPM_RC_SUCCESS;
    if(alg == ALG_AES_VALUE || alg == ALG_SM4_VALUE || alg == ALG_CAMELLIA_VALUE)
    {
```
// Will test the algorithm for all modes and key sizes
CLEAR_BOTH(alg);

// A test this algorithm for all modes
for(index = 0; index < NUM_SYMS; index++)
{
    if(c_symTestValues[index].alg == alg)
    {
        for(mode = TPM_SYM_MODE_FIRST;
            mode <= TPM_SYM_MODE_LAST;
            mode++)
        {
            if(TEST_BIT(mode, *toTest))
                TestSymmetricAlgorithm(&c_symTestValues[index], mode);
        }
    }
}

// if all the symmetric tests are done
if(AllSymsAreDone(toTest))
{
    // all symmetric algorithms tested so no modes should be set
    for(alg = TPM_SYM_MODE_FIRST; alg <= TPM_SYM_MODE_LAST; alg++)
        CLEAR_BOTH(alg);
}
else if(TPM_SYM_MODE_FIRST <= alg && alg <= TPM_SYM_MODE_LAST)
{
    // Test this mode for all key sizes and algorithms
    for(index = 0; index < NUM_SYMS; index++)
    {
        // The mode testing only comes into play when doing self tests
        // by command. When doing self tests by command, the block ciphers are
        // tested first. That means that all of their modes would have been
        // tested for all key sizes. If there is no block cipher left to
        // test, then clear this mode bit.
        if(!TEST_BIT(ALG_AES_VALUE, *toTest)
            && !TEST_BIT(ALG_SM4_VALUE, *toTest))
            CLEAR BOTH(alg);
    }
}
else
{
    for(index = 0; index < NUM_SYMS; index++)
    {
        if(TEST_BIT(c_symTestValues[index].alg, *toTest))
            TestSymmetricAlgorithm(&c_symTestValues[index], alg);
    }
    // have tested this mode for all algorithms
    CLEAR BOTH(alg);
}

if(AllModesAreDone(toTest))
{
    CLEAR BOTH(ALG_AES_VALUE);
    CLEAR BOTH(ALG_SM4_VALUE);
}
else
    pAssert(alg == 0 && alg != 0);
return TPM_RC_SUCCESS;
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.

```c
static void RsaKeyInitialize

O(B) *testObject

{
  MemoryCopy2B(&testObject->publicArea.unique.rsa.b, (P2B)&c_rsaPublicModulus,
  sizeof(c_rsaPublicModulus));
  MemoryCopy2B(&testObject->sensitive.sensitive.rsa.b, (P2B)&c_rsaPrivatePrime,
  sizeof(testObject->sensitive.sensitive.rsa.t.buffer));
  testObject->publicArea.parameters.rsaDetail.keyBits = RSA_TEST_KEY_SIZE * 8;
  // Use the default exponent
  testObject->publicArea.parameters.rsaDetail.exponent = 0;
}
```

10.2.1.5.3 TestRsaEncryptDecrypt()

These test are for an public key encryption that uses a random value

```c
static TPM_RC TestRsaEncryptDecrypt

TMP_ALG_ID scheme, // IN: the scheme
ALGORITHM_VECTOR *toTest //

{
  static TPM2B_PUBLIC_KEY_RSA testInput;
  static TPM2B_PUBLIC_KEY_RSA testOutput;
  static OBJECT testObject;
  const TPM2B_RSA_TEST_KEY *kvtValue = NULL;
  TPM_RC result = TPM_RC_SUCCESS;
  const TPM2B *testLabel = NULL;
  TPM2B_RSA_DECRYPT rsaScheme;
  // Don't need to initialize much of the test object
  RsaKeyInitialize(&testObject);
  rsaScheme.scheme = scheme;
  rsaScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;
  CLEAR_BOTH(scheme);
  CLEAR_BOTH(ALG_NULL_VALUE);
  if(scheme == ALG_NULL_VALUE)
  {
```
// This is an encryption scheme using the private key without any encoding.
memcpy(testInput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue));
testInput.t.size = sizeof(c_RsaTestValue);
if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b, 
    &testObject, &rsaScheme, NULL, NULL))
    SELF_TEST_FAILURE;
if(!MemoryEqual(testOutput.t.buffer, c_RsaeKvt.buffer, c_RsaeKvt.size))
    SELF_TEST_FAILURE;
MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b, 
    &testObject, &rsaScheme, NULL))
    SELF_TEST_FAILURE;
if(!MemoryEqual(testOutput.t.buffer, c_RsaTestValue, sizeof(c_RsaTestValue)))
    SELF_TEST_FAILURE;
}
else
{
// ALG_RSAES_VALUE:
// This is a decryption scheme using padding according to
// PKCS#1v2.1, 7.2. This padding uses random bits. To test a public
// key encryption that uses random data, encrypt a value and then
// decrypt the value and see that we get the encrypted data back.
// The hash is not used by this encryption so it can be TMP_ALG_NULL

// ALG_OAEP_VALUE:
// This is also an decryption scheme and it also uses a
// pseudo-random
// value. However, this also uses a hash algorithm. So, we may need
// to test that algorithm before use.
if(scheme == ALG_OAEP_VALUE)
{
    TEST_DEFAULT_TEST_HASH(toTest);
    kvtValue = &c_OaepKvt;
    testLabel = OAEP_TEST_STRING;
}
else if(scheme == ALG_RSAES_VALUE)
{
    kvtValue = &c_RsaesKvt;
    testLabel = NULL;
}
else
    SELF_TEST_FAILURE;
// Only use a digest-size portion of the test value
memcpy(testInput.t.buffer, c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
testInput.t.size = DEFAULT_TEST_DIGEST_SIZE;
// See if the encryption works
if(TPM_RC_SUCCESS != CryptRsaEncrypt(&testOutput, &testInput.b, 
    &testObject, &rsaScheme, testLabel, 
    NULL))
    SELF_TEST_FAILURE;
MemoryCopy2B(&testInput.b, &testOutput.b, sizeof(testInput.t.buffer));
// see if we can decrypt this value and get the original data back
if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b, 
    &testObject, &rsaScheme, testLabel))
    SELF_TEST_FAILURE;
// See if the results compare
if(testOutput.t.size != DEFAULT_TEST_DIGEST_SIZE 
    || !MemoryEqual(testOutput.t.buffer, c_RsaTestValue, 
    DEFAULT_TEST_DIGEST_SIZE))
    SELF_TEST_FAILURE;
// Now check that the decryption works on a known value
MemoryCopy2B(&testInput.b, (P2B)kvtValue, 
    sizeof(testInput.t.buffer));
if(TPM_RC_SUCCESS != CryptRsaDecrypt(&testOutput.b, &testInput.b, 
    &testObject, 
    &rsaScheme, 
    testLabel))
    SELF_TEST_FAILURE;
10.2.1.5.4 TestRsaSignAndVerify()

This function does the testing of the RSA sign and verification functions. This test does a KVT.

static TPM_RC
TestRsaSignAndVerify(  
    TPM_ALG_ID scheme,  
    ALGORITHM_VECTOR *toTest  
)  
{
    TPM_RC result = TPM_RC_SUCCESS;
    static OBJECT testObject;
    static TPM2B_DIGEST testDigest;
    static TPMT_SIGNATURE testSig;

    // Do a sign and signature verification.
    // RSASSA:
    // This is a signing scheme according to PKCS#1-v2.1 8.2. It does not
    // use random data so there is a KVT for the signing operation. On
    // first use of the scheme for signing, use the TPM's RSA key to
    // sign a portion of c_RsaTestData and compare the results to c_RsassaKvt. Then
    // decrypt the data to see that it matches the starting value. This verifies
    // the signature with a KVT
    // Clear the bits indicating that the function has not been checked. This is to
    // prevent looping
    CLEAR_BOTH(scheme);
    CLEAR_BOTH(ALG_NULL_VALUE);
    CLEAR_BOTH(ALG_RSA_VALUE);

    RsaKeyInitialize(&testObject);
    memcpy(testDigest.t.buffer, (BYTE *)c_RsaTestValue, DEFAULT_TEST_DIGEST_SIZE);
    testDigest.t.size = DEFAULT_TEST_DIGEST_SIZE;
    testSig.sigAlg = scheme;
    testSig.signature.rsapss.hash = DEFAULT_TEST_HASH;

    // RSAPSS:
    // This is a signing scheme a according to PKCS#1-v2.2 8.1 it uses
    // random data in the signature so there is no KVT for the signing
    // operation. To test signing, the TPM will use the TPM's RSA key
    // to sign a portion of c_RsaTestValue and then it will verify the
    // signature. For verification, c_RsapssKvt is verified before the
    // user signature blob is verified. The worst case for testing of this
    // algorithm is two private and one public key operation.
    // The process is to sign known data. If RSASSA is being done, verify that the
    // signature matches the precomputed value. For both, use the signed value and
    // see that the verification says that it is a good signature. Then
    // if testing RSAPSS, do a verify of a known good signature. This ensures that
    // the validation function works.
    if(TPM_RC_SUCCESS != CryptRsaSign(&testSig, &testObject, &testDigest, NULL))
        SELF_TEST_FAILURE;

    // For RSASSA, make sure the results is what we are looking for
if(testSig.sigAlg == ALG_RSASSA_VALUE)
{
    if(testSig.signature.rsassa.sig.t.size != RSA_TEST_KEY_SIZE
        || !MemoryEqual(c_RsassaKvt.buffer,
                       testSig.signature.rsassa.sig.t.buffer,
                       RSA_TEST_KEY_SIZE))
        SELF_TEST_FAILURE;
}

// See if the TPM will validate its own signatures
if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
                                                  &testDigest))
    SELF_TEST_FAILURE;

// If this is RSAPSS, check the verification with known signature
// Have to copy because CryptRsaValidateSignature() eats the signature
if(ALG_RSAPSS_VALUE == scheme)
{
    MemoryCopy2B(&testSig.signature.rsapss.sig.b, (P2B)&c_RsapssKvt,
                 sizeof(testSig.signature.rsapss.sig.t.buffer));
    if(TPM_RC_SUCCESS != CryptRsaValidateSignature(&testSig, &testObject,
                                                  &testDigest))
        SELF_TEST_FAILURE;
}

return result;

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

static TPM_RC
TestRsa(TPM_ALG_ID alg,
        ALGORITHM_VECTOR *toTest)
{
    TPM_RC result = TPM_RC_SUCCESS;

    switch(alg)
    {
        case ALG_NULL_VALUE:
            // This is the RSAEP/RSADP function. If we are processing a list, don't
            // need to test these now because any other test will validate
            // RSAEP/RSADP. Can tell this is list of test by checking to see if
            // 'toTest' is pointing at g_toTest. If so, this is an isolated test
            // an need to go ahead and do the test;
            if((toTest == &g_toTest)
                || (!TEST_BIT(ALG_RSASSA_VALUE, *toTest)
                        && !TEST_BIT(ALG_RSAES_VALUE, *toTest)
                        && !TEST_BIT(ALG_RSAPSS_VALUE, *toTest)
                        && !TEST_BIT(ALG_OAEP_VALUE, *toTest)))
                // Not running a list of tests or no other tests on the list
                // so run the test now
                result = TestRsaEncryptDecrypt(alg, toTest);
                // if not running the test now, leave the bit on, just in case things
                // get interrupted
                break;
        case ALG_OAEP_VALUE:
        case ALG_RSAES_VALUE:
            result = TestRsaEncryptDecrypt(alg, toTest);
            break;
        case ALG_RSAPSS_VALUE:
        case ALG_RSASSA_VALUE:
            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

#if ALG_ECC

10.2.1.6.1  LoadEccParameter()

This function is mostly for readability and type checking

static void
LoadEccParameter(
    TPM2B_ECC_PARAMETER          *to,       // target
    const TPM2B_EC_TEST          *from     // source
)
{
    MemoryCopy2B(&to->b, &from->b, sizeof(to->t.buffer));
}

10.2.1.6.2  LoadEccPoint()

static void
LoadEccPoint(
    TPMS_ECC_POINT               *point,   // target
    const TPM2B_EC_TEST          *x,       // source
    const TPM2B_EC_TEST          *y
)
{
    MemoryCopy2B(&point->x.b, (TPM2B *)x, sizeof(point->x.t.buffer));
    MemoryCopy2B(&point->y.b, (TPM2B *)y, sizeof(point->y.t.buffer));
}

10.2.1.6.3  TestECDH()

This test does a KVT on a point multiply.

static TPM_RC
TestECDH(
    TPM_ALG_ID          scheme,  // IN: for consistency
    ALGORITHM_VECTOR    *toTest   // IN/OUT: modified after test is run
)
{
    static TPMS_ECC_POINT       Z;
    static TPMS_ECC_POINT       Qe;
    static TPM2B_ECC_PARAMETER  ds;
    static TPM_RC                      result = TPM_RC_SUCCESS;
    // NOT_REFERENCED(scheme);
    CLEAR_BOTH(ALG_ECDH_VALUE);
    LoadEccParameter(&ds, &c_ecTestKey_ds);
    LoadEccPoint(&Qe, &c_ecTestKey_QeX, &c_ecTestKey_QeY);
    if(TPM_RC_SUCCESS != CryptEccPointMultiply(&Z, c_testCurve, &Qe, &ds, NULL, NULL))
        SELF_TEST_FAILURE;
    if(!MemoryEqual2B(&c_ecTestEcdh_X.b, &Z.x.b)
\begin{verbatim}
|| !MemoryEqual2B(&c_ecTestEcdh_Y.b, &Z.y.b))
        SELF_TEST_FAILURE;
        return result;
\}

10.2.1.6.4 TestEccSignAndVerify()

static TPM_RC
TestEccSignAndVerify(
    TPM_ALG_ID                   scheme,
    ALGORITHM_VECTOR            *toTest
)
{
    static OBJECT                testObject;
    static TPMT_SIGNATURE        testSig;
    static TPMT_ECC_SCHEME       eccScheme;

    testSig.sigAlg = scheme;
    testSig.signature.ecdsa.hash = DEFAULT_TEST_HASH;

    eccScheme.scheme = scheme;
    eccScheme.details.anySig.hashAlg = DEFAULT_TEST_HASH;

    CLEAR_BOTH(scheme);
    CLEAR_BOTH(ALG_ECDH_VALUE);

    // ECC signature verification testing uses a KVT.
    switch(scheme)
    {
        case ALG_ECDSA_VALUE:
            LoadEccParameter(&testSig.signature.ecdsa.signatureR, &c_TestEcDsa_r);
            LoadEccParameter(&testSig.signature.ecdsa.signatureS, &c_TestEcDsa_s);
            break;
        case ALG_ECSCHNORR_VALUE:
            LoadEccParameter(&testSig.signature.ecschnorr.signatureR, &c_TestEcSchnorr_r);
            LoadEccParameter(&testSig.signature.ecschnorr.signatureS, &c_TestEcSchnorr_s);
            break;
        case ALG_SM2_VALUE:
            // don't have a test for SM2
            return TPM_RC_SUCCESS;
        default:
            SELF_TEST_FAILURE;
            break;
    }

    TEST_DEFAULT_TEST_HASH(toTest);

    // Have to copy the key. This is because the size used in the test vectors
    // is the size of the ECC parameter for the test key while the size of a point
    // is TPM dependent
    MemoryCopy2B(&testObject.sensitive.sensitive.ecc.b, &c_ecTestKey_ds.b, sizeof(testObject.sensitive.sensitive.ecc.t.buffer));
    LoadEccPoint(&testObject.publicArea.unique.ecc, &c_ecTestKey_QsX, &c_ecTestKey_QsY);
    testObject.publicArea.parameters.eccDetail.curveID = c_testCurve;

    if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject,
            (TPM2B_DIGEST *)&c_ecTestValue.b))
    {
        SELF_TEST_FAILURE;
    }
    CHECK_CANCELED;

    // Now sign and verify some data
\end{verbatim}
if(TPM_RC_SUCCESS != CryptEccSign(&testSig, &testObject, 
(TPM2B_DIGEST *)&c_ecTestValue, 
&eccScheme, NULL)) 
SELF_TEST_FAILURE;
CHECK_CANCELED;

if(TPM_RC_SUCCESS != CryptEccValidateSignature(&testSig, &testObject, 
(TPM2B_DIGEST *)&c_ecTestValue)) 
SELF_TEST_FAILURE;
CHECK_CANCELED;

return TPM_RC_SUCCESS;

10.2.1.6.5 TestKDFa()

static TPM_RC 
TestKDFa(
ALGORITHM_VECTOR *toTest )
{
static TPM2B_KDF_TEST_KEY keyOut;
UINT32 counter = 0;
// CLEAR_BOTH(ALG_KDF1_SP800_108_VALUE);
keyOut.t.size = CryptKDFa(KDF_TEST_ALG, &c_kdfTestKeyIn.b, &c_kdfTestLabel.b, 
&c_kdfTestContextU.b, &c_kdfTestContextV.b, 
TEST_KDF_KEY_SIZE * 8, keyOut.t.buffer, 
&counter, FALSE);
if ( keyOut.t.size != TEST_KDF_KEY_SIZE 
|| !MemoryEqual(keyOut.t.buffer, c_kdfTestKeyOut.t.buffer, 
TEST_KDF_KEY_SIZE)) 
SELF_TEST_FAILURE;

return TPM_RC_SUCCESS;

10.2.1.6.6 TestEcc()

static TPM_RC 
TestEcc( 
TPM_ALG_ID alg, 
ALGORITHM_VECTOR *toTest )
{
TPM_RC result = TPM_RC_SUCCESS;
NOT_REFERENCED(toTest);
switch(alg)
{
case ALG_ECC_VALUE:
case ALG_ECDH_VALUE:
// If this is in a loop then see if another test is going to deal with 
// this.
// If toTest is not a self-test list 
if((toTest == &g_toTest) 
// or this is the only ECC test in the list
|| !TEST_BIT(ALG_ECDSA_VALUE, *toTest) 
|| TEST_BIT(ALG_ECSCHNORR, *toTest) 
|| TEST_BIT(ALG_SM2_VALUE, *toTest)) 
{ 
result = TestECDH(alg, toTest); 
}
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CANCELED</td>
<td>test was canceled</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT
TPM_RC
TestAlgorithm(TPM_ALG_ID               alg,
              ALGORITHM_VECTOR        *toTest)
{
    TPM_ALG_ID              first = (alg == ALG_ERROR_VALUE) ? ALG_FIRST_VALUE : alg;
    TPM_ALG_ID              last = (alg == ALG_ERROR_VALUE) ? ALG_LAST_VALUE : alg;
    BOOL                    doTest = (alg != ALG_ERROR_VALUE);
    TPM_RC                  result = TPM_RC_SUCCESS;

    if(toTest == NULL)
        toTest = &g_toTest;

    // This is kind of strange. This function will either run a test of the selected
    // algorithm or just clear a bit if there is no test for the algorithm. So,
    // either this loop will be executed once for the selected algorithm or once for
    // each of the possible algorithms. If it is executed more than once ('alg' ==
    // ALG_ERROR), then no test will be run but bits will be cleared for
    // unimplemented algorithms. This was done this way so that there is only one
    // case statement with all of the algorithms. It was easier to have one case
    // statement than to have multiple ones to manage whenever an algorithm ID is
    // added.
    for(alg = first; (alg <= last); alg++)
        { 
            if('alg' was TPM_ALG_ERROR, then we will be cycling through
            // values, some of which may not be implemented. If the bit in toTest
            // happens to be set, then we could either generated an assert, or just
            // silently CLEAR it. Decided to just clear.
            if(!TEST_BIT(alg, gImplementedAlgorithms))
```
```c
{  
    CLEAR_BIT(alg, *toTest);
    continue;
}
// Process whatever is left.
// NOTE: since this switch will only be called if the algorithm is
// implemented, it is not necessary to modify this list except to comment
// out the algorithms for which there is no test
switch(alg)
{
    // Symmetric block ciphers
    #if ALG_AES
      case ALG_AES_VALUE:
      #endif // ALG_AES
    #if ALG_SM4
      // if SM4 is implemented, its test is like other block ciphers but there
      // aren't any test vectors for it yet
      //            case ALG_SM4_VALUE:
      #endif // ALG_SM4
    #if ALG_CAMELLIA
      // no test vectors for camellia
      //            case ALG_CAMELLIA_VALUE:
    #endif
    // Symmetric modes
    #if !ALG_CFB
      #error CFB is required in all TPM implementations
    #endif // !ALG_CFB
    #if ALG_CTR
    #endif // ALG_CTR
    #if ALG_OFB
    #endif // ALG_OFB
    #if ALG_CBC
    #endif // ALG_CBC
    #if ALG_ECB
    #endif // ALG_ECB
    #if ALG_HMAC
      // Clear the bit that indicates that HMAC is required because
      // HMAC is used as the basic test for all hash algorithms.
      // Testing HMAC means test the default hash
      CLEAR_BOTH(alg);
      // Testing HMAC means test the default hash
      if(doTest)
        TestHash(DEFAULT_TEST_HASH, toTest);
      else
        // If not testing, then indicate that the hash needs to be
        // tested because this uses HMAC
        SET_BOTH(DEFAULT_TEST_HASH);
```

break;  
#if ALG_SHA1  
    case ALG_SHA1_VALUE:  
#endif // ALG_SHA1  
#if ALG_SHA256  
    case ALG_SHA256_VALUE:  
#endif // ALG_SHA256  
#if ALG_SHA384  
    case ALG_SHA384_VALUE:  
#endif // ALG_SHA384  
#if ALG_SHA512  
    case ALG_SHA512_VALUE:  
#endif // ALG_SHA512  
  // if SM3 is implemented its test is like any other hash, but there  
  // aren't any test vectors yet.  
#if ALG_SM3_256  
    //            case ALG_SM3_256_VALUE:  
#endif // ALG_SM3_256  
  if (doTest)  
    result = TestHash(alg, toTest);  
    break;  
#else  
    // RSA-dependent  
#if ALG_RSA  
    case ALG_RSA_VALUE:  
        CLEAR_BOTH(alg);  
#if (doTest)  
            result = TestRsa(ALG_NULL_VALUE, toTest);  
#else  
            SET_BOTH(ALG_NULL_VALUE);  
            break;  
#endif  
        case ALG_RSASSA_VALUE:  
        case ALG_RSAES_VALUE:  
        case ALG_RSAPSS_VALUE:  
        case ALG_OAEP_VALUE:  
        case ALG_NULL_VALUE:  // used or RSADP  
#if (doTest)  
            result = TestRsa(alg, toTest);  
#else  
            SET_BOTH(ALG_NULL_VALUE);  
            break;  
#endif  
#endif // ALG_RSA  
#if ALG_KDF1_SP800_108  
    case ALG_KDF1_SP800_108_VALUE:  
#if (doTest)  
            result = TestKDFa(toTest);  
#else  
            SET_BOTH(ALG_KDF1_SP800_108_VALUE);  
            break;  
#endif  
#endif // ALG_KDF1_SP800_108  
#if ALG_ECC  
    case ALG_ECC_VALUE:  
        CLEAR_BOTH(alg);  
#if (doTest)  
            result = TestEcc(ALG_ECDH_VALUE, toTest);  
#else  
            SET_BOTH(ALG_ECDH_VALUE);  
            break;  
#endif  
    case ALG_ECDSA_VALUE:  
    case ALG_ECDH_VALUE:  
    case ALG_ECSCHNORR_VALUE:  
    case ALG_NULL_VALUE:  // used or RSADP  
#if (doTest)  
            result = TestEcc(alg, toTest);  
#else  
            SET_BOTH(ALG_ECDH_VALUE);  
            break;  
#endif  
#endif // ALG_ECC
#endif // ALG_ECC

default:
    CLEAR_BIT(alg, *toTest);
    break;

    if(result != TPM_RC_SUCCESS)
        break;

    return result;

#endif // SELF_TESTS
10.2.2 BnConvert.c

10.2.2.1 Introduction

This file contains the basic conversion functions that will convert TPM2B to/from the internal format. The internal format is a bigNum.

10.2.2.2 Includes

```c
#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.

```c
LIB_EXPORT bigNum BnFromBytes(
    bigNum bn,
    const BYTE *bytes,
    NUMBYTES nBytes
)
{
    const BYTE *pFrom; // 'p' points to the least significant bytes of source
    BYTE *pTo; // points to least significant bytes of destination
    crypt_uword_t size;
    //
    size = (bytes != NULL) ? BYTES_TO_CRYPT_WORDS(nBytes) : 0;
    // If nothing in, nothing out
    if(bn == NULL)
        return NULL;
    // make sure things fit
    pAssert(BnGetAllocated(bn) >= size);
    //
    if(size > 0)
    {
        // Clear the topmost word in case it is not filled with data
        bn->d[size - 1] = 0;
        // Moving the input bytes from the end of the list (LSB) end
        pFrom = bytes + nBytes - 1;
        // To the LS0 of the LSW of the bigNum.
        pTo = (BYTE *)bn->d;
        // For a little-endian machine, the conversion is a straight byte
        // reversal. For a big-endian machine, we have to put the words in
        // big-endian byte order
        #if BIG_ENDIAN_TPM
            {
                crypt_word_t t;
                for(t = (crypt_word_t)size - 1; t >= 0; t--)
                    bn->d[t] = SWAP_CRYPT_WORD(bn->d[t]);
            }
        #endif
    }
}
```
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

LIB_EXPORT bigNum
BnFrom2B(
    bigNum     bn,    // OUT:
    const TPM2B *a2B    // IN: number to convert
)
{
    if(a2B != NULL)
        return BnFromBytes(bn, a2B->buffer, a2B->size);
    // Make sure that the number has an initialized value rather than whatever
    // was there before
    BnSetTop(bn, 0);    // Function accepts NULL
    return NULL;
}

10.2.2.3.3  BnFromHex()

Convert a hex string into a bigNum. This is primarily used in debugging.

LIB_EXPORT bigNum
BnFromHex(
    bigNum     bn,    // OUT:
    const char *hex    // IN:
)
{
    #define FromHex(a)  ((a) - (((a) > 'a') ? ('a' + 10) : ((a) > 'A') ? ('A' - 10) : '0'))
    unsigned i;
    unsigned wordCount;
    const char *p;
    BYTE                *d = (BYTE *)&(bn->d[0]);
    // pAssert(bn && hex);  
    i = strlen(hex);
    wordCount = BYTES_TO_CRYPT_WORDS((i + 1) / 2);
    if((i == 0) || (wordCount >= BnGetAllocated(bn)))
        BnSetWord(bn, 0);
    else
        {  
            bn->d[wordCount - 1] = 0;
            p = hex + i - 1;
            for(;i > 1; i -= 2)
                {  
                    BYTE a;
                    a = FromHex(*p);
                    p--;
                    *d++ = a + (FromHex(*p) << 4);
                    p--;
                }
            if(i == 1)
                *d = FromHex(*p);
            }
        #if !BIG_ENDIAN_TPM
        for(i = 0; i < wordCount; i++)
            bn->d[i] = SWAP_CRYPT_WORD(bn->d[i]);
        
        }
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.

```c
LIB_EXPORT BOOL
BnToBytes(
    bigConst             bn,
    BYTE                *buffer,
    NUMBYTES            *size
    // This the number of bytes that are
    // available in the buffer. The result
    // should be this big.
)
{
    crypt_uword_t        requiredSize;
    BYTE                *pFrom;
    BYTE                *pTo;
    crypt_uword_t        count;
    //
    // validate inputs
    pAssert(bn && buffer && size);
    requiredSize = (BnSizeInBits(bn) + 7) / 8;
    if(requiredSize == 0)
    {
        // If the input value is 0, return a byte of zero
        *size = 1;
        *buffer = 0;
    }
    else
    {
        #if BIG_ENDIAN_TPM
        // Copy the constant input value into a modifiable value
        BN_VAR(bnL, LARGEST_NUMBER_BITS * 2);
        BnCopy(bnL, bn);
        // byte swap the words in the local value to make them little-endian
        for(count = 0; count < bnL->size; count++)
            bnL->d[count] = SWAP_CRYPT_WORD(bnL->d[count]);
        bn = (bigConst)bnL;
        endif
        if(*size == 0)
            *size = (NUMBYTES)requiredSize;
        // Byte swap the number (not words but the whole value)
        count = *size;
        // Start from the least significant word and offset to the most significant
        // byte which is in some high word
        pFrom = (BYTE *)&bn->d[0] + requiredSize - 1;
        pTo = buffer;
        // If the number of output bytes is larger than the number bytes required
        // for the input number, pad with zeros
        for(count = *size; count > requiredSize; count--)
            *pTo++ = 0;
    }
    return bn;
}
```
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 BnTo2B{ 
157       bigConst         bn,   // IN:
158       TPM2B           *a2B,  // OUT:
159       NUMBYTES        size  // IN: the desired size
160       }
161 
162    // Set the output size
163    if(bn && a2B)
164    {
165       a2B->size = size;
166       return BnToBytes(bn, a2B->buffer, &a2B->size);
167    }
168    return FALSE;
169 }
170 #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

```c
#include "Tpm.h"
```

A constant value of zero as a stand in for NULL bigNum values

```c
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no carry out</td>
</tr>
<tr>
<td>1</td>
<td>carry out</td>
</tr>
</tbody>
</table>

```c
static BOOL AddSame(
    crypt_uword_t *result,
    const crypt_uword_t *op1,
    const crypt_uword_t *op2,
    int count
){
    int carry = 0;
    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

static int
CarryProp(crypt_uword_t *result,
           const crypt_uword_t *op,
           int count,
           int carry)
{
    for(; count; count--)
    {
        carry = ((*result++ = *op++ + carry) == 0) & carry;
        return carry;
    }
}

static void
CarryResolve(bigNum result,
              int stop,
              int carry)
{
    if(carry)
    {
        pAssert((unsigned)stop < result->allocated);
        result->d[stop++] = 1;
    }
    BnSetTop(result, stop);
}

10.2.3.3.3 BnAdd()

This function adds two bigNum values. This function always returns TRUE.

LIB_EXPORT BOOL
BnAdd(bigNum result,
      bigConst op1,
      bigConst op2)
{
    crypt_uword_t stop;
    int carry;
    const bignum_t *n1 = op1;
    const bignum_t *n2 = op2;
//
if (n2->size > n1->size) {
    n1 = op2;
    n2 = op1;
}
pAssert(result->allocated >= n1->size);
stop = MIN(n1->size, n2->allocated);
carry = AddSame(result->d, n1->d, n2->d, stop);
if (n1->size > stop)
    carry = CarryProp(&result->d[stop], &n1->d[stop], n1->size - stop, carry);
CarryResolve(result, n1->size, carry);
return TRUE;

10.2.3.3.4 BnAddWord()

This function adds a word value to a bigNum. This function always returns TRUE.

LIB_EXPORT BOOL BnAddWord(
    bigNum result,
    bigConst op,
    crypt_uword_t word
) {
    int carry;
    //
carry = (result->d[0] = op->d[0] + word) < word;
carry = CarryProp(&result->d[1], &op->d[1], op->size - 1, carry);
CarryResolve(result, op->size, carry);
return TRUE;
}

10.2.3.3.5 SubSame()

This function subtracts two values that have the same size.

static int SubSame(
    crypt_uword_t *result,
    const crypt_uword_t *op1,
    const crypt_uword_t *op2,
    int count
) {
    int borrow = 0;
    int i;
    for(i = 0; i < count; i++)
    {
        crypt_uword_t a = op1[i];
crypt_uword_t diff = a - op2[i];
        result[i] = diff - borrow;
        // generate | propagate
        borrow = (diff > a) | ((diff == 0) & borrow);
    }
    return borrow;
}
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 \texttt{bigNum} computations are on large positive numbers (primes) or on fields.

```c
114 static int
115 BorrowProp(
116    crypt_uword_t *result,
117    const crypt_uword_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 \texttt{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.

```c
126 LIB_EXPORT BOOL
127 BnSub(
128    bigNum result,
129    bigConst op1,
130    bigConst op2
131 )
132 {
133    int borrow;
134    crypt_uword_t stop = MIN(op1->size, op2->allocated);
135    // Make sure that op2 is not obviously larger than op1
136    pAssert(op1->size >= op2->size);
137    borrow = SubSame(result->d, op1->d, op2->d, stop);
138    if(op1->size > stop)
139        borrow = BorrowProp(&result->d[stop], &op1->d[stop], op1->size - stop,
140                 borrow);
141    pAssert(!borrow);
142    BnSetTop(result, op1->size);
143    return TRUE;
144 }
```

10.2.3.3.8 BnSubWord()

This function subtracts a word value from a \texttt{bigNum}. This function always returns TRUE.

```c
146 LIB_EXPORT BOOL
147 BnSubWord(
148    bigNum result,
149    bigConst op,
150    crypt_uword_t word
151 )
152 {
153    int borrow;
154    //
155    pAssert(op->size > 1 || word <= op->d[0]);
156    borrow = word > op->d[0];
```
result->d[0] = op->d[0] - word;
borrow = BorrowProp(&result->d[1], &op->d[1], op->size - 1, borrow);
pAssert(!borrow);
BnSetTop(result, op->size);
return TRUE;
}

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).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 0</td>
<td>op1 is less than op2</td>
</tr>
<tr>
<td>0</td>
<td>op1 is equal to op2</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>op1 is greater than op2</td>
</tr>
</tbody>
</table>

LIB_EXPORT int BnUnsignedCmp(
bigConst op1,
bigConst op2)
{
int retVal;
int diff;
int i;

// pAssert((op1 != NULL) && (op2 != NULL));
retVal = op1->size - op2->size;
if(retVal == 0)
{
    for(i = (int)(op1->size - 1); i >= 0; i--)
    {
        diff = (op1->d[i] < op2->d[i]) ? -1 : (op1->d[i] != op2->d[i]);
        retVal = retVal == 0 ? diff : retVal;
    }
}
else
    retVal = (retVal < 0) ? -1 : 1;
return retVal;
}

10.2.3.3.10 BnUnsignedCmpWord()

Compare a bigNum to a crypt_uword_t.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>op1 is less than word</td>
</tr>
<tr>
<td>0</td>
<td>op1 is equal to word</td>
</tr>
<tr>
<td>1</td>
<td>op1 is greater than word</td>
</tr>
</tbody>
</table>

LIB_EXPORT int BnUnsignedCmpWord(
bigConst op1,
crypt_uword_t word)
{

if(op1->size > 1)  
    return 1;  
else if(op1->size == 1)  
    return (op1->d[0] < word) ? -1 : (op1->d[0] > word);  
else // op1 is zero  
    // equal if word is zero  
    return (word == 0) ? 0 : -1;  
}

10.2.3.3.11 BnModWord()  

This function does modular division of a big number when the modulus is a word value.

LIB_EXPORT crypt_word_t  
BnModWord(  
    bigConst numerator,  
    crypt_word_t modulus  
)  
{  
    BN_MAX(remainder);  
    BN_VAR(mod, RADIX_BITS);  
    //  
    mod->d[0] = modulus;  
    mod->size = (modulus != 0);  
    BnDiv(NULL, remainder, numerator, mod);  
    return remainder->d[0];  
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>the word was zero</td>
</tr>
<tr>
<td>n</td>
<td>the bit number of the most significant bit in the word</td>
</tr>
</tbody>
</table>

LIB_EXPORT int  
Msb(  
    crypt_uword_t word  
)  
{  
    int retVal = -1;  
    //  
    #if RADIX_BITS == 64  
    if(word & 0xffffffff00000000) { retVal += 32; word >>= 32; }  
    #endif  
    if(word & 0xffff0000) { retVal += 16; word >>= 16; }  
    if(word & 0x0000ff00) { retVal += 8; word >>= 8; }  
    if(word & 0x000000f0) { retVal += 4; word >>= 4; }  
    if(word & 0x0000000c) { retVal += 2; word >>= 2; }  
    if(word & 0x00000002) { retVal += 1; word >>= 1; }  
    return retVal + (int)word;  
}

10.2.3.3.13 BnMsb()  

This function returns the number of the MSb of a bigNum value.
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>the word was zero or bn was NULL</td>
</tr>
<tr>
<td>n</td>
<td>the bit number of the most significant bit in the word</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT int BnMsb(bigConst bn)
{
    // If the value is NULL, or the size is zero then treat as zero and return -1
    if(bn != NULL && bn->size > 0)
    {
        int retVal = Msb(bn->d[bn->size - 1]);
        retVal += (bn->size - 1) * RADIX_BITS;
        return retVal;
    }
    else
    return -1;
}
```

**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.

```
LIB_EXPORT unsigned BnSizeInBits(bigConst n)
{
    int bits = BnMsb(n) + 1;
    // return bits < 0? 0 : (unsigned)bits;
}
```

**10.2.3.3.15 BnSetWord()**

Change the value of a bigNum_t to a word value.

```
LIB_EXPORT bigNum BnSetWord(bigNum n, crypt_uword_t w)
{
    if(n != NULL)
    {
        pAssert(n->allocated > 1);
        n->d[0] = w;
        BnSetTop(n, (w != 0) ? 1 : 0);
    }
    return n;
}
```

**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.

```
LIB_EXPORT BOOL
```
271  \texttt{BnSetBit}\{
272      \texttt{bigNum} \quad \texttt{bn}, \quad // \texttt{IN/OUT}: big number to modify
273      \texttt{unsigned int} \quad \texttt{bitNum} \quad // \texttt{IN}: Bit number to SET
274  \}
275  \{
276      \texttt{crypt_uword_t} \quad \texttt{offset} = \texttt{bitNum} / \texttt{RADIX\_BITS};
277      \texttt{pAssert}(\texttt{bn->allocated * RADIX\_BITS} >= \texttt{bitNum});
278      // Grow the number if necessary to set the bit.
279      \texttt{while}(\texttt{bn->size} <= \texttt{offset})
280        \texttt{bn->d[bn->size++]} = 0;
281      \texttt{bn->d[offset]} |= (1 << \texttt{RADIX\_MOD(bitNum)});
282      \texttt{return} \texttt{TRUE};
283  \}

10.2.3.3.17 \texttt{BnTestBit()}

This function is used to check to see if a bit is SET in a \texttt{bignum\_t}. The 0th bit is the LSb of \texttt{d[0]}.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>the bit is set</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>the bit is not set or the number is out of range</td>
</tr>
</tbody>
</table>

LIB\_EXPORT \texttt{BOOL}

\texttt{BnTestBit()}

\texttt{bigNum} \quad \texttt{bn}, \quad // \texttt{IN}: number to check
\texttt{unsigned int} \quad \texttt{bitNum} \quad // \texttt{IN}: bit to test
\{
\texttt{crypt_uword_t} \quad \texttt{offset} = \texttt{RADIX\_DIV(bitNum)};
\texttt{//}
\texttt{if}(\texttt{bn->size} > \texttt{offset})
\texttt{\quad return} ((\texttt{bn->d[offset]} & (((\texttt{crypt_uword_t}1) << \texttt{RADIX\_MOD(bitNum)})) != 0);
\texttt{else}
\texttt{\quad return} \texttt{FALSE};
\}

10.2.3.3.18 \texttt{BnMaskBits()}

This function is used to mask off high order bits of a big number. The returned value will have no more than \texttt{maskBit} bits set.

\textbf{NOTE:} There is a requirement that unused words of a \texttt{bignum\_t} are set to zero.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>result masked</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>the input was not as large as the mask</td>
</tr>
</tbody>
</table>

LIB\_EXPORT \texttt{BOOL}

\texttt{BnMaskBits()}

\texttt{bigNum} \quad \texttt{bn}, \quad // \texttt{IN/OUT}: number to mask
\texttt{crypt_uword_t} \quad \texttt{maskBit} \quad // \texttt{IN}: the bit number for the mask.
\{
\texttt{crypt_uword_t} \quad \texttt{finalSize};
\texttt{BOOL} \quad \texttt{retVal};
\texttt{finalSize} = \texttt{BITS\_TO\_CRYPT\_WORDS(maskBit)};
\texttt{retVal} = (\texttt{finalSize} <= \texttt{bn->allocated});
if (retVal && (finalSize > 0))
{
  crypt_uword_t   mask;
  mask = ~((crypt_uword_t)0) >> RADIX_MOD(maskBit);
  bn->d[finalSize - 1] &= mask;
}
BnSetTop(bn, finalSize);
return retVal;

10.2.3.3.19 BnShiftRight()

This function will shift a bigNum to the right by the shiftAmount. This function always returns TRUE.

LIB_EXPORT BOOL BnShiftRight(
  bigNum           result,
  bigConst         toShift,
  uint32_t         shiftAmount
)
{
  uint32_t         offset = (shiftAmount >> RADIX_LOG2);
  uint32_t         i;
  uint32_t         shiftIn;
  crypt_uword_t    finalSize;
  // shiftAmount = shiftAmount & RADIX_MASK;
  shiftIn = RADIX_BITS - shiftAmount;
  // The end size is toShift->size - offset less one additional
  // word if the shiftAmount would make the upper word == 0
  if(toShift->size > offset)
  {
    finalSize = toShift->size - offset;
    finalSize -= (toShift->d[toShift->size - 1] >> shiftAmount) == 0 ? 1 : 0;
  }
  else
    finalSize = 0;
  pAssert(finalSize <= result->allocated);
  if(finalSize != 0)
  {
    for(i = 0; i < finalSize; i++)
    {
      result->d[i] = (toShift->d[i + offset] >> shiftAmount)
      | (toShift->d[i + offset + 1] << shiftIn);
    }
    if(offset == 0)
      result->d[i] = toShift->d[i] >> shiftAmount;
  }
  BnSetTop(result, finalSize);
  return TRUE;

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 an the high order 7 bits are masked off (CLEAR).
LIB_EXPORT BOOL
BnGetRandomBits(
    bigNum           n,
    size_t           bits,
    RAND_STATE      *rand
) {
    // Since this could be used for ECC key generation using the extra bits method,
    // make sure that the value is large enough
    TPM2B_TYPE(LARGEST, LARGEST_NUMBER + 8);
    TPM2B_LARGEST    large;
    //
    large.b.size = (UINT16)BITS_TO_BYTES(bits);
    if(DRBG_Generate(rand, large.t.buffer, large.t.size) == large.t.size) {
        if(BnFrom2B(n, &large.b) != NULL) {
            if(BnMaskBits(n, bits))
                return TRUE;
        }
    }
    return FALSE;
}

10.2.3.3.21  BnGenerateRandomInRange()

This function is used to generate a random number r in the range 1 <= r < 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.

LIB_EXPORT BOOL
BnGenerateRandomInRange(
    bigNum           dest,
    bigConst         limit,
    RAND_STATE      *rand
) {
    size_t   bits = BnSizeInBits(limit);
    //
    if(bits < 2) {
        BnSetWord(dest, 0);
        return FALSE;
    } else {
        while(BnGetRandomBits(dest, bits, rand)
            && (BnEqualZero(dest) || (BnUnsignedCmp(dest, limit) >= 0)));
    }
return !g_inFailureMode;
}
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

```c
#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.

```c
LIB_EXPORT bigNum
BnSetTop(
    bigNum bn,        // IN/OUT: number to clean
    crypt_uword_t top  // IN: the new top
)
{
    if (bn != NULL)
    {
        pAssert(top <= bn->allocated);
        // If forcing the size to be decreased, make sure that the words being
        // discarded are being set to 0
        while(bn->size > top)
        {
            bn->d[--bn->size] = 0;
            bn->size = top;
            // Now make sure that the words that are left are 'normalized' (no high-order
            // words of zero.
            while((bn->size > 0) && (bn->d[bn->size - 1] == 0))
            {
                bn->size -= 1;
            }
        }
        return bn;
    }
}
```

10.2.4.3.2 BnClearTop()

This function will make sure that all unused words are zero.

```c
LIBEXPORT bigNum
BnClearTop(
    bigNum bn
)
{
    crypt_uword_t i;
    //
    if (bn != NULL)
    {
        for(i = bn->size; i < bn->allocated; i++)
            bn->d[i] = 0;
        while((bn->size > 0) && (bn->d[bn->size] == 0))
            bn->size -= 1;
        return bn;
    }
}
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.

LIB_EXPORT bigNum
BnInitializeWord(
bigNum bn,          // IN:
crypt_uword_t allocated, // IN:
crypt_uword_t word    // IN:
);
{
    bn->allocated = allocated;
    bn->size = (word != 0);
    bn->d[0] = word;
    while(allocated > 1)
        bn->d[--allocated] = 0;
    return bn;
}

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.

LIB_EXPORT bigNum
BnInit(
bigNum bn,          // IN:
crypt_uword_t allocated
);
{
    if(bn != NULL)
        {
            bn->allocated = allocated;
            bn->size = 0;
            while(allocated != 0)
                bn->d[--allocated] = 0;
            return bn;
        }
}

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.

LIB_EXPORT BOOL
BnCopy(
bigNum out,
bigConst in
);
{
    if(in == out)
        BnSetTop(out, BnGetSize(out));
    else if(out != NULL)
        {
            if(in != NULL)
                {
                    unsigned int i;
                    pAssert(BnGetAllocated(out) >= BnGetSize(in));
                    for(i = 0; i < BnGetSize(in); i++)
                        out->d[i] = in->d[i];
                }
            }
84        BnSetTop(out, BnGetSize(in));
85    }
86    else
87    BnSetTop(out, 0);
88    }
89    return TRUE;
90 }
91 #if ALG_ECC

10.2.4.3.6 BnPointCopy()

Function to copy a bn point.

92 LIB_EXPORT BOOL
93 BnPointCopy(
94    bigPoint                pOut,
95    pointConst             pIn
96 }
97 {
98    return BnCopy(pOut->x, pIn->x)
99        && BnCopy(pOut->y, pIn->y)
100        && BnCopy(pOut->z, pIn->z);
101 }

10.2.4.3.7 BnInitializePoint()

This function is used to initialize a point structure with the addresses of the coordinates.

102 LIB_EXPORT bn_point_t *
103 BnInitializePoint(
104    bigPoint                p,   // OUT: structure to receive pointers
105    bigNum                 x,   // IN: x coordinate
106    bigNum                 y,   // IN: y coordinate
107    bigNum                 z   // IN: x coordinate
108 }
109 {
110    p->x = x;
111    p->y = y;
112    p->z = z;
113    BnSetWord(z, 1);
114    return p;
115 }
116 #endif // ALG_ECC
10.2.5 CryptCmac.c

10.2.5.1 Introduction

This file contains the implementation of the message authentication codes based on a symmetric block cipher. These functions only use the single block encryption functions of the selected symmetric cryptographic library.

10.2.5.2 Includes, Defines, and Typedefs

```c
#define _CRYPT_HASH_C_
#include "Tpm.h"
#include "CryptSym.h"
#if ALG_CMAC
```

10.2.5.3 Functions

10.2.5.3.1 CryptCmacStart()

This is the function to start the CMAC sequence operation. It initializes the dispatch functions for the data and end operations for CMAC and initializes the parameters that are used for the processing of data, including the key, key size and block cipher algorithm.

```c
UINT16 CryptCmacStart(
    SMAC_STATE          *state,
    TPMU_PUBLIC_PARMS   *keyParms,
    TPM_ALG_ID           macAlg,
    TPM2B               *key
)
{
    tpmCmacState_t      *cState = &state->state.cmac;
    TPM_SYM_DEF_OBJECT *def = &keyParms->symDetail.sym;
    // if (macAlg != TPM_ALG_CMAC)
    //    return 0;
    // set up the encryption algorithm and parameters
    cState->symAlg = def->algorithm;
    cState->keySizeBits = def->keyBits.sym;
    cState->iv.t.size = CryptGetSymmetricBlockSize(def->algorithm,
                                                   def->keyBits.sym);
    MemoryCopy2B(&cState->symKey.b, key, sizeof(cState->symKey.t.buffer));
    // Set up the dispatch methods for the CMAC
    state->smacMethods.data = CryptCmacData;
    state->smacMethods.end = CryptCmacEnd;
    return cState->iv.t.size;
}
```

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.
void CryptCmacData(
    SMAC_STATES *state, 
    UINT32 size, 
    const BYTE *buffer
)
{
    tpmCmacState_t *cmacState = &state->cmac;
    TPM_ALG_ID algorithm = cmacState->symAlg;
    BYTE *key = cmacState->symKey.t.buffer;
    UINT16 keySizeInBits = cmacState->keySizeBits;
    tpmCryptKeySchedule_t keySchedule;
    TpmCryptSetSymKeyCall_t encrypt;
    
    // SELECT(ENCRYPT);
    while(size > 0)
    {
        if(cmacState->bcount == cmacState->iv.t.size)
        {
            ENCRYPT(&keySchedule, cmacState->iv.t.buffer, cmacState->iv.t.buffer);
            cmacState->bcount = 0;
        }
        for(;(size > 0) && (cmacState->bcount < cmacState->iv.t.size);
            size--, cmacState->bcount++)
        {
            cmacState->iv.t.buffer[cmacState->bcount] ^= *buffer++;
        }
    }
}

UINT16 CryptCmacEnd(
    SMAC_STATES *state, 
    UINT32 outSize, 
    BYTE *outBuffer
)
{
    tpmCmacState_t *cState = &state->cmac;
    TPM_ALG_ID algorithm = cState->symAlg;
    BYTE *key = cState->symKey.t.buffer;
    UINT16 keySizeInBits = cState->keySizeBits;
    tpmCryptKeySchedule_t keySchedule;
    TpmCryptSetSymKeyCall_t encrypt;
    TPM2B_IV subkey = {{0, {0}}};
    BOOL xorVal;
    UINT16 i;

    subkey.t.size = cState->iv.t.size;
    // Encrypt a block of zero
    SELECT(ENCRYPT);
    ENCRYPT(&keySchedule, subkey.t.buffer, subkey.t.buffer);
    
    // shift left by 1 and XOR with 0x0...87 if the MSb was 0
    xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0x87 :
        ShiftLeft(&subkey.b);
    subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
    // this is a sanity check to make sure that the algorithm is working properly.

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.
// remove this check when debug is done
pAssert(cState->bcount <= cState->iv.t.size);

// If the buffer is full then no need to compute subkey 2.
if (cState->bcount < cState->iv.t.size)
{
    // Pad the data
cState->iv.t.buffer[cState->bcount++] ^= 0x80;
    // The rest of the data is a pad of zero which would simply be XORed
    // with the iv value so nothing to do...
    // Now compute K2
    xorVal = ((subkey.t.buffer[0] & 0x80) == 0) ? 0 : 0x87;
    ShiftLeft(&subkey.b);
    subkey.t.buffer[subkey.t.size - 1] ^= xorVal;
}

// XOR the subkey into the IV
for(i = 0; i < subkey.t.size; i++)
    cState->iv.t.buffer[i] ^= subkey.t.buffer[i];

ENCRYPT(&keySchedule, cState->iv.t.buffer, cState->iv.t.buffer);
i = (UINT16)MIN(cState->iv.t.size, outSize);
MemoryCopy(outBuffer, cState->iv.t.buffer, i);

return i;

#endif
10.2.6 CryptUtil.c

10.2.6.1 Introduction

This module contains the interfaces to the CryptoEngine() and provides miscellaneous cryptographic functions in support of the TPM.

10.2.6.2 Includes

```c
#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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_HASH</td>
<td>not a valid hash</td>
</tr>
</tbody>
</table>

```c
static TPM_RC CryptHmacSign(
    TPMT_SIGNATURE *signature, // OUT: signature
    OBJECT *signKey,          // IN: HMAC key sign the hash
    TPM2B_DIGEST *hashData    // IN: hash to be signed
)
{
    HMAC_STATE hmacState;
    UINT32  digestSize;

    digestSize = CryptHmacStart2B(&hmacState, signature->signature.any.hashAlg,
                                  &signKey->sensitive.sensitive.bits.b);
    CryptDigestUpdate2B(&hmacState.hashState, &hashData->b);
    CryptHmacEnd(&hmacState, digestSize,
                 (BYTE *)signature->signature.hmac.digest);
    return TPM_RC_SUCCESS;
}
```

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SCHEMA</td>
<td>not the proper scheme for this key type</td>
</tr>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>if invalid input or signature is not genuine</td>
</tr>
</tbody>
</table>

```c
static TPM_RC CryptHMACVerifySignature(
    OBJECT              *signKey, // IN: HMAC key signed the hash
    TPM2B_DIGEST        *hashData, // IN: digest being verified
    TPMT_SIGNATURE      *signature // IN: signature to be verified
)
{
```
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 TMP_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  CryptHmacSign(&test, signKey, hashData);
46  // Compare digest
47  if(!MemoryEqual(&test.signature.hmac.digest,
48                  &signature->signature.hmac.digest,
49                  CryptHashGetDigestSize(signature->signature.any.hashAlg)))
50      return TPM_RC_SIGNATURE;
51  return TPM_RC_SUCCESS;
52 }
10.2.6.3.3 CryptGenerateKeyedHash()

This function creates a keyedHash object.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>cannot get values from random number generator</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>sensitive data size is larger than allowed for the scheme</td>
</tr>
</tbody>
</table>

static TPM_RC
CryptGenerateKeyedHash(
  TPMT_PUBLIC             *publicArea,  // IN/OUT: the public area template
  TPMT_SENSITIVE          *sensitive,   // IN: sensitive area
  TPMS_SENSITIVE_CREATE   *sensitiveCreate, // IN: sensitive creation data
  RAND_STATE              *rand         // IN: “entropy” source
) {
  TPMT_KEYEDHASH_SCHEME   *scheme;
  TPM_ALG_ID              hashAlg;
  UINT16                   hashBlockSize;
  UINT16                   digestSize;

  scheme = &publicArea->parameters.keyedHashDetail.scheme;

  if(publicArea->type != ALG_KEYEDHASH_VALUE)
    return TPM_RC_FAILURE;

  // Pick the limiting hash algorithm
  if(scheme->scheme == ALG_NULL_VALUE)
    hashAlg = publicArea->nameAlg;
  else if(scheme->scheme == ALG_XOR_VALUE)
```c
hashAlg = scheme->details.xor.hashAlg;
else
    hashAlg = scheme->details.hmac.hashAlg;
hashBlockSize = CryptHashGetBlockSize(hashAlg);
digestSize = CryptHashGetDigestSize(hashAlg);

// if this is a signing or a decryption key, then the limit
// for the data size is the block size of the hash. This limit
// is set because larger values have lower entropy because of the
// HMAC function. The lower limit is 1/2 the size of the digest
// If the user provided the key, check that it is a proper size
if(sensitiveCreate->data.t.size != 0)
{
    if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
        || IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
    {
        if(sensitiveCreate->data.t.size > hashBlockSize)
            #if 0 // May make this a FIPS-mode requirement
                if(sensitiveCreate->data.t.size < (digestSize / 2))
                    return TPM_RC_SIZE;
            #endif
    }
    // If this is a data blob, then anything that will get past the unmarshaling
    // is OK
    MemoryCopy2B(&sensitive->敏感.bits.b, &sensitiveCreate->data.b,
        sizeof(sensitive->敏感.bits.t.buffer));
}
else
{
    // The TPM is going to generate the data so set the size to be the
    // size of the digest of the algorithm
    sensitive->敏感.bits.t.size =
        DRBG_Generate(rand, sensitive->敏感.bits.t.buffer, digestSize);
    if(sensitive->敏感.bits.t.size == 0)
        return (g_inFailureMode) ? TPM_RC_FAILURE : TPM_RC_NO_RESULT;
}
return TPM_RC_SUCCESS;
```

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 ECDAA. ECDAA can be used to do things like U-Prove.

```c
BOOL CryptIsSchemeAnonymous(
    TPM_ALG_ID scheme // IN: the scheme algorithm to test
)
{
    return scheme == ALG_ECDAA_VALUE;
}
```

10.2.6.4 **Symmetric Functions**

10.2.6.4.1 **ParmDecryptSym()**

This function performs parameter decryption using symmetric block cipher.

```c
void ParmDecryptSym()
```
ParmEncryptSym()

This function performs parameter encryption using symmetric block cipher.

```
void ParmEncryptSym(
    TPM_ALG_ID symAlg,   // IN: symmetric algorithm
    TPM_ALG_ID hash,     // IN: hash algorithm for KDFa
    UINT16 keySizeInBits, // IN: symmetric key size in bits
    TPM2B *key,          // IN: KDF HMAC key
    TPM2B *nonceCaller,  // IN: nonce caller
    TPM2B *nonceTpm,     // IN: nonce TPM
    UINT32 dataSize,     // IN: size of parameter buffer
    BYTE *data           // OUT: buffer to be encrypted
)
{
    // KDF output buffer
    // It contains parameters for the CFB encryption
    BYTE symParmString[MAX_SYM_KEY_BYTES + MAX_SYM_BLOCK_SIZE];
    UINT16 keySize = (keySizeInBits + 7) / 8;
    TPM2B_IV iv;

    iv.t.size = CryptGetSymmetricBlockSize(symAlg, keySizeInBits);
    if(iv.t.size > 0)
    {
        // Generate key and iv
        CryptKDFa(hash, key, CFB_KEY, nonceTpm, nonceCaller, noncesize, symParmString, NULL, FALSE);
        MemoryCopy(iv.t.buffer, &symParmString[keySize], iv.t.size);
        CryptSymmetricDecrypt(data, symAlg, keySizeInBits, symParmString, &iv, ALG_CFB_VALUE, dataSize, data);
        return;
    }
}
```
187  
188  
189  
190  
191  
192  
193  
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>cannot get a random value</td>
</tr>
<tr>
<td>TPM_RC_KEY_SIZE</td>
<td>key size in the public area does not match the size in the sensitive creation area</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>provided key value is not allowed</td>
</tr>
</tbody>
</table>

```
static TPM_RC
CryptGenerateKeySymmetric(
    TPMT_PUBLIC    *publicArea, // IN/OUT: The public area template
    TPMT_SENSITIVE *sensitive, // OUT: sensitive area
    TPMS_SENSITIVE_CREATE *sensitiveCreate, // IN: sensitive creation data
    RAND_STATE      *rand      // IN: the "entropy" source for
)
{
    UINT16 keyBits = publicArea->parameters.symDetail.sym.keyBits.sym;
    TPM_RC result;
    // only do multiples of RADIX_BITS
    if((keyBits % RADIX_BITS) != 0)
        return TPM_RC_KEY_SIZE;
    // If this is not a new key, then the provided key data must be the right size
    if(sensitiveCreate->data.t.size != 0)
        {  
            result = CryptSymKeyValidate(&publicArea->parameters.symDetail.sym,  
                (TPM2B_SYM_KEY *)&sensitiveCreate->data);  
            if(result == TPM_RC_SUCCESS)  
                MemoryCopy2B(&sensitive->sym.b, &sensitiveCreate->data.b,  
                    sizeof(sensitive->sym.t.buffer));  
        }
    #if ALG_TDES
    else if(publicArea->parameters.symDetail.sym.algorithm == ALG_TDES_VALUE)
        {  
            result = CryptGenerateKeyDes(publicArea, sensitive, rand);
        }
    #endif
    else
        {  
            sensitive->sym.t.size =  
                DRBG_Generate(rand, sensitive->sym.t.buffer,  
                    BITS_TO_BYTES(keyBits));  
            if(g_inFailureMode)
                result = TPM_RC_FAILURE;
            else if(sensitive->sym.t.size == 0)
                result = TPM_RC_NO_RESULT;
            else
                result = TPM_RC_SUCCESS;
        }
```
10.2.6.4 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.

```c
void CryptXORObfuscation(
    TPM_ALG_ID       hash,       // IN: hash algorithm for KDF
    TPM2B           *key,       // IN: KDF key
    TPM2B           *contextU,   // IN: contextU
    TPM2B           *contextV,   // IN: contextV
    UINT32           dataSize,   // IN: size of data buffer
    BYTE            *data       // IN/OUT: data to be XORed in place
)
{
    BYTE    mask[MAX_DIGEST_SIZE];   // Allocate a digest sized buffer
    BYTE    *pm;
    UINT32   i;
    UINT32   counter = 0;
    UINT16   hLen = CryptHashGetDigestSize(hash);
    UINT32   requestSize = dataSize * 8;
    INT32    remainBytes = (INT32)dataSize;

    pAssert((key != NULL) && (data != NULL) && (hLen != 0));
    // Call KDFa to generate XOR mask
    for(; remainBytes > 0; remainBytes -= hLen)
    {
        // Make a call to KDFa to get next iteration
        CryptKDFa(hash, key, XOR_KEY, contextU, contextV,
                   requestSize, mask, &counter, TRUE);
        // XOR next piece of the data
        pm = mask;
        for(i = hLen < remainBytes ? hLen : remainBytes; i > 0; i--)
            *data++ ^= *pm++;
    }
    return;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>initializations succeeded</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>initialization failed and caller should place the TPM into Failure Mode</td>
</tr>
</tbody>
</table>

```c
BOOL CryptInit(
    void
)
```
276 } )
277 {
278     BOOL ok;
279     // Initialize the vector of implemented algorithms
280     AlgorithmGetImplementedVector(&g_implementedAlgorithms);
281     // Indicate that all test are necessary
282     CryptInitializeToTest();
283     // Do any library initializations that are necessary. If any fails,
284     // the caller should go into failure mode;
285     ok = SupportLibInit();
286     ok = ok && CryptSymInit();
287     ok = ok && CryptRandInit();
288     ok = ok && CryptHashInit();
289     #if ALG_RSA
290     ok = ok && CryptRsaInit();
291     #endif // ALG_RSA
292     #if ALG_ECC
293     ok = ok && CryptEccInit();
294     #endif // ALG_ECC
295     return ok;
296 }

10.2.6.5.2 CryptStartup()

This function is called by TPM2_Startup() to initialize the functions in this cryptographic library and in the
provided CryptoLibrary(). This function and CryptUtilInit() are both provided so that the implementation
may move the initialization around to get the best interaction.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>startup succeeded</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>startup failed and caller should place the TPM into Failure Mode</td>
</tr>
</tbody>
</table>

BOOL CryptStartup(STARTUP_TYPE type) // IN: the startup type
{
    BOOL ok;
    NOT_REFERENCED(type);
    ok = CryptSymStartup() && CryptRandStartup() && CryptHashStartup();
    #if ALG_RSA
    && CryptRsaStartup();
    #endif // ALG_RSA
    #if ALG_ECC
    && CryptEccStartup();
    #endif // ALG_ECC
    if (OK && (type != SU_RESTART) && (type != SU_RESUME))
    { // If the shutdown was orderly, then the values recovered from NV will
    // be OK to use.
    // Get a new random commit nonce
    gr.commitNonce.t.size = sizeof(gr.commitNonce.t.buffer);
    CryptRandomGenerate(gr.commitNonce.t.size, gr.commitNonce.t.buffer);
    // Reset the counter and commit array
    gr.commitCounter = 0;
    MemorySet(gr.commitArray, 0, sizeof(gr.commitArray));
    }
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>if it is an asymmetric algorithm</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>if it is not an asymmetric algorithm</td>
</tr>
</tbody>
</table>

```c
BOOL CryptIsAsymAlgorithm(
    TPM_ALG_ID       algID       // IN: algorithm ID
) {
    switch(algID) {
        #if ALG_RSA
        case ALG_RSA_VALUE: 
        #endif
        #if ALG_ECC
        case ALG_ECC_VALUE:
        #endif
        return TRUE;
        break;
        default:
            break;
        }
    return FALSE;
}
```

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().

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ATTRIBUTES</td>
<td>keyHandle does not reference a valid decryption key</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>invalid ECC key (public point is not on the curve)</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>RSA key with an unsupported padding scheme</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>numeric value of the data to be decrypted is greater than the RSA key modulus</td>
</tr>
</tbody>
</table>

```c
```
CryptSecretEncrypt(
    OBJECT *encryptKey, // IN: encryption key object
    const TPM2B *label, // IN: a null-terminated string as L
    TPM2B_DATA *data, // OUT: secret value
    TPM2B_ENCRYPTED_SECRET *secret // OUT: secret structure
)

// 
// If(data == NULL || secret == NULL)
return TPM_RC_FAILURE;
// 
// The output secret value has the size of the digest produced by the nameAlg.
data->t.size = CryptHashGetDigestSize(encryptKey->publicArea.nameAlg);
// The encryption scheme is OAEP using the nameAlg of the encryption key.
scheme.scheme = ALG_OAEP_VALUE;
scheme.details.anySig.hashAlg = encryptKey->publicArea.nameAlg;
if(!IS_ATTRIBUTE(encryptKey->publicArea.objectAttributes, TPMA_OBJECT, decrypt))
    return TPM_RC_ATTRIBUTES;
switch(encryptKey->publicArea.type)
{
#if ALG_RSA
    case ALG_RSA_VALUE:
        // Create secret data from RNG
        CryptRandomGenerate(data->t.size, data->t.buffer);
        // Encrypt the data by RSA OAEP into encrypted secret
        result = CryptRsaEncrypt((TPM2B_PUBLIC_KEY_RSA *)secret, &data->b,
            encryptKey, &scheme, label, NULL);
        break;
#endif // ALG_RSA
    #if ALG_ECC
    case ALG_ECC_VALUE:
        TPMS_ECC_POINT eccPublic;
        TPM2B_ECC_PARAMETER eccPrivate;
        TPMS_ECC_POINT eccSecret;
        BYTE *buffer = secret->t.secret;
        // Need to make sure that the public point of the key is on the
        // curve defined by the key.
        if(!CryptEccIsPointOnCurve(
            encryptKey->publicArea.parameters.eccDetail.curveID,
            &encryptKey->publicArea.unique.ecc))
            result = TPM_RC_KEY;
        else
            { // Call crypto engine to create an auxiliary ECC key
                // We assume crypt engine initialization should always success.
                // Otherwise, TPM should go to failure mode.
                CryptEccNewKeyPair(&eccPublic, &eccPrivate,
                    encryptKey->publicArea.parameters.eccDetail.curveID);
                // Marshal ECC public to secret structure. This will be used by the
                // recipient to decrypt the secret with their private key.
                secret->t.size = TPMS_ECC_POINT_Marshal(&eccPublic, &buffer, NULL);
                // Compute ECDH shared secret which is R = [d]Q where d is the
                // private part of the ephemeral key and Q is the public part of a
                // TPM key. TPM_RC_KEY error return from CryptComputeECDHSecret
                // because the auxiliary ECC key is just created according to the
// parameters of input ECC encrypt key.
if (CryptEccPointMultiply(&eccSecret,
    encryptKey->publicArea.parameters.eccDetail.curveID,
    &encryptKey->publicArea.unique.ecc, &eccPrivate,
    NULL, NULL)
 != TPM_RC_SUCCESS)
    result = TPM_RC_KEY;
else
{
    // The secret value is computed from Z using KDFe as:
    // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
    // Where:
    // HashID  the nameAlg of the decrypt key
    // Z   the x coordinate (Px) of the product (P) of the point
    //      (Q) of the secret and the private x coordinate (de,V)
    // of the decryption key
    // Use a null-terminated string containing "SECRET"
    // PartyUInfo  the x coordinate of the point in the secret
    //      (Qe,U )
    // PartyVInfo  the x coordinate of the public key (Qs,V )
    // bits  the number of bits in the digest of HashID
    // Retrieve seed from KDFe
    CryptKDFe(encryptKey->publicArea.nameAlg, &eccSecret.x.b,
        label, &eccPublic.x.b,
        &encryptKey->publicArea.unique.ecc.x.b,
        data->t.size * 8, data->t.buffer);
}
#endif // ALG_ECC
default:
    FAIL(FATAL_ERROR_INTERNAL);
break;

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 keyedHash or symmetric key, the secret is larger than the size of the digest produced by the name algorithm.
TPM_RC_FAILURE | internal error

```c
TPM_RC
CryptSecretDecrypt(
    OBJECT                  *decryptKey, // IN: decrypt key
    TPM2B_NONCE             *nonceCaller, // IN: nonceCaller. It is needed for symmetric decryption. For asymmetric decryption, this parameter is NULL
    const TPM2B             *label, // IN: a value for L
    TPM2B_ENCRYPTED_SECRET  *secret, // IN: input secret
    TPM2B_DATA              *data // OUT: decrypted secret value
)
{
    TPM_RC      result = TPM_RC_SUCCESS;
    // Decryption for secret
    switch(decryptKey->publicArea.type)
    {
        #if ALG_RSA
        case ALG_RSA_VALUE:
        {
            TPMT_RSA_DECRYPT        scheme;
            TPMT_RSA_SCHEME         *keyScheme
                = &decryptKey->publicArea.parameters.rsaDetail.scheme;
            UINT16                   digestSize;
            scheme = *(TPMT_RSA_DECRYPT *)keyScheme;
            // If the key scheme is ALG_NULL_VALUE, set the scheme to OAEP and set the algorithm to the name algorithm.
            if(scheme.scheme == ALG_NULL_VALUE)
            {
                // Use OAEP scheme
                scheme.scheme = ALG_OAEP_VALUE;
                scheme.details.oaep.hashAlg = decryptKey->publicArea.nameAlg;
            }
            // use the digestSize as an indicator of whether or not the scheme
            // is using a supported hash algorithm.
            // Note: depending on the scheme used for encryption, a hashAlg might not be needed. However, the return value has to have some upper
            // limit on the size. In this case, it is the size of the digest of the
            // hash algorithm. It is checked after the decryption is done but, there
            // is no point in doing the decryption if the size is going to be 'wrong' anyway.
            digestSize = CryptHashGetDigestSize(scheme.details.oaep.hashAlg);
            if(scheme.scheme != ALG_OAEP_VALUE || digestSize == 0)
            return TPM_RC_SCHEMA;
        }```
```
// Set the output buffer capacity
data->t.size = sizeof(data->t.buffer);

// Decrypt seed by RSA OAEP
result = CryptRsaDecrypt(&data->b, &secret->b,
decryptKey, &scheme, label);
if ((result == TPM_RC_SUCCESS) && (data->t.size > digestSize))
    result = TPM_RC_VALUE;

break;
#endif // ALG_RSA
#if ALG_ECC

} // ALG_ECC
#endif

} // ALG_ECC_VALUE:
{ TPMS_ECC_POINT eccPublic;
TPMS_ECC_POINT eccSecret;
BYTE *buffer = secret->t.secret;
INT32 size = secret->t.size;

// Retrieve ECC point from secret buffer
result = TPMS_ECC_POINT_Unmarshal(&eccPublic, &buffer, &size);
if (result == TPM_RC_SUCCESS)
    { result = CryptEccPointMultiply(&eccSecret,
decryptKey->publicArea.parameters.eccDetail.curveID,
    &eccPublic, &decryptKey->sensitive.sensitive.ecc,
    NULL, NULL);
if (result == TPM_RC_SUCCESS)
    { // Set the size of the "recovered" secret value to be the size
    // of the digest produced by the nameAlg.
    data->t.size =
    CryptHashGetDigestSize(decryptKey->publicArea.nameAlg);
    // The secret value is computed from Z using KDFe as:
    // secret := KDFe(HashID, Z, Use, PartyUInfo, PartyVInfo, bits)
    // Where:
    // HashID -- the nameAlg of the decrypt key
    // Z -- the x coordinate (Px) of the product (P) of the point
    // (Q) of the secret and the private x coordinate (de,V)
    // of the decryption key
    // Use -- a null-terminated string containing "SECRET"
    // PartyUInfo -- the x coordinate of the point in the secret
    // (Qe,U )
    // PartyVInfo -- the x coordinate of the public key (Qs,V )
    // bits -- the number of bits in the digest of HashID
    // Retrieve seed from KDFe
    CryptKDFe(decryptKey->publicArea.nameAlg, &eccSecret.x.b, label,
    &eccPublic.x.b,
    &decryptKey->publicArea.unique.ecc.x.b,
    data->t.size * 8, data->t.buffer);
    }
    }
break;
#endif // ALG_ECC
#if !ALG_KEYEDHASH
#error "KEYEDHASH support is required"
#endif

} // ALG_KEYEDHASH_VALUE:
{ // The seed size can not be bigger than the digest size of nameAlg
if (secret->t.size >
    CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
    result = TPM_RC_VALUE;
else
    //
566
567 {  
568     // Retrieve seed by XOR Obfuscation:
569     // seed = XOR(secret, hash, key, nonceCaller, nullNonce)
570     // where:
571     // secret the secret parameter from the TPM2_StartAuthHMAC
572     // command that contains the seed value
573     // hash nameAlg of tpmKey
574     // key the key or data value in the object referenced by
575     // entityHandle in the TPM2_StartAuthHMAC command
576     // nonceCaller the parameter from the TPM2_StartAuthHMAC command
577     // nullNonce a zero-length nonce
578     // XOR Obfuscation in place
579     CryptXORObfuscation(decryptKey->publicArea.nameAlg,
580                     &decryptKey->sensitive.sensitive.bits.b,
581                     &nonceCaller->b, NULL,
582                     secret->t.size, secret->t.secret);
583     // Copy decrypted seed
584     MemoryCopy2B(&data->b, &secret->b, sizeof(data->buffer));
585 }
586 break;
587 case ALG_SYMCIPHER_VALUE:
588 {
589     TPM2B_IV iv = {{0}};
590     TPMT_SYM_DEF_OBJECT *symDef;
591     // The seed size can not be bigger than the digest size of nameAlg
592     if(secret->t.size >
593         CryptHashGetDigestSize(decryptKey->publicArea.nameAlg))
594         result = TPM_RC_VALUE;
595     else
596     {
597         symDef = &decryptKey->publicArea.parameters.symDetail.sym;
598         iv.t.size = CryptGetSymmetricBlockSize(symDef->algorithm,
599             symDef->keyBits.sym);
600         if(iv.t.size == 0)
601             return TPM_RC_FAILURE;
602         if(nonceCaller->t.size >= iv.t.size)
603             {   
604                 MemoryCopy(iv.t.buffer, nonceCaller->t.buffer, iv.t.size);
605             }
606         else
607         {
608             if(nonceCaller->t.size > sizeof(iv.t.buffer))
609                 return TPM_RC_FAILURE;
610             MemoryCopy(iv.b.buffer, nonceCaller->t.buffer,
611                     nonceCaller->t.size);
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 break;
624 default:
625     FAIL(FATAL_ERROR_INTERNAL);
626     break;
627 }
628 return result;
10.2.6.5 CryptParameterEncryption()

This function does in-place encryption of a response parameter.

void CryptParameterEncryption(
    TPM_HANDLE handle,       // IN: encrypt session handle
    TPM2B *nonceCaller,      // IN: nonce caller
    UINT16 leadingSizeInByte,  // IN: the size of the leading size field in bytes
    TPM2B_AUTH *extraKey,     // IN: additional key material other than sessionAuth
    BYTE *buffer              // IN/OUT: parameter buffer to be encrypted
)
{
    SESSION *session = SessionGet(handle);   // encrypt session
    TPM2B_TYPE TEMP_KEY, ((sizeof(extraKey->t.buffer) + sizeof(session->sessionKey.t.buffer)))
    TPM2B_TEMP_KEY key;                      // encryption key
    UINT32 cipherSize = 0;                    // size of cipher text
    // Retrieve encrypted data size.
    if(leadingSizeInByte == 2)
    {
        // Extract the first two bytes as the size field as the data size
        // encrypt
        cipherSize = (UINT32)BYTE_ARRAY_TO_UINT16(buffer);
        // advance the buffer
        buffer = &buffer[2];
    }
    #ifdef TPM4B
    else if(leadingSizeInByte == 4)
    {
        // use the first four bytes to indicate the number of bytes to encrypt
        cipherSize = BYTE_ARRAY_TO_UINT32(buffer);
        // advance pointer
        buffer = &buffer[4];
    }
    #endif
    else
    {
        FAIL(FATAL_ERROR_INTERNAL);
    }
    // Compute encryption key by concatenating sessionKey with extra key
    MemoryCopy2B(&key.b, &session->sessionKey.b, sizeof(key.t.buffer));
    MemoryConcat2B(&key.b, &extraKey->b, sizeof(key.t.buffer));
    if(session->symmetric.algorithm == ALG_XOR_VALUE)
    {
        // XOR parameter encryption formulation:
        // XOR(parameter, hash, sessionAuth, nonceNewer, nonceOlder)
        CryptXORObfuscation(session->authHashAlg, &key.b),
        &(session->nonceTPM.b),
        nonceCaller, cipherSize, buffer);
    }
    else
    {
        ParmEncryptSym(session->symmetric.algorithm, session->authHashAlg,
        session->symmetric.keyBits.aes, &key.b),
        nonceCaller, &(session->nonceTPM.b),
        cipherSize, buffer);
        return;
    }
}
10.2.6.6 CryptParameterDecryption()

This function does in-place decryption of a command parameter.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIZE</td>
<td>The number of bytes in the input buffer is less than the number of bytes to be decrypted.</td>
</tr>
</tbody>
</table>
10.2.6.7 CryptComputeSymmetricUnique()

This function computes the unique field in public area for symmetric objects.

```c
void CryptComputeSymmetricUnique(
    TPMT_PUBLIC     *publicArea, // IN: the object's public area
    TPMT_SENSITIVE  *sensitive, // IN: the associated sensitive area
    TPM2B_DIGEST    *unique     // OUT: unique buffer
)
{
    // For parents (symmetric and derivation), use an HMAC to compute
    // the 'unique' field
    if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
        & IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt))
    {
        // Unique field is HMAC(sensitive->seedValue, sensitive->sensitive)
        HMAC_STATE hmacState;
        unique->b.size = CryptHmacStart2B(&hmacState, publicArea->nameAlg,
            &sensitive->seedValue.b);
        CryptDigestUpdate2B(&hmacState.hashState,
            &sensitive->sensitive.any.b);
        CryptHmacEnd2B(&hmacState, &unique->b);
    }
    else
    {
        HASH_STATE hashState;
        // Unique := Hash(sensitive->seedValue || sensitive->sensitive)
        unique->t.size = CryptHashStart(&hashState, publicArea->nameAlg);
        CryptDigestUpdate2B(&hashState, &sensitive->seedValue.b);
        CryptDigestUpdate2B(&hashState, &sensitive->sensitive.any.b);
        CryptHashEnd2B(&hashState, &unique->b);
    }
    return;
}
```

10.2.6.8 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 a 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_KEY</td>
<td>a provided key is not an allowed value</td>
</tr>
<tr>
<td>TPM_RC_KEY_SIZE</td>
<td>key size in the public area does not match the size in the sensitive creation area for a symmetric key</td>
</tr>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>unable to get random values (only in derivation)</td>
</tr>
<tr>
<td>TPM_RC_RANGE</td>
<td>for an RSA key, the exponent is not supported</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>sensitive data size is larger than allowed for the scheme for a keyed hash object</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>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</td>
</tr>
</tbody>
</table>

```c
TPM_RC
CryptCreateObject(
    OBJECT                  *object,       // IN: new object structure pointer
    TPMS_SENSITIVE_CREATE   *sensitiveCreate, // IN: sensitive creation
    RAND_STATE              *rand           // IN: the random number generator to use
)
{
    TPMT_PUBLIC             *publicArea = &object->publicArea;
    TPMT_SENSITIVE          *sensitive = &object->sensitive;
    TPM_RC                   result = TPM_RC_SUCCESS;
    // Set the sensitive type for the object
    sensitive->sensitiveType = publicArea->type;
    // For all objects, copy the initial authorization data
    sensitive->authValue = sensitiveCreate->userAuth;
    // If the TPM is the source of the data, set the size of the provided data to
    // zero so that there's no confusion about what to do.
    if(IS_ATTRIBUTE(publicArea->objectAttributes,
        TPMA_OBJECT, sensitiveDataOrigin))
        sensitiveCreate->data.t.size = 0;
    // Generate the key and unique fields for the asymmetric keys and just the
    // sensitive value for symmetric object
    switch(publicArea->type)
    {
        #if ALG_RSA
            // Create RSA key
            case ALG_RSA_VALUE:
                // RSA uses full object so that it has a place to put the private
                // exponent
                result = CryptRsaGenerateKey(publicArea, sensitive, rand);
                break;
            #endif // ALG_RSA
        #if ALG_ECC
            // Create ECC key
            case ALG_ECC_VALUE:
                result = CryptEccGenerateKey(publicArea, sensitive, rand);
                break;
            #endif // ALG_ECC
        case ALG_SYMCIPHER_VALUE:
```
result = CryptGenerateKeySymmetric(publicArea, sensitive,
sensitiveCreate, rand);
break;
case ALG_KEYEDHASH_VALUE:
    result = CryptGenerateKeyedHash(publicArea, sensitive,
sensitiveCreate, rand);
    break;
default:
    FAIL(FATAL_ERROR_INTERNAL);
    break;
}
if (result != TPM_RC_SUCCESS)
    return result;

// Create the sensitive seed value
// If this is a primary key in the endorsement hierarchy, stir the DRBG state
// This implementation uses both shProof and ehProof to make sure that there
// is no leakage of either.
if (object->attributes.primary && object->attributes.epsHierarchy)
{
    DRBG_AdditionalData((DRBG_STATE *)rand, &gp.shProof.b);
    DRBG_AdditionalData((DRBG_STATE *)rand, &gp.ehProof.b);
}
// Generate a seedValue that is the size of the digest produced by nameAlg
sensitive->seedValue.t.size =
    DRBG_Generate(rand, sensitive->seedValue.t.buffer,
                  CryptHashGetDigestSize(publicArea->nameAlg));
if (g_inFailureMode)
    return TPM_RC_FAILURE;
else if (sensitive->seedValue.t.size == 0)
    return TPM_RC_NO_RESULT;
// For symmetric objects, need to compute the unique value for the public area
if (publicArea->type == ALG_SYMCIPHER_VALUE
    || publicArea->type == ALG_KEYEDHASH_VALUE)
{
    CryptComputeSymmetricUnique(publicArea, sensitive, &publicArea->unique.sym);
}
else
{
    // if this is an asymmetric key and it isn't a parent, then
    // get rid of the seed.
    if (IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign)
        || !IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted))
    {
        memset(&sensitive->seedValue, 0, sizeof(sensitive->seedValue));
    }
}
// Compute the name
PublicMarshalAndComputeName(publicArea, &object->name);
return result;

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

TPMI_ALG_HASH
CryptGetSignHashAlg(
TPMT_SIGNATURE *auth // IN: signature
)
{
    if (auth->sigAlg == ALG_NULL_VALUE)
        FAIL(FATAL_ERROR_INTERNAL);
    // Get authHash algorithm based on signing scheme
    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     #  if ALG_SM2 
904     case ALG_SM2_VALUE: 
905     #  endif 
906     #  if ALG_ECSCHNORR 
907     case ALG_ECSCHNORR_VALUE: 
908     #  endif 
909     // all ECC signatures look the same 
910     return auth->signature.ecdsa.hash; 
911 
912     #  if ALG_ECDAA 
913     // Don't know how to verify an ECDAA signature 
914     case ALG_ECDAA_VALUE: 
915     break; 
916     #  endif 
917     #endif // ALG_ECC 
918 
919     case ALG_HMAC_VALUE: 
920     return auth->signature.hmac.hashAlg; 
921     default: 
922     break; 
923     } 
924     return ALG_NULL_VALUE; 
925 } 
926 } 
927 
928 10.2.6.6.10 CryptIsSplitSign() 
929 
930 This function us used to determine if the signing operation is a split signing operation that required a 
931 TPM2_Commit(). 
932 
933 BOOL CryptIsSplitSign( 
934     TPM_ALG_ID       scheme // IN: the algorithm selector 
935 ) 
936 { 
937     switch(scheme) 
938     { 
939     #  if ALG_ECDAA 
940     case ALG_ECDAA_VALUE: 
941     return TRUE; 
942     break; 
943     #  endif // ALG_ECDAA 
944     default: 
945     break; 
946     } 
947 }
10.2.6.6.11 CryptIsAsymSignScheme()

This function indicates if a scheme algorithm is a sign algorithm.

```c
BOOL CryptIsAsymSignScheme(
    TPMI_ALG_PUBLIC          publicType,  // IN: Type of the object
    TPMI_ALG_ASYM_SCHEME     scheme    // IN: the scheme
)
{
    BOOL isSignScheme = TRUE;
    switch(publicType)
    {
        #if ALG_RSA
        case ALG_RSA_VALUE:
            switch(scheme)
            {
                # if !ALG_RSASSA || !ALG_RSAPSS
                #   error "RSASSA and PSAPSS required if RSA used."
                # endif
                case ALG_RSASSA_VALUE:
                    break;
                case ALG_RSAPSS_VALUE:
                    break;
                default:
                    isSignScheme = FALSE;
                    break;
                # endif // ALG_RSA
        # if ALG_ECC
        // If ECC is implemented ECDSA is required
        case ALG_ECC_VALUE:
            switch(scheme)
            {
                // Support for ECDSA is required for ECC
                case ALG_ECDSA_VALUE:
                #if ALG_ECDAA // ECDA is optional
                    break;
                #endif
                #if ALG_ECSCHNORR // Schnorr is also optional
                case ALG_ECSCHNORR_VALUE:
                #endif
                #if ALG_SM2 // SM2 is optional
                case ALG_SM2_VALUE:
                #endif
                    break;
                default:
                    isSignScheme = FALSE;
                    break;
                # endif // ALG_ECC
                #endif
        } break;
    #endif // ALG_ECC
        default:
            isSignScheme = FALSE;
            break;
    } return isSignScheme;
```

10.2.6.6.12 CryptIsAsymDecryptScheme()

This function indicates if a scheme algorithm is a decrypt algorithm.

```c
BOOL CryptIsAsymDecryptScheme(
    TPMI_ALG_PUBLIC publicType, // IN: Type of the object
    TPMI_ALG_ASYM_SCHEME scheme   // IN: the scheme
)
{
    BOOL isDecryptScheme = TRUE;
    switch(publicType)
    {
        #if ALG_RSA
            case ALG_RSA_VALUE:
                switch(scheme)
                {
                    case ALG_RSAES_VALUE:
                    case ALG_OAEP_VALUE:
                        break;
                    default:
                        isDecryptScheme = FALSE;
                        break;
                }
                break;
        #endif // ALG_RSA
        #if ALG_ECC
            // If ECC is implemented ECDH is required
            case ALG_ECC_VALUE:
                switch(scheme)
                {
                    #if !ALG_ECDH
                        # error "ECDH is required for ECC"
                    #endif
                    #if ALG_SM2
                        case ALG_SM2_VALUE:
                            #endif
                    #if ALG_ECMQV
                        case ALG_ECMQV_VALUE:
                            #endif
                    default
                        break;
                    default:
                        isDecryptScheme = FALSE;
                        break;
                }
                break;
        #endif // ALG_ECC
        default:
            isDecryptScheme = FALSE;
            break;
    }
    return isDecryptScheme;
}
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>scheme selected</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>both scheme and key's default scheme are empty; or scheme is empty while key's default scheme requires explicit input scheme (split signing); or non-empty default key scheme differs from scheme</td>
</tr>
</tbody>
</table>

```c
BOOL CryptSelectSignScheme(
  OBJECT *signObject,  // IN: signing key
  TPMT_SIG_SCHEME *scheme  // IN/OUT: signing scheme
)
{
  TPMT_SIG_SCHEME *objectScheme;
  TPMT_PUBLIC *publicArea;
  BOOL OK;

  if (signObject == NULL)
  {
    OK = TRUE;
    scheme->scheme = ALG_NULL_VALUE;
    scheme->details.any.hashAlg = ALG_NULL_VALUE;
  }
  else
  {
    // assignment to save typing
    publicArea = &signObject->publicArea;

    // A symmetric cipher can be used to encrypt and decrypt but it can't
    // be used for signing
    if (publicArea->type == ALG_SYMCIPHER_VALUE)
      return FALSE;

    // Point to the scheme object
    if (CryptIsAsymAlgorithm(publicArea->type))
      objectScheme =
        (TPMT_SIG_SCHEME *)&publicArea->parameters.asymDetail.scheme;
    else
      objectScheme =
        (TPMT_SIG_SCHEME *)&publicArea->parameters.keyedHashDetail.scheme;

    // If the object doesn't have a default scheme, then use the
    // input scheme.
    if (objectScheme->scheme == ALG_NULL_VALUE)
    {
      // Input and default can't both be NULL
      OK = (scheme->scheme != ALG_NULL_VALUE);
      // Assume that the scheme is compatible with the key. If not,
      // an error will be generated in the signing operation.
    }
    else if (scheme->scheme == ALG_NULL_VALUE)
  }
```
1099  {
1100      // input scheme is NULL so use default
1101      // First, check to see if the default requires that the caller
1102      // provided scheme data
1103      OK = !CryptIsSplitSign(objectScheme->scheme);
1104      if(OK)
1105      {
1106          // The object has a scheme and the input is TPM_ALG_NULL so copy
1107          // the object scheme as the final scheme. It is better to use a
1108          // structure copy than a copy of the individual fields.
1109          *scheme = *objectScheme;
1110      }
1111      else
1112      {
1113          // Both input and object have scheme selectors
1114          // If the scheme and the hash are not the same then...
1115          // NOTE: the reason that there is no copy here is that the input
1116          // might contain extra data for a split signing scheme and that
1117          // data is not in the object so, it has to be preserved.
1118          OK = (objectScheme->scheme == scheme->scheme)
1119              && (objectScheme->details.any.hashAlg
1120                 == scheme->details.any.hashAlg);
1121      }
1122      }
1123  }
1124  return OK;
1125}

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SCHEME</td>
<td>signScheme is not compatible with the signing key type</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>digest value is greater than the modulus of signHandle or size of hashData does not match hash algorithm in signScheme (for an RSA key); invalid commit status or failed to generate r value (for an ECC key)</td>
</tr>
</tbody>
</table>

1127
1128  TPM_RC
1129  CryptSign(
1130      OBJECT              *signKey,       // IN: signing key
1131      TPM_T_SIG_SCHEME     *signScheme,    // IN: sign scheme.
1132      TPM2B_DIGEST         *digest,        // IN: The digest being signed
1133      TPM_T_SIGNATURE      *signature      // OUT: signature
1134  )
1135  
1136  TPM_RC result = TPM_RC_SCHEME;
1137  
1138  // Initialize signature scheme
1139  signature->sigAlg = signScheme->scheme;
1140  
1141  // If the signature algorithm is TPM_ALG_NULL or the signing key is NULL,
1142  // then we are done
1143  if((signature->sigAlg == TPM_ALG_NULL_VALUE) || (signKey == NULL))
1144      return TPM_RC_SUCCESS;
1145  
1146  // Initialize signature hash

Family "2.0" TCG PUBLIC REVIEW Page 571
Level 00 Revision 01.55 Copyright © TCG 2006-2019 April 15, 2019
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>the signature is not genuine</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>the scheme is not supported</td>
</tr>
<tr>
<td>TPM_RC_HANDLE</td>
<td>an HMAC key was selected but the private part of the key is not loaded</td>
</tr>
</tbody>
</table>

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_SCHEMA;
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            // Call RSA code to verify signature
1202            result = CryptRsaValidateSignature(signature, signObject, digest);
1203            break;
1204        }
1205      #endif // ALG_RSA
1206
1207      #if ALG_ECC
1208      case ALG_ECC_VALUE:
1209          result = CryptEccValidateSignature(signature, signObject, digest);
1210          break;
1211      #endif // ALG_ECC
1212
1213      case ALG_KEYEDHASH_VALUE:
1214          if(signObject->attributes.publicOnly)
1215              result = TPM_RCS_HANDLE;
1216          else
1217              result = CryptHMACVerifySignature(signObject, digest, signature);
1218          break;
1219      default:
1220          break;
1221      }
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>sizes are consistent</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>sizes are not consistent</td>
</tr>
</tbody>
</table>

```c
BOOL CryptIsUniqueSizeValid(
    TPMT_PUBLIC *publicArea) // IN: the public area to check
{
    BOOL consistent = FALSE;
    UINT16 keySizeInBytes;

    switch(publicArea->type)
    {
        #if ALG_RSA
        case ALG_RSA_VALUE:
            keySizeInBytes = BITS_TO_BYTES(publicArea->parameters.rsaDetail.keyBits);
            consistent = publicArea->unique.rsa.t.size == keySizeInBytes;
            break;
        #endif // ALG_RSA

        #if ALG_ECC
        case ALG_ECC_VALUE:
        {
            keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(publicArea->parameters.eccDetail.curveID));
            consistent = keySizeInBytes > 0 && publicArea->unique.ecc.x.t.size <= keySizeInBytes
                         && publicArea->unique.ecc.y.t.size <= keySizeInBytes;
        }
        break;
        #endif // ALG_ECC

        default:
        {
            // For SYMCIPHER and KEYEDHASH objects, the unique field is the size of the nameAlg digest.
            consistent = publicArea->unique.sym.t.size
                         == CryptHashGetDigestSize(publicArea->nameAlg);
            break;
        }
    }

    return consistent;
}
```

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.

```c
BOOL CryptIsSensitiveSizeValid(
    TPMT_PUBLIC *publicArea,       // IN: the object's public part
    TPMT_SENSITIVE *sensitiveArea // IN: the object's sensitive part
)
{
    BOOL consistent;
    UINT16 keySizeInBytes;

    switch(publicArea->type)
    {
        #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     // 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 } return consistent;
1316
10.2.6.6.19 CryptValidateKeys()

This function is used to verify that the key material of and 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_BINDING</td>
<td>the public and private parts are not cryptographically bound</td>
</tr>
<tr>
<td>TPM_RC_HASH</td>
<td>cannot have a publicOnly key with nameAlg of TPM_ALG_NULL</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>the public unique is not valid</td>
</tr>
<tr>
<td>TPM_RC_KEY_SIZE</td>
<td>the private area key is not valid</td>
</tr>
<tr>
<td>TPM_RC_TYPE</td>
<td>the types of the sensitive and private parts do not match</td>
</tr>
</tbody>
</table>

1319 TPM_RC
1320 CryptValidateKeys(
1321     TPMT_PUBLIC *publicArea,
1322     TPMT_SENSITIVE *sensitive,
TPM_RC    blamePublic,
TPM_RC    blameSensitive
)

TPM_RC    result;
UINT16    keySizeInBytes;
UINT16    digestSize = CryptHashGetDigestSize(publicArea->nameAlg);
TPMU_PUBLIC_PARMS *params = &publicArea->parameters;
TPMU_PUBLIC_ID    *unique = &publicArea->unique;

if(sensitive != NULL)
{
    // Make sure that the types of the public and sensitive are compatible
    if(publicArea->type != sensitive->sensitiveType)
        return TPM_RCS_TYPE + blameSensitive;
    // Make sure that the authValue is not bigger than allowed
    // If there is no name algorithm, then the size just needs to be less than
    // the maximum size of the buffer used for authorization. That size check
    // was made during unmarshaling of the sensitive area
    if((sensitive->authValue.t.size > digestSize && (digestSize > 0))
        return TPM_RCS_SIZE + blameSensitive;
}

switch(publicArea->type)
{
#if ALG_RSA
    case ALG_RSA_VALUE:
        keySizeInBytes = BITS_TO_BYTES(params->rsaDetail.keyBits);
        // Regardless of whether there is a sensitive area, the public modulus
        // needs to have the correct size. Otherwise, it can't be used for
        // any public key operation nor can it be used to compute the private
        // exponent.
        // NOTE: This implementation only supports key sizes that are multiples
        // of 1024 bits which means that the MSB of the 0th byte will always be
        // SET in any prime and in the public modulus.
        if((unique->rsa.t.size != keySizeInBytes)
            || (unique->rsa.t.buffer[0] < 0x80))
            return TPM_RCS_KEY + blamePublic;
        if(params->rsaDetail.exponent != 0
            && params->rsaDetail.exponent < 7)
            return TPM_RCS_VALUE + blamePublic;
        if(sensitive != NULL)
        {
            // If there is a sensitive area, it has to be the correct size
            // including having the correct high order bit SET.
            if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))
            {  // Validate the public key size
                if(unique->ecc.x.t.size != keySizeInBytes
                    || unique->ecc.y.t.size != keySizeInBytes)
                    return TPM_RCS_KEY + blamePublic;
                if(publicArea->nameAlg != ALG_NULL_VALUE)
                {
                    if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))
                    {
#endif
                
#if ALG_ECC
            case ALG_ECC_VALUE:
                TPMI_ECC_CURVE curveId;
                curveId = params->eccDetail.curveID;
                keySizeInBytes = BITS_TO_BYTES(CryptEccGetKeySizeForCurve(curveId));
                if(sensitive == NULL)
                {
                    // Validate the public key size
                    if(unique->ecc.x.t.size != keySizeInBytes
                        || unique->ecc.y.t.size != keySizeInBytes)
                        return TPM_RCS_KEY + blamePublic;
                
                if(publicArea->nameAlg != ALG_NULL_VALUE)
                {
                    if(!CryptEccIsPointOnCurve(curveId, &unique->ecc))
                    {
#endif
                
#if ALG_DSA
            case ALG_DSA_VALUE:
                keySizeInBytes = BITS_TO_BYTES((params->dsaDetail.keyBits/8));
                // DSA public modulus must be of a certain size
                // if there is a sensitive area
                if(sensitive)
return TPM_RCS_ECC_POINT + blamePublic;
}
else
{
    // If the nameAlg is TPM_ALG_NULL, then only verify that the
    // private part of the key is OK.
    if(!CryptEccIsValidPrivateKey(&sensitive->sensitive.ecc, curveId))
        return TPM_RCS_KEY_SIZE;
    if(publicArea->nameAlg != ALG_NULL_VALUE)
    {
        // Full key load, verify that the public point belongs to the
        // private key.
        TPMS_ECC_POINT toCompare;
        result = CryptEccPointMultiply(&toCompare, curveId, NULL, &sensitive->sensitive.ecc, NULL, NULL);

        if(result != TPM_RC_SUCCESS)
            return TPM_RC_BINDING;
        else
        {
            // Make sure that the private key generated the public key.
            // The input values and the values produced by the point
            // multiply may not be the same size so adjust the computed
            // value to match the size of the input value by adding or
            // removing zeros.
            AdjustNumberB(&toCompare.x.b, unique->ecc.x.t.size);
            AdjustNumberB(&toCompare.y.b, unique->ecc.y.t.size);
            if(!MemoryEqual2B(&unique->ecc.x.b, &toCompare.x.b)
                || !MemoryEqual2B(&unique->ecc.y.b, &toCompare.y.b))
                return TPM_RC_BINDING;
        }
    }
    break;
}
#endif
default:
{
    // Checks for SYMCIPHER and KEYEDHASH are largely the same
    // If public area has a nameAlg, then validate the public area size
    // and if there is also a sensitive area, validate the binding
    // For consistency, if the object is public-only just make sure that
    // the unique field is consistent with the name algorithm
    if(sensitive == NULL)
    {
        if(unique->sym.t.size != digestSize)
            return TPM_RCS_KEY + blamePublic;
    }
    else
    {
        // Make sure that the key size in the sensitive area is consistent.
        if(publicArea->type == ALG_SYMCIPHER_VALUE)
        {
            result = CryptSymKeyValidate(&params->symDetail.sym, &sensitive->sensitive.sym);
            if(result != TPM_RC_SUCCESS)
                return result + blameSensitive;
        }
        else
        {
            // For a keyed hash object, the key has to be less than the
            // smaller of the block size of the hash used in the scheme or
            // 128 bytes. The worst case value is limited by the
            // unmarshaling code so the only thing left to be checked is
// that it does not exceed the block size of the hash.
// by the hash algorithm of the scheme.
TPMT_KEYEDHASH_SCHEME *scheme;
UINT16 maxSize;
scheme = &params->keyedHashDetail.scheme;
if(scheme->scheme == ALG_XOR_VALUE)
{
    maxSize = CryptHashGetBlockSize(scheme->details.xor.hashAlg);
} else if(scheme->scheme == ALG_HMAC_VALUE)
{
    maxSize = CryptHashGetBlockSize(scheme->details.hmac.hashAlg);
} else if(scheme->scheme == ALG_NULL_VALUE)
{
    // Not signing or xor so must be a data block
    maxSize = 128;
} else
    return TPM_RCS_SCHEME + blamePublic;
if(sensitive->sensitive.bits.t.size > maxSize)
    return TPM_RCS_KEY_SIZE + blameSensitive;
}
// If there is a nameAlg, check the binding
if(publicArea->nameAlg != ALG_NULL_VALUE)
{
    TPM2B_DIGEST compare;
    if(sensitive->seedValue.t.size != digestSize)
        return TPM_RCS_KEY_SIZE + blameSensitive;
    CryptComputeSymmetricUnique(publicArea, sensitive, &compare);
    if(!MemoryEqual2B(&unique->sym.b, &compare.b))
        return TPM_RC_BINDING;
}

// For a parent, need to check that the seedValue is the correct size for
// protections. It should be at least half the size of the nameAlg
if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, restricted)
    && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, decrypt)
    && sensitive != NULL
    && publicArea->nameAlg != ALG_NULL_VALUE)
{
    if((sensitive->seedValue.t.size < (digestSize / 2))
        || (sensitive->seedValue.t.size > digestSize))
        return TPM_RCS_SIZE + blameSensitive;
}
return TPM_RC_SUCCESS;

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

void CryptAlgsSetImplemented()
{
    AlgorithmGetImplementedVector(&g_implementedAlgorithms);
}
10.2.6.6.21 CryptSelectMac()

This function is used to set the MAC scheme based on the key parameters and the input scheme.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SCHEME</td>
<td>the scheme is not a valid mac scheme</td>
</tr>
<tr>
<td>TPM_RC_TYPE</td>
<td>the input key is not a type that supports a mac</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>the input scheme and the key scheme are not compatible</td>
</tr>
</tbody>
</table>

```
TPM_RC
CryptSelectMac(
    TMT_PUBLIC *publicArea,
    TPMI_ALG_MAC_SCHEME *inMac
)
{
    TPM_ALG_ID macAlg = ALG_NULL_VALUE;
    switch(publicArea->type)
    {
        case ALG_KEYEDHASH_VALUE:
        {
            // Local value to keep lines from getting too long
            TPM_KEYEDHASH_SCHEME *scheme;
            scheme = &publicArea->parameters.keyedHashDetail.scheme;
            // Expect that the scheme is either HMAC or NULL
            if(scheme->scheme != ALG_NULL_VALUE)
                macAlg = scheme->details.hmac.hashAlg;
            break;
        }
        case ALG_SYMCIPHER_VALUE:
        {
            TPMT_SYM_DEF_OBJECT *scheme;
            scheme = &publicArea->parameters.symDetail.sym;
            // Expect that the scheme is either valid symmetric cipher or NULL
            if(scheme->algorithm != ALG_NULL_VALUE)
                macAlg = scheme->mode.sym;
            break;
        }
        default:
            return TPM_RCS_TYPE;
    }
    // If the input value is not TPM_ALG_NULL ...
    if(*inMac != ALG_NULL_VALUE)
    {
        // ... then either the scheme in the key must be TPM_ALG_NULL or the input
        // value must match
        if((macAlg != ALG_NULL_VALUE) && (*inMac != macAlg))
            return TPM_RCS_VALUE;
    }
    else
    {
        // Since the input value is TPM_ALG_NULL, then the key value can't be
        // TPM_ALG_NULL
        if(macAlg == ALG_NULL_VALUE)
            return TPM_RCS_VALUE;
        *inMac = macAlg;
    }
    if(!CryptMacIsValidForKey(publicArea->type, *inMac, FALSE))
        return TPM_RCS_SCHEME;
    return TPM_RC_SUCCESS;
}```
10.2.6.6.22 CryptMacIsValidForKey()

Check to see if the key type is compatible with the mac type

```c
BOOL CryptMacIsValidForKey(
    TPM_ALG_ID keyType,
    TPM_ALG_ID macAlg,
    BOOL flag
)
{
    switch(keyType)
    {
        case ALG_KEYEDHASH_VALUE:
            return CryptHashIsValidAlg(macAlg, flag);
        break;
        case ALG_SYMCIPHER_VALUE:
            return CryptSmacIsValidAlg(macAlg, flag);
        break;
        default:
            break;
    }
    return FALSE;
}
```

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.

```c
BOOL CryptSmacIsValidAlg(
    TPM_ALG_ID alg,
    BOOL FLAG // IN: Indicates if TPM_ALG_NULL is valid
)
{
    switch (alg)
    {
        #if ALG_CMAC
        case ALG_CMAC_VALUE:
            return TRUE;
        break;
        #endif
        case ALG_NULL_VALUE:
            return FLAG;
        break;
        default:
            return FALSE;
    }
}
```

10.2.6.6.24 CryptSymModeIsValid()

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.

```c
BOOL CryptSymModeIsValid(
    TPM_ALG_ID mode,
    BOOL flag
)
{ 
```
switch (mode) {
    #if ALG_CTR
    case ALG_CTR_VALUE:
    #endif // ALG_CTR
    #if ALG_OFB
    case ALG_OFB_VALUE:
    #endif // ALG_OFB
    #if ALG_CBC
    case ALG_CBC_VALUE:
    #endif // ALG_CBC
    #if ALG_CFB
    case ALG_CFB_VALUE:
    #endif // ALG_CFB
    #if ALG_ECB
    case ALG_ECB_VALUE:
    #endif // ALG_ECB
    return TRUE;
    case ALG_NULL_VALUE:
    return flag;
    default:
    break;
  }
  break;
  return FALSE;
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

```c
#include "Tpm.h"
```

10.2.7.2  Functions

10.2.7.2.1  RunSelfTest()

Local function to run self-test

```c
static TPM_RC
CryptRunSelfTests(
    ALGORITHM_VECTOR    *toTest
        // IN: the vector of the algorithms to test
    )
{
    TPM_ALG_ID alg;

    // For each of the algorithms that are in the toTestVecor, need to run a
    // test
    for(alg = TPM_ALG_FIRST; alg <= TPM_ALG_LAST; alg++)
    {
        if(TEST_BIT(alg, *toTest))
        {
            TPM_RC result = CryptTestAlgorithm(alg, toTest);
            if(result != TPM_RC_SUCCESS)
                return result;
        }
    }

    return TPM_RC_SUCCESS;
}
```

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.
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CANCELED</td>
<td>processing of this command was canceled</td>
</tr>
<tr>
<td>TPM_RC_TESTING</td>
<td>if toTest list is not empty</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>an algorithm in the toTest list is not implemented</td>
</tr>
</tbody>
</table>

```
TPM_RC CryptIncrementalSelfTest(
    TPML_ALG *toTest,  // IN: list of algorithms to be tested
    TPML_ALG *toDoList // OUT: list of algorithms needing test
)
{
    ALGORITHM_VECTOR toTestVector = {0};
    TPM_ALG_ID alg;
    UINT32 i;

    pAssert(toTest != NULL && toDoList != NULL);
    if(toTest->count > 0)
    {
        // Transcribe the toTest list into the toTestVector
        for(i = 0; i < toTest->count; i++)
        {
            alg = toTest->algorithms[i];
            // make sure that the algorithm value is not out of range
            if((alg > TPM_ALG_LAST) || !TEST_BIT(alg, g_implementedAlgorithms))
            {
                // Transcribe the toTest list into the toTestVector
                // make sure that the algorithm value is not out of range
                if((alg > TPM_ALG_LAST) || !TEST_BIT(alg, g_implementedAlgorithms))
                {
                    // Transcribe the toTest list into the toTestVector
                    // make sure that the algorithm value is not out of range
                }
            }
        }
    }
}
```
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.

```c
void CryptInitializeToTest()
{
    // Indicate that nothing has been tested
    memset(&g_cryptoSelfTestState, 0, sizeof(g_cryptoSelfTestState));

    // Copy the implemented algorithm vector
    MemoryCopy(g_toTest, g_implementedAlgorithms, sizeof(g_toTest));

    // Setting the algorithm to null causes the test function to just clear
    // out any algorithms for which there is no test.
    CryptTestAlgorithm(TPM_ALG_ERROR, &g_toTest);

    return;
}
```

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CANCELED</td>
<td>test was canceled</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT
TPM_RC
CryptTestAlgorithm(
    TPM_ALG_ID alg,
    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) \n34 const struct { \n35 crypt_uword_t allocate, size, d[BN_MIN_ALLOC(bytes)]; \n36 } 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) \n40 const TPM2B_BYTE_VALUE NAME = {bytes, {initializer}}
41 #endif

Have to special case ECC_ZERO

42 TPM2B_BYTE_VALUE(1);
43 TPM2B_1_BYTE_VALUE ECC_ZERO = {1, {0}};
44 #endif
45 ECC_CONST(ECC_ONE, 1, 1);
46 #if USE_BN_ECC_DATA
47 define TO_ECC_192(a, b, c) a, b, c
48 TPM2B_BYTE_VALUE(24);
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);
#define TO_ECC_384(a, b, c, d, e, f) a, b, c, d, e, f
TPM2B_BYTE_VALUE(66);
#define TO_ECC_528(a, b, c, d, e, f, g, h, i) a, b, c, d, e, f, g, h, i
TPM2B_BYTE_VALUE(80);
#define TO_ECC_640(a, b, c, d, e, f, g, h, i, j) a, b, c, d, e, f, g, h, i, j
#define TO_ECC_192(a, b, c) c, b, a
#define TO_ECC_224(a, b, c, d) d, c, b, a
#define TO_ECC_256(a, b, c, d) d, c, b, a
#define TO_ECC_384(a, b, c, d, e, f, g, h, i, j) f, e, d, c, b, a
#define TO_ECC_528(a, b, c, d, e, f, g, h, i, j) i, h, g, f, e, d, c, b, a
#define TO_ECC_640(a, b, c, d, e, f, g, h, i, j) j, i, h, g, f, e, d, c, b, a
#endif // !USE_BN_ECC_DATA
#define NIST_P192_h ECC_ONE
#define NIST_P192_gZ ECC_ONE
#if USE_BN_ECC_DATA
const ECC_CURVE_DATA NIST_P192 = {
  (bigNum)&NIST_P192_p, (bigNum)&NIST_P192_n, (bigNum)&NIST_P192_h,
  (bigNum)&NIST_P192_a, (bigNum)&NIST_P192_b,
  {(bigNum)&NIST_P192_gX, (bigNum)&NIST_P192_gY, (bigNum)&NIST_P192_gZ};
#else
const ECC_CURVE_DATA NIST_P192 = {
  &NIST_P192_p.b, &NIST_P192_n.b, &NIST_P192_h.b,
  &NIST_P192_a.b, &NIST_P192_b.b,
  {&NIST_P192_gX.b, &NIST_P192_gY.b, &NIST_P192_gZ.b};
#endif // USE_BN_ECC_DATA
#endif // ECC_NIST_P192
#if ECC_NIST_P224
ECC_CONST(NIST_P224_p, 28, TO_ECC_224(
  TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
  TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00)),
ECC_CONST(NIST_P224_a, 28, TO_ECC_224(
  TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)),
ECC_CONST(NIST_P224_gX, 28, TO_ECC_224(
  TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)),
ECC_CONST(NIST_P224_gY, 28, TO_ECC_224(
  TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)),
ECC_CONST(NIST_P224_gZ, 28, TO_ECC_224(
  TO_ECC_32(0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
  TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)),
#endif // ECC_NIST_P224

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, 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, 0x01),
154          TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00),
155          TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00))),
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(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
159          TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00),
160          TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0x00, 0x00, 0x00, 0x00))),
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, 0x33, 0x57, 0x6B, 0x31, 0x5E, 0xCE),
166          TO_ECC_64(0xCB, 0xB6, 0x40, 0x68, 0x37, 0xBF, 0x51, 0xF5)));
167          ECC_CONST(NIST_P256_gX, 32, TO_ECC_256(
168          TO_ECC_64(0x6B, 0x17, 0xD1, 0xF2, 0xE1, 0x2C, 0x42, 0x47),
169          TO_ECC_64(0xF8, 0x8C, 0xE6, 0xE5, 0x63, 0xA4, 0x40, 0xF2),
170          TO_ECC_64(0x77, 0x03, 0x7D, 0x81, 0x2D, 0xEB, 0x33, 0xA0),
171          TO_ECC_64(0xF4, 0xA1, 0x39, 0x45, 0xD8, 0x98, 0xC2, 0x96)));
172          ECC_CONST(NIST_P256_gY, 32, TO_ECC_256(
173          TO_ECC_64(0x4F, 0xE3, 0x42, 0xE2, 0xFE, 0x1A, 0x7F, 0x9B),
174          TO_ECC_64(0x8E, 0xEB, 0xB5, 0xE6, 0xB5, 0xA4, 0x0F, 0акс),
175          TO_ECC_64(0.AppendFormat(0xF4, 0x8A, 0x93, 0x25, 0x51)),
176          TO_ECC_64(0x03, 0x8B, 0x0C, 0x60, 0x63, 0x25, 0x51)));
177          #define NIST_P256_h         ECC_ONE
178          #define NIST_P256_gZ        ECC_ONE
179          #if USE_BN_ECC_DATA
180          const ECC_CURVE_DATA NIST_P256 = {
181          (bigNum)&NIST_P256_p, (bigNum)&NIST_P256_n, (bigNum)&NIST_P256_h,
182          (bigNum)&NIST_P256_a, (bigNum)&NIST_P256_b,
183          {(bigNum)&NIST_P256_gX, (bigNum)&NIST_P256_gY, (bigNum)&NIST_P256_gZ}};
184          #endif // USE_BN_ECC_DATA
185          #endif // ECC_NIST_P256
186          }
(bigNum)&NIST_P256_p, (bigNum)&NIST_P256_n, (bigNum)&NIST_P256_h,
(bigNum)&NIST_P256_a, (bigNum)&NIST_P256_b,
{(bigNum)&NIST_P256_gX, (bigNum)&NIST_P256_gY, (bigNum)&NIST_P256_gZ});

#else

const ECC_CURVE_DATA NIST_P256 = {
    &NIST_P256_p.b, &NIST_P256_n.b, &NIST_P256_h.b,
    &NIST_P256_a.b, &NIST_P256_b.b,
    {&NIST_P256_gX.b, &NIST_P256_gY.b, &NIST_P256_gZ.b});
#endif // USE_BN_ECC_DATA

#if ECC_NIST_P384

#define NIST_P384_h ECC_ONE
#define NIST_P384_gZ        ECC_ONE
#if USE_BN_ECC_DATA

const ECC_CURVE_DATA NIST_P384 = {
    (bigNum)&NIST_P384_p, (bigNum)&NIST_P384_n, (bigNum)&NIST_P384_h,
    (bigNum)&NIST_P384_a, (bigNum)&NIST_P384_b,
    {(bigNum)&NIST_P384_gX, (bigNum)&NIST_P384_gY, (bigNum)&NIST_P384_gZ});
#else

const ECC_CURVE_DATA NIST_P384 = {
    &NIST_P384_p.b, &NIST_P384_n.b, &NIST_P384_h.b,
    &NIST_P384_a.b, &NIST_P384_b.b,
    {&NIST_P384_gX.b, &NIST_P384_gY.b, &NIST_P384_gZ.b});
#endif // USE_BN_ECC_DATA
# ifndef // ECC_NIST_P384
# if ECC_NIST_P521

ECC_CONST(NIST_P521_p, 66, TO_ECC_528(
    TO_EC_16(0x01, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)))),

ECC_CONST(NIST_P521_a, 66, TO_ECC_528(
    TO_EC_16(0x01, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)))),

ECC_CONST(NIST_P521_b, 66, TO_ECC_528(
    TO_EC_16(0x01, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)))),

ECC_CONST(NIST_P521_gX, 66, TO_ECC_528(
    TO_EC_16(0x01, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)))),

ECC_CONST(NIST_P521_gY, 66, TO_ECC_528(
    TO_EC_16(0x01, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)))),

ECC_CONST(NIST_P521_n, 66, TO_ECC_528(
    TO_EC_16(0x01, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
    TO_EC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)))),

#define NIST_P521_h

#define NIST_P521_gZ

#define USE_BN_ECC_DATA

const ECC_CURVE_DATA NIST_P521 = {

};
# USE_BN_ECC_DATA
326 #endif // ECC_BN_P256
327 #if ECC_BN_P638
328 ECC_CONST(BN_P638_p, 80, TO_ECC_640(
329 TO_ECC_64(0x23, 0xFF, 0xFF, 0xFD, 0xC0, 0x00, 0x00, 0x0D),
330 TO_ECC_64(0x7F, 0xFF, 0xFF, 0xFF, 0xB8, 0x00, 0x00, 0x01, 0xD3),
331 TO_ECC_64(0x0F, 0xFF, 0x0F, 0x94, 0x87, 0x00, 0x00, 0x00, 0xD5, 0x2F),
332 TO_ECC_64(0x0F, 0xFD, 0x0D, 0xE0, 0x00, 0x00, 0x08, 0xDE, 0x55),
333 TO_ECC_64(0xC0, 0x00, 0x86, 0x52, 0x00, 0x21, 0x5E, 0x5B),
334 TO_ECC_64(0xFF, 0xFD, 0xD0, 0xE0, 0x00, 0x08, 0xDE, 0x55),
335 TO_ECC_64(0x0F, 0 xAxis, 0x1F, 0xFF, 0xF4, 0xEB, 0x80),
336 TO_ECC_64(0x00, 0x00, 0x00, 0x4C, 0x80, 0x01, 0x5A, 0xCD),
337 TO_ECC_64(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x4E, 0x67)),
338 #define BN_P638_gZ, 1, TO_ECC_8(0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x66)));
#define BN_P638_h           ECC_ONE
#define BN_P638_gZ          ECC_ONE
#if USE_BN_ECC_DATA
const ECC_CURVE_DATA BN_P638 = {
(bignum)&BN_P638_p, (bignum)&BN_P638_n, (bignum)&BN_P638_h,
(bignum)&BN_P638_a, (bignum)&BN_P638_b,
{(bignum)&BN_P638_gX, (bignum)&BN_P638_gY, (bignum)&BN_P638_gZ}};
#else
const ECC_CURVE_DATA BN_P638 = {
&BN_P638_p.b, &BN_P638_n.b, &BN_P638_h.b,
&BN_P638_a.b, &BN_P638_b.b,
{&BN_P638_gX.b, &BN_P638_gY.b, &BN_P638_gZ.b}};
#endif // USE_BN_ECC_DATA
#endif // ECC_BN_P638
#if ECC_SM2_P256
ECC_CONST(SM2_P256_p, 32, TO_ECC_256(
TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
ECC_CONST(SM2_P256_a, 32, TO_ECC_256(
TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFE, 0xFF, 0xFF, 0xFF, 0xFF),
TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF),
TO_ECC_64(0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF)));
ECC_CONST(SM2_P256_gX, 32, TO_ECC_256(
TO_ECC_64(0x28, 0x9E, 0x99, 0x9F, 0x9E, 0x9F, 0x9E, 0x34),
TO_ECC_64(0x99, 0x46, 0x6A, 0xF4, 0xF6, 0x77, 0x9C, 0x59),
TO_ECC_64(0x8F, 0x32, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7)));
ECC_CONST(SM2_P256_gY, 32, TO_ECC_256(
TO_ECC_64(0x28, 0x9E, 0x99, 0x9F, 0x9E, 0x9F, 0x9E, 0x34),
TO_ECC_64(0x99, 0x46, 0x6A, 0xF4, 0xF6, 0x77, 0x9C, 0x59),
TO_ECC_64(0x8F, 0x32, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7)));
ECC_CONST(SM2_P256_n, 32, TO_ECC_256(
TO_ECC_64(0x28, 0x9E, 0x99, 0x9F, 0x9E, 0x9F, 0x9E, 0x34),
TO_ECC_64(0x99, 0x46, 0x6A, 0xF4, 0xF6, 0x77, 0x9C, 0x59),
TO_ECC_64(0x8F, 0x32, 0x45, 0x89, 0x33, 0x4C, 0x74, 0xC7)));
#define SM2_P256_h          ECC_ONE
#define SM2_P256_gZ         ECC_ONE
#if USE_BN_ECC_DATA
const ECC_CURVE_DATA SM2_P256 = {
(bignum)&SM2_P256_p, (bignum)&SM2_P256_n, (bignum)&SM2_P256_h,
(bignum)&SM2_P256_a, (bignum)&SM2_P256_b,
{(bignum)&SM2_P256_gX, (bignum)&SM2_P256_gY, (bignum)&SM2_P256_gZ}};
#else
const ECC_CURVE_DATA SM2_P256 = {
&SM2_P256_p.b, &SM2_P256_n.b, &SM2_P256_h.b,
&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   {TPM_ECC_NIST_P192,
456    192,
457    {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
458    {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
459    &NIST_P192,
460    OID_ECC_NIST_P192
461    CURVE_NAME("NIST_P192")
462   }
463 #  undef comma
464 #  define comma ,
465 #endif // ECC_NIST_P192
466 #if ECC_NIST_P224
467   {TPM_ECC_NIST_P224,
468    224,
469    {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
470    {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
471    &NIST_P224,
472    OID_ECC_NIST_P224
473    CURVE_NAME("NIST_P224")
474   }
475 #  undef comma
476 #  define comma ,
477 #endif // ECC_NIST_P224
478 #if ECC_NIST_P256
479   {TPM_ECC_NIST_P256,
480    256,
481    {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA256_VALUE}}},
482    {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
483    &NIST_P256,
484    OID_ECC_NIST_P256
485    CURVE_NAME("NIST_P256")
486   }
487 #  undef comma
488 #  define comma ,
489 #endif // ECC_NIST_P256
490 #if ECC_NIST_P384
491   {TPM_ECC_NIST_P384,
492    384,
493    {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA384_VALUE}}},
494    {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
495    &NIST_P384,
496    OID_ECC_NIST_P384
497    CURVE_NAME("NIST_P384")
498   }
499 #  undef comma
500 #  define comma ,
501 #endif // ECC_NIST_P384
502 #if ECC_NIST_P521
503   {TPM_ECC_NIST_P521,
504    521,
505    {ALG_KDF1_SP800_56A_VALUE, {{ALG_SHA512_VALUE}}},
506    {ALG_NULL_VALUE, {{ALG_NULL_VALUE}}},
507    &NIST_P521,
508    OID_ECC_NIST_P521
509    CURVE_NAME("NIST_P521")
510   }
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
Part 4: Supporting Routines

10.2.9

Trusted Platform Module Library

CryptDes.c

10.2.9.1

Introduction

This file contains the extra functions required for TDES.
10.2.9.2
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36

Includes, Defines, and Typedefs

#include "Tpm.h"
#if ALG_TDES
#define DES_NUM_WEAK 64
const UINT64 DesWeakKeys[DES_NUM_WEAK] = {
0x0101010101010101ULL, 0xFEFEFEFEFEFEFEFEULL,
0xE0E0E0E0F1F1F1F1ULL, 0x1F1F1F1F0E0E0E0EULL,
0x011F011F010E010EULL, 0x1F011F010E010E01ULL,
0x01E001E001F101F1ULL, 0xE001E001F101F101ULL,
0x01FE01FE01FE01FEULL, 0xFE01FE01FE01FE01ULL,
0x1FE01FE00EF10EF1ULL, 0xE01FE01FF10EF10EULL,
0x1FFE1FFE0EFE0EFEULL, 0xFE1FFE1FFE0EFE0EULL,
0xE0FEE0FEF1FEF1FEULL, 0xFEE0FEE0FEF1FEF1ULL,
0x01011F1F01010E0EULL, 0x1F1F01010E0E0101ULL,
0xE0E01F1FF1F10E0EULL, 0x0101E0E00101F1F1ULL,
0x1F1FE0E00E0EF1F1ULL, 0xE0E0FEFEF1F1FEFEULL,
0x0101FEFE0101FEFEULL, 0x1F1FFEFE0E0EFEFEULL,
0xE0FE011FF1FE010EULL, 0x011F1F01010E0E01ULL,
0x1FE001FE0EF101FEULL, 0xE0FE1F01F1FE0E01ULL,
0x011FE0FE010EF1FEULL, 0x1FE0E01F0EF1F10EULL,
0xE0FEFEE0F1FEFEF1ULL, 0x011FFEE0010EFEF1ULL,
0x1FE0FE010EF1FE01ULL, 0xFE0101FEFE0101FEULL,
0x01E01FFE01F10EFEULL, 0x1FFE01E00EFE01F1ULL,
0xFE011FE0FE010EF1ULL, 0xFE01E01FFE01F10EULL,
0x1FFEE0010EFEF101ULL, 0xFE1F01E0FE0E01F1ULL,
0x01E0E00101F1F101ULL, 0x1FFEFE1F0EFEFE0EULL,
0xFE1FE001FE0EF101ULL, 0x01E0FE1F01F1FE0EULL,
0xE00101E0F10101F1ULL, 0xFE1F1FFEFE0E0EFEULL,
0x01FE1FE001FE0EF1ULL, 0xE0011FFEF1010EFEULL,
0xFEE0011FFEF1010EULL, 0x01FEE01F01FEF10EULL,
0xE001FE1FF101FE0EULL, 0xFEE01F01FEF10E01ULL,
0x01FEFE0101FEFE01ULL, 0xE01F01FEF10E01FEULL,
0xFEE0E0FEFEF1F1FEULL, 0x1F01011F0E01010EULL,
0xE01F1FE0F10E0EF1ULL, 0xFEFE0101FEFE0101ULL,
0x1F01E0FE0E01F1FEULL, 0xE01FFE01F10EFE01ULL,
0xFEFE1F1FFEFE0E0EULL, 0x1F01FEE00E01FEF1ULL,
0xE0E00101F1F10101ULL, 0xFEFEE0E0FEFEF1F1ULL};

10.2.9.2.1

CryptSetOddByteParity()

This function sets the per byte parity of a 64-bit value. The least-significant bit is 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
38
39
40
41
42
43
44
45
46
47

UINT64
CryptSetOddByteParity(
UINT64
)

k

{
#define PMASK 0x0101010101010101ULL
UINT64
out;
k |= PMASK;
// set the parity bit
out = k;
k ^= k >> 4;
k ^= k >> 2;

Family “2.0”
Level 00 Revision 01.55

TCG PUBLIC REVIEW
Copyright © TCG 2006-2019

Page 595
April 15, 2019


48       k ^= k >> 1;          // odd parity extracted
49       k &= PMASK;         // out is now even parity because parity bit was already set
50       out ^= k;           // out is now even parity
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>DES key is weak</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>DES key is not weak</td>
</tr>
</tbody>
</table>

54    static BOOL CryptDesIsWeakKey(
55    UINT64            k
56    )
57    {
58        int          i;
59        //
60        for(i = 0; i < DES_NUM_WEAK; i++)
61        {
62            if(k == DesWeakKeys[i])
63                return TRUE;
64        }
65        return FALSE;
66    }

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.

BOOL CryptDesValidateKey(
    TPM2B_SYM_KEY       *desKey,  // IN: key to validate
)
{
    UINT64               k[3];
    int                  i;
    int                  keys = (desKey->t.size + 7) / 8;
    BYTE                *pk = desKey->t.buffer;
    BOOL                 ok;
    //
    // Note: 'keys' is the number of keys, not the maximum index for 'k'
    ok = ((keys == 2) || (keys == 3)) && ((desKey->t.size % 8) == 0);
    for(i = 0; ok && i < keys; pk += 8, i++)
    {
        k[i] = CryptSetOddByteParity(BYTE_ARRAY_TO_UINT64(pk));
        ok = !CryptDesIsWeakKey(k[i]);
    }
    ok = ok && k[0] != k[1];
    if(keys == 3)
        ok = ok && k[1] != k[2];
    return ok;
}
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.

```c
10.2.9.2.4 CryptGenerateKeyDes()
11     TPM_RC
12     CryptGenerateKeyDes(
13             TPMT_PUBLIC     *publicArea,  // IN/OUT: The public area template
14             // for the new key.
15             TPMT_SENSITIVE   *sensitive,  // OUT: sensitive area
16             RAND_STATE     *rand     // IN: the "entropy" source for
17         )
18     {
19         // Assume that the publicArea key size has been validated and is a supported
20         // number of bits.
21         sensitive->sensitive.sym.t.size =
22         BITS_TO_BYTES(publicArea->parameters.symDetail.sym.keyBits.sym);
23         do
24             {
25                 BYTE                    *pK = sensitive->sensitive.sym.t.buffer;
26                 int              i = (sensitive->sensitive.sym.t.size + 7) / 8;
27                 // Use the random number generator to generate the required number of bits
28                 if(DRBI_Generate(rand, pK, sensitive->sensitive.sym.t.size) == 0)
29                     return TPM_RC_NO_RESULT;
30                 for(; i > 0; pK += 8, i--)
31                     {
32                         UINT64      k = BYTE_ARRAY_TO_UINT64(pK);
33                         k = CryptSetOddByteParity(k);
34                         UINT64_TO_BYTE_ARRAY(k, pK);
35                     }
36             } while(!CryptDesValidateKey(&sensitive->sensitive.sym));
37         return TPM_RC_SUCCESS;
38     }
39 #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.

```
#include "Tpm.h"
#if CC_ZGen_2Phase == YES

10.2.10.2 Functions

#else ALG_ECMQV

10.2.10.2.1 avf1()

This function does the associated value computation required by MQV key exchange. Process:
   a)  Convert xQ to an integer xqi using the convention specified in Appendix C.3.
   b)  Calculate xqm = xqi mod 2^cei(f/2) (where f = ceil(log2(n)).
   c)  Calculate the associate value function avf(Q) = xqm + 2 ceil(f / 2) Always returns TRUE(1).

static BOOL

avf1(
   bigNum bnX,   // IN/OUT: the reduced value
   bigNum bnN    // IN: the order of the curve
)
{
   // compute f = 2^(cei(ceil(log2(n)) / 2))
   int f = (BnSizeInBits(bnN) + 1) / 2;
   // x' = 2^f + (x mod 2^f)
   BnMaskBits(bnX, f);  // This is mod 2*2^f but it doesn't matter because
   BnSetBit(bnX, f);    // the next operation will SET the extra bit anyway
   return TRUE;
}

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>the value for dsA does not give a valid point on the curve</td>
</tr>
</tbody>
</table>

```
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 {  
29  CURVE_INITIALIZED(E, curveId);  
30  const ECC_CURVE_DATA *C;  
31  POINT(pQeA);  
32  POINT_INITIALIZED(pQeB, QeB);  
33  POINT_INITIALIZED(pQsB, QsB);  
34  ECC_NUM(bnTa);  
35  ECC_INITIALIZED(bnDeA, deA);  
36  ECC_INITIALIZED(bnDsA, dsA);  
37  ECC_NUM(bnN);  
38  ECC_NUM(bnXeB);  
39  TPM_RC retVal;  
40  
41  // Parameter checks  
42  if (E == NULL)  
43     ERROR_RETURN(TPM_RC_VALUE);  
44  pAssert(outZ != NULL && pQeB != NULL && pQsB != NULL && deA != NULL  
45     && dsA != NULL);  
46  C = AccessCurveData(E);  
47  // Process:  
48  //  1. implicitsigA = (de,A + avf(Qe,A)ds,A ) mod n.  
49  //  2. P = h(implicitsigA)(Qe,B + avf(Qe,B)Qs,B).  
50  //  3. If P = O, output an error indicator.  
51  //  4. Z=xP, where xP is the x-coordinate of P.  
52  // 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  if (BnEqualZero(pQeB->z))  
89     ERROR_RETURN(TPM_RC_NO_RESULT);  
90  
91  // Convert BIGNUM E to TPM2B E
BnPointTo2B(outZ, pQeB, E);

Exit:
CURVE_FREE(E);
return retVal;
}
#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).

static TPM_RC
C_2_2_ECDH(
TPMS_ECC_POINT *outZs, // OUT: Zs
TPMS_ECC_POINT *outZe, // OUT: Ze
TPM_ECC_CURVE curveId, // IN: the curve for the computations
TPM2B_ECC_PARAMETER *dsA, // IN: static private TPM key
TPM2B_ECC_PARAMETER *deA, // IN: ephemeral private TPM key
TPMS_ECC_POINT *QsB, // IN: static public party B key
TPMS_ECC_POINT *QeB // IN: ephemeral public party B key
)
{
CURVE_INITIALIZED(E, curveId);
ECC_INITIALIZED(bnAs, dsA);
ECC_INITIALIZED(bnAe, deA);
POINT_INITIALIZED(ecBs, QsB);
POINT_INITIALIZED(ecBe, QeB);
POINT(ecZ);
TPM_RC            retVal;
// Parameter checks
if(E == NULL)
ERROR_RETURN(TPM_RC_CURVE);
pAssert(outZs != NULL && dsA != NULL && deA != NULL && QsB != NULL
&& QeB != NULL);
// Do the point multiply for the Zs value ([dsA]QsB)
retVal = BnPointMult(ecZ, ecBs, bnAs, NULL, NULL, E);
if(retVal == TPM_RC_SUCCESS)
{
    // Convert the Zs value.
    BnPointTo2B(outZs, ecZ, E);
    // Do the point multiply for the Ze value ([deA]QeB)
    retVal = BnPointMult(ecZ, ecBe, bnAe, NULL, NULL, E);
    if(retVal == TPM_RC_SUCCESS)
    { BnPointTo2B(outZe, ecZ, E);
    }
    Exit:
    CURVE_FREE(E);
    return retVal;
}

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.
139  LIB_EXPORT TPM_RC
140     CryptEcc2PhaseKeyExchange(
141                                TPM_ECC_POINT *outZ1,     // OUT: a computed point
142                                TPM_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                                TPM_ECC_POINT *QsB,            // IN: static public party B key
148                                TPM_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     #if ALG_SM2
183
184     10.2.10.2.5 ComputeWForSM2()
185
186     ComputeWForSM2(
187                     bigCurve        E
188                     )
189     {
190         // w := ceil(ceil(log2(n)) / 2) - 1
191         return (BnMsb(CurveGetOrder(AccessCurveData(E))) / 2 - 1);
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, \( W_s \) (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.

```c
static bigNum
avfSm2(  
  bigNum bn, // IN/OUT: the reduced value
  UINT32 w  // IN: the value of w
)
{
  // a) set w := ceil(ceil(log2(n)) / 2) - 1
  // b) set x' := 2^w + (x & (2^w - 1))
  // This is just like the avf for MQV where x' = 2^w + (x mod 2^w)
  BnMaskBits(bn, w);  // as with avf1, this is too big by a factor of 2 but
  // it doesn't matter because we SET the extra bit
  BnSetBit(bn, w);
  return bn;
}
```

10.2.10.2.7 SM2KeyExchange()

This function performs the key exchange defined in SM2. The first step is to compute \( tA = (dsA + deA \ avf(XeA)) \ mod \ n \) Then, compute the \( Z \) value from \( outZ = (h \ tA \ mod \ n) \ (QsA + [avf(QeB.x)](QeB)) \). 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>the value for ( dsA ) does not give a valid point on the curve</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT TPM_RC
SM2KeyExchange(  
  TPMS_ECC_POINT *outZ, // OUT: the computed point
  TPM_ECC_CURVE curveId, // IN: the curve for the computations
  TPM2B_ECC_PARAMETER *dsAIn, // IN: static private TPM key
  TPM2B_ECC_PARAMETER *deAIn, // IN: ephemeral private TPM key
  TPM2B_ECC_PARAMETER *QsBIn, // IN: static public party B key
  TPM2B_ECC_PARAMETER *QeBIn // IN: ephemeral public party B key
)
{
  CURVE_INITIALIZED(E, curveId);
  const ECC_CURVE_DATA *C;
  ECC_INITIALIZED(dsA, dsAIn);
  ECC_INITIALIZED(deA, deAIn);
  POINT_INITIALIZED(QsB, QsBIn);
  POINT_INITIALIZED(QeB, QeBIn);
  BN_WORD_INITIALIZED(One, 1);
  POINT(QeA);
  ECC_NUM(XeB);
  POINT(Z);
  ECC_NUM(Ta);
  UINT32 w;
  TPM_RC retVal = TPM_RC_NO_RESULT;
  // Parameter checks
  if(E == NULL)
```
```c
    ERROR_RETURN(TPM_RC_CURVE);
    C = AccessCurveData(E);
    pAssert(outZ != NULL && dsA != NULL && deA != NULL && QsB != NULL
         && QeB != NULL);

    // Compute the value for w
    w = ComputeWForSM2(E);

    // Compute the public ephemeral key pQeA = [de,A]G
    if(!BnEccModMult(QeA, CurveGetG(C), deA, E))
        goto Exit;

    // tA := (ds,A + de,A avf(Xe,A)) mod n    (3)
    // Compute 'tA' = ('dsA' + 'deA' avf('XeA')) mod n
    // Ta = avf(XeA);
    // do Ta = de,A * Ta = deA * avf(XeA)
    BnMult(Ta, deA, avfSm2(QeA->x, w));
    // now Ta = dsA + Ta = dsA + deA * avf(XeA)
    BnAdd(Ta, dsA, Ta);
    BnMod(Ta, CurveGetOrder(C));

    // outZ = [h tA mod n] (Qs,B + [avf(Xe,B)](Qe,B)) (4)
    if(!BnEqualWord(CurveGetCofactor(C), 1))
        // Cofactor is not 1 so compute Ta := Ta * h mod n
        BnModMult(Ta, Ta, CurveGetCofactor(C), CurveGetOrder(C));
    // Now that 'tA' is (h * 'tA' mod n)
    // 'outZ' = [tA](QsB + [avf(QeB.x)](QeB)).
    BnCopy(XeB, QeB->x);
    if(!BnEccModMult2(Z, QsB, One, QeB, avfSm2(XeB, w), E))
        goto Exit;
    // QeB := [tA]QeB = [tA](QsB + [Xe,B]QeB) and check for at infinity
    if(!BnEccModMult(Z, Z, Ta, E))
        goto Exit;
    // Convert BIGNUM E to TPM2B E
    BnPointTo2B(outZ, Z, E);
    retVal = TPM_RC_SUCCESS;

    CURVE_FREE(E);
    return retVal;
}
```
10.2.11 CryptEccMain.c

10.2.11.1 Includes and Defines

```c
#include "Tpm.h"
#if ALG_ECC

This version requires that the new format for ECC data be used

#elif !USE_BN_ECC_DATA
#error "Need to SET USE_BN_ECC_DATA to YES in Implementation.h"
#endif
```

10.2.11.2 Functions

```c
#if SIMULATION
void
EccSimulationEnd(
  void
)
{
#if SIMULATION
  // put things to be printed at the end of the simulation here
#endif
}
#endif // SIMULATION
```

10.2.11.2.1 CryptEccInit()

This function is called at _TPM_Init().

```c
BOOL
CryptEccInit(
  void
)
{
  return TRUE;
}
```

10.2.11.2.2 CryptEccStartup()

This function is called at TPM2_Startup().

```c
BOOL
CryptEccStartup(
  void
)
{
  return TRUE;
}
```

10.2.11.2.3 ClearPoint2B(generic)

Initialize the size values of a TPMS_ECC_POINT structure.

```c
void
ClearPoint2B(
  TPMS_ECC_POINT *p
    // IN: the point
```
10.2.11.2.4 CryptEccGetParametersByCurveId()

This function returns a pointer to the curve data that is associated with the indicated curve ID. 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>curve with the indicated TPM_ECC_CURVE is not implemented</td>
</tr>
<tr>
<td>!= NULL</td>
<td>pointer to the curve data</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT const ECC_CURVE *
CryptEccGetParametersByCurveId(
    TPM_ECC_CURVE curveId // IN: the curveID
)
{
    int i;
    for(i = 0; i < ECC_CURVE_COUNT; i++)
    {
        if(eccCurves[i].curveId == curveId)
            return &eccCurves[i];
    }
    return NULL;
}
```

10.2.11.2.5 CryptEccGetKeySizeForCurve()

This function returns the key size in bits of the indicated curve.

```
LIB_EXPORT UINT16
CryptEccGetKeySizeForCurve(
    TPM_ECC_CURVE curveId // IN: the curve
)
{
    const ECC_CURVE *curve = CryptEccGetParametersByCurveId(curveId);
    UINT16 keySizeInBits;
    // keySizeInBits = (curve != NULL) ? curve->keySizeBits : 0;
    return keySizeInBits;
}
```

10.2.11.2.6 GetCurveData()

This function returns the a pointer for the parameter data associated with a curve.

```
const ECC_CURVE_DATA *
GetCurveData(
    TPM_ECC_CURVE curveId // IN: the curveId
)
{
    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”.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>curve exists and parameter returned</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>curve does not exist or parameter selector</td>
</tr>
</tbody>
</table>

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     if(curve != NULL)
101         {
102             switch(p)
103             {
104                 case 'p':
105                     parameter = CurveGetPrime(curve);
106                     break;
107                 case 'n':
108                     parameter = CurveGetOrder(curve);
109                     break;
case 'a':
    parameter = CurveGet_a(curve);
    break;

case 'b':
    parameter = CurveGet_b(curve);
    break;

case 'x':
    parameter = CurveGetGx(curve);
    break;

case 'y':
    parameter = CurveGetGy(curve);
    break;

case 'h':
    parameter = CurveGetCofactor(curve);
    break;

default:
    FAIL(FATAL_ERROR_INTERNAL);
    break;
}

// If not debugging and we get here with parameter still NULL, had better
// not try to convert so just return FALSE instead.
return (parameter != NULL) ? BnTo2B(parameter, &out->b, 0) : 0;

10.2.11.2.10 CryptCapGetECCCurve()

This function returns the list of implemented ECC curves.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>if no more ECC curve is available</td>
</tr>
<tr>
<td>NO</td>
<td>if there are more ECC curves not reported</td>
</tr>
</tbody>
</table>

TPMI_YES_NO

CryptCapGetECCurve(
    TPM_ECC_CURVE    curveID,  // IN: the starting ECC curve
    UINT32           maxCount,  // IN: count of returned curves
    TPML_ECC_CURVE  *curveList  // OUT: ECC curve list
)
{
    TPMI_YES_NO       more = NO;
    UINT16            i;
    UINT32            count = ECC_CURVE_COUNT;
    TPM_ECC_CURVE     curve;

    // Initialize output property list
    curveList->count = 0;

    // The maximum count of curves we may return is MAX_ECC_CURVES
    if(maxCount > MAX_ECC_CURVES) maxCount = MAX_ECC_CURVES;

    // Scan the eccCurveValues array
    for(i = 0; i < count; i++)
    {
        curve = CryptEccGetCurveByIndex(i);
        // If curveID is less than the starting curveID, skip it
        if(curve < curveID) continue;
        if(curveList->count < maxCount)
        {
            // If we have not filled up the return list, add more curves to
            // 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 = CryptEccGetParametersByCurvid(curveId);
183   if(curve != NULL)
184       return & (curve->sign);
185   else
186       return NULL;
187 }

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>r value computed</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>no r value computed</td>
</tr>
</tbody>
</table>

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))
return FALSE;

if (c != NULL) {
   if (!TEST_BIT((c & COMMIT_INDEX_MASK), gr.commitArray))
      return FALSE;

   // If it is the commit phase, use the current value of the commit counter
   if (c != NULL) {
      // if the array bit is not set, can't use the value.
      if (!TEST_BIT((*c & COMMIT_INDEX_MASK), gr.commitArray))
         return FALSE;

      // If it is the sign phase, figure out what the counter value was
      // when the commitment was made.
      // When gr.commitArray has less than 64K bits, the extra
      // bits of 'c' are used as a check to make sure that the
      // signing operation is not using an out of range count value
      t1 = (UINT16)currentCount;

      // If the lower bits of c are greater or equal to the lower bits of t1
      // then the upper bits of t1 must be one more than the upper bits
      // of 'c'
      if ((c & COMMIT_INDEX_MASK) >= (t1 & COMMIT_INDEX_MASK))
         // Since the counter is behind, reduce the current count
         // and set the counter to the value that was
         // present when the commitment was made
         currentCount = (currentCount & 0xffffffffffff0000) | *c;

   } // Marshal the count value to a TPM2B buffer for the KDF
   cntr.t.size = sizeof(currentCount);
   UINT64_TO_BYTE_ARRAY(currentCount, cntr.t.buffer);
   // Now can do the KDF to create the random value for the signing operation
   // During the creation process, we may generate an r that does not meet the
   // requirements of the random value.
   // want to generate a new r.
   r->t.size = n.t.size;
   for (iterations = 1; iterations < 1000000;)
      { int i;
        CryptKDFa(CONTEXT_INTEGRITY_HASH_ALG, &gr.commitNonce.b, COMMIT_STRING,
                   (TPM2B *)name, &cntr.b, n.t.size * 8, r->t.buffer,
                   &iterations, FALSE);
        // "random" value must be less than the prime
        if (UnsignedCompareB(r->b.size, r->b.buffer, n.t.size, n.t.buffer) >= 0)
           continue;

        // in this implementation it is required that at least bit
        // in the upper half of the number be set
        for (i = n.t.size / 2; i >= 0; i--)
           if ((r->b.buffer[i] != 0)
              return TRUE;

      } // return FALSE;
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.

```c
 UINT16
 CRYPTOMODULE
 void
 CryptCommit()
{
    UINT16 oldCount = (UINT16)gr.commitCounter;
    gr.commitCounter++;
    SET_BIT(oldCount & COMMIT_INDEX_MASK, gr.commitArray);
    return oldCount;
}
```

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.

```c
 void
 CRYPTOMODULE
 CryptEndCommit(UINT16 c) // IN: the counter value of the commitment
{
    ClearBit((c & COMMIT_INDEX_MASK), gr.commitArray, sizeof(gr.commitArray));
}
```

10.2.11.2.15 CryptEccGetParameters()

This function returns the ECC parameter details of the given curve.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>unsupported ECC curve ID</td>
</tr>
</tbody>
</table>

```c
 BOOL
 CRYPTOMODULE
 CryptEccGetParameters(TPM_ECC_CURVE curveId, // IN: ECC curve ID
                        TPMS_ALGORITHM_DETAIL_ECC *parameters // OUT: ECC parameters
) {
    const ECC_CURVE *curve = CryptEccGetParametersByCurveId(curveId);
    const ECC_CURVE_DATA *data;
    BOOL found = curve != NULL;
    if(found)
    {
        data = curve->curveData;
        parameters->curveID = curve->curveId;
        parameters->keySize = curve->keySizeBits;
        parameters->kdf = curve->kdf;
        parameters->sign = curve->sign;
        // BnTo2B(data->prime, &parameters->p.b, 0);
        BnTo2B(data->prime, &parameters->p.b, parameters->p.t.size);
        BnTo2B(data->a, &parameters->a.b, 0);
        BnTo2B(data->b, &parameters->b.b, 0);
        BnTo2B(data->base.x, &parameters->gX.b, parameters->p.t.size);
```
10.2.11.2.16 BnGetCurvePrime()

This function is used to get just the prime modulus associated with a curve.

```c
const bignum_t * BnGetCurvePrime (TPM_ECC_CURVE curveId)
{
    const ECC_CURVE_DATA *C = GetCurveData(curveId);
    return (C != NULL) ? CurveGetPrime(C) : NULL;
}
```

10.2.11.2.17 BnGetCurveOrder()

This function is used to get just the curve order.

```c
const bignum_t * BnGetCurveOrder (TPM_ECC_CURVE curveId)
{
    const ECC_CURVE_DATA *C = GetCurveData(curveId);
    return (C != NULL) ? CurveGetOrder(C) : NULL;
}
```

10.2.11.2.18 BnIsOnCurve()

This function checks if a point is on the curve.

```c
BOOL BnIsOnCurve (pointConst Q, const ECC_CURVE_DATA *C)
{
    BN_VAR(right, (MAX_ECC_KEY_BITS * 3));
    BN_VAR(left, (MAX_ECC_KEY_BITS * 2));
    bigConst prime = CurveGetPrime(C);
    // Show that point is on the curve y^2 = x^3 + ax + b;
    // Or y^2 = x(x^2 + a) + b
    // y^2
    BnMult(left, Q->y, Q->y);
    BnMod(left, prime);
    // x^2
    BnMult(right, Q->x, Q->x);
    // x^2 + a
    BnAdd(right, right, CurveGet_a(C));
```
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 }

LIB_EXPORT BOOL
CryptEccIsValidPrivateKey(  
    TPM2B_ECC_PARAMETER     *d,  
    TPM_ECC_CURVE            curveId
)
383  {  
384      BN_INITIALIZED(bnD, MAX_ECC_PARAMETER_BYTES * 8, d);
385      return !BnEqualZero(bnD) && (BnUnsignedCmp(bnD, BnGetCurveOrder(curveId)) < 0);
386  }

10.2.11.2.20 BnPointMult()

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>result of multiplication is a point at infinity</td>
</tr>
<tr>
<td>TPM_RC_ECC_POINT</td>
<td>S or Q is not on the curve</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>d or u is not &lt; n</td>
</tr>
</tbody>
</table>

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
typedef bigCurve E;  // IN: curve parameters

{  
    BOOL OK;

    // Need one scalar
    OK = (d != NULL || u != NULL);

    // If S is present, then d has to be present. If S is not present, then d may or may not be present
    OK = OK && (((S == NULL) == (d == NULL)) || (d != NULL));

    // either both u and Q have to be provided or neither can be provided (don't know what to do if only one is provided.
    OK = OK && ((u == NULL) == (Q == NULL));

    OK = OK && (E != NULL);

    return (OK ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT);

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 <= d < q. This is the method of FIPS 186-4 Section B.4.1 “Key Pair Generation Using Extra Random Bits”.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure generating private key</td>
</tr>
</tbody>
</table>

BOOL BnEccGetPrivate(
    bigNum dOut,       // OUT: the qualified random value
    const ECC_CURVE_DATA *C,  // IN: curve for which the private key needs to be appropriate
    RAND_STATE *rand     // IN: state for DRBG
)
bigConst           order = CurveGetOrder(C);
BOOL                OK;
UINT32              orderBits = BnSizeInBits(order);
UINT32              orderBytes = BITS_TO_BYTES(orderBits);
BN_VAR(bnExtraBits, MAX_ECC_KEY_BITS + 64);
BN_VAR(nMinus1, MAX_ECC_KEY_BITS);

//
OK = BnGetRandomBits(bnExtraBits, (orderBytes * 8) + 64, rand);
OK = OK && BnSubWord(nMinus1, order, 1);
OK = OK && BnMod(bnExtraBits, nMinus1);
OK = OK && BnAddWord(dOut, bnExtraBits, 1);
return OK && !g_inFailureMode;

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.

BOOL BnEccGenerateKeyPair(
    bigNum               bnD,       // OUT: private scalar
    bn_point_t *ecQ,      // OUT: public point
    bigCurve             E,         // IN: curve for the point
    RAND_STATE *rand      // IN: DRBG state to use
)
{
    BOOL            OK = FALSE;
    // Get a private scalar
    OK = BnEccGetPrivate(bnD, AccessCurveData(E), rand);
    // Do a point multiply
    if(!OK)
        BnSetWord(ecQ->z, 0);
    else
        BnSetWord(ecQ->z, 1);
    return OK;
}

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.

LIB_EXPORT TPM_RC
CryptEccNewKeyPair(
    TPMS_ECC_POINT *Qout,       // OUT: the public point
    TPM2B_ECC_PARAMETER *dOut,  // OUT: the private scalar
    TPM_ECC_CURVE curveId       // IN: the curve for the key
)
{
    CURVE_INITIALIZED(E, curveId);
    POINT(ecQ);
    ECC_NUM(bnD);
    BOOL          OK;
    if(E == NULL)
        return TPM_RC_CURVE;
    TEST(TPM_ALG_ECDH);
    OK = BnEccGenerateKeyPair(bnD, ecQ, E, NULL);
### 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 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_ECC_POINT</td>
<td>the point ( Pin ) or ( Qin ) is not on the curve</td>
</tr>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>the product point is at infinity</td>
</tr>
<tr>
<td>TPM_RC_CURVE</td>
<td>bad curve</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>( dIn ) or ( uIn ) out of range</td>
</tr>
</tbody>
</table>

#### 505 LIB_EXPORT TPM_RC

**CryptEccPointMultiply**

```c
507  TPM_ECC_POINT *Rout,          // OUT: the product point R
508  TPM_ECC_CURVE curveId,       // IN: the curve to use
509  TPM_ECC_POINT *Pin,          // IN: first point (can be null)
510  TPM2B_ECC_PARAMETER *dIn,    // IN: scalar value for [dIn]Qin
511  //     the Pin
512  TPM_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  if(retVal == TPM_RC_SUCCESS)
527     BnPointTo2B(Rout, ecR, E);
528  else
529      ClearPoint2B(Rout);
530  CURVE_FREE(E);
```
return retval;
}

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
\[ y^2 \mod p = x^3 + ax + b \mod p. \]

It is a fatal error if \( Q \) is not specified (is NULL).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>point is on curve</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>point is not on curve or curve is not supported</td>
</tr>
</tbody>
</table>

LIB_EXPORT BOOL CryptEccIsPointOnCurve(
    TPM_ECC_CURVE curveId, // IN: the curve selector
    TPMS_ECC_POINT *Qin   // IN: the point.
)
{
    const ECC_CURVE_DATA *C = GetCurveData(curveId);
    POINT_INITIALIZED(ecQ, Qin);
    BOOL OK = (C != NULL && (BnIsOnCurve(ecQ, C)));
    return OK;
}

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 <= 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CURVE</td>
<td>curve is not supported</td>
</tr>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>could not verify key with signature (FIPS only)</td>
</tr>
</tbody>
</table>

LIB_EXPORT TPM_RC CryptEccGenerateKey(
    TPMT_PUBLIC *publicArea, // IN/OUT: The public area template for
    TPMT_SENSITIVE *sensitive, // the new key. The public key
    RAND_STATE *rand          // OUT: the sensitive area will be
    // area will be replaced computed
    // ECC public key
    // updated to contain the private
    // ECC key and the symmetric
    // encryption key
    // IN: if not NULL, the deterministic
    // RNG state
)
{
    CURVE_INITIALIZED(E, publicArea->parameters.eccDetail.curveID);
ECC_NUM(bnD);
POINT(ecQ);
BOOL OK;
TPM_RC retVal;

// TEST(TPM_ALG_ECDSA); // ECDSA is used to verify each key

// Validate parameters
if(E == NULL)
   ERROR_RETURN(TPM_RC_CURVE);

publicArea->unique.ecc.x.t.size = 0;
publicArea->unique.ecc.y.t.size = 0;
sensitive->sensitive.ecc.t.size = 0;

OK = BnEccGenerateKeyPair(bnD, ecQ, E, rand);
if(OK)
{
   BnPointTo2B(&publicArea->unique.ecc, ecQ, E);
   BnTo2B(bnD, &sensitive->sensitive.ecc.b, publicArea->unique.ecc.x.t.size);
}

#if FIPS_COMPLIANT
   // See if PWCT is required
   if(OK && IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
   {
      ECC_NUM(bnT);
      ECC_NUM(bnS);
      TPM2B_DIGEST digest;
      // TEST(TPM_ALG_ECDSA);
      digest.t.size = MIN(sensitive->sensitive.ecc.t.size, sizeof(digest.t.buffer));
      // Get a random value to sign using the built in DRBG state
      DRBG_Generate(NULL, digest.t.buffer, digest.t.size);
      if(g_inFailureMode)
         return TPM_RC_FAILURE;
      BnSignEcdsa(bnT, bnS, E, bnD, &digest, NULL);
      // and make sure that we can validate the signature
      OK = BnValidateSignatureEcdsa(bnT, bnS, E, ecQ, &digest) == TPM_RC_SUCCESS;
   }
#endif

retVal = (OK) ? TPM_RC_SUCCESS : TPM_RC_NO_RESULT;
Exit:
   CURVE_FREE(E);
   return retVal;

#endif // ALG_ECC
10.2.12 CryptEccSignature.c

10.2.12.1 Includes and Defines

```c
#include "Tpm.h"
#include "CryptEccSignature_fp.h"
#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.

```c
static bigNum
EcdsaDigest(
    bigNum               bnD,                     // OUT: the adjusted digest
    const TPM2B_DIGEST  *digest,               // IN: digest to adjust
    bigConst             max                      // IN: value that indicates the maximum
) {
    int bitsInMax = BnSizeInBits(max);
    // if (digest == NULL)
    BnSetWord(bnD, 0);
    else {
        BnFromBytes(bnD, digest->t.buffer,
                    (NUMBYTES)MIN(digest->t.size, BITS_TO_BYTES(bitsInMax)));
        shift = BnSizeInBits(bnD) - bitsInMax;
        if(shift > 0)
            BnShiftRight(bnD, bnD, shift);
        return bnD;   
    }
}
```

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: \( s = k + r \cdot d \pmod{n} \)

a) \( s \) is the signature
b) \( k \) is a random value
c) \( r \) is the value to sign
d) \( d \) is the private EC key
e) \( n \) 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

```c
static TPM_RC
BnSchnorrSign(
    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     // Need a local temp value to store the intermediate computation because product
36     // size can be larger than will fit in bnS.
37     BN_VAR(bnT1, MAX_ECC_PARAMETER_BYTES * 2 * 8);
38     // Reduce bnR without changing the input value
39     if(!BnEqualZero(bnT1))
40         return TPM_RC_NO_RESULT;
41     // compute s = (k + r * d)(mod n)
42     BnMult(bnT1, bnT1, bnD);
43     BnAdd(bnT1, bnT1, bnK);
44     BnDiv(NULL, bnS, bnT1, bnN);
45     return (BnEqualZero(bnS)) ? TPM_RC_NO_RESULT : TPM_RC_SUCCESS;

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         bigNum          bnD,   // IN: private signing key
59         const TPM2B_DIGEST *digest, // IN: the digest to sign
60         RAND_STATE      *rand   // IN: used in debug of signing
61 )
62     {
63         ECC_NUM(bnK);
64         ECC_NUM(bnIk);
65         BN_VAR(bnE, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE) * 8);
66         POINT(ecR);
67         bigConst        order = CurveGetOrder(AccessCurveData(E));
68         TPM_RC          retVal = TPM_RC_SUCCESS;
69         INT32            tries = 10;
70         BOOL            OK = FALSE;
71     // pAssert(digest != NULL);
72     // The algorithm as described in "Suite B Implementer's Guide to FIPS
73     // 186-3 (ECDSA)"
74     // 1. Use one of the routines in Appendix A.2 to generate (k, k^-1), a
75     // per-message secret number and its inverse modulo n. Since n is prime,
76     // the output will be invalid only if there is a failure in the RBG.
77     // 2. Compute the elliptic curve point R = [k]G = (xR, yR) using EC scalar
78     // multiplication (see [Routines]), where G is the base point included in
79     // the set of domain parameters.
80     // 3. Compute r = xR mod n. If r = 0, then return to Step 1.1.
81     // 4. Use the selected hash function to compute H = Hash(M).
82     // 5. Convert the bit string H to an integer e as described in Appendix B.2.
83     // 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  } // !_OK
110              
111         goto Exit;
112  
113         EcdsaDigest(bnE, digest, order);
114         
115         // now have inverse of K (bnIk), e (bnE), r (bnR), d (bnD) and
116         // CurveGetOrder(E)
117         // Compute s = k^-1 (e + r*d) (mod q)
118         // first do s = r*d mod q
119         BnModMult(bnS, bnR, bnD, order);
120         // s = e + s = e + r * d
121         BnAdd(bnS, bnE, bnS);
122         // s = k^-1s (mod n) = k^-1(e + r * d) (mod n)
123         BnModMult(bnS, bnIk, bnS, order);
124         
125         // If S is zero, try again
126         while(BnEqualZero(bnS));
127  Exit:
128         return retVal;
129  
130  #if ALG_ECDAA

10.2.12.3.2  BnSignEcdaa()

This function performs $s = r + T^* d \mod q$ where

a) $r$ is a random, or pseudo-random value created in the commit phase

b) $nonceK$ is a TPM-generated, random value $0 < nonceK < n$

(c) $T$ is mod $q$ of $\text{Hash}(nonceK || digest)$, and

d) $d$ is a private key.

The signature is the tuple $(nonceK, s)$

Regrettably, the parameters in this function kind of collide with the parameter names used in ECSCHNORR making for a lot of confusion.
**Error Returns**  |  **Meaning**  
--- | ---  
TPM_RC_SCHEME  |  unsupported hash algorithm  
TPM_RC_NO_RESULT  |  cannot get values from random number generator

```c
static TPM_RC
BnSignEcdaa(
  TPM2B_ECC_PARAMETER     *nonceK,  // OUT: 'nonce' component of the signature
  bigNum                   bnS,    // OUT: 's' component of the signature
  bigCurve                 E,      // IN: the curve used in signing
  bigNum                   bnD,    // IN: the private key
  const TPM2B_DIGEST      *digest, // IN: the value to sign (mod 'q')
  TPMT_ECC_SCHEME         *scheme,  // IN: signing scheme (contains the
       // commit count value).
  OBJECT                  *eccKey,  // IN: The signing key
  RAND_STATE              *rand    // IN: a random number state
)

TPM_RC                   retVal;
TPM2B_ECC_PARAMETER      r;
HASH_STATE               state;
TPM2B_DIGEST             T;
BN_MAX(bnT);

NOT_REFERENCED(rand);
if(!CryptGenerateR(&r, &scheme->details.ecdaa.count,
  eccKey->publicArea.parameters.eccDetail.curveID,
  &eccKey->name))
  retVal = TPM_RC_VALUE;
else
  {  // This allocation is here because 'r' doesn't have a value until
      // CryptGenerateR() is done.
    ECC_INITIALIZED(bnR, &r);
    do
      {  // generate nonceK such that 0 < nonceK < n
          // use bnT as a temp.
        if(!BnEccGetPrivate(bnT, AccessCurveData(E), rand))
          {
            retVal = TPM_RC_NO_RESULT;
            break;
          }
        BnTo2B(bnT, &nonceK->b, 0);  
        T.t.size = CryptHashStart(&state, scheme->details.ecdaa.hashAlg);
        if(T.t.size == 0)
          {  
            retVal = TPM_RC_SCHEME;
          }
        else
          {  
            CryptDigestUpdate2B(&state, &nonceK->b);
            CryptDigestUpdate2B(&state, &digest->b);
            CryptHashEnd2B(&state, &T.b);
            BnFrom2B(bnT, &T.b);
            // Watch out for the name collisions in this call!!
            retVal = BnSchnorrSign(bnS, bnR, bnT, bnD,
                              AccessCurveData(E)->order);
          }
        }  
      while(retVal == TPM_RC_NO_RESULT);
    // Because the rule is that internal state is not modified if the command
      // fails, only end the commit if the command succeeds.
    // NOTE that if the result of the Schnorr computation was zero
```
// it will probably not be worthwhile to run the same command again because
// the result will still be zero. This means that the Commit command will
// need to be run again to get a new commit value for the signature.
if(retVal == TPM_RC_SUCCESS)
    CryptEndCommit(scheme->details.ecdaa.count);

return retVal;
#endif // ALG_ECDAA
#if ALG_ECSCHNORR

10.2.12.3.3 SchnorrReduce()

Function to reduce a hash result if it's magnitude is too large. The size of number is set so that it has no
more bytes of significance than reference value. If the resulting number can have more bits of
significance than reference.

static void
SchnorrReduce(
    TPM2B       *number, // IN/OUT: Value to reduce
    bigConst     reference // IN: the reference value
)
{
    UINT16      maxBytes = (UINT16)BITS_TO_BYTES(BnSizeInBits(reference));
    if(number->size > maxBytes)
        number->size = maxBytes;
}

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
a) \((xR, yR) = [k]G\)
b) \(r = \text{Hash}(xR || P) \pmod q\)
c) \(rT = \text{truncated } r\)
d) \(s = k + rT * ds \pmod q\)
e) return the tuple \(rT, s\)

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>failure in the Schnorr sign process</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>hashAlg can't produce zero-length digest</td>
</tr>
</tbody>
</table>

static TPM_RC
BnSignEcSchnorr(
    bigNum                   bnR,   // OUT: 'r' component of the signature
    bigNum                   bnS,   // OUT: 's' component of the signature
    bigCurve                 E,     // IN: the curve used in signing
    bigNum                   bnD,   // IN: the signing key
    const TPM2B_DIGEST      *digest, // IN: the digest to sign
    TPM_ALG_ID               hashAlg, // IN: signing scheme (contains a hash)
    RAND_STATE              *rand    // IN: non-NULL when testing

    )
{
    HASH_STATE hashState;
    digestSize = CryptHashGetDigestSize(hashAlg);
    TPM2B_TYPE(T, MAX(DIGEST_SIZE, ECC_KEY_BYTES));
TPM2B_T T2b;
*e = &T2b.b;

TPM_RC
retVal = TPM_RC_NO_RESULT;

const ECC_CURVE_DATA *C;
bigConst order;
bigConst prime;

ECC_NUM(bnK);
POINT(ecR);

// Parameter checks
if (E == NULL)
ERROR_RETURN(TPM_RC_VALUE);

C = AccessCurveData(E);
order = CurveGetOrder(C);
prime = CurveGetOrder(C);

// If the digest does not produce a hash, then null the signature and return
// a failure.
if (digestSize == 0)
{
BnSetWord(bnR, 0);
BnSetWord(bnS, 0);
ERROR_RETURN(TPM_RC_SCHEME);
}

do
{
// Generate a random key pair
if (!BnEccGenerateKeyPair(bnK, ecR, E, rand))
break;

// Convert R.x to a string
BnTo2B(ecR->x, e, (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(prime)));

// f) compute r = \text{Hash}(e || P) \pmod n
CryptHashStart(&hashState, hashAlg);
CryptDigestUpdate2B(&hashState, e);
CryptDigestUpdate2B(&hashState, &digest->b);
e->size = CryptHashEnd(&hashState, digestSize, e->buffer);
// Reduce the hash size if it is larger than the curve order
SchnorrReduce(e, order);

BnFrom2B(bnR, e);

// Do the Schnorr computation
retVal = BnSchnorrSign(bnS, bnK, bnR, bnD, CurveGetOrder(C));
}
while(retVal == TPM_RC_NO_RESULT);

Exit:
return retVal;

#endif // ALG_ECSCHNORR
#endif

10.2.12.3.5 BnHexEqual()

This function compares a bignum value to a hex string.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>values equal</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>values not equal</td>
</tr>
</tbody>
</table>

static BOOL BnHexEqual(
    bigNum
    bn,       //IN: big number value
10.2.12.3.6 \texttt{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 \(e\).

\begin{table}[h]
\begin{tabular}{|c|c|}
\hline
\textbf{Error Returns} & \textbf{Meaning} \\
\hline
TPM\_RC\_VALUE & bad curve \\
\hline
\end{tabular}
\end{table}

\begin{verbatim}
static TPM\_RC
BnSignEcSm2(
  bigNum                   bnR,       // OUT: 'r' component of the signature
  bigNum                   bnS,       // OUT: 's' component of the signature
  bigCurve                 E,        // IN: the curve used in signing
  bigNum                   bnD,       // IN: the private key
  const TPM\_2\_DIGEST      *digest, // IN: the digest to sign
  RAND\_STATE              *rand     // IN: random number generator (mostly for
                                       // debug)
)
{
  BN\_MAX\_INITIALIZED(bnE, digest); // Don't know how big digest might be
  ECC\_NUM(bnN);
  ECC\_NUM(bnK);
  ECC\_NUM(bnT); // temp
  POINT(\texttt{Q1});
  bigConst  order = (E != NULL)
    ? CurveGetOrder(\texttt{AccessCurveData(E)}) : NULL;
    // A3: Use random number generator to generate random number 1 <= k <= n-1;
    // NOTE: Ax: numbers are from the SM2 standard
  loop:
    {
      // Get a random number 0 < k < n
      BnGenerateRandomInRange(bnK, order, rand);
      // A4: Figure out the point of elliptic curve \((x1, y1)=[k]G\), and according
      // to details specified in 4.2.7 in Part 1 of this document, transform the
      // data type of x1 into an integer:
      if (!BnEccModMult(\texttt{Q1}, NULL, bnK, E))
        goto loop;
      // A5: Figure out \('r' = ('e' + 'x1') mod 'n',
      BnAdd(bnR, bnE, Q1->x);
      BnMod(bnR, order);
      // A6: Figure out \('s' = ('e' - 'r') mod 'n',
      BnAdd(bnS, bnE, -bnR->x);
      BnMod(bnS, order);
    }
  return (BnUnsignedCmp(bn, bnC) == 0);
}
#endif // _SM2\_SIGN\_DEBUG
\end{verbatim}
329    "94F79FB1EED2CAA55BACDB49C4E755D1"));
330  
331  #endif
332  // if r=0 or r+k=n, return to A3;
333  if (BnEqualZero(bnR))
334     goto loop;
335  BnAdd(bnT, bnK, bnR);
336  if (BnUnsignedCmp(bnT, bnN) == 0)
337     goto loop;
338  // A6: Figure out s = ((1 + dA)^-1  (k - r  dA)) mod n,
339  // if s=0, return to A3;
340  // compute t = (1+dA)^-1
341  BnAddWord(bnT, bnD, 1);
342  BnModInverse(bnT, bnT, order);
343  #ifdef _SM2_SIGN_DEBUG
344     pAssert(BnHexEqual(bnT, "79BFCF3052C80DA7B939E0C6914A18CB"
345      "B2D96D8555256E83122743A7D4F5F956"));
346  #endif
347  // compute s = t * (k - r * dA) mod n
348  BnModMult(bnS, bnR, bnD, order);
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      "67A457872FB09EC56327A67EC7DEEEB7"));
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      "67A457872FB09EC56327A67EC7DEEEB7"));
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 ECDAA. 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SCHEME</td>
<td>scheme is not supported</td>
</tr>
</tbody>
</table>

LIB_EXPORT TPM_RC

cryptEccSign(
    TPMT_SIGNATURE *signature, // OUT: signature
    OBJECT *signKey, // IN: ECC key to sign the hash
    const TPM2B_DIGEST *digest, // IN: digest to sign
)
TPMT_ECC_SCHEME *scheme, // IN: signing scheme
RAN'D_STATE *rand
{
    CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
    ECC_INITIALIZED(bnD, &signKey->sensitive.sensitive.ecc.b);
    ECC_NUM(bnR);
    ECC_NUM(bnS);
    const ECC_CURVE_DATA *C;
    TPM_RC retVal = TPM_RC_SCHEME;

    NOT_REFERENCED(scheme);
    if (E == NULL)
        ERROR_RETURN(TPM_RC_VALUE);
    C = AccessCurveData(E);
    signature->signature.ecdaa.signatureR.t.size
        = sizeof(signature->signature.ecdaa.signatureR.t.buffer);
    signature->signature.ecdaa.signatureS.t.size
        = sizeof(signature->signature.ecdaa.signatureS.t.buffer);
    TEST(signature->sigAlg);
    switch(signature->sigAlg)
        {
    case ALG_ECDSA_VALUE:
        retVal = BnSignEcdsa(bnR, bnS, E, bnD, digest, rand);
        break;
    #if ALG_ECDAA
    case ALG_ECDAA_VALUE:
        retVal = BnSignEcdaa(&signature->signature.ecdaa.signatureR, bnS, E,
            bnD, digest, scheme, signKey, rand);
        bnR = NULL;
        break;
    #endif
    #if ALG_ECSCHNORR
    case ALG_ECSCHNORR_VALUE:
        retVal = BnSignEcSchnorr(bnR, bnS, E, bnD, digest,
            signature->signature.ecschnorr.hash, rand);
        break;
    #endif
    #if ALG_SM2
    case ALG_SM2_VALUE:
        retVal = BnSignEcSm2(bnR, bnS, E, bnD, digest, rand);
        break;
    #endif
    default:
        break;
    }

    // If signature generation worked, convert the results.
    if(retVal == TPM_RC_SUCCESS)
    {
        NUMBYTES orderBytes =
            (NUMBYTES)BITS_TO_BYTES(BnSizeInBits(CurveGetOrder(C)));
        if(bnR != NULL)
            BnTo2B(bnR, &signature->signature.ecdaa.signatureR.b, orderBytes);
        if(bnS != NULL)
            BnTo2B(bnS, &signature->signature.ecdaa.signatureS.b, orderBytes);
    }

    Exit:
    CURVE_FREE(E);
    return retVal;
    }
# if ALG_ECDSA
10.2.12.3.8 BnValidateSignatureEcdsa()

This function validates an ECDSA signature. \( rIn \) and \( sIn \) should have been checked to make sure that they are in the range \( 0 < v < n \)

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>signature not valid</td>
</tr>
</tbody>
</table>

```c
TPM_RC
BnValidateSignatureEcdsa(
        bigNum                bnR,  // IN: 'r' component of the signature
        bigNum                bnS,  // IN: 's' component of the signature
        bigCurve              E,    // IN: the curve used in the signature
        bn_point_t            *ecQ, // IN: the public point of the key
        const TPM2B_DIGEST    *digest // IN: the digest that was signed
    )
{
    // Make sure that the allocation for the digest is big enough for a maximum
    // digest
    BN_VAR(bnE, MAX(MAX_ECC_KEY_BYTES, MAX_DIGEST_SIZE) * 8);
    POINT(ecR);
    ECC_NUM(bnU1);
    ECC_NUM(bnU2);
    ECC_NUM(bnW);
    bigConst                 order = CurveGetOrder(AccessCurveData(E));
    TPM_RC                   retVal = TPM_RC_SIGNATURE;
    // Get adjusted digest
    EcdsaDigest(bnE, digest, order);
    // 1. If r and s are not both integers in the interval [1, n - 1], output
    //    INVALID.
    // 2. Use the selected hash function to compute \( H0 = \text{Hash}(M0) \).
    // 3. Convert the bit string \( H0 \) to an integer \( p \) as described in Appendix B.2.
    // 4. Compute \( w = (s')^-1 \mod n \), using the routine in Appendix B.1.
    if(!BnModInverse(bnW, bnS, order))
        goto Exit;
    // 5. Compute \( u1 = (e' \times w) \mod n \), and compute \( u2 = (r' \times w) \mod n \).
    BnModMult(bnU1, bnE, bnW, order);
    BnModMult(bnU2, bnR, bnW, order);
    // 6. Compute the elliptic curve point \( R = (xR, yR) = u1G+u2Q \), using EC
    // 7. scalar multiplication and EC addition (see [Routines]). If R is equal to
    //    the point at infinity \( O \), output INVALID.
    if(BnPointMult(ecR, CurveGetG(AccessCurveData(E)), bnU1, ecQ, bnU2, E)
        != TPM_RC_SUCCESS)
        goto Exit;
    // 8. Compare \( v \) and \( r0 \). If \( v = r0 \), output VALID; otherwise, output INVALID
    if(BnUnsignedCmp(ecR->x, bnR) != 0)
        goto Exit;
    retVal = TPM_RC_SUCCESS;
    Exit:
    return retVal;
}
```

#pragma ALG_ECDSA
#pragma ALG_SM2
10.2.12.3.9  BnValidateSignatureEcSm2()

This function is used to validate an SM2 signature.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>signature not valid</td>
</tr>
</tbody>
</table>

```c
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     bigPoint ecQ,     // IN: the public point of the key
498     const TPM2B_DIGEST *digest // IN: the digest that was signed
499   )
500 {
501     POINT(P);
502     ECC_NUM(bnRp);
503     ECC_NUM(bnT);
504     BN_MAX_INITIALIZED(bnE, digest);
505     BOOL          OK;
506     bigConst      order = CurveGetOrder(AccessCurveData(E));
507     #ifdef _SM2_SIGN_DEBUG
508     // Make sure that the input signature is the test signature
509     pAssert(BnHexEqual(bnR,
510            "40F1EC59F793D9F49E09DCEF49130D41"
511            "94F79FB1EE02C5594BACDB49C4E755D1"));
512     pAssert(BnHexEqual(bnS,
513            "6FC6DAC32C5D5CF10C777FB20F7C2EB6"
514            "67A457872FB095C6327A67EC7DEEBE7"));
515     #endif
516     // b) compute t := (r + s) mod n
517     BnAdd(bnT, bnR, bnS);
518     BnMod(bnT, order);
519     #ifdef _SM2_SIGN_DEBUG
520     pAssert(BnHexEqual(bnT,
521            "2B75F07ED7ECE7CCC1C8986B991F441A"
522            "D324D619FE06DD63ED32E0C997C801"));
523     #endif
524     // c) verify that t > 0
525     OK = !BnEqualZero(bnT);
526     if(!OK)
527     // set T to a value that should allow rest of the computations to run
528     // without trouble
529     BnCopy(bnT, bnS);
530     // d) compute (x, y) := [s]G + [t]Q
531     OK = BnEccModMult2(P, NULL, bnS, ecQ, bnT, E);
532     #ifdef _SM2_SIGN_DEBUG
533     pAssert(OK && BnHexEqual(P->x,
534            "110FCDA57615705D5E7B9324AC4B856D"
535            "23E6D9188B2AE47759514657CE25D112"));
536     #endif
537     // e) compute r' := (e + x) mod n (the x coordinate is in bnT)
538     OK = OK && BnAdd(bnRp, bnE, P->x);
539     OK = OK && BnMod(bnRp, order);
540     // f) verify that r' = r
541     OK = OK && (BnUnsignedCmp(bnR, bnRp) == 0);
542     if(!OK)
543     return TPM_RC_SIGNATURE;
544   else
545     return TPM_RC_OK;
```
return TPM_RC_SUCCESS;
#endif  // ALG_SM2
#endif  // ALG_ECSCHNORR

10.2.12.3.10 BnValidateSignatureEcSchnorr()

This function is used to validate an EC Schnorr signature.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>signature not valid</td>
</tr>
</tbody>
</table>

static TPM_RC
BnValidateSignatureEcSchnorr(
bigNum bnR,  // IN: 'r' component of the signature
bigNum bnS,  // IN: 's' component of the signature
TPM_ALG_ID hashAlg,  // IN: hash algorithm of the signature
bigCurve E,  // IN: the curve used in the signature
bigPoint ecQ,  // IN: the public point of the key
const TPM2B_DIGEST *digest  // IN: the digest that was signed
)
{
BN_MAX(bnRn);
POINT(ecE);
BN_MAX(bnEx);
const ECC_CURVE_DATA *C = AccessCurveData(E);
bigConst order = CurveGetOrder(C);
UINT16 digestSize = CryptHashGetDigestSize(hashAlg);
HASH_STATE hashState;
TPM2B_TYPE(BUFFER, MAX(MAX_ECC_PARAMETER_BYTES, MAX_DIGEST_SIZE));
TPM2B_BUFFER Ex2 = {{sizeof(Ex2.t.buffer), {0}}};
BOOL OK;

// E = [s]G - [r]Q
BnMod(bnR, order);
// Make -r = n - r
BnSub(bnRn, order, bnR);
// E = [s][G] + [-r]Q
OK = BnPointMult(ecE, CurveGetG(C), bnS, ecQ, bnRn, E) == TPM_RC_SUCCESS;
// reduce the x portion of E mod q
// OK = OK && BnMod(ecE->x, order);
// Convert to byte string
OK = OK && BnTo2B(ecE->x, &Ex2.b,
(NUMBYTES)(BITS_TO_BYTES(BnSizeInBits(order))));
if(OK)
{
// Ex = h(pE.x || digest)
CryptHashStart(&hashState, hashAlg);
CryptDigestUpdate(&hashState, Ex2.t.size, Ex2.t.buffer);
CryptDigestUpdate(&hashState, digest->t.size, digest->t.buffer);
Ex2.t.size = CryptHashEnd(&hashState, digestSize, Ex2.t.buffer);
SchnorrReduce(&Ex2.b, order);
BnFrom2B(bnEx, &Ex2.b);
// see if Ex matches R
OK = BnUnsignedCmp(bnEx, bnR) == 0;
}
return (OK) ? TPM_RC_SUCCESS : TPM_RC_SIGNATURE;
}
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>not a valid signature</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT TPM_RC
CryptEccValidateSignature(
    TPM_T_SIGNATURE *signature, // IN: signature to be verified
    OBJECT *signKey,           // IN: ECC key signed the hash
    const TPM2B_DIGEST *digest  // IN: digest that was signed
)
{
    CURVE_INITIALIZED(E, signKey->publicArea.parameters.eccDetail.curveID);
    ECC_NUM(bnR);
    ECC_NUM(bnS);
    POINT_INITIALIZED(ecQ, &signKey->publicArea.unique.ecc);
    bigConst order;
    TPM_RC retVal;

    if (E == NULL)
        ERROR_RETURN(TPM_RC_VALUE);

    order = CurveGetOrder(AccessCurveData(E));

    // Make sure that the scheme is valid
    switch(signature->sigAlg)
    {
        case ALG_ECDSA_VALUE:
            #if ALG_ECSCHNORR
                case ALG_ECSCHNORR_VALUE:
            #endif
        #if ALG_SM2
            case ALG_SM2_VALUE:
        #endif
            break;
        default:
            ERROR_RETURN(TPM_RC_SCHEME);
            break;
    }

    // Can convert r and s after determining that the scheme is an ECC scheme. If
    // this conversion doesn't work, it means that the unmarshaling code for
    // an ECC signature is broken.
    BnFrom2B(bnR, &signature->signature.ecdsa.signatureR.b);
    BnFrom2B(bnS, &signature->signature.ecdsa.signatureS.b);

    // r and s have to be greater than 0 but less than the curve order
    if (BnEqualZero(bnR) || BnEqualZero(bnS))
        ERROR_RETURN(TPM_RC_SIGNATURE);
    if ((BnUnsignedCmp(bnS, order) >= 0)
        || (BnUnsignedCmp(bnR, order) >= 0))
        ERROR_RETURN(TPM_RC_SIGNATURE);

    switch(signature->sigAlg)
    {
        case ALG_ECDSA_VALUE:
            retVal = BnValidateSignatureEcdsa(bnR, bnS, E, ecQ, digest);
            break;
        #if ALG_ECSCHNORR
            case ALG_ECSCHNORR_VALUE:
                retVal = BnValidateSignatureEcSchnorr(bnR, bnS,
```
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 \( \text{curveId} \). 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>if ( K ), ( L ) or ( E ) was computed to be the point at infinity</td>
</tr>
<tr>
<td>TPM_RC_CANCELED</td>
<td>a cancel indication was asserted during this function</td>
</tr>
</tbody>
</table>

\[
\text{LIB_EXPORT TPM_RC CryptEccCommitCompute(}
\begin{align*}
\text{TPMS_ECC_POINT} & \quad *K, & \text{// OUT: } [d]B \text{ or } [r]Q \\
\text{TPMS_ECC_POINT} & \quad *L, & \text{// OUT: } [r]B \\
\text{TPMS_ECC_POINT} & \quad *E, & \text{// OUT: } [r]M \\
\text{TPM_ECC_CURVE} & \quad \text{curveId}, & \text{// IN: the curve for the computations} \\
\text{TPMS_ECC_POINT} & \quad *M, & \text{// IN: } M \text{ (optional)} \\
\text{TPMS_ECC_POINT} & \quad *B, & \text{// IN: } B \text{ (optional)} \\
\text{TPM2B_ECC_PARAMETER} & \quad *d, & \text{// IN: } d \text{ (optional)} \\
\text{TPM2B_ECC_PARAMETER} & \quad *r & \text{// IN: the computed } r \text{ value (required)} \\
\end{align*}
\]

\[
\text{\{ }
\begin{align*}
\text{CURVE_INITIALIZED(curve, curveId);} & \quad \text{// Normally initialize } E \text{ as the curve, but} \\
\text{ECC_INITIALIZED(bnR, r);} & \quad \text{// } E \text{ means something else in this function} \\
\text{TPM_RC} & \quad \text{retVal = TPM_RC_SUCCESS;} \\
\end{align*}
\]

\[
\text{// Validate that the required parameters are provided.} \\
\text{// Note: } E \text{ has to be provided if computing } E := [r]Q \text{ or } E := [r]M. \text{ Will do} \\
\text{// } E := [r]Q \text{ if both } M \text{ and } B \text{ are NULL.} \\
\text{pAssert(r != NULL && E != NULL);} \\
\text{// Initialize the output points in case they are not computed} \\
\text{ClearPoint2B(K);} \\
\text{ClearPoint2B(L);} \\
\text{ClearPoint2B(E);} \\
\text{// Sizes of the } r \text{ parameter may not be zero} \\
\text{pAssert(r->t.size > 0);} \\
\text{// If } B \text{ is provided, compute } K=[d]B \text{ and } L=[r]B \\
\text{if(B != NULL)}}


    }  

    ECC_INITIALIZED(bnD, d);  
    POINT_INITIALIZED(pB, B);  
    POINT(pK);  
    POINT(pL);  

    //
    // pAssert(d != NULL && K != NULL && L != NULL);  

    if(!BnIsOnCurve(pB, AccessCurveData(curve)))  
        ERROR_RETURN(TPM_RC_VALUE);  
    // do the math for K = [d]B  
    if((retVal = BnPointMult(pK, pB, bnD, NULL, NULL, curve)) != TPM_RC_SUCCESS)  
        goto Exit;  
    // Convert BN K to TPM2B K  
    BnPointTo2B(K, pK, curve);  
    // compute L = [r]B after checking for cancel  
    if(_plat__IsCanceled())  
        ERROR_RETURN(TPM_RC_CANCELED);  
    // compute L = [r]B  
    if(!BnIsValidPrivateEcc(bnR, curve))  
        ERROR_RETURN(TPM_RC_VALUE);  
    if((retVal = BnPointMult(pL, pB, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)  
        goto Exit;  
    // Convert BN L to TPM2B L  
    BnPointTo2B(L, pL, curve);  

    if((M != NULL) || (B == NULL))  
    {  
        POINT_INITIALIZED(pM, M);  
        POINT(pE);  

        // Make sure that a place was provided for the result  
        pAssert(E != NULL);  

        // if this is the third point multiply, check for cancel first  
        if((B != NULL) && _plat__IsCanceled())  
            ERROR_RETURN(TPM_RC_CANCELED);  

        // If M provided, then pM will not be NULL and will compute E = [r]M.  
        // However, if M was not provided, then pM will be NULL and E = [r]G  
        // will be computed  
        if((retVal = BnPointMult(pE, pM, bnR, NULL, NULL, curve)) != TPM_RC_SUCCESS)  
            goto Exit;  
        // Convert E to 2B format  
        BnPointTo2B(E, pE, curve);  
    }  

    Exit:  
        CURVE_FREE(curve);  
        return retVal;  
    }  

#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

```c
#define _CRYPT_HASH_C_
#include "Tp.m"
#include "CryptHash_fp.h"
#include "CryptHash.h"
#include "OIDS.h"
#define HASH_TABLE_SIZE (HASH_COUNT + 1)
#if ALG_SHA1
HASH_DEF_TEMPLATE(SHA1, Sha1);
#endif
#if ALG_SHA256
HASH_DEF_TEMPLATE(SHA256, Sha256);
#endif
#if ALG_SHA384
HASH_DEF_TEMPLATE(SHA384, Sha384);
#endif
#if ALG_SHA512
HASH_DEF_TEMPLATE(SHA512, Sha512);
#endif
#if ALG_SM3_256
HASH_DEF_TEMPLATE(SM3_256, Sm3_256);
#endif
HASH_DEF NULL_Def = {{0}};
PHASH_DEF HashDefArray[] = {
#if ALG_SHA1
    &Sha1_Def,
#endif
#if ALG_SHA256
    &Sha256_Def,
#endif
#if ALG_SHA384
    &Sha384_Def,
#endif
#if ALG_SHA512
    &Sha512_Def,
#endif
#if ALG_SM3_256
    &Sm3_256_Def,
#endif
    &NULL_Def
};
```

10.2.13.3 Obligatory Initialization Functions

10.2.13.3.1 CryptHashInit()

This function is called by _TPM_Init() do perform the initialization operations for the library.

```c
BOOL CryptHashInit()
{
    void
}
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.

```c
BOOL CryptHashStartup(
    void
)
{
    int i = sizeof(HashDefArray) / sizeof(PHASH_DEF) - 1;
    return (i == HASH_COUNT);
}
```

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.

```c
PHASH_DEF CryptGetHashDef(
    TPM_ALG_ID       hashAlg,
)
{
    INT16 i;
    #define HASHES (sizeof(HashDefArray) / sizeof(PHASH_DEF))
    for(i = 0; i < HASHES; i++)
    {
        PHASH_DEF p = HashDefArray[i];
        if(p->hashAlg == hashAlg)
            return p;
    }
    return &NULL_Def;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td><code>hashAlg</code> is a valid, implemented hash on this TPM</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td><code>hashAlg</code> is not valid for this TPM</td>
</tr>
</tbody>
</table>

```c
BOOL CryptHashIsValidAlg(
    TPM_ALG_ID       hashAlg, // IN: the algorithm to check
)
BOOL flag // IN: TRUE if TPM_ALG_NULL is to be treated
    // as a valid hash

if (hashAlg == TPM_ALG_NULL)
    return flag;
return CryptGetHashDef(hashAlg) != &NULL_Def;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_ALG_xxx</td>
<td>a hash algorithm</td>
</tr>
<tr>
<td>TPM_ALG_NULL</td>
<td>this can be used as a stop value</td>
</tr>
</tbody>
</table>

LIB_EXPORT TPM_ALG_ID
CryptHashGetAlgByIndex(
    UINT32           index // IN: the index
)
{
    TPM_ALG_ID       hashAlg;
    if (index >= HASH_COUNT)
        hashAlg = TPM_ALG_NULL;
    else
        hashAlg = HashDefArray[index] - > hashAlg;
    return hashAlg;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the digest size</td>
</tr>
</tbody>
</table>

LIB_EXPORT UINT16
CryptHashGetDigestSize(
    TPM_ALG_ID   hashAlg // IN: hash algorithm to look up
)
{
    return CryptGetHashDef(hashAlg) - > digestSize;
}

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.
<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the digest size</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT UINT16
CryptHashGetBlockSize(
    TPM_ALG_ID hashAlg // IN: hash algorithm to look up
)
{
    return CryptGetHashDef(hashAlg)->blockSize;
}
```

### 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.

```c
LIB_EXPORT const BYTE *
CryptHashGetOid(
    TPM_ALG_ID hashAlg
)
{
    return CryptGetHashDef(hashAlg)->OID;
}
```

### 10.2.13.4.8 CryptHashGetContextAlg()

This function returns the hash algorithm associated with a hash context.

```c
TPM_ALG_ID
CryptHashGetContextAlg(
    PHASH_STATE state // IN: the context to check
)
{
    return state->hashAlg;
}
```

### 10.2.13.5 State Import and Export

#### 10.2.13.5.1 CryptHashCopyState

This function is used to clone a HASH_STATE.

```c
LIB_EXPORT void
CryptHashCopyState(
    HASH_STATE *out, // OUT: destination of the state
    const HASH_STATE *in // IN: source of the state
)
{
    pAssert(out->type == in->type);
    out->hashAlg = in->hashAlg;
    out->def = in->def;
    if(in->hashAlg != TPM_ALG_NULL)
    {
        HASH_STATE_COPY(out, in);
    }
    if(in->type == HASH_STATE_HMAC)
    {  
```
const HMAC_STATE *hIn = (HMAC_STATE *)in;
HMAC_STATE *hOut = (HMAC_STATE *)out;
hOut->hmacKey = hIn->hmacKey;
return;
}

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.

```c
void CryptHashExportState(
    PCHASH_STATE internalFmt, // IN: the hash state formatted for use by library
    PEXPORT_HASH_STATE externalFmt // OUT: the exported hash state
)
{
    BYTE *outBuf = (BYTE *)externalFmt;
    cAssert(sizeof(HASH_STATE) <= sizeof(EXPORT_HASH_STATE));
    // the following #define is used to move data from an aligned internal data structure to a byte buffer (external format data).
    #define CopyToOffset(value) memcpy(&outBuf[offsetof(HASH_STATE,value)], &internalFmt->value, sizeof(internalFmt->value))
    // Copy the hashAlg
    CopyToOffset(hashAlg);
    CopyToOffset(type);
    #ifdef HASH_STATE_SMAC
    if(internalFmt->type == HASH_STATE_SMAC)
    {
        memcpy(outBuf, internalFmt, sizeof(HASH_STATE));
        return;
    }
    #endif
    if(internalFmt->type == HASH_STATE_HMAC)
    {
        HMAC_STATE *from = (HMAC_STATE *)internalFmt;
        memcpy(&outBuf[offsetof(HASH_STATE, hmacKey)], &from->hmacKey, sizeof(from->hmacKey));
    }
    if(internalFmt->hashAlg != TPM_ALG_NULL)
    HASH_STATE_EXPORT(externalFmt, internalFmt);
}
```

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.

```c
void CryptHashImportState(
    PHASH_STATE internalFmt, // OUT: the hash state formatted for use by the library
    PCEXPORT_HASH_STATE externalFmt // IN: the exported hash state
)
{
    BYTE *inBuf = (BYTE *)externalFmt;
```
\#define CopyFromOffset(value) \ 
    memcpy(&internalFmt->value, &inBuf[offsetof(HASH_STATE,value)], \ 
    sizeof(internalFmt->value))

    // Copy the hashAlg of the byte-aligned input structure to the structure-aligned
    // internal structure.
    CopyFromOffset(hashAlg);
    CopyFromOffset(type);
    if(internalFmt->hashAlg != TPM_ALG_NULL)
    {
        \#ifdef HASH_STATE_SMAC
        if(internalFmt->type == HASH_STATE_SMAC)
        {
            memcpy(internalFmt, inBuf, sizeof(HASH_STATE));
            return;
        }
        \#endif
        internalFmt->def = CryptGetHashDef(internalFmt->hashAlg);
        HASH_STATE_IMPORT(internalFmt, inBuf);
        if(internalFmt->type == HASH_STATE_HMAC)
        {
            HMAC_STATE *to = (HMAC_STATE *)internalFmt;
            memcpy(&to->hmacKey, inBuf[offsetof(HMAC_STATE, hmacKey)],
            sizeof(to->hmacKey));
        }
    }
}

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.

static UINT16 HashEnd(
    PHASH_STATE hashState,   // IN: the hash state
    UINT32 dOutSize,         // IN: the size of receive buffer
    PBYTE dOut               // OUT: the receive buffer
)
{
    BYTE temp[MAX_DIGEST_SIZE];
    if((hashState->hashAlg == TPM_ALG_NULL)
    || (hashState->type != HASH_STATE_HASH))
    dOutSize = 0;
    if(dOutSize > 0)
    {
        hashState->def = CryptGetHashDef(hashState->hashAlg);
        // Set the final size
        dOutSize = MIN(dOutSize, hashState->def->digestSize);
        // Complete into the temp buffer and then copy
        HASH_END(hashState, temp);
        // Don't want any other functions calling the HASH_END method
        // directly.
        \#undef HASH_END
        memcpy(dOut, &temp, dOutSize);
        }
    hashState->type = HASH_STATE_EMPTY;
    return (UINT16)dOutSize;
}
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).

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hash is TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt;0</td>
<td>digest size</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT UINT16 CryptHashStart(
PHASH_STATE hashState,       // OUT: the running hash state
TPM_ALG_ID hashAlg           // IN: hash algorithm
)
{
    UINT16           retVal;
    TEST(hashAlg);
    hashState->hashAlg = hashAlg;
    if(hashAlg == TPM_ALG_NULL)
    {
        retVal = 0;
    }
    else
    {
        hashState->def = CryptGetHashDef(hashAlg);
        HASH_START(hashState);
        retVal = hashState->def->digestSize;
    }
    #undef HASH_START
    hashState->type = HASH_STATE_HASH;
    return retVal;
}
```

10.2.13.6.3 CryptDigestUpdate()

Add data to a hash or HMAC, SMAC stack.

```c
void CryptDigestUpdate(
PHASH_STATE hashState,       // IN: the hash context information
UINT32           dataSize,    // IN: the size of data to be added
const BYTE      *data         // IN: data to be hashed
)
{
    if(hashState->hashAlg != TPM_ALG_NULL)
    {
        if((hashState->type == HASH_STATE_HASH)
           || (hashState->type == HASH_STATE_HMAC))
            HASH_DATA(hashState, dataSize, (BYTE *)data);
        #if SMAC_IMPLEMENTED
        else if(hashState->type == HASH_STATE_SMAC)
            (hashState->state.smac.smacMethods.data)(hashState->state.smac.state,
                dataSize, data);
        #endif
    }
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>no data returned</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the number of bytes in the digest or <code>dOutSize</code>, whichever is smaller</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT UINT16 CryptHashEnd(
    PHASH_STATE hashState,       // IN: the state of hash stack
    UINT32 dOutSize,             // IN: size of digest buffer
    BYTE *dOut                   // OUT: hash digest
) {
    pAssert(hashState->type == HASH_STATE_HASH);
    return HashEnd(hashState, dOutSize, dOut);
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>number of bytes placed in <code>dOut</code></td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT UINT16 CryptHashBlock(
    TPM_ALG_ID hashAlg,       // IN: The hash algorithm
    UINT32 dataSize,          // IN: size of buffer to hash
    const BYTE *data,         // IN: the buffer to hash
    UINT32 dOutSize,          // IN: size of the digest buffer
    BYTE *dOut                // OUT: digest buffer
) {
    HASH_STATE state;
    CryptHashStart(&state, hashAlg);
    CryptDigestUpdate(&state, dataSize, data);
    return HashEnd(&state, dOutSize, dOut);
}
```

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.

```c
LIB_EXPORT void
CryptDigestUpdate2B(
    PHASH_STATE      state,    // IN: the digest state
    const TPM2B     *bIn       // IN: 2B containing the data
)
{
    // Only compute the digest if a pointer to the 2B is provided.
    // In CryptDigestUpdate(), if size is zero or buffer is NULL, then no change
    // to the digest occurs. This function should not provide a buffer if bIn is
    // not provided.
    pAssert(bIn != NULL);
    CryptDigestUpdate(state, bIn->size, bIn->buffer);
    return;
}
```

### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>the number of bytes placed in digest.buffer</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT UINT16
CryptHashEnd2B(
    PHASH_STATE      state,    // IN: the hash state
    P2B              digest    // IN: the size of the buffer Out: requested
)
{
    return CryptHashEnd(state, digest->size, digest->buffer);
}
```

### 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().

```c
LIB_EXPORT void
CryptDigestUpdateInt(
    void          *state,     // IN: the state of hash stack
    UINT32        intSize,    // IN: the size of 'intValue' in bytes
    UINT64        intValue    // IN: integer value to be hashed
)
{
#if LITTLE_ENDIAN_TPM
    intValue = REVERSE_ENDIAN_64(intValue);
#endif
    CryptDigestUpdate(state, intSize, &((BYTE *)&intValue)[8 - intSize]);
}
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>number of bytes in digest produced by hashAlg (may be zero)</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT UINT16 CryptHmacStart(
    PHMAC_STATE state, // IN/OUT: the state buffer
    TPM_ALG_ID hashAlg, // IN: the algorithm to use
    UINT16 keySize, // IN: the size of the HMAC key
    const BYTE *key // IN: the HMAC key
)
{
    PHASH_DEF hashDef;
    BYTE *pb;
    UINT32 i;

    // hashDef = CryptGetHashDef(hashAlg);
    if(hashDef->digestSize != 0)
    {
        // If the HMAC key is larger than the hash block size, it has to be reduced to fit. The reduction is a digest of the hashKey.
        if(keySize > hashDef->blockSize)
        {
            // if the key is too big, reduce it to a digest of itself
            state->hmacKey.t.size = CryptHashBlock(hashAlg, keySize, key,
                hashDef->digestSize,
                state->hmacKey.t.buffer);
        }
        else
        {
            memcpy(state->hmacKey.t.buffer, key, keySize);
            state->hmacKey.t.size = keySize;
        }
        // XOR the key with iPad (0x36)
        pb = state->hmacKey.t.buffer;
        for(i = state->hmacKey.t.size; i > 0; i--)
            *pb++ ^= 0x36;

        // if the keySize is smaller than a block, fill the rest with 0x36
        for(i = hashDef->blockSize - state->hmacKey.t.size; i > 0; i--)
            *pb++ ^= 0x36;

        // Increase the oPadSize to a full block
        state->hmacKey.t.size = hashDef->blockSize;

        // Start a new hash with the HMAC key
        // This will go in the caller’s state structure and may be a sequence or not
        CryptHashStart((PHASH_STATE)state, hashAlg);
        CryptDigestUpdate((PHASH_STATE)state, state->hmacKey.t.size,
            state->hmacKey.t.buffer);

        // XOR the key block with 0x5c ^ 0x36
        for(pb = state->hmacKey.t.buffer, i = hashDef->blockSize; i > 0; i--)
            *pb++ ^= (0x5c ^ 0x36);
    }
    // Set the hash algorithm
```

```c
    // Set the hash algorithm
```
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;= 0</td>
<td>number of bytes in dOut (may be zero)</td>
</tr>
</tbody>
</table>

LIB_EXPORT UINT16 CryptHmacEnd(
    PHMAC_STATE state,  // IN: the hash state buffer
    UINT32 dOutSize,   // IN: size of digest buffer
    BYTE *dOut);     // OUT: hash digest

#if SMAC_IMPLEMENTED
    if (hState->type == HASH_STATE_SMAC)
        return (state->hashState.state.smac.state,
                &state->hashState.state.smac.smacMethods.end)
            (&state->hashState.state.smac.state,
                dOutSize,
                dOut);
#endif

    pAssert(hState->type == HASH_STATE_HMAC);
    hState->def = CryptGetHashDef(hState->hashAlg);
    // Change the state type for completion processing
    hState->type = HASH_STATE_HASH;
    if (hState->hashAlg == TPM_ALG_NULL)
        dOutSize = 0;
    else
    {
        // Complete the current hash
        HashEnd(hState, hState->def->digestSize, temp);
        // Do another hash starting with the oPad
        CryptHashStart(hState, hState->hashAlg);
        CryptDigestUpdate(hState, state->hmacKey.t.size, state->hmacKey.t.buffer);
        CryptDigestUpdate(hState, hState->def->digestSize, temp);
    }
    return HashEnd(hState, dOutSize, dOut);
}

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.
### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>the number of bytes placed in digest</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT UINT16 CryptHmacEnd2B(
    PHMAC_STATE hmacState, // IN: the state of HMAC stack
    P2B digest, // OUT: HMAC
)
{
    return CryptHmacEnd(hmacState, digest->size, digest->buffer);
}
```

### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hash algorithm was TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>should be the same as mSize</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT UINT16 CryptMGF1(
    UINT32 mSize, // IN: length of the mask to be produced
    BYTE *mask, // OUT: buffer to receive the mask
    TPM_ALG_ID hashAlg, // IN: hash to use
    UINT32 seedSize, // IN: size of the seed
    BYTE *seed // IN: seed size
)
{
    HASH_STATE hashState;
    PHASH_DEF hDef = CryptGetHashDef(hashAlg);
    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     if(remaining <= hDef->digestSize)
491         break;
492     mask = &mask[hDef->digestSize];
493     counter++;
494 
495     return (UINT16)mSize;
496 }

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hash algorithm is not supported or is TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the number of bytes in the keyStream buffer</td>
</tr>
</tbody>
</table>

498 LIBEXPORT 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
UINT16 blocks // IN: If non-zero, this is the maximum number
// of blocks to be returned, regardless
// of sizeInBits
}

UINT32 counter = 0; // counter value
INT16 bytes; // number of bytes to produce
UINT16 generated; // number of bytes generated
BYTE *stream = keyStream;
HMAC_STATE hState;
UINT16 digestSize = CryptHashGetDigestSize(hashAlg);

pAssert(key != NULL && keyStream != NULL);

TEST(TPM_ALG_KDF1_SP800_108);
if(digestSize == 0)
    return 0;

if(counterInOut != NULL)
    counter = *counterInOut;

// If the size of the request is larger than the numbers will handle,
// it is a fatal error.
pAssert(((sizeInBits + 7) / 8) <= INT16_MAX);

// The number of bytes to be generated is the smaller of the sizeInBits bytes or
// the number of requested blocks. The number of blocks is the smaller of the
// number requested or the number allowed by sizeInBits. A partial block is
// a full block.
bytes = (blocks > 0) ? blocks * digestSize : (UINT16)BITS_TO_BYTES(sizeInBits);
generated = bytes;

// Generate required bytes
for(; bytes > 0; bytes -= digestSize)
{
    counter++;  // Start HMAC
    if(CryptHmacStart(&hState, hashAlg, key->size, key->buffer) == 0)
        return 0;
    // Adding counter
    CryptDigestUpdateInt(&hState.hashState, 4, counter);
    if(label != NULL)
        HASH_DATA(&hState.hashState, label->size, (BYTE *)label->buffer);
    // Add a null. SP108 is not very clear about when the 0 is needed but to
    // make this like the previous version that did not add an 0x00 after
    // a null-terminated string, this version will only add a null byte
    // if the label parameter did not end in a null byte, or if no label
    // is present.
    if((label == NULL)
        || (label->size == 0)
        || (label->buffer[label->size - 1] != 0))
        CryptDigestUpdateInt(&hState.hashState, 1, 0);
    // Adding contextU
    if(contextU != NULL)
        HASH_DATA(&hState.hashState, contextU->size, contextU->buffer);
    // Adding contextV
    if(contextV != NULL)
        HASH_DATA(&hState.hashState, contextV->size, contextV->buffer);
    // Adding size in bits
    CryptDigestUpdateInt(&hState.hashState, 4, sizeInBits);
}

// Complete and put the data in the buffer
CryptHmacEnd(&hState, bytes, stream);
stream = &stream[digestSize];

// Masking in the KDF is disabled. If the calling function wants something
// less than even number of bytes, then the caller should do the masking
// because there is no universal way to do it here
if(counterInOut != NULL)
    *counterInOut = counter;
return generated;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hash algorithm is not supported or is TPM_ALG_NULL</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the number of bytes in the keyStream buffer</td>
</tr>
</tbody>
</table>

LIB_EXPORT UINT16 CryptKDFe(
    TPM_ALG_ID hashAlg,    // IN: hash algorithm used in HMAC
    TPM2B *Z,            // IN: Z
    const TPM2B *label,  // IN: a label value for the KDF
    TPM2B *partyUInfo,  // IN: PartyUInfo
    TPM2B *partyVInfo,  // IN: PartyVInfo
    UINT32 sizeInBits,  // IN: size of generated key in bits
    BYTE *keyStream    // OUT: key buffer
)

{  
    HASH_STATE hashState;
    PHASH_DEF hashDef = CryptGetHashDef(hashAlg);
    UINT32 counter = 0;      // counter value
    UINT16 hLen;
    BYTE *stream = keyStream;
    INT16 bytes;            // number of bytes to generate

    pAssert(keyStream != NULL && Z != NULL && ((sizeInBits + 7) / 8) < INT16_MAX);
    hLen = hashDef->digestSize;
    bytes = (INT16)((sizeInBits + 7) / 8);
    if(hashAlg == TPM_ALG_NULL || bytes == 0)
        return 0;

    // Generate required bytes
    // The inner loop of that KDF uses:
    // Hash[i] := H(counter | Z | OtherInfo) (5)
    // Where:
    // Hash[i] the hash generated on the i-th iteration of the loop.
    // H() an approved hash function
    // counter a 32-bit counter that is initialized to 1 and incremented
    // on each iteration
    // Z the X coordinate of the product of a public ECC key and a
different private ECC key.
    // OtherInfo a collection of qualifying data for the KDF defined below.
    // In this specification, OtherInfo will be constructed by:
OtherInfo := Use | PartyUInfo | PartyVInfo

for(;; bytes > 0; stream = &stream[hLen], bytes = bytes - hLen)
{
    if(bytes < hLen)
        hLen = bytes;
    counter++;
    // Do the hash
    CryptHashStart(&hashState, hashAlg);
    // Add counter
    CryptDigestUpdateInt(&hashState, 4, counter);

    // Add Z
    if(Z != NULL)
        CryptDigestUpdate2B(&hashState, Z);
    // Add label
    if(label != NULL)
        CryptDigestUpdate2B(&hashState, label);
    // Add a null. SP108 is not very clear about when the 0 is needed but to
    // make this like the previous version that did not add an 0x00 after
    // a null-terminated string, this version will only add a null byte
    // if the label parameter did not end in a null byte, or if no label
    // is present.
    if((label == NULL)
        || (label->size == 0)
        || (label->buffer[label->size - 1] != 0))
        CryptDigestUpdateInt(&hashState, 1, 0);
    // Add PartyUInfo
    if(partyUInfo != NULL)
        CryptDigestUpdate2B(&hashState, partyUInfo);
    // Add PartyVInfo
    if(partyVInfo != NULL)
        CryptDigestUpdate2B(&hashState, partyVInfo);

    // Compute Hash. hLen was changed to be the smaller of bytes or hLen
    // at the start of each iteration.
    CryptHashEnd(&hashState, hLen, stream);
}

// Mask off bits if the required bits is not a multiple of byte size
if((sizeInBits % 8) != 0)
    keyStream[0] &= ((1 << (sizeInBits % 8)) - 1);
return (UINT16)((sizeInBits + 7) / 8);
10.2.14 CryptPrime.c

10.2.14.1 Introduction

This file contains the code for prime validation.

```
#include "Tpm.h"
#include "CryptPrime_fp.h"
//#define CPRI_PRIME
//#include "PrimeTable.h"
#include "CryptPrimeSieve_fp.h"

extern const uint32_t s_LastPrimeInTable;
extern const uint32_t s_PrimeTableSize;
extern const uint32_t s_PrimesInTable;
extern const unsigned char s_PrimeTable[];
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.

```
static uint32_t Root2(uint32_t n)
{
  int32_t last = (int32_t)(n >> 2);
  int32_t next = (int32_t)(n >> 1);
  int32_t diff;
  int32_t stop = 10;

  // get a starting point
  for(; next != 0; last >>= 1, next >>= 2);
  last++;
  do
  {
    next = (last + (n / last)) >> 1;
    diff = next - last;
    last = next;
    if(stop-- == 0)
      FAIL(FATAL_ERROR_INTERNAL);
  } while(diff < -1 || diff > 1);
  if(n / next > (unsigned)next)
    next++;
  pAssert(next != 0);
  pAssert((n / next) <= (unsigned)next) && (n / (next + 1) < (unsigned)next));
  return next;
}
```

10.2.14.2.2 IsPrimeInt()

This will do a test of a word of up to 32-bits in size.

```
BOOL IsPrimeInt(uint32_t n)
{
}
```
```
uint32_t i;
uint32_t stop;
if (n < 3 || (n & 1) == 0)
    return (n == 2);
if (n <= s_LastPrimeInTable)
    
    n >>= 1;
    return ((s_PrimeTable[n >> 3] >> (n & 7)) & 1);
}
// Need to search
stop = Root2(n) >> 1;
// starting at 1 is equivalent to staring at \((1 << 1) + 1\) = 3
for (i = 1; i < stop; i++)
    
    if ((s_PrimeTable[i >> 3] >> (i & 7)) & 1)
        // see if this prime evenly divides the number
        if ((n % ((i << 1) + 1)) == 0)
            return FALSE;
    }
return TRUE;
}

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.

```
BOOL
BnIsProbablyPrime(
    bigNum prime, // IN:
    RAND_STATE *rand // IN: the random state just
)
{
    #if RADIX_BITS > 32
        if (BnUnsignedCmpWord(prime, UINT32_MAX) <= 0)
    #else
        if (BnGetSize(prime) == 1)
    #endif
        return IsPrimeInt(prime->d[0]);
    
    if (BnIsEven(prime))
        return FALSE;
    if (BnUnsignedCmpWord(prime, s_LastPrimeInTable) <= 0)
    {
        crypt_uword_t temp = prime->d[0] >> 1;
        return ((s_PrimeTable[temp >> 3] >> (temp & 7)) & 1);
    }
    
    {  
        BN_VAR(n, LARGEST_NUMBER_BITS);
        BnGcd(n, prime, s_CompositeOfSmallPrimes);
        if (!BnEqualWord(n, 1))
            return FALSE;
    }
    return MillerRabin(prime, rand);
}

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.

```
MillerRabinRounds(
    UINT32 bits, // IN: Number of bits in the RSA prime
)

if (bits < 511) return 8; // don't really expect this
if (bits < 1536) return 5; // for 512 and 1K primes
return 4; // for 3K public modulus and greater

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>probably prime</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>composite</td>
</tr>
</tbody>
</table>

BOOL MillerRabin(
    bigNum bnW, RAND_STATE *rand
)

BN_MAX(bnWm1);
BN_PRIME(bnM);
BN_PRIME(bnB);
BN_PRIME(bnZ);
BOOL ret = FALSE; // Assumed composite for easy exit
unsigned int a;
unsigned int j;
int wLen;
int i;
int iterations = MillerRabinRounds(BnSizeInBits(bnW));

// INSTRUMENT_INC(MillerRabinTrials[PrimeIndex]);
pAssert(bnW->size > 1);
// Let a be the largest integer such that 2^a divides w1.
BnSubWord(bnWm1, bnW, 1);
pAssert(bnWm1->size != 0);

// Since w is odd (w-1) is even so start at bit number 1 rather than 0
// Get the number of bits in bnWm1 so that it doesn't have to be recomputed
// on each iteration.
i = bnWm1->size * RADIX_BITS;
// Now find the largest power of 2 that divides w1
for(a = 1;
    (a < (bnWm1->size * RADIX_BITS)) &&
    (BnTestBit(bnWm1, a) == 0);
a++);
// 2. m = (w1) / 2^a
BnShiftRight(bnM, bnWm1, a);
// 3. wlen = len (w).
wLen = BnSizeInBits(bnW);
// 4. For i = 1 to iterations do
for(i = 0; i < iterations; i++)
{
    // 4.1 Obtain a string b of wlen bits from an RBG.
    // Ensure that 1 < b < w1.
    // 4.2 If ((b <= 1) or (b >= w1)), then go to step 4.1.
    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 \( z = b^m \mod w \).
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 \((z == 1) \text{ or } (z = w == 1)\), 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 \( a \) do.
160     for(j = 1; j < a; j++)
161     {
162       // 4.5.1 \( z = z^2 \mod w \).
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    // 5. Return PROBABLY PRIME
179    ret = TRUE;
180 end:
181    return ret;
182 }
183 #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     NOT_REFERENCED(rand);
196 #if (modE == 0)

Page 652 TCG PUBLIC REVIEW Family “2.0”
April 15, 2019 Copyright © TCG 2006-2019 Level 00 Revision 01.55

DRAFT
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 
213 #else
214 return PrimeSelectWithSieve(prime, exponent, rand);
215 #endif
216

10.2.14.2.7 AdjustPrimeCandidate()

This function adjusts the candidate prime so that it is odd and > root(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 root(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.

LIB_EXPORT void
RsaAdjustPrimeCandidate(
  bigNum          prime,
)
{
  crypt_uword_t       msw = prime->d[prime->size - 1];
  crypt_uword_t       adjusted;

#if RADIX_BITS == 64
  #define ADD_CONST ((crypt_uword_t)0xB505000000000000ULL)
#else
  #define ADD_CONST ((crypt_uword_t)0xB5050000UL)
#endif

  // Multiplying 0xff...f by 0x4AFB gives 0xff...f - 0xB5050...0
  adjusted = (crypt_uword_t)(msw >> 16) * (crypt_uword_t)0x4AFB;
  adjusted += ((msw & 0xFFFF) * (crypt_uword_t)0x4AFB) >> 16;
  adjusted += ADD_CONST;
  prime->d[prime->size - 1] = adjusted;
  // make sure the number is odd
  prime->d[0] |= 1;
}

10.2.14.2.8 BnGeneratePrimeForRSA()

Function to generate a prime of the desired size with the proper attributes for an RSA prime.
UINT32          bits,
UINT32          exponent,
RAND_STATE      *rand
}
BOOL            found = FALSE;
// Make sure that the prime is large enough
pAssert(prime->allocated >= BITS_TO_CRYPT_WORDS(bits));
// Only try to handle specific sizes of keys in order to save overhead
pAssert((bits % 32) == 0);
prime->size = BITS_TO_CRYPT_WORDS(bits);
while(!found)
{
  DRBG_Generate(rand, (BYTE *)prime->d, (UINT16)BITS_TO_BYTES(bits));
  if(g_inFailureMode)
    return TPM_RC_FAILURE;
  RsaAdjustPrimeCandidate(prime);
  found = RsaCheckPrime(prime, exponent, rand) == TPM_RC_SUCCESS;
}
return TPM_RC_SUCCESS;
#endif  // ALG_RSA
10.2.15 CryptPrimeSieve.c

10.2.15.1 Includes and defines

```c
#include "Tpkm.h"
#if RSA_KEY_SIEVE
#include "CryptPrimeSieve_fp.h"
#endif
```

This determines the number of bits in the largest sieve field.

```c
#define MAX_FIELD_SIZE 2048
extern const uint32_t s_LastPrimeInTable;
extern const uint32_t s_PrimeTableSize;
extern const uint32_t s_PrimesInTable;
extern const unsigned char s_PrimeTable[];
```

This table is set of prime markers. Each entry is the prime value for the \((n + 1) \times 1024\) 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.

```c
const uint32_t s_PrimeMarkersCount = 6;
const uint32_t s_PrimeMarkers[] = {
  8167, 17881, 28183, 38891, 49871, 60961
};
```

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.

```c
LIB_EXPORT void RsaAdjustPrimeLimit(
    uint32_t requestedPrimes
)
{
    if (requestedPrimes == 0 || requestedPrimes > s_PrimesInTable)
        requestedPrimes = s_PrimesInTable;
    requestedPrimes = (requestedPrimes - 1) / 1024;
    if (requestedPrimes < s_PrimeMarkersCount)
        primeLimit = s_PrimeMarkers[requestedPrimes];
    else
        primeLimit = s_LastPrimeInTable;
    primeLimit >>= 1;
}
```

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.

```c
LIB_EXPORT uint32_t RsaNextPrime(
```
```c
uint32_t lastPrime

{  
    if (lastPrime == 0)
        return 0;
    lastPrime >>= 1;
    for (lastPrime += 1; lastPrime <= primeLimit; lastPrime++)
    {  
        if (((s_PrimeTable[lastPrime >> 3] >> (lastPrime & 0x7)) & 1) == 1)
            return ((lastPrime << 1) + 1);
    }
    return 0;
}
```

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.

```c
const BYTE seedValues[] = {
    0x16, 0x29, 0xcb, 0xaa, 0x65, 0xda, 0x30, 0x6c,
    0x99, 0x96, 0x4c, 0xa4, 0x65, 0xda, 0x30, 0x6c,
    0x49, 0xcb, 0xb4, 0x61, 0xd8, 0x32, 0x2d, 0x99,
    0xa6, 0x44, 0x5b, 0xa4, 0x2c, 0x93, 0x96, 0x69,
    0xc3, 0xb0, 0x65, 0x5a, 0x32, 0x4d, 0x89, 0xb6,
    0x48, 0x59, 0x26, 0x2d, 0xd3, 0x86, 0x61, 0xcb,
    0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x91, 0xb2, 0x4c,
    0x5a, 0xa6, 0x0d, 0xc3, 0x96, 0xc9, 0x34, 0x25,
    0xda, 0x22, 0x65, 0x99, 0xb4, 0x64, 0x9a, 0x12,
    0x86, 0x2d, 0x92, 0x69, 0x4a, 0xb4, 0x45, 0xca,
    0x32, 0x6d, 0x18, 0xb6, 0x4c, 0x4b, 0xa6, 0x29,
    0xda1};
#define USE_NIBBLE
#undef USE_NIBBLE

static const BYTE bitsInByte[256] = {
    0x00, 0x01, 0x01, 0x02, 0x01, 0x02, 0x02, 0x03,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x01, 0x02, 0x02, 0x03, 0x02, 0x03, 0x03, 0x04,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x04, 0x05, 0x05, 0x06, 0x05, 0x06, 0x06, 0x07,
    0x02, 0x03, 0x03, 0x04, 0x03, 0x04, 0x04, 0x05,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
    0x03, 0x04, 0x04, 0x05, 0x04, 0x05, 0x05, 0x06,
10.2.15.2.3 BitsInArray()

This function counts the number of bits set in an array of bytes.

```c
static int BitsInArray(
    const unsigned char *a,          // IN: A pointer to an array of bytes
    unsigned int aSize              // IN: the number of bytes to sum
) {
    int j = 0;
    for(; aSize; a++, aSize--)
        j += BitsInByte(*a);
    return j;
}
```

10.2.15.2.4 FindNthSetBit()

This function finds the nth 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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;0</td>
<td>no bit is set or no bit with the requested number is set</td>
</tr>
<tr>
<td>&gt;=0</td>
<td>the number of the bit in the array that is the nth set</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT int FindNthSetBit(
    const UINT16 aSize,         // IN: the size of the array to check
    const BYTE *a,              // IN: the array to check
    const UINT32 n               // IN, the number of the SET bit
) {
    UINT16 i;
    int retval;
    UINT32 sum = 0;
    BYTE sel;

    //find the bit
    for(i = 0; (i < (int)aSize) && (sum < n); i++)
        sum += BitsInByte(a[i]);
    i--;

    // Compute the offset to the start of that byte
    retval = i * 8 - 1;
    sel = a[i];

    // Subtract the bits in the last byte added.
    sum -= BitsInByte(sel);

    // Now process the byte, one bit at a time.
```
for(; (sel != 0) && (sum != n); retValue++, sel = sel >> 1)
    sum += (sel & 1) != 0;
return (sum == n) ? retValue : -1;
}
typedef struct
{
    UINT16      prime;
    UINT16      count;
} SIEVE_MARKS;
const SIEVE_MARKS sieveMarks[5] = {
    {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.

LIB_EXPORT UINT32
PrimeSieve(
bigNum           bnN,  // IN/OUT: number to sieve
UINT32           fieldSize, // IN: size of the field area in bytes
BYTE            *field   // IN: field
)
{
    INT32            i;
    INT32            j;
    UINT32           fieldBits = fieldSize * 8;
    UINT32           r;
    BYTE            *pField;
    INT32            iter;
    INT32            adjust;
    UINT32           mark = 0;
    INT32            count = sieveMarks[0].count;
    INT32            stop = sieveMarks[0].prime;
    UINT32           composite;
    UINT32           pList[8];
    UINT32           next;
    pAssert(field != NULL && bnN != NULL);
    // If the remainder is odd, then subtracting the value will give an even number,
    // but we want an odd number, so subtract the 105+rem. Otherwise, just subtract
    // the even remainder
    adjust = BnModWord(bnN, 105);
    if(adjust & 1)
        adjust += 105;
    // Adjust the input number so that it points to the first number in a
    // aligned field.
    BnSubWord(bnN, bnN, adjust);
    // pAssert (BnModWord (bnN, 105) == 0);
    pField = field;
    for(i = fieldSize; i >= sizeof(seedValues);)
        pField += sizeof(seedValues), i -= sizeof(seedValues))
        
{
memcpy(pField, seedValues, sizeof(seedValues));

if(i != 0)
    memcpy(pField, seedValues, i);

// Cycle through the primes, clearing bits
// Have already done 3, 5, and 7
iter = 7;

#define NEXT_PRIME(iter)    (iter = RsaNextPrime(iter))

// Get the next N primes where N is determined by the mark in the sieveMarks
while((composite = NEXT_PRIME(iter)) != 0)
{
    next = 0;
    i = count;
    pList[i--] = composite;
    for(; i > 0; i--)
    {
        next = NEXT_PRIME(iter);
        pList[i] = next;
        if(next != 0)
        {
            composite *= next;
        }
    }
    // Get the remainder when dividing the base field address
    // by the composite
    composite = BnModWord(bnN, composite);
    // 'composite' is divisible by the composite components. for each of the
    // composite components, divide 'composite'. That remainder (r) is used to
    // pick a starting point for clearing the array. The stride is equal to the
    // composite component. Note, the field only contains odd numbers. If the
    // field were expanded to contain all numbers, then half of the bits would
    // have already been cleared. We can save the trouble of clearing them a
    // second time by having a stride of 2*next. Or we can take all of the even
    // numbers out of the field and use a stride of 'next'
    for(i = count; i > 0; i--)
    {
        next = pList[i];
        if(next == 0)
            goto done;
        r = composite % next;
        // these computations deal with the fact that the field starts at some
        // arbitrary offset within the number space. If the field were all numbers,
        // then we would have gone through some number of bit clearings before we
        // got to the start of this range. We don't know how many there were before,
        // but we can tell from the remainder whether we are on an even or odd
        // stride when we hit the beginning of the table. If we are on an odd stride
        // (r & 1), we would start half a stride in (next - r)/2. If we are on an
        // even stride, we need 1.5 strides (next + r/2) because the table only has
        // odd numbers. If the remainder happens to be zero, then the start of the
        // table is on stride so no adjustment is necessary.
        if(r & 1)
            j = (next - r) / 2;
        else if(r == 0)
            j = 0;
        else
            j = next - r / 2;
        for(; j < fieldBits; j += next)
            ClearBit(j, field, fieldSize);
    }
    if(next >= stop)
    {
        mark++;
        count = sieveMarks[mark].count;
        stop = sieveMarks[mark].prime;
    }
}
done:
INSTRUMENT_INC(totalFieldsSieved[PrimeIndex]);
i = BitsInArray(field, fieldSize);
10.2.15.2.6 SetFieldSize()

Function to set the field size used for prime generation. Used for tuning.

```
LIB_EXPORT uint32_t SetFieldSize(
    uint32_t newFieldSize
)
{
    if (newFieldSize == 0 || newFieldSize > MAX_FIELD_SIZE)
        fieldSize = MAX_FIELD_SIZE;
    else
        fieldSize = newFieldSize;
    return fieldSize;
}
```

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, \( p_{NP} \) 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_FAILURE</td>
<td>TPM in failure mode, probably due to entropy source</td>
</tr>
<tr>
<td>TPM_RC_SUCCESS</td>
<td>candidate is probably prime</td>
</tr>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>candidate is not prime and couldn't find and alternative in the field</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT TPM_RC PrimeSelectWithSieve(
    bigNum candidate, // IN/OUT: The candidate to filter
    UINT32 e,        // IN: the exponent
    RAND_STATE *rand  // IN: the random number generator state
)
{
    BYTE field[MAX_FIELD_SIZE];
    UINT32 first;
    UINT32 ones;
    INT32 chosen;
    BN_PRIME(test);
    UINT32 modE;
    #ifndef SIEVE_DEBUG
    UINT32 fieldSize = MAX_FIELD_SIZE;
    #endif
    UINT32 primeSize;
    //
    // Adjust the field size and prime table list to fit the size of the prime
    // being tested. This is done to try to optimize the trade-off between the
    // dividing done for sieving and the time for Miller-Rabin. When the size
```
if (primeSize <= 512)
{
    RsaAdjustPrimeLimit(1024); // Use just the first 1024 primes
}
else if (primeSize <= 1024)
{
    RsaAdjustPrimeLimit(4096); // Use just the first 4K primes
}
else
{
    RsaAdjustPrimeLimit(0); // Use all available
}

// Save the low-order word to use as a search generator and make sure that
// it has some interesting range to it
first = candidate->d[0] | 0x80000000;

// Sieve the field
ones = PrimeSieve(candidate, fieldSize, field);
pAssert(ones > 0 && ones < (fieldSize * 8));
for(; ones > 0; ones--)
{
    // Decide which bit to look at and find its offset
    chosen = FindNthSetBit((UINT16)fieldSize, field, ((first % ones) + 1));
    if((chosen < 0) || (chosen >= (INT32)(fieldSize * 8)))
        FAIL(FATAL_ERROR_INTERNAL);
    // Set this as the trial prime
    BnAddWord(test, candidate, (crypt_uword_t)(chosen * 2));
    // The exponent might not have been one of the tested primes so
    // make sure that it isn't divisible and make sure that 0 != (p-1) mod e
    // Note: This is the same as 1 != p mod e
    modE = BnModWord(test, e);
    if((modE != 0) && (modE != 1) && MillerRabin(test, rand))
    {
        BnCopy(candidate, test);
        return TPM_RC_SUCCESS;
    }
    // Clear the bit just tested
    ClearBit(chosen, field, fieldSize);
}

#endif

#define CLEAR_VALUE(x)    memset(x, 0, sizeof(x))

void
RsaSimulationEnd(
    void
)
{
    int i;
    UINT32 averages[3];
    UINT32 nonFirst = 0;
    if((PrimeCounts[0] + PrimeCounts[1] + PrimeCounts[2]) != 0)
    {
        printf("Primes generated = %s\n", PrintTuple(PrimeCounts));
        printf("Fields sieved = %s\n", PrintTuple(totalFieldsSieved));
        printf("Fields with no primes = %s\n", PrintTuple(noPrimeFields));
        printf("Primes checked with Miller-Rabin = %s\n",
            PrintTuple(MillerRabinTrials));
        for(i = 0; i < 3; i++)
            averages[i] = (totalFieldsSieved[i]
                != 0 ? bitsInFieldAfterSieve[i] / totalFieldsSieved[i]
                : 0);
        printf("Average candidates in field %s\n", PrintTuple(averages));
        for(i = 1; i < (sizeof(failedAtIteration) / sizeof(failedAtIteration[0]));
            i++)
        {
            nonFirst += failedAtIteration[i];
            printf("Miller-Rabin failures not in first round = %d\n", nonFirst);
        }
    }
    CLEAR_VALUE(PrimeCounts);
    CLEAR_VALUE(totalFieldsSieved);
    CLEAR_VALUE(noPrimeFields);
    CLEAR_VALUE(MillerRabinTrials);
    CLEAR_VALUE(bitsInFieldAfterSieve);
}

LIB_EXPORT void
GetSieveStats(
    uint32_t        *trials,
    uint32_t        *emptyFields,
    uint32_t        *averageBits
)
{
    uint32_t totalBits;
    uint32_t fields;
    *emptyFields = noPrimeFields[0] + noPrimeFields[1] + noPrimeFields[2];
    fields = totalFieldsSieved[0] + totalFieldsSieved[1]
        + totalFieldsSieved[2];
    totalBits = bitsInFieldAfterSieve[0] + bitsInFieldAfterSieve[1]
        + bitsInFieldAfterSieve[2];
    if(fields != 0)
        *averageBits = totalBits / fields;
    else
        *averageBits = 0;
    CLEAR_VALUE(PrimeCounts);
    CLEAR_VALUE(totalFieldsSieved);
    CLEAR_VALUE(noPrimeFields);
    CLEAR_VALUE(MillerRabinTrials);
    CLEAR_VALUE(bitsInFieldAfterSieve);
}

#endif
#endif
#if !RSA_INSTRUMENT

void 
RsaSimulationEnd();
#endif // RSA_KEY_SIEVE
#if !RSA_INSTRUMENT
423     }
424     {
425     }
426 #endif
10.2.16 CryptRand.c

10.2.16.1 Introduction

This file implements a DRBG with a behavior according to SP800-90A using a block cypher. This is also compliant to ISO/IEC 18031:2011(E) C.3.2.

A state structure is created for use by TPM.lib and functions within the CryptoEngine() my 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 DRGB 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

```c
#define DF_COUNT (DRBG_KEY_SIZE_WORDS / DRBG_IV_SIZE_WORDS + 1)

#if DRBG_KEY_SIZE_BITS != 128 && DRBG_KEY_SIZE_BITS != 256
  #error "CryptRand.c only written for AES with 128- or 256-bit keys."
#endif
typedef struct
{
  DRBG_KEY_SCHEDULE keySchedule;
  DRBG_IV iv[DF_COUNT];
  DRBG_IV out1;
  DRBG_IV buf;
  int contents;
} 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.

```c
static void DfCompute(
    PDF_STATE dfState
)
{
    int i;
    int iv;
    crypt_uword_t *pIv;
    crypt_uword_t temp[DRBG_IV_SIZE_WORDS] = {0};
    //
    for (iv = 0; iv < DF_COUNT; iv++)
    {
        pIv = (crypt_uword_t *)&dfState->iv[iv].words[0];
        for (i = 0; i < DRBG_IV_SIZE_WORDS; i++)
        {
            temp[i] ^= pIv[i] ^ dfState->buf.words[i];
        }
        DRBG.ENCRYPT(&dfState->keySchedule, &temp, pIv);
    }
    for (i = 0; i < DRBG_IV_SIZE_WORDS; i++)
    {
        dfState->buf.words[i] = 0;
    }
    dfState->contents = 0;
}
```

### 10.2.16.2.4 DfStart()

This initializes the output blocks with an encrypted counter value and initializes the key schedule.

```c
static void DfStart(
    PDF_STATE dfState,
    uint32_t inputLength
)
{
    BYTE init[8];
    int i;
    UINT32 drbgSeedSize = sizeof(DRBG_SEED);
    const BYTE dfKey[DRBG_KEY_SIZE_BYTES] = {
        0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
        0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f
```
#if DRBG_KEY_SIZE_BYTES > 16
    0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d, 0x1e, 0x1f
#endif
memset(dfState, 0, sizeof(DF_STATE));
DRBG_ENCRYPT_SETUP(&dfKey[0], DRBG_KEY_SIZE_BITS, &dfState->keySchedule);
// Create the first chaining values
for(i = 0; i < DF_COUNT; i++)
    ((BYTE *)&dfState->iv[i])[3] = (BYTE)i;
DfCompute(dfState);
// initialize the first 64 bits of the IV in a way that doesn't depend
// on the size of the words used.
UINT32_TO_BYTE_ARRAY(inputLength, init);
UINT32_TO_BYTE_ARRAY(drbgSeedSize, &init[4]);
memcpy(&dfState->iv[0], init, 8);
dfState->contents = 4;
}

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().

static void
DfUpdate(
    PDF_STATE        dfState,
    int              size,
    const BYTE      *data
    )
{
    while(size > 0)
    {
        int toFill = DBRG_IV_SIZE_BYTES - dfState->contents;
        if(size < toFill)
            toFill = size;
        // Copy as many bytes as there are or until the state buffer is full
        memcpy(&dfState->buf.bytes[dfState->contents], data, toFill);
        // Reduce the size left by the amount copied
        size -= toFill;
        // Advance the data pointer by the amount copied
        data += toFill;
        // increase the buffer contents count by the amount copied
        dfState->contents += toFill;
        pAssert(dfState->contents <= DBRG_IV_SIZE_BYTES);
        // If we have a full buffer, do a computation pass.
        if(dfState->contents == DBRG_IV_SIZE_BYTES)
            DfCompute(dfState);
    }
}

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 DBRG_SEED value so that the
function can return a pointer to the DBRG_SEED value in the DF_STATE structure.

static DBRG_SEED *
DfEnd(
    PDF_STATE        dfState
    )
{
// Since DfCompute is always called when a buffer is full, there is always
// space in the buffer for the terminator
dfState->buf.bytes[dfState->contents++] = 0x80;
// If the buffer is not full, pad with zeros
while(dfState->contents < DRGB_IV_SIZE_BYTES)
    dfState->buf.bytes[dfState->contents++] = 0;
// Do a final state update
DfCompute(dfState);
return (DRBG_SEED *)&dfState->iv;
}

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();

static DRBG_SEED *
DfBuffer(
    DRBG_SEED *output,  // OUT: receives the result
    int size,    // IN: size of the buffer to add
    BYTE *buf    // IN: address of the buffer
) {
    DF_STATE dfState;
    if(size == 0 || buf == NULL)
        return NULL;
    // Initialize the derivation function
    DfStart(&dfState, size);
    DfUpdate(&dfState, size, buf);
    DfEnd(&dfState);
    memcpy(output, &dfState.iv[0], sizeof(DRBG_SEED));
    return output;
}

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 an non-orderly shutdown.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>requested entropy returned</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>entropy Failure</td>
</tr>
</tbody>
</table>

BOOL DRBG_GetEntropy(
    UINT32 requiredEntropy,   // IN: requested number of bytes of full
    BYTE *entropy)            // OUT: buffer to return collected entropy
)

#ifdef USE_DEBUG_RNG

// If in debug mode, always use the self-test values for initialization
if(IsSelfTest())
    {
#endif

Family "2.0" TCG PUBLIC REVIEW Page 667
Level 00 Revision 01.55 Copyright © TCG 2006-2019 April 15, 2019

DRAFT
// If doing simulated DRBG, then check to see if the
// entropyFailure condition is being tested
if(!IsEntropyBad())
{
    // In self-test, the caller should be asking for exactly the seed
    // size of entropy.
pAssert(requiredEntropy == sizeof(DRBG_NistTestVector_Entropy));
    memcpy(entropy, DRBG_NistTestVector_Entropy,
           sizeof(DRBG_NistTestVector_Entropy));
}
#else
#endif

return !IsEntropyBad();
}

10.2.16.2.9 IncrementIv()

This function increments the IV value by 1. It is used by EncryptDRBG().

void
IncrementIv(
    DRBG_IV         *iv)
{
    BYTE      *ivP = ((BYTE *)iv) + DRBG_IV_SIZE_BYTES;
    while((--ivP >= (BYTE *)iv) && (*ivP = ((*ivP + 1) & 0xFF)) == 0));

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.

static BOOL
EncryptDRBG(
    BYTE        *dOut,
    UINT32      dOutBytes,
    DRBG_KEY_SCHEDULE   *keySchedule,
    DRBG_IV         *iv,
    UINT32              *lastValue     // Points to the last output value
)
{
    #if FIPS_COMPLIANT
    // For FIPS compliance, the DRBG has to do a continuous self-test to make sure that
    // no two consecutive values are the same. This overhead is not incurred if the TPM
    // is not required to be FIPS compliant
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.
static BOOL

DRBG_Update(
    DRBG_STATE *drbgState,  // IN:OUT state to update
    DRBG_KEY_SCHEDULE *keySchedule,  // IN: the key schedule (optional)
    DRBG_SEED *providedData  // IN: additional data
)
{
    UINT32 i;
    BYTE *temp = (BYTE *)&drbgState->seed;
    DRBG_KEY *key = pDRBG_KEY(&drbgState->seed);
    DRBG_IV *iv = pDRBG_IV(&drbgState->seed);
    DRBG_KEY_SCHEDULE localKeySchedule;
    // pAssert(drbgState->magic == DRBG_MAGIC);

    // If an key schedule was not provided, make one
    if(keySchedule == NULL)
    {
        if(DRNG_ENCRYPT_SETUP((BYTE *)key,
            DRBG_KEY_SIZE_BITS, &localKeySchedule) != 0)
        {
            LOG_FAILURE(FATAL_ERROR_INTERNAL);
            return FALSE;
        }
        keySchedule = &localKeySchedule;
    }

    // Encrypt the temp value
    EncryptDRBG(temp, sizeof(DRBG_SEED), keySchedule, iv,
        drbgState->lastValue);
    if(providedData != NULL)
    {
        BYTE *pP = (BYTE *)providedData;
        for(i = DRBG_SEED_SIZE_BYTES; i != 0; i--)
        {
            *temp++ ^= *pP++;
        }
    }
    // Since temp points to the input key and IV, we are done and
    // don't need to copy the resulting 'temp' to drbgState->seed
    return TRUE;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>reseed succeeded</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>reseed failed, probably due to the entropy generation</td>
</tr>
</tbody>
</table>

BOOL

DRBG_Reseed(
    DRBG_STATE *drbgState,  // IN: the state to update
    DRBG_SEED *providedEntropy,  // IN: entropy
    DRBG_SEED *additionalData  // IN:
)
{
    DRBG_SEED seed;
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         // XOR the provided data into the provided entropy
316         for(i = 0; i < sizeof(DRBG_SEED); i++)
317             ((BYTE *)providedEntropy)[i] ^= ((BYTE *)additionalData)[i];
318     }
319     DRBG_Update(drbgState, NULL, providedEntropy);
320     drbgState->reseedCounter = 1;
321
322     return TRUE;
323 }
324
325 10.2.16.2.13 DRBG_SelfTest()
326     This is run when the DRBG is instantiated and at startup
327
328     | Return Value | Meaning     |
329     | TRUE(1) | test OK     |
330     | FALSE(0) | test failed |
331
326     BOOL DRBG_SelfTest(
327     void
328     )
329     {
330         BYTE         buf[sizeof(DRBG_NistTestVector_Generated)];
331         DRBG_SEED    seed;
332         UINT32       i;
333         BYTE        *p;
334         DRBG_STATE   testState;
335         //
336         pAssert(!IsSelfTest());
337         SetSelfTest();
338         SetDrbgTested();
339         // Do an instantiate
340         if(!DRBG_Instantiate(&testState, 0, NULL))
341             return FALSE;
342         #if DRBG_DEBUG_PRINT
343             dbgDumpMemBlock(pDRBG_KEY(&testState), DRBG_KEY_SIZE_BYTES,
344                             "Key after Instantiate");
345             dbgDumpMemBlock(pDRBG_IV(&testState), DRBG_IV_SIZE_BYTES,
346                             "Value after Instantiate");
347         #endif
348         if(DRBG_Generate((RAND_STATE *)&testState, buf, sizeof(buf)) == 0)
349             return FALSE;
350         #if DRBG_DEBUG_PRINT
351             dbgDumpMemBlock(pDRBG_KEY(&testState.seed), DRBG_KEY_SIZE_BYTES,
352                             "Key after 1st Generate");
353             dbgDumpMemBlock(pDRBG_IV(&testState.seed), DRBG_IV_SIZE_BYTES,
354                             "Value after 1st Generate");
355         #endif
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_NO_RESULT</td>
<td>failure of the entropy generator</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT TPM_RC CryptRandomStir(
    UINT16 additionalDataSize,
    BYTE *additionalData
)
{
    #if !USE_DEBUG_RNG
    DRBG_SEED tmpBuf;
    DRBG_SEED dfResult;

    // All reseed with outside data starts with a buffer full of entropy
    if (!DRBG_GetEntropy(sizeof(tmpBuf), (BYTE *)&tmpBuf))
    ```
return TPM_RC_NO_RESULT;

DRBG_Reseed(&drbgDefault, &tmpBuf,
            DfBuffer(&dfResult, additionalDataSize, additionalData));
drbgDefault.reseedCounter = 1;

return TPM_RC_SUCCESS;

#else

// If doing debug, use the input data as the initial setting for the RNG state
// so that the test can be reset at any time.
// Note: If this is called with a data size of 0 or less, nothing happens. The
// presumption is that, in a debug environment, the caller will have specific
// values for initialization, so this check is just a simple way to prevent
// inadvertent programming errors from screwing things up. This doesn’t use an
// pAssert() because the non-debug version of this function will accept these
// parameters as meaning that there is no additionalData and only hardware
// entropy is used.
if((additionalDataSize > 0) && (additionalData != NULL))
{
    memset(drbgDefault.seed.bytes, 0,
            sizeof(drbgDefault.seed.bytes));
    memcpy(drbgDefault.seed.bytes, additionalData,
            MIN(additionalDataSize, sizeof(drbgDefault.seed.bytes)));
}
drbgDefault.reseedCounter = 1;

#endif

10.2.16.3.3 CryptRandomGenerate()

Generate a randomSize number or random bytes.

LIB_EXPORT UINT16 CryptRandomGenerate
(INT32            randomSize,
 BYTE            *buffer
 )
{
    if(randomSize > UINT16_MAX)
        randomSize = UINT16_MAX;
    return DRBG_Generate((RAND_STATE *)&drbgDefault, buffer, (UINT16)randomSize);
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.

LIB_EXPORT BOOL
DRBG_InstantiateSeededKdf(
    KDF_STATE   *state,       // OUT: buffer to hold the state
    TPM_ALG_ID  hashAlg,      // IN: hash algorithm
    TPM_ALG_ID  kdf,          // IN: the KDF to use
    TPM2B       *seed,        // IN: the seed to use
    const TPM2B *label,       // IN: a label for the generation process.
    TPM2B       *context,     // IN: the context value
    UINT32      limit         // IN: Maximum number of bits from the KDF
)
{
    state->magic = KDF_MAGIC;
    state->limit = limit;
    state->seed = seed;
    state->hash = hashAlg;
    state->kdf = kdf;
    state->label = label;
    state->context = context;
    state->digestSize = CryptHashGetDigestSize(hashAlg);
    state->counter = 0;
    state->residual.t.size = 0;
    return TRUE;
}
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.

```c
LIB_EXPORT void
DRBG_AdditionalData(
    DRBG_STATE *drbgState,   // IN:OUT state to update
    TPM2B *additionalData   // IN: value to incorporate
)
{
    DRBG_SEED dfResult;
    if(drbgState->magic == DRBG_MAGIC)
        DfBuffer(&dfResult, additionalData->size, additionalData->buffer);
    DRBG_Reseed(drbgState, &dfResult, NULL);
}
```
10.2.16.3.3 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.

```
LIB_EXPORT TPM_RC
DRBG_InstantiateSeeded(
    DRBG_STATE *drbgState, // IN/OUT: buffer to hold the state
    const TPM2B *seed, // IN: the seed to use
    const TPM2B *purpose, // IN: a label for the generation process.
    const TPM2B *name, // IN: name of the object
    const TPM2B *additional // IN: additional data
){
    DF_STATE dfState;
    int totalInputSize;
    // DRBG should have been tested, but...
    if(!IsDrbgTested() && !DRBG_SelfTest()) {
        LOG_FAILURE(FATAL_ERROR_SELF_TEST);
        return TPM_RC_FAILURE;
    }
    // Initialize the DRBG state
    memset(drbgState, 0, sizeof(DRBG_STATE));
    drbgState->magic = DRBG_MAGIC;
    // Size all of the values
    totalInputSize = (seed != NULL) ? seed->size : 0;
    totalInputSize += (purpose != NULL) ? purpose->size : 0;
    totalInputSize += (name != NULL) ? name->size : 0;
    totalInputSize += (additional != NULL) ? additional->size : 0;
    // Initialize the derivation
    DfStart(&dfState, totalInputSize);
    // Run all the input strings through the derivation function
    if(seed != NULL)
        DfUpdate(&dfState, seed->size, seed->buffer);
    if(purpose != NULL)
        DfUpdate(&dfState, purpose->size, purpose->buffer);
    if(name != NULL)
        DfUpdate(&dfState, name->size, name->buffer);
    if(additional != NULL)
        DfUpdate(&dfState, additional->size, additional->buffer);
    // Used the derivation function output as the "entropy" input. This is not
    // how it is described in SP800-90A but this is the equivalent function
    DRBG_Reseed(((DRBG_STATE *)drbgState), DfEnd(&dfState), NULL);
    return TPM_RC_SUCCESS;
}
```
10.2.16.3.3.4 CryptRandStartup()

This function is called when TPM_Startup is executed. This function always returns TRUE.

```c
LIB_EXPORT BOOL
CryptRandStartup(
    void
)
{
#if ! _DRBG_STATE_SAVE
    // If not saved in NV, re-instantiate on each startup
    DRBG_Instantiate(&drbgDefault, 0, NULL);
#else
    // If the running state is saved in NV, NV has to be loaded before it can
    // be updated
    if(go.drbgState.magic == DRBG_MAGIC)
        DRBG_Reseed(&go.drbgState, NULL, NULL);
    else
        DRBG_Instantiate(&go.drbgState, 0, NULL);
#endif
    return TRUE;
}
```
10.2.16.3.3.5 CryptRandInit()

This function is called when _TPM_Init() is being processed.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT BOOL
CryptRandInit(
    void
)
{
    #if !USE_DEBUG_RNG
        _plat__GetEntropy(NULL, 0);
    #endif
    return DRBG_SelfTest();
}
```

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.

```c
LIB_EXPORT UINT16
DRBG_Generate(
    RAND_STATE      *state,
    BYTE            *random,   // OUT: buffer to receive the random values
    UINT16           randomSize // IN: the number of bytes to generate
)
{
    if(state == NULL)
        state = (RAND_STATE *)&drbgDefault;
    // If the caller used a KDF state, generate a sequence from the KDF not to
    // exceed the limit.
    if(state->kdf.magic == KDF_MAGIC)
    {
        KDF_STATE       *kdf = (KDF_STATE *)state;
        INT32           counter = (UINT32)kdf->counter;
        INT32           bytesLeft = randomSize;
        if(random == NULL)
            return 0;
        // If the number of bytes to be returned would put the generator
        // over the limit, then return 0
        if(((kdf->counter * kdf->digestSize) + randomSize) * 8) > kdf->limit)
            return 0;
        // Process partial and full blocks until all requested bytes provided
        while(bytesLeft > 0)
        {
            // If there is any residual data in the buffer, copy it to the output
            // buffer
            if(kdf->residual.t.size > 0)
            {
                INT32        size;
                //
```
584  // Don't use more of the residual than will fit or more than are
585  size = MIN(kdf->residual.t.size, bytesLeft);
586  // Copy some or all of the residual to the output. The residual is
587  // at the end of the buffer. The residual might be a full buffer.
588  MemoryCopy(random,
589    &kdf->residual.t.buffer
590    [kdf->digestSize - kdf->residual.t.size], size);
591
592  // Advance the buffer pointer
593  random += size;
594
595  // Reduce the number of bytes left to get
596  bytesLeft -= size;
597
598  // And reduce the residual size appropriately
599  kdf->residual.t.size -= (UINT16)size;
600  }
601  else
602  {
603     UINT16   blocks = (UINT16)(bytesLeft / kdf->digestSize);
604
605     // Get the number of required full blocks
606     if(blocks > 0)
607     {
608         UINT16   size = blocks * kdf->digestSize;
609
610         // Get some number of full blocks and put them in the return buffer
611         CryptKDFa(kdf->hash, kdf->seed, kdf->label, kdf->context, NULL,
612             kdf->limit, random, &counter, blocks);
613
614         // reduce the size remaining to be moved and advance the pointer
615         bytesLeft -= size;
616         random += size;
617     }
618     else
619     {
620         // Fill the residual buffer with a full block and then loop to
621         // top to get part of it copied to the output.
622         kdf->residual.t.size = CryptKDFa(kdf->hash, kdf->seed,
623             kdf->label, kdf->context, NULL,
624             kdf->limit, kdf->residual.t.buffer,
625             &counter, 1);
626
627     }
628  }
629
630  kdf->counter = counter;
631  return randomSize;
632 }
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         {
// If this is a PRNG then the only way to get
// here is if the SW has run away.
LOG_FAILURE(FATAL_ERROR_INTERNAL);
return 0;
}

// if the allowed number of bytes in a request is larger than the
// less than the number of bytes that can be requested, then check
#if UINT16_MAX >= CTR_DRBG_MAX_BYTES_PER_REQUEST
if(randomSize > CTR_DRBG_MAX_BYTES_PER_REQUEST)
    randomSize = CTR_DRBG_MAX_BYTES_PER_REQUEST;
#endif

// Create encryption schedule
if(DRBG_ENCRYPT_SETUP((BYTE *)pDRBG_KEY(seed),
    DRBG_KEY_SIZE_BITS, &keySchedule) != 0)
{
    LOG_FAILURE(FATAL_ERROR_INTERNAL);
    return 0;
}

// Generate the random data
EncryptDRBG(random, randomSize, &keySchedule, pDRBG_IV(seed),
    drbgState->lastValue);

// Do a key update
DRBG_Update(drbgState, &keySchedule, NULL);

// Increment the reseed counter
drbgState->reseedCounter += 1;

else
{
    LOG_FAILURE(FATAL_ERROR_INTERNAL);
    return FALSE;
}

return randomSize;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>instantiation succeeded</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>instantiation failed</td>
</tr>
</tbody>
</table>

LIB_EXPORT BOOL
DRBG_Instantiate(
    DRBG_STATE      *drbgState,  // OUT: the instantiated value
    UINT16           pSize,     // IN: Size of personalization string
    BYTE            *personalization // IN: The personalization string
)
{
    DRBG_SEED        seed;
    DRBG_SEED        dfResult;

    // pAssert((pSize == 0) || (pSize <= sizeof(seed)) || (personalization != NULL));
    // If the DRBG has not been tested, test when doing an instantiation. Since
    // Instantiation is called during self test, make sure we don't get stuck in a
    // loop.
    if(!IsDrbgTested() && !IsSelfTest() && !DRBG_SelfTest())
        return FALSE;

    // If doing a self test, DRBG_GetEntropy will return the NIST
Part 4: Supporting Routines

// test vector value.
if (!DRBG_GetEntropy(sizeof(seed), (BYTE *)&seed))
    return FALSE;

// set everything to zero
memset(drbgState, 0, sizeof(DRBG_STATE));

// DRBG_Reseed(drbgState, &seed, DfBuffer(&dfResult, pSize, personalization));
return TRUE;

10.2.16.3.6   DRBG_Uninstantiate()

This is Uninstantiate_function() from [SP 800-90A 9.4].

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_VALUE</td>
<td>not a valid state</td>
</tr>
</tbody>
</table>

LIB_EXPORT TPM_RC
DRBG_Uninstantiate(
    DRBG_STATE      *drbgState    // IN/OUT: working state to erase
)
{
    if ((drbgState == NULL) || (drbgState->magic != DRBG_MAGIC))
        return TPM_RC_VALUE;
    memset(drbgState, 0, sizeof(DRBG_STATE));
    return TPM_RC_SUCCESS;
}
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

```c
#define CRYPT_RSA_C
#include "Tpm.h"
#if ALG_RSA
```

10.2.17.3 Obligatory Initialization Functions

10.2.17.3.1 CryptRsaInit()

Function called at _TPM_Init().

```c
BOOL CryptRsaInit(
void
)
{
    return TRUE;
}
```

10.2.17.3.2 CryptRsaStartup()

Function called at TPM2_Startup()

```c
BOOL CryptRsaStartup(
void
)
{
    return TRUE;
}
```

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.

```c
static privateExponent *
RsaInitializeExponent( 
privateExponent *Z
)
{
    bigNum *bn = (bigNum *)Z->P;
    int i;
    //
```
for(i = 0; i < 5; i++)
{
    bn[i] = (bigNum)|Z->entries[i];
    BnInit(bn[i],BYTES_TO_CRYPT_WORDS(sizeof(Z->entries[0].d)));
}
return Z;

10.2.17.4.2 MakePgreaterThanQ()

This function swaps the pointers for P and Q if Q happens to be larger than Q.

static void MakePgreaterThanQ(
    privateExponent *Z
)
{
    if(BnUnsignedCmp(Z->P, Z->Q) < 0)
    {
        bigNum bnT = Z->P;
        Z->P = Z->Q;
        Z->Q = bnT;
    }
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure // The data is too big to fit</td>
</tr>
</tbody>
</table>

static BOOL PackExponent(
    TPM2B_PRIVATE_KEY_RSA *packed,
    privateExponent *Z
)
{
    int i;
    UINT16 primeSize = (UINT16)BITS_TO_BYTES(BnMsb(Z->P));
    UINT16 pS = primeSize;
    // pAssert((primeSize * 5) <= sizeof(packed->t.buffer));
    packed->t.size = (primeSize * 5) + RSA_prime_flag;
    for(i = 0; i < 5; i++)
        if(!BnToBytes((bigNum)&Z->entries[i], &packed->t.buffer[primeSize * i], &pS))
            return FALSE;
    if(pS != primeSize)
        return FALSE;
    return TRUE;
10.2.17.4.4 UnpackExponent()

This function unpacks the private exponent from its TPM2B form into its bignum form.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>TPM2B is not the correct size</td>
</tr>
</tbody>
</table>

```c
static BOOL UnpackExponent(
    TPM2B_PRIVATE_KEY_RSA *b,
    privateExponent *Z
)
{
    UINT16 primeSize = b->t.size & ~RSA_prime_flag;
    int i;
    bigNum *bn = &Z->P;
    //
    VERIFY(b->t.size & RSA_prime_flag);
    RsaInitializeExponent(Z);
    VERIFY((primeSize % 5) == 0);
    primeSize /= 5;
    for (i = 0; i < 5; i++)
        VERIFY(BnFromBytes(bn[i], &b->t.buffer[primeSize * i], primeSize) != NULL);
    MakePgreaterThanQ(Z);
    return TRUE;
    Error:
    return FALSE;
}
```

10.2.17.4.5 ComputePrivateExponent()

This function computes the private exponent from the primes.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
static BOOL ComputePrivateExponent(
    bigNum pubExp, // IN: the public exponent
    privateExponent *Z // IN/OUT: on input, has primes P and Q. On
    // output, has P, Q, dP, dQ, and pInv
)
{
    BOOL pOK;
    BOOL qOK;
    BN_PRIME(pT);
    // make p the larger value so that m2 is always less than p
    MakePgreaterThanQ(Z);
    //dP = (1/e) mod (p-1)
    pOK = BnSubWord(pT, Z->P, 1);
    pOK = pOK && BnModInverse(Z->dP, pubExp, pT);
    //dQ = (1/e) mod (q-1)
    qOK = BnSubWord(pT, Z->Q, 1);
    qOK = qOK && BnModInverse(Z->dQ, pubExp, pT);
    // 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure</td>
</tr>
</tbody>
</table>

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    // MakePgreaterThanQ(Z);
126    // m1 = cdP mod p
127    VERIFY(BnModExp(M1, inOut, Z->dP, Z->P));
128    // m2 = cdQ mod q
129    VERIFY(BnModExp(M2, inOut, Z->dQ, Z->Q));
130    // h = qInv * (m1 - m2) mod p = qInv * (m1 + P - m2) mod P because Q < P
131    // so m2 < P
132    VERIFY(BnSub(H, Z->P, M2));
133    VERIFY(BnAdd(H, H, M1));
134    VERIFY(BnModMult(H, H, Z->qInv, Z->P));
135    // m = m2 + h * q
136    VERIFY(BnMult(M, H, Z->Q));
137    VERIFY(BnAdd(inOut, M2, M));
138    return TRUE;
139 }
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).

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_VALUE</td>
<td>number to exponentiate is larger than the modulus</td>
</tr>
</tbody>
</table>

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    TPMM2B_TYPE(4BYTES, 4);
154    TPMM2B_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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIZE</td>
<td>the value to decrypt is larger than the modulus</td>
</tr>
</tbody>
</table>

static TPM_RC

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(UnsignedCompareB(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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_VALUE</td>
<td>hashAlg is not valid or message size is too large</td>
</tr>
</tbody>
</table>

```c
static TPM_RC
OaepEncode(
    TPM2B *padded,  // OUT: the pad data
    TPM_ALG_ID hashAlg,  // IN: algorithm to use for padding
    const TPM2B *label,  // IN: null-terminated string (may be NULL)
    TPM2B *message,  // IN: the message being padded
    RAND_STATE *rand  // IN: the random number generator to use
)
{
    INT32        padLen;
    INT32        dbSize;
    INT32        i;
    BYTE         mySeed[MAX_DIGEST_SIZE];
    BYTE        *seed = mySeed;
    INT32        hLen = CryptHashGetDigestSize(hashAlg);
    BYTE         mask[MAX_RSA_KEY_BYTES];
    BYTE        *pp;
    BYTE        *pm;
    TPM_RC       retVal = TPM_RC_SUCCESS;

    pAssert(padded != NULL && message != NULL);

    // A value of zero is not allowed because the KDF can’t produce a result
    // if the digest size is zero.
    if(hLen <= 0)
        return TPM_RC_VALUE;

    // Basic size checks
    // make sure digest isn’t too big for key size
    if(padded->size < (2 * hLen) + 2)
        ERROR_RETURN(TPM_RC_HASH);

    // and that message will fit messageSize <= k - 2hLen - 2
    if(message->size > (padded->size - (2 * hLen) - 2))
        ERROR_RETURN(TPM_RC_VALUE);

    // Hash L even if it is null
    // Offset into padded leaving room for masked seed and byte of zero
    pp = &padded->buffer[hLen + 1];
    if(CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
        hLen, pp) != hLen)
        ERROR_RETURN(TPM_RC_FAILURE);

    // The total size of db = hLen + pad + mSize;
    dbSize = hLen + padLen + message->size;

    // If testing, then use the provided seed. Otherwise, use values
    // from the RNG
    CryptRandomGenerate(hLen, mySeed);
```
DBG_Generate(rand, mySeed, (UINT16)hLen);

if(g_inFailureMode)
   ERROR_RETURN(TPM_RC_FAILURE);

// mask = MGF1 (seed, nSize hLen 1)
CryptMGF1(dbSize, mask, hashAlg, hLen, seed);

// Create the masked db
pm = mask;
for(i = dbSize; i > 0; i--)
   *pp++ ^= *pm++;
pp = &padded->buffer[hLen + 1];

// Run the masked data through MGF1
if(CryptMGF1(hLen, &padded->buffer[1], hashAlg, dbSize, pp) != (unsigned)hLen)
   ERROR_RETURN(TPM_RC_VALUE);

// Now XOR the seed to create masked seed
pp = &padded->buffer[1];
pm = seed;
for(i = hLen; i > 0; i--)
   *pp++ ^= *pm++;

// Set the first byte to zero
padded->buffer[0] = 0x00;
Exit:
return retVal;

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_VALUE</td>
<td>the value to decode was larger than the modulus, or the padding is wrong or the buffer to receive the results is too small</td>
</tr>
</tbody>
</table>

static TPM_RC
OaepDecode(
TPM2B *dataOut,    // OUT: the recovered data
TPM_ALG_ID hashAlg, // IN: algorithm to use for padding
const TPM2B *label, // IN: null-terminated string (may be NULL)
TPM2B *padded      // IN: the padded data
)

i;
BYTE seedMask[MAX_DIGEST_SIZE];
UINT32 hLen = CryptHashGetDigestSize(hashAlg);
BYTE mask[MAX_RSA_KEY_BYTES];
BYTE *pp;
BYTE *pm;
TPM_RC retVal = TPM_RC_SUCCESS;

// Strange size (anything smaller can't be an OAEP padded block)
// Also check for no leading 0
if((padded->size < (unsigned)((2 * hLen) + 2)) || (padded->buffer[0] != 0))
   ERROR_RETURN(TPM_RC_VALUE);

// Use the hash size to determine what to put through MGF1 in order
// to recover the seedMask
CryptMGF1(hLen, seedMask, hashAlg, padded->size - hLen - 1,
   &padded->buffer[hLen + 1]);
300 // Recover the seed into seedMask
301 pAssert(hLen <= sizeof(seedMask));
302 pp = &padded->buffer[1];
303 pm = seedMask;
304 for (i = hLen; i > 0; i--)
305     *pm++ ^= *pp++;
306
307 // Use the seed to generate the data mask
308 CryptMGF1(padded->size - hLen - 1, mask, hashAlg, hLen, seedMask);
309
310 // Use the seed generated from seed to recover the padded data
311 pp = &padded->buffer[hLen + 1];
312 pm = mask;
313 for (i = (padded->size - hLen - 1); i > 0; i--)
314     *pm++ ^= *pp++;
315
316 // Make sure that the recovered data has the hash of the label
317 // Put trial value in the seed mask
318 if ((CryptHashBlock(hashAlg, label->size, (BYTE *)label->buffer,
319     hLen, seedMask)) != hLen)
320     FAIL(FATAL_ERROR_INTERNAL);
321 if (memcmp(seedMask, mask, hLen) != 0)
322     ERROR_RETURN(TPM_RC_VALUE);
323
324 // find the start of the data
325 pm = &mask[hLen];
326 for (i = (UINT32)padded->size - (2 * hLen) - 1; i > 0; i--)
327 {
328     if (*pm++ != 0)
329         break;
330 }
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 memcp(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

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_VALUE</td>
<td>message size is too large</td>
</tr>
</tbody>
</table>

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 )
{  UINT32 ps = padded->size - message->size - 3;
  // if(message->size > padded->size - 11)
  return TPM_RC_VALUE;
  // move the message to the end of the buffer
  memcpy(&padded->buffer[padded->size - message->size], message->buffer,
         message->size);
  // Set the first byte to 0x00 and the second to 0x02
  padded->buffer[0] = 0;
  padded->buffer[1] = 2;
  // Fill with random bytes
  DRBG_Generate(rand, &padded->buffer[2], (UINT16)ps);
  if(g_inFailureMode)
    return TPM_RC_FAILURE;
  // Set the delimiter for the random field to 0
  padded->buffer[2 + ps] = 0;
  // Now, the only messy part. Make sure that all the 'ps' bytes are non-zero
  // In this implementation, use the value of the current index
  for(ps++; ps > 1; ps--)
  {
    if(padded->buffer[ps] == 0)
      break;
  }
  psSize++;
  // Make sure that pSize has not gone over the end and that there are at least 8
  // bytes of pad data.
  fail = (pSize > coded->size) | fail;
  fail = (pSize - 2 < 8) | fail;
  if((message->size < (UINT16)(coded->size - pSize)) || fail)
    return TPM_RC_FAIL;
}

10.2.17.4.12 RSAES_Decode()

This function performs the decoding for RSAES-PKCS1-V1_5-ENCRYPT as defined in PKCS#1V2.1

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_FAIL</td>
<td>decoding error or results would no fit into provided buffer</td>
</tr>
</tbody>
</table>

static TPM_RC RSAES_Decode(
    TPM2B *message,  // OUT: the recovered message
    TPM2B *coded    // IN: the encoded message
)
{
    BOOL fail = FALSE;
    UINT16 pSize;
    fail = (coded->size < 11);
    fail = (coded->buffer[0] != 0x00) | fail;
    fail = (coded->buffer[1] != 0x02) | fail;
    for(pSize = 2; pSize < coded->size; pSize++)
    {
      if(coded->buffer[pSize] == 0)
        break;
    }
    pSize++;
    // Make sure that pSize has not gone over the end and that there are at least 8
    // bytes of pad data.
    fail = (pSize > coded->size) | fail;
    fail = (pSize - 2 < 8) | fail;
    if((message->size < (UINT16)(coded->size - pSize)) || fail)
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.

```c
INT16 CryptRsaPssSaltSize(
    INT16 hashSize,
    INT16 outSize
)
{
    INT16 saltSize;

    // (Mask Length) = (outSize - hashSize - 1);
    // Max saltSize is (Mask Length) - 1
    saltSize = (outSize - hashSize - 1) - 1;
    // Use the maximum salt size allowed by FIPS 186-4
    if(saltSize > hashSize)
        saltSize = hashSize;
    else if(saltSize < 0)
        saltSize = 0;
    return saltSize;
}
```

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.

```c
static TPM_RC
PssEncode(
    TPM2B *out,    // OUT: the encoded buffer
    TPM_ALG_ID hashAlg, // IN: hash algorithm for the encoding
    TPM2B *digest, // IN: the digest
    RAND_STATE *rand   // IN: random number source
)
{
    UINT32 hLen = CryptHashGetDigestSize(hashAlg);
    BYTE salt[MAX_RSA_KEY_BYTES - 1];
    UINT16 saltSize;
    BYTE *ps = salt;
    BYTE *pOut;
    UINT16 mLen;
    HASH_STATE hashState;

    // These are fatal errors indicating bad TPM firmware
    pAssert(out != NULL && hLen > 0 && digest != NULL);

    // Get the size of the mask
    mLen = (UINT16)(out->size - hLen - 1);

    // Set the salt size
    saltSize = CryptRsaPssSaltSize((INT16)hLen, (INT16)mLen);
    //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] = 0x00;
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 enforce by this implementation. This implementation does allow for a
variable size salt value to have been used by the creator of the signature.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SCHEME</td>
<td>hashAlg is not a supported hash algorithm</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>decode operation failed</td>
</tr>
</tbody>
</table>

```c
499 static TPM_RC
500 PssDecode(
501     TPM_ALG_ID   hashAlg,   // IN: hash algorithm to use for the encoding
502     TPM2B        *dIn,      // In: the digest to compare
503     TPM2B        *eIn      // IN: the encoded data
504 )
505 {
506     UINT32       hLen = CryptHashGetDigestSize(hashAlg);
507     BYTE         mask[MAX_RSA_KEY_BYTES];
```
BYTE *pm = mask;
BYTE *pe;
BYTE pad[8] = {0};
UINT32 i;
UINT32 mLen;
BYTE fail;
TPM_RC retVal = TPM_RC_SUCCESS;
HASH_STATE hashState;

// These errors are indicative of failures due to programmer error
pAssert(dIn != NULL && eIn != NULL);
pe = eIn->buffer;

// check the hash scheme
if (hLen == 0)
    ERROR_RETURN(TPM_RC_SCHEME);

// most significant bit must be zero
fail = pe[0] & 0x80;

// last byte must be 0xbc
fail |= pe[eIn->size - 1] ^ 0xbc;

// Use the hLen bytes at the end of the buffer to generate a mask
// Doesn't start at the end which is a flag byte
mLen = eIn->size - hLen - 1;
CryptMGF1(mLen, mask, hashAlg, hLen, &pe[mLen]);

// Clear the MSO of the mask to make it consistent with the encoding.
mask[0] &= 0x7F;

pAssert(mLen <= sizeof(mask));
// XOR the data into the mask to recover the salt. This sequence
// advances eIn so that it will end up pointing to the seed data
// which is the hash of the signature data
for (i = mLen; i > 0; i--)
    *pm++ ^= *pe++;

// Find the first byte of 0x01 after a string of all 0x00
for (pm = mask, i = mLen; i > 0; i--)
    { 
        if (*pm == 0x01)
            break;
        else
            fail |= *pm++;
    }
// i should not be zero
fail |= (i == 0);

// if we have failed, will continue using the entire mask as the salt value so
// that the timing attacks will not disclose anything (I don't think that this
// is a problem for TPM applications but, usually, we don't fail so this
// doesn't cost anything).
if (fail)
    { 
        i = mLen;
        pm = mask;
    }
else
    { 
        pm++;
        i--;
    }
// i contains the salt size and pm points to the salt. Going to use the input
// hash and the seed to recreate the hash in the lower portion of eIn.
CryptHashStart(&hashState, hashAlg);
572  // add the pad of 8 zeros
573  CryptDigestUpdate(&hashState, 8, pad);
574
575  // add the provided digest value
576  CryptDigestUpdate(&hashState, dIn->size, dIn->buffer);
577
578  // and the salt
579  CryptDigestUpdate(&hashState, i, pm);
580
581  // get the result
582  fail |= (CryptHashEnd(&hashState, hLen, mask) != hLen);
583
584  // Compare all bytes
585  for(pm = mask; hLen > 0; hLen--)
586     // don’t use fail = because that could skip the increment and compare
587     // operations after the first failure and that gives away timing
588     // information.
589     fail |= *pm++ ^ *pe++;
590
591  retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;
592 Exit:
593  return retval;
594}

10.2.17.4.16 MakeDerTag()

Construct the DER value that is used in RSASSA

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>size of value</td>
</tr>
<tr>
<td>&lt;= 0</td>
<td>no hash exists</td>
</tr>
</tbody>
</table>

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;
*buffer++ = 0x00;  // Add a NULL
*buffer++ = 0x04;
*buffer++ = (BYTE)(info->digestSize);
return oidSize + 8;

Error:
return 0;

}

10.2.17.4.17 RSASSA_Encode()

Encode a message using PKCS1v1.5 method.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SCHEME</td>
<td>hashAlg is not a supported hash algorithm</td>
</tr>
<tr>
<td>TPM_RC_SIZE</td>
<td>eOutSize is not large enough</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>hInSize does not match the digest size of hashAlg</td>
</tr>
</tbody>
</table>

static TPM_RC

RSASSA_Encode(

TPM2B               *pOut,   // IN:OUT on in, the size of the public key
TPM_ALG_ID           hashAlg, // IN: hash algorithm for PKCS1v1_5
TPM2B               *hIn     // IN: digest value to encode
)

BYTE             DER[20];
BYTE             *der = DER;
INT32            derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
BYTE             *eOut;
INT32            fillSize;
TPM_RC           retVal = TPM_RC_SUCCESS;

// Can't use this scheme if the algorithm doesn't have a DER string defined.
if(derSize == 0)
  ERROR_RETURN(TPM_RC_SCHEME);

// If the digest size of 'hashAlg' doesn't match the input digest size, then
// the DER will misidentify the digest so return an error
if(CryptHashGetDigestSize(hashAlg) != hIn->size)
  ERROR_RETURN(TPM_RC_VALUE);
fillSize = pOut->size - derSize - hIn->size - 3;
eOut = pOut->buffer;

// Make sure that this combination will fit in the provided space
if(fillSize < 8)
  ERROR_RETURN(TPM_RC_SIZE);

// Start filling
*eOut++ = 0; // initial byte of zero
*eOut++ = 1; // byte of 0x01
for(; fillSize > 0; fillSize--)
  *eOut++ = 0xff; // bunch of 0xff
*eOut++ = 0; // another 0
for(; derSize > 0; derSize--)
  *eOut++ = *der++; // copy the DER
der = hIn->buffer;
for(fillSize = hIn->size; fillSize > 0; fillSize--)
  *eOut++ = *der++; // copy the hash

Exit:
return retVal;
}

10.2.17.4.18 RSASSA_Decode()

This function performs the RSASSA decoding of a signature.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_VALUE</td>
<td>decode unsuccessful</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>hasAlg is not supported</td>
</tr>
</tbody>
</table>

static TPM_RC
RSASSA_Decode(

TPM_ALG_ID       hashAlg,       // IN: hash algorithm to use for the encoding
TPM2B           *hIn,         // In: the digest to compare
TPM2B           *eIn           // IN: the encoded data
)
{
    BYTE fail;
    BYTE DER[20];
    BYTE *pe;
    INT32 derSize = MakeDerTag(hashAlg, sizeof(DER), DER);
    BYTE *digest;
    UINT16 digestSize;
    pAssert(hIn != NULL && eIn != NULL);
    pe = eIn->buffer;

    // Can't use this scheme if the algorithm doesn't have a DER string
    // defined or if the provided hash isn't the right size
    if(derSize == 0 || (unsigned)hashSize != hIn->size)
        ERROR_RETURN(TPM_RC_SCHEME);

    // Make sure that this combination will fit in the provided space
    // Since no data movement takes place, can just walk though this
    // and accept nearly random values. This can only be called from
    // CryptValidateSignature() so eInSize is known to be in range.
    fillSize = eIn->size - derSize - hashSize - 3;

    // Start checking (fail will become non-zero if any of the bytes do not have
    // the expected value.
    fail = *pe++;       // initial byte of zero
    fail |= *pe++ ^ 1;  // byte of 0x01
    for(; fillSize > 0; fillSize--)
        fail |= *pe++ ^ 0xff;  // bunch of 0xFF
    fail |= *pe++;      // another 0
    for(; derSize > 0; derSize--)
        fail |= *pe++ ^ *der++;  // match the DER
    digestSize = hIn->size;
    digest = hIn->buffer;
    for(; digestSize > 0; digestSize--)
        fail |= *pe++ ^ *digest++;  // match the hash
    retVal = (fail != 0) ? TPM_RC_VALUE : TPM_RC_SUCCESS;

    Exit:
    return retVal;
}
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.

```c
TPMT_RSA_DECRYPT* CryptRsaSelectScheme(
    TPMI_DH_OBJECT rsaHandle, // IN: handle of an RSA key
    TPMublicrypt *)scheme // IN: a sign or decrypt scheme
    )
{
    OBJECT *rsaObject;
    TPMT_ASYM_SCHEME *keyScheme;
    TPMT_RSA_DECRYPT *retVal = NULL;

    // Get sign object pointer
    rsaObject = HandleToObject(rsaHandle);
    keyScheme = &rsaObject->publicArea.parameters.asymDetail.scheme;

    // if the default scheme of the object is TPM_ALG_NULL, then select the
    // input scheme
    if(keyScheme->scheme == TPM_ALG_NULL)
    {
        retVal = scheme;
    }

    // if the object scheme is not TPM_ALG_NULL and the input scheme is
    // TPM_ALG_NULL, then select the default scheme of the object.
    else if(scheme->scheme == TPM_ALG_NULL)
    {
        // if input scheme is NULL
        retVal = (TPMT_RSA_DECRYPT *)keyScheme;
    }

    // get here if both the object scheme and the input scheme are
    // not TPM_ALG_NULL. Need to insure that they are the same.
    // IMPLEMENTATION NOTE: This could cause problems if future versions have
    // schemes that have more values than just a hash algorithm. A new function
    // (IsSchemeSame()) might be needed then.
    else if(keyScheme->scheme == scheme
        && keyScheme->details.anySig.hashAlg == scheme->details.anySig.hashAlg)
    {
        retVal = scheme;
    }

    // two different, incompatible schemes specified will return NULL
    return retVal;
}
```

10.2.17.5.2 CryptRsaLoadPrivateExponent()

This function is called to generate the private exponent of an RSA key.
### Error Returns

<table>
<thead>
<tr>
<th>Error Return</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_BINDING</td>
<td>public and private parts of rsaKey are not matched</td>
</tr>
</tbody>
</table>

```c
TPM_RC
CryptRsaLoadPrivateExponent(
    TPMT_PUBLIC             *publicArea,
    TPMT_SENSITIVE          *sensitive
)
{
    if((sensitive->sensitive.rsa.t.size & RSA_prime_flag) == 0)
    {
        if((sensitive->sensitive.rsa.t.size * 2) == publicArea->unique.rsa.t.size)
        {
            NEW_PRIVATE_EXPONENT(Z);
            BN_RSA_INITIALIZED(bnN, &publicArea->unique.rsa);
            BN_RSA(bnQr);
            BN_VAR(bnE, RADIX_BITS);
            TEST(ALG_NULL_VALUE);
            VERIFY((sensitive->sensitive.rsa.t.size * 2) == publicArea->unique.rsa.t.size);
            // Initialize the exponent
            BnSetWord(bnE, publicArea->parameters.rsaDetail.exponent);
            if(BnEqualZero(bnE))
                BnSetWord(bnE, RSA_DEFAULT_PUBLIC_EXPONENT);
            // Convert first prime to 2B
            VERIFY(BnFrom2B(Z->P, &sensitive->sensitive.rsa.b) != NULL);
            // Find the second prime by division. This uses 'bQ' rather than Z->Q
            // because the division could make the quotient larger than a prime during
            // some intermediate step.
            VERIFY(BnDiv(Z->Q, bnQr, bnN, Z->P));
            VERIFY(BnEqualZero(bnQr));
            // Compute the private exponent and return it if found
            VERIFY(ComputePrivateExponent(bnE, Z));
            VERIFY(PackExponent(&sensitive->sensitive.rsa, Z));
        }
        else
        {
            VERIFY(((sensitive->sensitive.rsa.t.size / 5) * 2)
                == publicArea->unique.rsa.t.size);
            sensitive->sensitive.rsa.t.size |= RSA_prime_flag;
        }
    }
    return TPM_RC_SUCCESS;
    Error:
    return TPM_RC_BINDING;
}
```

### 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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_VALUE</td>
<td>$cOutSize$ is too small (must be the size of the modulus)</td>
</tr>
<tr>
<td>TPM_RC SCHEME</td>
<td>$padType$ is not a supported scheme</td>
</tr>
</tbody>
</table>

```c
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    TPM_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 // if the input and output buffers are the same, copy the input to a scratch
825 // buffer so that things don’t get messed up.
826 if (dIn == &cOut->b)
827 {
828    MemoryCopy2B(&dataIn.b, dIn, sizeof(dataIn.t.buffer));
829    dIn = &dataIn.b;
830 }
831 // All encryption schemes return the same size of data
832 cOut->t.size = key->publicArea.unique.rsa.t.size;
833 TEST(scheme->scheme);
834 switch(scheme->scheme)
835 {
836     case ALG_NULL_VALUE: // 'raw' encryption
837         {
838             INT32 i;
839             INT32 dSize = dIn->size;
840             // dIn can have more bytes than cOut as long as the extra bytes
841             // are zero. Note: the more significant bytes of a number in a byte
842             // buffer are the bytes at the start of the array.
843             for(i = 0; (i < dSize) && (dIn->buffer[i] == 0); i++);
844             dSize -= i;
845             if(dSize > cOut->t.size)
846                 ERROR_RETURN(TPM_RC_VALUE);
847             // Pad cOut with zeros if dIn is smaller
848             memset(cOut->t.buffer, 0, cOut->t.size - dSize);
849             // And copy the rest of the value
850             memcpy(&cOut->t.buffer[cOut->t.size - dSize], &dIn->buffer[i], dSize);
851             // If the size of dIn is the same as cOut dIn could be larger than
852             // the modulus. If it is, then RSAEP() will catch it.
853         }
854     break;
855     case ALG_RSAES_VALUE:
856         retVal = RSAES_PKCS1v1_5Encode(&cOut->b, dIn, rand);
857     break;
858     case ALG_OAEP_VALUE:
859         retVal = OaepEncode(&cOut->b, scheme->details.oaep.hashAlg, label, dIn,
860                             rand);
861     ...  
```
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIZE</td>
<td>cInSize is not the same as the size of the public modulus of key; or numeric value of the encrypted data is greater than the modulus</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>dOutSize is not large enough for the result</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>padType is not supported</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT TPM_RC
CryptRsaDecrypt(
    TPM2B               *dOut,     // OUT: the decrypted data
    TPM2B               *cIn,      // IN: the data to decrypt
    OBJECT              *key,      // IN: the key to use for decryption
    TPMT_RSA_DECRYPT    *sch,      // IN: the padding scheme
    const TPM2B         *label     // IN: in case it is needed for the scheme
) {

    TPM_RC                 retVal;

    // Make sure that the necessary parameters are provided
    pAssert(cIn != NULL && dOut != NULL && key != NULL);

    // Size is checked to make sure that the encrypted value is the right size
    if(cIn->size != key->publicArea.unique.rsa.t.size)
        ERROR_RETURN(TPM_RC_SIZE);

    TEST(sch->scheme);

    // For others that do padding, do the decryption in place and then
    // go handle the decoding.
    retVal = RSADP(cIn, key);
    if(retVal == TPM_RC_SUCCESS)
    {
        // Remove padding
        switch(sch->scheme)
        {
            case ALG_NULL_VALUE:
                if(dOut->size < cIn->size)
                    return TPM_RC_VALUE;
                MemoryCopy2B(dOut, cIn, dOut->size);
                break;
            case ALG_RSAES_VALUE:
                retVal = RSAES_Decode(dOut, cIn);
                break;
            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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SCHEME</td>
<td>scheme or hashAlg are not supported</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>hInSize does not match hashAlg (for RSASSA)</td>
</tr>
</tbody>
</table>

LIB_EXPORT TPM_RC
CryptRsaSign(
   TPM_SIGNATURE  *sigOut,
   OBJECT          *key,      // IN: key to use
   TPM2B_DIGEST    *hIn,      // IN: the digest to sign
   RAND_STATE      *rand      // IN: the random number generator
   //      to use (mostly for testing)
   })
927     TPM_RC                retVal = TPM_RC_SUCCESS;
928     UINT16                modSize;
929     // parameter checks
930     pAssert(sigOut != NULL && key != NULL && hIn != NULL);
931     modSize = key->publicArea.unique.rsa.t.size;
932     // for all non-null signatures, the size is the size of the key modulus
933     sigOut->signature.rsapss.sig.t.size = modSize;
934     TEST(sigOut->sigAlg);
935     switch(sigOut->sigAlg)
936     {
937       case ALG_NULL_VALUE:
938         sigOut->signature.rsapss.sig.t.size = 0;
939         return TPM_RC_SUCCESS;
940       case ALG_RSAPSS_VALUE:
941         retVal = PssEncode(&sigOut->signature.rsapss.sig.b,
942          sigOut->signature.rsapss.hash, &hIn->b, rand);
943         break;
944       case ALG_RSASSA_VALUE:
945         retVal = RSASSA_Encode(&sigOut->signature.rsassa.sig.b,
946          sigOut->signature.rsassa.hash, &hIn->b);
947         break;
948       default:
949         retVal = TPM_RC_SCHEME;
950     }
951     if(retVal == TPM_RC_SUCCESS)
952     {
953       // Do the encryption using the private key
954       retVal = RSADP(&sigOut->signature.rsapss.sig.b, key);
955     }
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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIGNATURE</td>
<td>the signature does not check</td>
</tr>
<tr>
<td>TPM_RC_SCHEME</td>
<td>unsupported scheme or hash algorithm</td>
</tr>
</tbody>
</table>

LIB_EXPORT TPM_RC
CryptRsaValidateSignature(
    TPMT_SIGNATURE *sig,  // IN: signature
    OBJECT *key,         // IN: public modulus
    TPM2B_DIGEST *digest  // IN: The digest being validated
)
{
    TPM_RC            retVal;
    // Fatal programming errors
    pAssert(key != NULL && sig != NULL && digest != NULL);
    switch(sig->sigAlg)
    {
        case ALG_RSAPSS_VALUE:
        case ALG_RSASSA_VALUE:
            break;
        default:
            return TPM_RC_SCHEME;
    }

    // Errors that might be caused by calling parameters
    if(sig->signature.rsassa.sig.t.size != key->publicArea.unique.rsa.t.size)
        ERROR_RETURN(TPM_RC_SIGNATURE);

    TEST(sig->sigAlg);

    // Decrypt the block
    retVal = RSAEP(&sig->signature.rsassa.sig.b, key);
    if(retVal == TPM_RC_SUCCESS)
    {
        switch(sig->sigAlg)
        {
            case ALG_RSAPSS_VALUE:
                retVal = PssDecode(sig->signature.any.hashAlg, &digest->b,
                    &sig->signature.rsassa.sig.b);
                break;
            case ALG_RSASSA_VALUE:
                retVal = RSASSA_Decode(sig->signature.any.hashAlg, &digest->b,
                    &sig->signature.rsassa.sig.b);
                break;
            default:
                return TPM_RC_SCHEME;
        }
    }

    Exit:
    return (retVal != TPM_RC_SUCCESS) ? TPM_RC_SIGNATURE : TPM_RC_SUCCESS;
}
#else
extern int s_rsaKeyCacheEnabled;
int GetCachedRsaKey(TPMT_PUBLIC *publicArea, TPMT_SENSITIVE *sensitive, 
                  RAND_STATE *rand);
#define GET_CACHED_KEY(publicArea, sensitive, rand) 
    (s_rsaKeyCacheEnabled && GetCachedRsaKey(publicArea, sensitive, rand))
#define GET_CACHED_KEY(key, rand)

10.2.17.5.7 CryptRsaGenerateKey()

Generate an RSA key from a provided seed

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_CANCELED</td>
<td>operation was canceled</td>
</tr>
<tr>
<td>TPM_RC_RANGE</td>
<td>public exponent is not supported</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>could not find a prime using the provided parameters</td>
</tr>
</tbody>
</table>

LIB_EXPORT TPM_RC
CryptRsaGenerateKey(
    TPMT_PUBLIC *publicArea,
    TPMT_SENSITIVE *sensitive,
    RAND_STATE *rand) // IN: if not NULL, the deterministic
                      // RNG state
{
    UINT32 i;
    BN_RSA(bnD);
    BN_RSA(bnN);
    BN_WORD(bnPubExp);
    UINT32 e = publicArea->parameters.rsaDetail.exponent;
    int keySizeInBits;
    TPM_RC retVal = TPM_RC_NO_RESULT;
    NEW_PRIVATE_EXPONENT(Z);
    // Need to make sure that the caller did not specify an exponent that is
    // not supported
    e = publicArea->parameters.rsaDetail.exponent;
    if (e == 0)
        e = RSA_DEFAULT_PUBLIC_EXPONENT;
    else
    {
        if (e < 65537)
            ERROR_RETURN(TPM_RC_RANGE);
        // Check that e is prime
        if (!IsPrimeInt(e))
            ERROR_RETURN(TPM_RC_RANGE);
    }
    BnSetWord(bnPubExp, e);
    // check for supported key size.
    keySizeInBits = publicArea->parameters.rsaDetail.keyBits;
    if (((keySizeInBits % 1024) != 0) 
        || (keySizeInBits > MAX_RSA_KEY_BITS) // this might be redundant, but...
        || (keySizeInBits == 0))
        ERROR_RETURN(TPM_RC_VALUE);
    // Set the prime size for instrumentation purposes
    INSTRUMENT_SET(PrimeIndex, PRIME_INDEX(keySizeInBits / 2));
    #if SIMULATION && USE_RSA_KEY_CACHE
    if (GET_CACHED_KEY(publicArea, sensitive, rand))
return TPM_RC_SUCCESS;
#endif

Make sure that key generation has been tested
TEST(ALG_NULL_VALUE);

The prime is computed in P. When a new prime is found, Q is checked to
see if it is zero. If so, P is copied to Q and a new P is found.
When both P and Q are non-zero, the modulus and
private exponent are computed and a trial encryption/decryption is
performed. If the encrypt/decrypt fails, assume that at least one of the
primes is composite. Since we don't know which one, set Q to zero and start
over and find a new pair of primes.

for(i = 1; (retVal == TPM_RC_NO_RESULT) && (i != 100); i++)
{
    if(_plat__IsCanceled())
        ERROR_RETURN(TPM_RC_CANCELED);
    if(BnGeneratePrimeForRSA(Z->P, keySizeInBits / 2, e, rand) == TPM_RC_FAILURE)
    {
        retVal = TPM_RC_FAILURE;
        goto Exit;
    }
    INSTRUMENT_INC(PrimeCounts[PrimeIndex]);
    // If this is the second prime, make sure that it differs from the
    // first prime by at least 2^100
    if(BnEqualZero(Z->Q))
    {
        // copy p to q and compute another prime in p
        BnCopy(Z->Q, Z->P);
        continue;
    }
    // Make sure that the difference is at least 100 bits. Need to do it this
    // way because the big numbers are only positive values
    if(BnUnsignedCmp(Z->P, Z->Q) < 0)
        BnSub(bnD, Z->Q, Z->P);
    else
        BnSub(bnD, Z->P, Z->Q);
    if(BnMsb(bnD) < 100)
        continue;
    // Form the public modulus and set the unique value
    BnMult(bnN, Z->P, Z->Q);
    BnTo2B(bnN, &publicArea->unique.rsa.b,
        (NUMBYTES)BITS_TO_BYTES(keySizeInBits));
    // Make sure everything came out right. The MSb of the values must be one
    if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
        || (publicArea->unique.rsa.t.size
            != (NUMBYTES)BITS_TO_BYTES(keySizeInBits)))
        FAIL(FATAL_ERROR_INTERNAL);
    // Make sure that we can form the private exponent values
    if(ComputePrivateExponent(bnPubExp, Z) != TRUE)
    {
        // If ComputePrivateExponent could not find an inverse for
        // Q, then copy P and recompute P. This might
        // cause both to be recomputed if P is also zero
        if(BnEqualZero(Z->Q))
            BnCopy(Z->Q, Z->P);
            continue;
    }
    // Pack the private exponent into the sensitive area
PackExponent(&sensitive->sensitive.rsa, Z);

// Make sure everything came out right. The MSb of the values must be one
if(((publicArea->unique.rsa.t.buffer[0] & 0x80) == 0)
    || ((sensitive->sensitive.rsa.t.buffer[0] & 0x80) == 0))
    FAIL(FATAL_ERROR_INTERNAL);

retVal = TPM_RC_SUCCESS;

// Do a trial encryption decryption if this is a signing key
if(IS_ATTRIBUTE(publicArea->objectAttributes, TPMA_OBJECT, sign))
{

    BN_RSA(temp1);
    BN_RSA(temp2);
    BnGenerateRandomInRange(temp1, bnN, rand);

    // Encrypt with public exponent...
    BnModExp(temp2, temp1, bnPubExp, bnN);
    // ... then decrypt with private exponent
    RsaPrivateKeyOp(temp2, Z);

    // If the starting and ending values are not the same,
    // start over ;-
    if(BnUnsignedCmp(temp2, temp1) != 0)
    {
        BnSetWord(Z->Q, 0);
        retVal = TPM_RC_NO_RESULT;
    }

    exit:
    return retVal;
}

#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

```c
#define _CRYPT_HASH_C_
#include "Tpm.h"
#if SMAC_IMPLEMENTED
```

10.2.18.2.1 CryptSmacStart()

Function to start an SMAC.

```c
UINT16 CryptSmacStart( HASH_STATE *state,
TPMU_PUBLIC_PARMS *keyParameters,
TPM_ALG_ID macAlg,       // IN: the type of MAC
TPM2B *key
)
{
    UINT16 retVal = 0;
    // Make sure that the key size is correct. This should have been checked
    // at key load, but...
    if (BITS_TO_BYTES(keyParameters->symDetail.sym.keyBits.sym) == key->size)
    {
        switch (macAlg)
        {
        #if ALG_CMAC
        case ALG_CMAC_VALUE:
            retVal = CryptCmacStart(&state->state.smac, keyParameters,
                                      macAlg, key);
            break;
        #endif
        default:
                break;
        }
        state->type = (retVal != 0) ? HASH_STATE_SMAC : HASH_STATE_EMPTY;
        return retVal;
    }
```

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.

```c
UINT16 CryptMacStart( HMAC_STATE *state,
TPMU_PUBLIC_PARMS *keyParameters,
TPM_ALG_ID macAlg,       // IN: the type of MAC
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

```c
#include "Tpm.h"
#include "CryptSym.h"
define KEY_BLOCK_SIZES(ALG, alg)
static const INT16 alg##KeyBlockSizes[] = {
    ALG##_KEY_SIZE_BITS, -1, ALG##_BLOCK_SIZES }
#endif // ALG_AES
#if ALG_SM4
    KEY_BLOCK_SIZES(SM4, sm4);
#endif
#if ALG_CAMELLIA
    KEY_BLOCK_SIZES(CAMELLIA, camellia);
#endif
#if ALG_TDES
    KEY_BLOCK_SIZES(TDES, tdes);
#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

```c
BOOL CryptSymInit( void )
{
    return TRUE;
}
```

10.2.19.3.2 CryptSymStartup()

This function is called to do TPM2_Startup() processing

```c
BOOL CryptSymStartup( void )
{
    return TRUE;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 0</td>
<td>cipher not supported</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>the cipher block size in bytes</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT INT16
CryptGetSymmetricBlockSize(
    TPM_ALG_ID symmetricAlg, // IN: the symmetric algorithm
    UINT16 keySizeInBits      // IN: the key size
)
{
    const INT16 *sizes;
    INT16 i;

    #define ALG_CASE(SYM, sym) case ALG_##SYM##_VALUE: sizes = sym##KeyBlockSizes; break
    switch(symmetricAlg)
    {
        #if ALG_AES
            ALG_CASE(AES, aes);
        #endif
        #if ALG_SM4
            ALG_CASE(SM4, sm4);
        #endif
        #if ALG_CAMELLIA
            ALG_CASE(CAMELLIA, camellia);
        #endif
        #if ALG_TDES
            ALG_CASE(TDES, tdes);
        #endif
        default:
        {
            return 0;
        }
        // Find the index of the indicated keySizeInBits
        for(i = 0; *sizes >= 0; i++, sizes++)
        {
            if(*sizes == keySizeInBits)
                break;
        }
        // If sizes is pointing at the end of the list of key sizes, then the desired key size was not found so set the block size to zero.
        if(*sizes++ < 0)
            return 0;
        // Advance until the end of the list is found
        while(*sizes++ >= 0);
        // sizes is pointing to the first entry in the list of block sizes. Use the ith index to find the block size for the corresponding key size.
        return sizes[i];
    }
}
```

10.2.19.4 Symmetric Encryption

This function performs symmetric encryption based on the mode.
<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_SIZE</td>
<td>dSize is not a multiple of the block size for an algorithm that requires it</td>
</tr>
<tr>
<td>TPM_RC_FAILURE</td>
<td>Fatal error</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT TPM_RC
CryptSymmetricEncrypt(
    BYTE *dOut,       // OUT:
    TPM_ALG_ID algorithm, // IN: the symmetric algorithm
    UINT16 keySizeInBits, // IN: key size in bits
    const BYTE *key,    // IN: key buffer. The size of this buffer
    int i;              // in bytes is (keySizeInBits + 7) / 8
    TPM2B_IV *ivInOut,  // IN/OUT: IV for decryption.
    TPM_ALG_ID mode,    // IN: Mode to use
    INT32 dSize,        // IN: data size (may need to be a multiple of the blockSize)
    const BYTE *dIn)    // IN: data buffer
    {
        BYTE *pT;
        BYTE tmp[MAX_SYM_BLOCK_SIZE];
        BYTE *pIv;
        int i;
        BYTE *iv = defaultIv;
        pIv = iv;
        // Create encrypt key schedule and set the encryption function pointer.
        SELECT(ENCRYPT);
        switch(mode)
        {
            #if ALG_CTR
                case ALG_CTR_VALUE:
                    for(; dSize > 0; dSize -= blockSize)
                    {
                        // Encrypt the current value of the IV(counter)
                        ENCRYPT(&keySchedule, iv, tmp);
                    }
                    //increment the counter (counter is big-endian so start at end)
```
for(i = blockSize - 1; i >= 0; i--)
    if((iv[i] += 1) != 0)
        break;
    // XOR the encrypted counter value with input and put into output
    pT = tmp;
    for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
        *dOut++ = *dIn++ ^ *pT++;
}
break;
#endif
#if ALG_OFB
    case ALG_OFB_VALUE:
        // This is written so that dIn and dOut may be the same
        for(; dSize > 0; dSize -= blockSize)
        {
            // Encrypt the current value of the "IV"
            ENCRYPT(&keySchedule, iv, iv);
            // XOR the encrypted IV into dIn to create the cipher text (dOut)
            pIv = iv;
            for(i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                *dOut++ = (*pIv++ ^ *dIn++);
        }
        break;
#endif
#if ALG_CBC
    case ALG_CBC_VALUE:
        // For CBC the data size must be an even multiple of the cipher block size
        if((dSize % blockSize) != 0)
            return TPM_RC_SIZE;
        // XOR the data block into the IV, encrypt the IV into the IV and then copy the IV to the output
        for(; dSize > 0; dSize -= blockSize)
        {
            pIv = iv;
            for(i = blockSize; i > 0; i--)
                *pIv++ ^= *dIn++;
            ENCRYPT(&keySchedule, iv, iv);
            pIv = iv;
            for(i = blockSize; i > 0; i--)
                *dOut++ = *pIv++;
        }
        break;
#endif
    // CFB is not optional
    case ALG_CFB_VALUE:
        // Encrypt the IV into the IV, XOR in the data, and copy to output
        for(; dSize > 0; dSize -= blockSize)
        {
            // Encrypt the current value of the IV
            ENCRYPT(&keySchedule, iv, iv);
            pIv = iv;
            for(i = (int)(dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                *pIv++ ^= *dIn++;
            // XOR the data into the IV to create the cipher text
            // and put into the output
            *dOut++ = *pIv++ ^ *dIn++;
        }
        // If the inner loop (i loop) was smaller than blockSize, then dSize would have been smaller than blockSize and it is now negative. If it is negative, then it indicates how many bytes are needed to pad out the IV for the next round.
        for(; dSize < 0; dSize++)
            *pIv++ = 0;
        break;
#endif
#if ALG_ECB

case ALG_ECB_VALUE:
    // For ECB the data size must be an even multiple of the
    // cipher block size
    if((dSize % blockSize) != 0)
        return TPM_RC_SIZE;
    // Encrypt the input block to the output block
    for(; dSize > 0; dSize -= blockSize)
    {
        ENCRYPT(&keySchedule, dIn, dOut);
        dIn = &dIn[blockSize];
        dOut = &dOut[blockSize];
    }
    break;
#endif
default:
    return TPM_RC_FAILURE;
}
return TPM_RC_SUCCESS;

10.2.19.4.1 CryptSymmetricDecrypt()

This function performs symmetric decryption based on the mode.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_FAILURE</td>
<td>A fatal error</td>
</tr>
<tr>
<td>TPM_RCS_SIZE</td>
<td>dSize is not a multiple of the block size for an algorithm that requires it</td>
</tr>
</tbody>
</table>

LIB_EXPORT TPM_RC
CryptSymmetricDecrypt(
BYTE                *dOut,       // OUT: decrypted data
TPM_ALG_ID           algorithm,  // IN: the symmetric algorithm
UINT16               keySizeInBits, // IN: key size in bits
const BYTE          *key,         // IN: key buffer. The size of this buffer
                    // in bytes is (keySizeInBits + 7) / 8
TPM2B_IV            *ivInOut,     // IN/OUT: IV for decryption.
TPM_ALG_ID           mode,        // IN: Mode to use
INT32                dSize,       // IN: data size (may need to be a
                    // multiple of the blockSize)
const BYTE          *dIn          // IN: data buffer
)
{
    BYTE                *pIv;
    int i;
    BYTE                tmp[MAX_SYM_BLOCK_SIZE];
    BYTE                *pT;
    tpmCryptKeySchedule_t    keySchedule;
    INT16                blockSize;
    BYTE                *iv;
    TpmCryptSetSymKeyCall_t        encrypt;
    TpmCryptSetSymKeyCall_t        decrypt;
    BYTE                defaultIv[MAX_SYM_BLOCK_SIZE] = {0};
    // These are used but the compiler can't tell because they are initialized
    // in case statements and it can't tell if they are always initialized
    // when needed, so... Comment these out if the compiler can tell or doesn't
    // care that these are initialized before use.
    encrypt = NULL;
    decrypt = NULL;

    pAssert(dOut != NULL && key != NULL && dIn != NULL);
if (dSize == 0)
    return TPM_RC_SUCCESS;

TEST(algorithm);
blockSize = CryptGetSymmetricBlockSize(algorithm, keySizeInBits);
if (blockSize == 0)
    return TPM_RC_FAILURE;
// If the iv is provided, then it is expected to be block sized. In some cases,
// the caller is providing an array of 0's that is equal to [MAX_SYM_BLOCK_SIZE]
// with no knowledge of the actual block size. This function will set it.
if ((ivInOut != NULL) && (mode != ALG_ECB_VALUE))
{
    ivInOut->t.size = blockSize;
    iv = ivInOut->t.buffer;
}
else
    iv = defaultIv;

pIv = iv;
// Use the mode to select the key schedule to create. Encrypt always uses the
// encryption schedule. Depending on the mode, decryption might use either
// the decryption or encryption schedule.
switch (mode)
{
    #if ALG_CBC || ALG_ECB
        case ALG_CBC_VALUE: // decrypt = decrypt
            case ALG_ECB_VALUE: // For ECB and CBC, the data size must be an even multiple of the
                // cipher block size
            if ((dSize % blockSize) != 0)
                return TPM_RC_SIZE;
            SELECT(DECRYPT);
            break;
    #endif
    default: // For the remaining stream ciphers, use encryption to decrypt
        SELECT(ENCRYPT);
        break;
    #endif

    // Now do the mode-dependent decryption
    switch (mode)
    {
        #if ALG_CBC
            case ALG_CBC_VALUE:
                // Copy the input data to a temp buffer, decrypt the buffer into the
                // output, XOR in the IV, and copy the temp buffer to the IV and repeat.
                for (; dSize > 0; dSize -= blockSize)
                {
                    pT = tmp;
                    for (i = blockSize; i > 0; i--)
                        *pT++ = *dIn++;
                    DECRYPT(&keySchedule, tmp, dOut);
                    pIv = iv;
                    pT = tmp;
                    for (i = blockSize; i > 0; i--)
                    {
                        *dOut++ ^= *pIv;
                        *pIv++ = *pT++;
                    }
                    break;
                }
        #endif
        case ALG_CFB_VALUE:
            for (; dSize > 0; dSize -= blockSize)
            {
                // Encrypt the IV into the temp buffer
                }
ENCYPRT(&keySchedule, iv, tmp);
    pT = tmp;
pIV = iv;
    for (i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
        // Copy the current cipher text to IV, XOR
        // with the temp buffer and put into the output
        *dOut++ = *pT++ ^ (*pIV++ = *dIn++);

    // If the inner loop (i loop) was smaller than blockSize, then dSize
    // would have been smaller than blockSize and it is now negative
    // If it is negative, then it indicates how many fill bytes
    // are needed to pad out the IV for the next round.
    for (; dSize < 0; dSize++)
        *pIV++ = 0;

    break;

#if ALG_CTR
    case ALG_CTR_VALUE:
        for (; dSize > 0; dSize -= blockSize)
            { // Encrypt the current value of the IV(counter)
                ENCRYPT(&keySchedule, iv, tmp);

                //increment the counter (counter is big-endian so start at end)
                for (i = blockSize - 1; i >= 0; i--)
                    if ((iv[i] += 1) != 0)
                        break;

                // XOR the encrypted counter value with input and put into output
                pT = tmp;
                for (i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                    *dOut++ = *dIn++ ^ *pT++;
            }
        break;
#endif

#if ALG_ECB
    case ALG_ECB_VALUE:
        for (; dSize > 0; dSize -= blockSize)
            { // Encrypt the current value of the IV
                ENCRYPT(&keySchedule, iv, tmp);

                // XOR the encrypted IV into dIn to create the cipher text (dOut)
                pIV = iv;
                for (i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                    *dOut++ = (*pIV++ ^ *dIn++);
            }
        break;
#endif

#if ALG_OFB
    case ALG_OFB_VALUE:
        // This is written so that dIn and dOut may be the same
        for (; dSize > 0; dSize -= blockSize)
            { // Encrypt the current value of the "IV"
                ENCRYPT(&keySchedule, iv, iv);

                // XOR the encrypted IV into dIn to create the cipher text (dOut)
                pIV = iv;
                for (i = (dSize < blockSize) ? dSize : blockSize; i > 0; i--)
                    *dOut++ = (*pIV++ ^ *dIn++);
            }
        break;
#endif

    default:
    return TPM_RC_FAILURE;
}
return TPM_RC_SUCCESS;
10.2.19.4.2 CryptSymKeyValidate()

Validate that a provided symmetric key meets the requirements of the TPM

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RC_KEY_SIZE</td>
<td>Key size specifiers do not match</td>
</tr>
<tr>
<td>TPM_RC_KEY</td>
<td>Key is not allowed</td>
</tr>
</tbody>
</table>

```c
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

```c
#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.

```const BN_STRUCT(43 * RADIX_BITS) s_CompositeOfSmallPrimes_ =
{ 44, 44,
 0x2ED42696, 0x2BBF4177, 0x4820594F, 0xF73F4841,
 0xBFA3C313A, 0xC43ECB81, 0xF6F26BF8, 0x7FAB5061,
 0x5974FB7, 0xF71377F6, 0x3B19B59B, 0xCBDD0312,
 0xBB92E1F1, 0x3AC3152C, 0xE87C8273, 0xCOAE0E69,
 0x7A492E95, 0x448CCE86, 0x3CA1907, 0x8A0BF944,
 0xF8C3CBE0, 0xC26F0A5F5, 0xC501C02F, 0x6579441A,
 0xD1099DCA, 0x6BC76A00, 0xC81A3228, 0xBF81A825,
 0x70FA3841, 0x51B3D076, 0xCC2359ED, 0xD9EE0769,
 0x75E47AF0, 0xD45FF31E, 0x52CCE4F6, 0x04DBC991,
 0x9665EE2D, 0x1753EF85, 0x3AE4A5A6, 0x8FD4A97F,
 0x8B15E7EB, 0x0243C3E1, 0xE0FC3D1D, 0x0000000B };
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 do redo IsPrimeInt().

```c
const uint32_t s_LastPrimeInTable = 65537;
const uint32_t s_PrimeTableSize = 4097;
const uint32_t s_PrimesInTable = 6542;
const unsigned char s_PrimeTable[] = {
 0x6e, 0xcb, 0xb4, 0x64, 0x9a, 0x12, 0x6d, 0x81, 0x32, 0xc4, 0x4a, 0x86,
 0x0d, 0x82, 0x96, 0x21, 0xc9, 0x34, 0x04, 0x5a, 0x20, 0x61, 0x89, 0xa4,
 0x44, 0x11, 0x86, 0x29, 0xd1, 0x82, 0x28, 0x4a, 0x30, 0x40, 0x42, 0x32,
 0x21, 0x99, 0x34, 0x08, 0x4b, 0x06, 0x25, 0x42, 0x84, 0x48, 0x8a, 0x14,
 0x05, 0x42, 0x30, 0x6c, 0x08, 0xb4, 0x40, 0x0b, 0xa0, 0x08, 0x51, 0x12,
 0x28, 0x89, 0x04, 0x65, 0x98, 0x30, 0x4c, 0x80, 0x96, 0x44, 0x12, 0x80,
 0x21, 0x42, 0x12, 0x41, 0xc9, 0x04, 0x21, 0xc0, 0x32, 0x2d, 0x96, 0x00,
 0x00, 0x49, 0x04, 0x08, 0x81, 0x96, 0x68, 0x82, 0xb0, 0x25, 0x08, 0x22,
 0x48, 0x89, 0xa2, 0x40, 0x59, 0x26, 0x04, 0x90, 0x06, 0x40, 0x43, 0x30,
 0x44, 0x92, 0x00, 0x69, 0x10, 0x82, 0x08, 0x08, 0xa4, 0x04, 0xa1, 0x12,
 0x60, 0xc0, 0x00, 0x24, 0xd2, 0x22, 0x61, 0x08, 0x84, 0x04, 0x1b, 0x82,
 0x01, 0xd3, 0x10, 0x01, 0x02, 0xa0, 0x44, 0xc0, 0x22, 0x60, 0x91, 0x14,
 0x0c, 0x40, 0xa6, 0x04, 0xd2, 0x94, 0x20, 0x09, 0x94, 0x20, 0x52, 0x00,
 0x08, 0x10, 0xa2, 0x4c, 0x00, 0x81, 0x20, 0x51, 0x10, 0x08, 0xb8, 0xa4,
 0x25, 0x9a, 0x30, 0x44, 0xb1, 0x10, 0x4c, 0x03, 0x02, 0x25, 0x52, 0x80,
 0x08, 0x49, 0x84, 0x20, 0x50, 0x32, 0x30, 0x18, 0xa2, 0x40, 0x11, 0x24,
 0x28, 0x01, 0x84, 0x01, 0x01, 0xa0, 0x41, 0x0a, 0x12, 0x45, 0x00, 0x36,
 0x08, 0x08, 0x26, 0x29, 0x83, 0x82, 0x61, 0xc0, 0x80, 0x10, 0x10,
 0x6d, 0x00, 0x22, 0x48, 0x58, 0x26, 0x0c, 0x2c, 0x10, 0x48, 0x89, 0x24,
 0x20, 0x58, 0x20, 0x45, 0x88, 0x24, 0x00, 0x19, 0x02, 0x25, 0xc0, 0x10,
 0x68, 0x08, 0x14, 0x0a, 0xc0, 0x32, 0x28, 0x80, 0x00, 0x04, 0x4b, 0x26,
 0x00, 0x13, 0x90, 0x60, 0x82, 0x80, 0x25, 0xd0, 0x00, 0x10, 0x32,
 0x0c, 0x43, 0x86, 0x21, 0x11, 0x00, 0x08, 0x43, 0x24, 0x04, 0x48, 0x10,
 0x0c, 0x90, 0x92, 0x00, 0x43, 0x20, 0x2d, 0x00, 0x06, 0x09, 0x88, 0x24,
 0x40, 0xc0, 0x32, 0x09, 0x09, 0x82, 0x00, 0x53, 0x80, 0x08, 0x80, 0x96,
 0x41, 0x81, 0x00, 0x40, 0x48, 0x10, 0x48, 0x08, 0x96, 0x48, 0x56, 0x20,
 0x29, 0xc3, 0x80, 0x20, 0x02, 0x94, 0x60, 0x92, 0x00, 0x20, 0x81, 0x22,
 0x44, 0x10, 0xa0, 0x05, 0x40, 0x90, 0x01, 0x49, 0x20, 0x04, 0xa0, 0x00,
 0x24, 0x89, 0x34, 0x48, 0x13, 0x80, 0x2c, 0xc0, 0x82, 0x29, 0x00, 0x24,
 0x45, 0x08, 0x00, 0x08, 0x98, 0x36, 0x04, 0x52, 0x84, 0x04, 0xd0, 0x04,
 0x00, 0x8a, 0x90, 0x44, 0x82, 0x32, 0x65, 0x18, 0x90, 0x00, 0xa0, 0x02,
```
```c
#if RSA_KEY_SIEVE && SIMULATION && RSA_INSTRUMENT

UINT32  PrimeIndex = 0;
INT32   failedAtIteration[10] = {0};
INT32   PrimeCounts[3] = {0};
INT32   MillerRabinTrials[3] = {0};
INT32   totalFieldsSieved[3] = {0};
INT32   bitsInFieldAfterSieve[3] = {0};
INT32   emptyFieldsSieved[3] = {0};
INT32   noPrimeFields[3] = {0};
INT32   primesChecked[3] = {0};
INT16   lastSievePrime = 0;
#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 TMP 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

```c
#include "Tpm.h"
#include <stdio.h>
#include "Platform_fp.h"
#include "RsaKeyCache_fp.h"
#if CRT_FORMAT_RSA == YES
#define CACHE_FILE_NAME "RsaKeyCacheCrt.data"
#else
#define CACHE_FILE_NAME "RsaKeyCacheNoCrt.data"
#endif

typedef struct _RSA_KEY_CACHE_
{
    TPM2B_PUBLIC_KEY_RSA        publicModulus;
    TMP2B_PRIVATE_KEY_RSA       privateExponent;
} RSA_KEY_CACHE;

Determine the number of RSA key sizes for the cache

TPMI_RSA_KEY_BITS SupportedRsaKeySizes[] = {
#if RSA_1024
    1024,
#endif
#if RSA_2048
    2048,
#endif
#if RSA_3072
    3072,
#endif
#if RSA_4096
    4096,
#endif
};
#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

```c
RSA_KEY_CACHE    s_rsaKeyCache[RSA_KEY_CACHE_ENTRIES];
```

Indicates if the key cache is loaded. It can be loaded and enabled or disabled.

```c
BOOL             s_keyCacheLoaded = 0;
```

Indicates if the key cache is enabled

```c
int             s_rsaKeyCacheEnabled = FALSE;
```

### 10.2.21.2.1 RsaKeyCacheControl()

Used to enable and disable the RSA key cache.

```c
LIB_EXPORT void RsaKeyCacheControl(
    int state
)
{
    s_rsaKeyCacheEnabled = state;
}
```

### 10.2.21.2.2 InitializeKeyCache()

This will initialize the key cache and attempt to write it to a file for later use.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
static BOOL InitializeKeyCache(
    TPMT_PUBLIC   *publicArea,
    TPMT_SENSITIVE *sensitive,
    RAND_STATE    *rand // IN: if not NULL, the deterministic
                    // RNG state
)
{
    int index;
    TPM_KEY_BITS keySave = publicArea->parameters.rsaDetail.keyBits;
    BOOL OK = TRUE;
    // s_rsaKeyCacheEnabled = FALSE;
    for(index = 0; OK && index < RSA_KEY_CACHE_ENTRIES; index++)
    {
        publicArea->parameters.rsaDetail.keyBits
            = SupportedRSAKeySizes[index];
        OK = (CryptRsaGenerateKey(publicArea, sensitive, rand) == TPM_RC_SUCCESS);
        if(OK)
        {
            s_rsaKeyCache[index].publicModulus = publicArea->unique.rsa;
            s_rsaKeyCache[index].privateExponent = sensitive->sensitive.rsa;
        }
    }
    publicArea->parameters.rsaDetail.keyBits = keySave;
    s_keyCacheLoaded = OK;
    #if SIMULATION & USE_RSA_KEY_CACHE & USE_KEY_CACHE_FILE
```
#if defined _MSC_VER
if(fopen_s(&cacheFile, fn, "w+b") != 0)
#else
    cacheFile = fopen(fn, "w+b");
    if(NULL == cacheFile)
#endif
{
    printf("Can't open %s for write.\n", fn);
} else
{
    fseek(cacheFile, 0, SEEK_SET);
    if(fwrite(s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
    != sizeof(s_rsaKeyCache))
    {
        printf("Error writing cache to %s.", fn);
    }
    if(cacheFile)
        fclose(cacheFile);
} #endif
return s_keyCacheLoaded;

10.2.21.2.3 KeyCacheLoaded()

Checks that key cache is loaded.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>cache loaded</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>cache not loaded</td>
</tr>
</tbody>
</table>

static BOOL KeyCacheLoaded(
    TPMT_PUBLIC *publicArea,
    TPMT_SENSITIVE *sensitive,
    RAND_STATE *rand     // IN: if not NULL, the deterministic
                       // RNG state
)
{
#if SIMULATION && USE_RSA_KEY_CACHE && USE_KEY_CACHE_FILE
    if(!s_keyCacheLoaded)
    {
        FILE *cacheFile;
        const char * fn = CACHE_FILE_NAME;
        #if defined _MSC_VER & & 1
            if(fopen_s(&cacheFile, fn, "r+b") == 0)
        #else
            cacheFile = fopen(fn, "r+b");
            if(NULL != cacheFile)
        #endif
        {
            fseek(cacheFile, 0L, SEEK_END);
            if(ftell(cacheFile) == sizeof(s_rsaKeyCache))
            {
                fseek(cacheFile, 0L, SEEK_SET);
            }
        }
    }
}
s_keyCacheLoaded = {
    fread(&s_rsaKeyCache, 1, sizeof(s_rsaKeyCache), cacheFile)
    == sizeof(s_rsaKeyCache));
    fclose(cacheFile);
}
#endif
if(!s_keyCacheLoaded)
    s_rsaKeyCacheEnabled = InitializeKeyCache(publicArea, sensitive, rand);
return s_keyCacheLoaded;
}

10.2.21.2.4 GetCachedRsaKey()

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>key loaded</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>key not loaded</td>
</tr>
</tbody>
</table>

BOOL GetCachedRsaKey()
{
    TPMT_PUBLIC *publicArea,
    TPMT_SENSITIVE *sensitive,
    RAND_STATE *rand     // IN: if not NULL, the deterministic
    // RNG state

    // if(KeyCacheLoaded(publicArea, sensitive, rand))
    {
        int keyBits = publicArea->parameters.rsaDetail.keyBits;
        int index;

        // if((s_rsaKeyCache[index].publicModulus.t.size * 8) == keyBits)
        {
            if(publicArea->unique.rsa = s_rsaKeyCache[index].publicModulus;
                sensitive->sensitive.rsa = s_rsaKeyCache[index].privateExponent;
            return TRUE;
        }
        return FALSE;
    }
    return s_keyCacheLoaded;
}
10.2.22 Ticket.c

10.2.22.1 Introduction

This clause contains the functions used for ticket computations.

10.2.22.2 Includes

```c
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>safe to produce ticket</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>not safe to produce ticket</td>
</tr>
</tbody>
</table>

```c
BOOL TicketIsSafe(             
    TPM2B           *buffer        
) {                           
   TPM_GENERATED   valueToCompare = TPM_GENERATED_VALUE;   
   BYTE            bufferToCompare[sizeof(valueToCompare)];    
   BYTE            *marshalBuffer;   
   // If the buffer size is less than the size of TPM_GENERATED_VALUE, assume   
   // it is not safe to generate a ticket   
   if(buffer->size < sizeof(valueToCompare))   
      return FALSE;   
   marshalBuffer = bufferToCompare;   
   TPM_GENERATED_Marshal(&valueToCompare, &marshalBuffer, NULL);   
   if(MemoryEqual(buffer->buffer, bufferToCompare, sizeof(valueToCompare)))   
      return FALSE;   
   else   
      return TRUE;   
}
```

10.2.22.3.2 TicketComputeVerified()

This function creates a TPMT_TK_VERIFIED ticket.

```c
void TicketComputeVerified(      
    TPMI_RH_HIERARCHY    hierarchy,   // IN: hierarchy constant for ticket   
    TPM2B_DIGEST        *digest,    // IN: digest   
    TPM2B_NAME          *keyName,   // IN: name of key that signed the values   
    TPMT_TK_VERIFIED    *ticket     // OUT: verified ticket   
) {                           
   TPM2B_PROOF         *proof;   
   HMAC_STATE          hmacState;  
```
// Fill in ticket fields
ticket->tag = TPM_ST_VERIFIED;
ticket->hierarchy = hierarchy;
proof = HierarchyGetProof(hierarchy);

// Start HMAC using the proof value of the hierarchy as the HMAC key
ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
&proof->b);

// TPM_ST_VERIFIED
CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
// digest
CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
// key name
CryptDigestUpdate2B(&hmacState.hashState, &keyName->b);
// done
CryptHmacEnd2B(&hmacState, &ticket->digest.b);

return;
}

10.2.22.3.3 TicketComputeAuth()

This function creates a TPMT_TK_AUTH ticket.

void
TicketComputeAuth(

TPM_ST               type,       // IN: the type of ticket.
TPMI_RH_HIERARCHY    hierarchy, // IN: hierarchy constant for ticket
UINT64               timeout,    // IN: timeout
BOOL                 expiresOnReset, // IN: flag to indicate if ticket expires on
//      TPM Reset
TPM2B_DIGEST         *cpHashA,   // IN: input cpHashA
TPM2B_NONCE          *policyRef, // IN: input policyRef
TPM2B_NAME           *entityName, // IN: name of entity
TPMT_TK_AUTH         *ticket    // OUT: Created ticket
)
{
TPM2B_PROOF         *proof;
HMACE STATE          hmacState;

// Get proper proof
proof = HierarchyGetProof(hierarchy);

// Fill in ticket fields
ticket->tag = type;
ticket->hierarchy = hierarchy;

// Start HMAC with hierarchy proof as the HMAC key
ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
&proof->b);

// TPM_ST_AUTH_SECRET or TPM_ST_AUTH_SIGNED,
CryptDigestUpdateInt(&hmacState, sizeof(UINT16), ticket->tag);
// cpHash
CryptDigestUpdate2B(&hmacState.hashState, &cpHashA->b);
// policyRef
CryptDigestUpdate2B(&hmacState.hashState, &policyRef->b);
// keyName
CryptDigestUpdate2B(&hmacState.hashState, &entityName->b);
// timeout
CryptDigestUpdateInt(&hmacState, sizeof(timeout), timeout);

if(timeout != 0)
{
  // epoch
CryptDigestUpdateInt(&hmacState.hashState, sizeof(CLOCK_NONCE),
   g_timeEpoch);
   // reset count
   if (expiresOnReset)
      CryptDigestUpdateInt(&hmacState.hashState, sizeof(gp.totalResetCount),
                           gp.totalResetCount);
   }
   // done
   CryptHmacEnd2B(&hmacState, &ticket->digest.b);
   return;
}

10.2.22.3.4 TicketComputeHashCheck()  
This function creates a TPMT_TK_HASHCHECK ticket.

void TicketComputeHashCheck(
   TPMI_RH_HIERARCHY hierarchy,  // IN: hierarchy constant for ticket
   TPM_ALG_ID hashAlg,  // IN: the hash algorithm for 'digest'
   TPM2B_DIGEST *digest,  // IN: input digest
   TPMT_TK_HASHCHECK *ticket // OUT: Created ticket
) {
   TPM2B_PROOF *proof;
   HMAC_STATE hmacState;
   // Get proper proof
   proof = HierarchyGetProof(hierarchy);
   // Fill in ticket fields
   ticket->tag = TPM_ST_HASHCHECK;
   ticket->hierarchy = hierarchy;
   // Start HMAC using hierarchy proof as HMAC key
   ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
                                            &proof->b);
   // TPM_ST_HASHCHECK
   CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
   // hash algorithm
   CryptDigestUpdateInt(&hmacState, sizeof(hashAlg), hashAlg);
   // digest
   CryptDigestUpdate2B(&hmacState.hashState, &digest->b);
   // done
   CryptHmacEnd2B(&hmacState, &ticket->digest.b);
   return;
}

10.2.22.3.5 TicketComputeCreation()  
This function creates a TPMT_TK_CREATION ticket.

void TicketComputeCreation(
   TPMI_RH_HIERARCHY hierarchy,  // IN: hierarchy for ticket
   TPM2B_NAME *name,  // IN: object name
   TPM2B_DIGEST *creation,  // IN: creation hash
   TPMT_TK_CREATION *ticket // OUT: created ticket
) {
   TPM2B_PROOF *proof;
HMAC_STATE hmacState;

// Get proper proof
proof = HierarchyGetProof(hierarchy);

// Fill in ticket fields
ticket->tag = TPM_ST_CREATION;
ticket->hierarchy = hierarchy;

// Start HMAC using hierarchy proof as HMAC key
ticket->digest.t.size = CryptHmacStart2B(&hmacState, CONTEXT_INTEGRITY_HASH_ALG,
                                    &proof->b);

    // TPM_ST_CREATION
    CryptDigestUpdateInt(&hmacState, sizeof(TPM_ST), ticket->tag);
    // name if provided
    if (name != NULL)
        CryptDigestUpdate2B(&hmacState.hashState, &name->b);
    // creation hash
    CryptDigestUpdate2B(&hmacState.hashState, &creation->b);
    // Done
    CryptHmacEnd2B(&hmacState, &ticket->digest.b);

return;
10.2.23 TpmAsn1.c

10.2.23.1 Includes

```c
#include "Tpm.h"
#define _OIDS_
#include "OIDS.h"
#include "TpmASN1.h"
#include "TpmASN1_fp.h"
```

10.2.23.2 Unmarshaling Functions

10.2.23.2.1 ASN1UnmarshalContextInitialize()

Function does standard initialization of a context.

```c
typedef struct

  // Return Value  Meaning
  TRUE(1) success
  FALSE(0) failure

BOOL
ASN1UnmarshalContextInitialize(
  ASN1UnmarshalContext *ctx,
  INT16 size,
  BYTE *buffer
)
{
  VERIFY(buffer != NULL);
  VERIFY(size > 0);
  ctx->buffer = buffer;
  ctx->size = size;
  ctx->offset = 0;
  ctx->tag = 0xFF;
  return TRUE;
}
Error:
return FALSE;
}
```

10.2.23.2.2 ASN1DecodeLength()

This function extracts the length of an element from buffer starting at offset.

```c
typedef int

  // Return Value  Meaning
  >=0 the extracted length
  <0 an error

INT16
ASN1DecodeLength(
  ASN1UnmarshalContext *ctx
)
{
  BYTE first; // Next octet in buffer
  INT16 value;
  // VERIFY(ctx->offset < ctx->size);
  first = NEXT_OCTET(ctx);
  ```
// If the number of octets of the entity is larger than 127, then the first octet
// is the number of octets in the length specifier.
if (first >= 0x80)
{
    // Make sure that this length field is contained with the structure being
    // parsed
    CHECK_SIZE(ctx, (first & 0x7F));
    if (first == 0x82)
    {
        // Two octets of size
        // get the next value
        value = (INT16)NEXT_OCTET(ctx);
        // Make sure that the result will fit in an INT16
        VERIFY(value < 0x0080);
        // Shift up and add next octet
        value = (value << 8) + NEXT_OCTET(ctx);
    }
    else if (first == 0x81)
    {
        value = NEXT_OCTET(ctx);
        // Sizes larger than will fit in a INT16 are an error
        else
            goto Error;
    }
    else
    {
        value = first;
        // Make sure that the size defined something within the current context
        CHECK_SIZE(ctx, value);
        return value;
    }
Error:
    ctx->size = -1; // Makes everything fail from now on.
    return -1;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;=0</td>
<td>the number of octets in type</td>
</tr>
<tr>
<td>&lt;0</td>
<td>an error</td>
</tr>
</tbody>
</table>

INT16
ASN1NextTag(ASN1UnmarshalContext *ctx)
{
    // A tag to get?
    VERIFY(ctx->offset < ctx->size);
    // Get it
    ctx->tag = NEXT_OCTET(ctx);
    // Make sure that it is not an extended tag
    VERIFY((ctx->tag & 0x1F) != 0x1F);
    // Get the length field and return that
    return ASN1DecodeLength(ctx);
Error:
    // Attempt to read beyond the end of the context or an illegal tag
    ctx->size = -1; // Persistent failure
    ctx->tag = 0xFF;
    return -1;
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
BOOL
ASN1GetBitStringValue(
    ASN1UnmarshalContext *ctx,
    UINT32               *val
)
{
    int        shift;
    INT16      length;
    UINT32     value = 0;

    //
    VERIFY((length = ASN1NextTag(ctx)) >= 1);
    VERIFY(ctx->tag == ASN1_BITSTRING);
    // Get the shift value for the bit field (how many bits to loop off of the end)
    shift = NEXT_OCTET(ctx);
    length--;
    // the shift count has to make sense
    VERIFY((shift < 8) && ((length > 0) || (shift == 0)));
    // if there are any bytes left
    for(; length > 0; length--)
    {
        if(length > 1)
        {
            // for all but the last octet, just shift and add the new octet
            VERIFY((value & 0xFF000000) == 0); // can’t loose significant bits
            value = (value << 8) + NEXT_OCTET(ctx);
        }
        else
        {
            // for the last octet, just shift the accumulated value enough to
            // accept the significant bits in the last octet and shift the last
            // octet down
            VERIFY(((value & 0xFF000000) << (8 - shift))) == 0);
            value = (value << (8 - shift)) + (NEXT_OCTET(ctx) >> shift);
        }
    }

    *val = value;
    return TRUE;
}
```

Error:
```c
ctx->size = -1;
return FALSE;
```

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. On 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.

```c
void
ASN1InitializeMarshalContext(ASN1MarshalContext      *ctx,
INT16       length,
BYTE        *buffer)
{
  ctx->buffer = buffer;
  if(buffer)
    ctx->offset = length;
  else
    ctx->offset = INT16_MAX;
  ctx->end = ctx->offset;
  ctx->depth = -1;
}
```

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.

```c
void
ASN1StartMarshalContext(ASN1MarshalContext      *ctx)
{
  pAssert((ctx->depth + 1) < MAX_DEPTH);
  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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>the size of the encapsulated structure that was just ended</td>
</tr>
<tr>
<td>&lt;= 0</td>
<td>an error</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of octets in the encapsulation</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

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
Part 4: Supporting Routines

Trusted Platform Module Library

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>count bytes</td>
</tr>
<tr>
<td>== 0</td>
<td>failure unless count was zero</td>
</tr>
</tbody>
</table>

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 }
215 Error:
216     ctx->offset = -1;
217     return 0;
218 }

10.2.23.3.8 ASN1PushNull()

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>count bytes</td>
</tr>
<tr>
<td>== 0</td>
<td>failure unless count was zero</td>
</tr>
</tbody>
</table>

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;
10.2.23.3.9 **ASN1PushLength()**

Push a length value. This will only handle length values that fit in an INT16.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of bytes added</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
10.2.23.3.9

ASN1PushLength(
ASN1MarshalContext *ctx,
INT16 len
)
{
    UINT16 start = ctx->offset;
    VERIFY(len >= 0);
    if (len <= 127)
        ASN1PushByte(ctx, (BYTE)len);
    else
        { 
            ASN1PushByte(ctx, (BYTE)(len & 0xFF));
            len >>= 8;
            if (len == 0)
                ASN1PushByte(ctx, 0x81);
            else
                { 
                    ASN1PushByte(ctx, (BYTE)len);
                    ASN1PushByte(ctx, 0x82);
                }
            goto Exit;
        }
    Error:
    ctx->offset = -1;
    Exit:
    return (ctx->offset > 0) ? start - ctx->offset : 0;
}
```

10.2.23.3.10 **ASN1PushTagAndLength()**

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of bytes added</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
10.2.23.3.10

ASN1PushTagAndLength(
ASN1MarshalContext *ctx,
BYTE tag,
INT16 length
)
{
    INT16 bytes;
    bytes = ASN1PushLength(ctx, length);
    bytes += (INT16)ASN1PushByte(ctx, tag);
    return (ctx->offset < 0) ? 0 : bytes;
}
```
10.2.23.3.11 ASN1PushTaggedOctetString()

This function will push a random octet string.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of bytes added</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
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     // PushTagAndLenght 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>count bytes</td>
</tr>
<tr>
<td>== 0</td>
<td>failure unless count was zero</td>
</tr>
</tbody>
</table>

```c
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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>the number of bytes marshaled for the integer</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
291 INT16
292 ASN1PushInteger(
293     ASN1MarshalContext *ctx,       // IN/OUT: buffer context
294     INT16                iLen,      // IN: octets of the integer
```
BYTE *integer  // IN: big-endian integer
}

while(*integer == 0) && (--iLen > 0))
    integer++;

while(*integer & 0x80)
    iLen += (INT16)ASN1PushByte(ctx, 0);

// PushTagAndLength just tells how many octets it added so the total size of this
// element is the sum of those octets and the adjusted input size.
iLen += ASN1PushTagAndLength(ctx, ASN1_INTEGER, iLen);

return iLen;
}

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>the number of bytes marshaled for the integer</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

INT16 ASN1PushOID(ASN1MarshalContext *ctx, const BYTE *OID)
{
    if(*OID == ASN1_OBJECT_IDENTIFIER) && ((OID[1] & 0x80) == 0))
    {
        return ASN1PushBytes(ctx, OID[1] + 2, OID);
    }
    ctx->offset = -1;
    return 0;
}
10.2.24  X509_ECC.c

10.2.24.1  Includes

```c
#include "Tpm.h"
#include "X509.h"
#include "OIDS.h"
#include "TpmASN1_fp.h"
#include "X509_spt_fp.h"
#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...

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of bytes added</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
INT16 X509PushPoint(
    ASN1MarshalContext  *ctx,
    TPMS_ECC_POINT      *p
) {
    // Push a bit string containing the public key. For now, push the x, and y
    // coordinates of the public point, bottom up
    ASN1StartMarshalContext(ctx); // BIT STRING
    {
        ASN1PushBytes(ctx, p->y.t.size, p->y.t.buffer);
        ASN1PushBytes(ctx, p->x.t.size, p->x.t.buffer);
        ASN1PushByte(ctx, 0x04);
    }
    return ASN1EndEncapsulation(ctx, ASN1_BITSTRING); // Ends BIT STRING
}
```

10.2.24.2.2  X509AddSigningAlgorithmECC()

This creates the signing algorithm data.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of bytes added</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
INT16 X509AddSigningAlgorithmECC(
    OBJECT      *signKey,
    TPMT_SIG_SCHEME  *scheme,
    ASN1MarshalContext  *ctx
) {
    PHASH_DEF hashDef = CryptGetHashDef(scheme->details.any.hashAlg);
    // NOT_REFERENCED(signKey);
```
```c
// If the desired hashAlg definition wasn't found...
if(hashDef->hashAlg != scheme->details.any.hashAlg)
    return 0;

switch(scheme->scheme)
{
    case ALG_ECDSA_VALUE:
        // Make sure that we have an OID for this hash and ECC
        if((hashDef->ECDSA) != ASN1_OBJECT_IDENTIFIER)
            break;
        // if this is just an implementation check, indicate that this
        // combination is supported
        if(!ctx)
            return 1;
        ASN1StartMarshalContext(ctx);
        ASN1PushOID(ctx, hashDef->ECDSA);
        return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
    default:
        break;
}
```

### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of bytes added</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
INT16 X509AddPublicECC(
    OBJECT *object,
    ASN1MarshalContext *ctx
)
{
    const BYTE *curveOid = CryptEccGetOID(object->publicArea.parameters.eccDetail.curveID);
    if((curveOid == NULL) || (*curveOid != ASN1_OBJECT_IDENTIFIER))
        return 0;
    // SEQUENCE (2 elem) 1st
    //   SEQUENCE (2 elem) 2nd
    //     OBJECT IDENTIFIER 1.2.840.10045.2.1 ecPublicKey (ANSI X9.62 public key type)
    //     OBJECT IDENTIFIER 1.2.840.10045.3.1.7 prime256v1 (ANSI X9.62 named curve)
    //     BIT STRING (520 bit) 000001001010000111010101010111001001101101000100000010...
    // If this is a check to see if the key can be encoded, it can.
    // Need to mark the end sequence
    if(ctx == NULL)
        return 1;
    ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st
    { X509PushPoint(ctx, &object->publicArea.unique.ecc); // BIT STRING
        ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 2nd
        { ASN1PushOID(ctx, curveOid); // curve dependent
            ASN1PushOID(ctx, OID_ECC_PUBLIC); // (1.2.840.10045.2.1)
        }
    }
```
ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 2nd
}

return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // Ends SEQUENCE 1st
10.2.25 X509_RSA.c

10.2.25.1 Includes

```c
#include "Tpm.h"
#include "X509.h"
#include "TpmASN1_fp.h"
#include "X509_spt_fp.h"
#include "CryptHash_fp.h"
#include "CryptRsa_fp.h"
```

10.2.25.2 Functions

```c
#if ALG_RSA

10.2.25.2.1 X509AddSigningAlgorithmRSA()

This creates the signing algorithm data.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of bytes added</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```

```c
INT16
X509AddSigningAlgorithmRSA(
    OBJECT              *signKey,
    TPM_SIG_SCHEME     *scheme,
    ASN1MarshalContext *ctx
)
{
    TPM_ALG_ID           hashAlg = scheme->details.any.hashAlg;
    PHASH_DEF            hashDef = CryptGetHashDef(hashAlg);

    // NOT_REFERENCED(signKey);
    // return failure if hash isn't implemented
    if(hashDef->hashAlg != hashAlg)
        return 0;
    switch(scheme->scheme)
    {
        case ALG_RSASSA_VALUE:
        {
            // if the hash is implemented but there is no PKCS1 OID defined
            // then this is not a valid signing combination.
            if(hashDef->PKCS1[0] != ASN1_OBJECT_IDENTIFIER)
                break;
            if(ctx == NULL)
                return 1;
            ASN1StartMarshalContext(ctx);
            ASN1PushOID(ctx, hashDef->PKCS1);
            return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
        }
        case ALG_RSAPSS_VALUE:
        {
            // leave if this is just an implementation check
            if(ctx == NULL)
                return 1;
            // In the case of SHA1, everything is default and RFC4055 says that
            // implementations that do signature generation MUST omit the parameter
            // when defaults are used. }-
            if(hashDef->hashAlg == ALG_SHA1_VALUE)
                {  
```
return X509PushAlgorithmIdentifierSequence(ctx, OID_RSAPSS);
}  
else  
{  
// Going to build something that looks like:  
// SEQUENCE (2 elem)  
//   OBJECT IDENTIFIER 1.2.840.113549.1.1.10 rsaPSS (PKCS #1)  
//   SEQUENCE (3 elem)  
//     [0] (1 elem)  
//      SEQUENCE (2 elem)  
//       OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256  
//       NULL  
//     [1] (1 elem)  
//      SEQUENCE (2 elem)  
//       OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF  
//       SEQUENCE (2 elem)  
//        OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256  
//        NULL  
//     [2] (1 elem) salt length  
//      INTEGER 32  
// The indentation is just to keep track of where we are in the  
// structure  
ASN1StartMarshalContext(ctx); // SEQUENCE (2 elements)  
{  
ASN1StartMarshalContext(ctx); // SEQUENCE (3 elements)  
{  
// [2] (1 elem) salt length  
//   INTEGER 32  
ASN1StartMarshalContext(ctx);  
{  
INT16 saltSize =  
CryptRsaPssSaltSize((INT16)hashDef->digestSize,  
(INT16)signKey->publicArea.unique.rsa.t.size);  
ASN1PushUINT(ctx, saltSize);  
}  
ASN1EndEncapsulation(ctx, ASN1_APPLICATION_SPECIFIC + 2);  
}  
// Add the mask generation algorithm  
// [1] (1 elem)  
//   SEQUENCE (2 elem) 1st  
//    OBJECT IDENTIFIER 1.2.840.113549.1.1.8 pkcs1-MGF  
//    SEQUENCE (2 elem) 2nd  
//     OBJECT IDENTIFIER 2.16.840.1.101.3.4.2.1 sha-256  
//     NULL  
ASN1StartMarshalContext(ctx); // mask context [1] (1 elem)  
{  
ASN1StartMarshalContext(ctx); // SEQUENCE (2 elem) 1st  
// Handle the 2nd Sequence (sequence (object, null))  
{  
X509PushAlgorithmIdentifierSequence(ctx,  
hashDef->OID);  
// add the pkcs1-MGF OID  
ASN1PushOID(ctx, OID_MGF1);  
}  
// End outer sequence  
ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);  
}  
// End the [1]  
ASN1EndEncapsulation(ctx, ASN1_APPLICATION_SPECIFIC + 1);  
}  
// Add the hash algorithm  
// [0] (1 elem)  
//   SEQUENCE (2 elem) (done by  
//    X509PushAlgorithmIdentifierSequence)  
//    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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of bytes added</td>
</tr>
<tr>
<td>== 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

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 // Get public exponent in big-endian byte order.
159 if(exp == 0)
160     exp = RSA_DEFAULT_PUBLIC_EXPONENT;
// Push a 4 byte integer. This might get reduced if there are leading zeros or
// extended if the high order byte is negative.
ASN1PushUINT(ctx, exp);
// Push the public key as an integer
ASN1PushInteger(ctx, object->publicArea.unique.rsa.t.size,
                 object->publicArea.unique.rsa.t.buffer);
// Embed this in a SEQUENCE tag and length in for the key, exponent sequence
ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE); // SEQUENCE (3rd)

// Embed this in a BIT STRING
ASN1EndEncapsulation(ctx, ASN1_BITSTRING);

// Now add the formatted SEQUENCE for the RSA public key OID. This is a
// fully constructed value so it doesn't need to have a context started
X509PushAlgorithmIdentifierSequence(ctx, OID_PKCS1_PUB);
return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);

#endif // ALG_RSA
10.2.26 X509_spt.c

10.2.26.1 Includes

```c
#include "Tpm.h"
#include "TpmASN1.h"
#include "TpmASN1_fp.h"
#define _X509_SPT_
#include "X509.h"
#include "X509_spt_fp.h"
#if ALG_RSA
#include "X509_RSA_fp.h"
#endif // ALG_RSA
#if ALG_ECC
#include "X509_ECC_fp.h"
#endif // ALG_ECC
#if ALG_SM2
#ifdef "X509_SM2_fp.h"
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure (could be catastrophic)</td>
</tr>
</tbody>
</table>

```c
BOOL X509FindExtensionByOID(
    ASN1UnmarshalContext *ctxIn,    // IN: the context to search
    const BYTE *OID,               // IN: oid to search for
    ASN1UnmarshalContext *ctx,     // OUT: the extension context
    INT16 length;                  // length
)
{
    INT16 length;
    // Make the search non-destructive of the input if ctx provided. Otherwise, use
    // the provided context.
    if (ctx == NULL)
        ctx = ctxIn;
    else if(ctx != ctxIn)
        *ctx = *ctxIn;
    for(;ctx->size > ctx->offset; ctx->offset += length)
    {
        VERIFY((length = ASN1NextTag(ctx)) >= 0);
        // If this is not a constructed sequence, then it doesn't belong
        // in the extensions.
        VERIFY(ctx->tag == ASN1_CONSTRUCTED_SEQUENCE);
        // Make sure that this entry could hold the OID
        if (length >= OID_SIZE(OID))
            {
                // See if this is a match for the provided object identifier.
                if (MemoryEqual(OID, &(ctx->buffer[ctx->offset]), OID_SIZE(OID)))
```
Part 4: Supporting Routines

Trusted Platform Module Library

10.2.26.2 X509GetExtensionBits()

This function will extract a bit field from an extension. If the extension doesn't contain a bit string, it will fail.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
UINT32
X509GetExtensionBits(
    ASN1UnmarshalContext *ctx,
    UINT32               *value
)
```

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.

<table>
<thead>
<tr>
<th>Error Returns</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPM_RCS_ATTRIBUTES</td>
<td>the attributes of object are not consistent with the extension setting</td>
</tr>
<tr>
<td>TPM_RC_VALUE</td>
<td>problem parsing the extensions</td>
</tr>
</tbody>
</table>

```c
TPM_RC
X509ProcessExtensions()
```
OBJECT   *object, // IN: The object with the attributes to
// check
stringRef *extension // IN: The start and length of the extensions

{  
  ASN1UnmarshalContext   ctx;
  ASN1UnmarshalContext  extensionCtx;
  INT16                   length;
  UINT32                   value;
  
  // Get the extension for the TPMA_OBJ
  // Get the keyUsage extension. This one is required

  if(!ASN1UnmarshalContextInitialize(&ctx, extension->len, extension->buf)
    || ((length = ASN1NextTag(&ctx)) != X509_EXTENSIONS))
    return TPM_RCS_VALUE;
  
  if( ((length = ASN1NextTag(&ctx)) < 0)
    || (ctx.tag != (ASN1_CONSTRUCTED_SEQUENCE)))
    return TPM_RCS_VALUE;

  // Get the extension for the TPMA_OBJ if there is one
  if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_TCG_TPMA_OBJECT) &&
    X509GetExtensionBits(&extensionCtx, &value))
  {
    // If an keyAttributes extension was found, it must be exactly the same as the
    // attributes of the object.
    // This cast will work because we know that a TPMA_OBJECT is in a UINT32.
    // Set RUNTIME_SIZE_CHECKS to YES to force a check to verify this assumption
    // during debug. Doing this is lot easier than having to revisit the code
    // any time a new attribute is added.
    // NOTE: MemoryEqual() is used to avoid type-punned pointer warning/error.
    if(!MemoryEqual(&value, &object->publicArea.objectAttributes, sizeof(value)))
      return TPM_RCS_ATTRIBUTES;
  }

  // Make sure the failure to find the value wasn't because of a fatal error
  else if(extensionCtx.size < 0)
    return TPM_RCS_VALUE;

  // Get the keyUsage extension. This one is required
  if(X509FindExtensionByOID(&ctx, &extensionCtx, OID_KEY_USAGE_EXTENSION) &&
    X509GetExtensionBits(&extensionCtx, &value))
  {
    x509KeyUsageUnion   keyUsage;
    TPMA_OBJECT         attributes = object->publicArea.objectAttributes;
    // keyUsage.integer = value;
    // For KeyUsage:
    // the 'sign' attribute is SET if Key Usage includes signing
    if(((keyUsageSign.integer & keyUsage.integer) != 0)
      && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, sign)
      && !IS_ATTRIBUTE(attributes, TPMA_OBJECT, fixedTPM))
      // and the 'decrypt' attribute is Set if Key Usage includes decryption
      return TPM_RCS_ATTRIBUTES;
  }

  // The KeyUsage extension is required
  return TPM_RCS_VALUE;
}
10.2.26.3 Marshaling Functions

10.2.26.3.1 X509AddSigningAlgorithm()

This creates the signing algorithm data.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of octets added</td>
</tr>
<tr>
<td>&lt;= 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 0</td>
<td>number of octets added</td>
</tr>
<tr>
<td>&lt;= 0</td>
<td>failure</td>
</tr>
</tbody>
</table>

```c
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 X509AddPublicRSA(object, ctx);
184 #endif
185 #if ALG_ECC
```
case ALG_ECC_VALUE:
    return X509AddPublicECC(object, ctx);
#endif
#if ALG_SM2
    case ALG_SM2_VALUE:
        break;
    #endif
    default:
        break;
    }
    return FALSE;
}

10.2.26.3.3 X509PushAlgorithmIdentifierSequence()

Return Value | Meaning |
-------------|---------|
> 0          | number of bytes added |
== 0         | failure |

INT16
X509PushAlgorithmIdentifierSequence(
    ASN1MarshalContext *ctx,
    const BYTE *OID
)
{
    ASN1StartMarshalContext(ctx);  // hash algorithm
    ASN1PushNull(ctx);
    ASN1PushOID(ctx, OID);
    return ASN1EndEncapsulation(ctx, ASN1_CONSTRUCTED_SEQUENCE);
}
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
typedef struct {
9   TPML_AC_CAPABILITIES acData0001 = {1,
10       {{TPM_AT_PV1, 0x01234567}}};
11   acCapabilities ac[1] = {0x0001, &acData0001};
12 #define NUM_AC  (sizeof(ac) / sizeof(acCapabilities))
13 #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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOOL</td>
<td></td>
</tr>
</tbody>
</table>

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

```c
TPMI_YES_NO

AcCapabilitiesGet(

    TPMI_RH_AC component,        // IN: the component
    TPM_AT      type,             // IN: start capability type
    TPMI_AC_CAPABILITIES *capabilityList // OUT: list of handle
)
{
    TPMI_YES_NO more = NO;
    UINT32           i;
    TPMI_AC_CAPABILITIES *capabilities = AcToCapabilities(component);
    pAssert(HandleGetType(component) == TPM_HT_AC);
    // Initialize output handle list
    capabilityList->count = 0;
    if(capabilities != NULL)
    {
        // Find the first capability less than or equal to type
        for(i = 0; i < capabilities->count; i++)
        {
            if(capabilities->acCapabilities[i].tag >= type)
            {
                // copy the capabilities until we run out or fill the list
                for(; (capabilityList->count < MAX_AC_CAPABILITIES) && (i < capabilities->count); i++)
                {
                    capabilityList->acCapabilities[capabilityList->count] = capabilities->acCapabilities[i];
                    capabilityList->count++;
                }
            more = i < capabilities->count;
        }
    }
    return more;
}
```

10.2.27.1.4 AcSendObject()

Stub to handle sending of an AC object

```c
TPM_RC

AcSendObject(

    TPM_HANDLE acHandle,        // IN: Handle of AC receiving object
    OBJECT     *object,         // IN: object structure to send
    TPMS_AC_OUTPUT *acDataOut   // OUT: results of operation
)
{
    NOT_REFERENCED(object);
    NOT_REFERENCED(acHandle);
    acDataOut->tag = TPM_AT_ERROR; // indicate that the response contains an
    // error code
    acDataOut->data = TPM_AE_NONE; // but there is no error.
```
return TPM_RC_SUCCESS;
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

```cpp
#ifndef _TPM_PROFILE_H_
#define _TPM_PROFILE_H_

Table 2:4 - Defines for Logic Values

#define TRUE                1
#define FALSE               0
#define YES                 1
#define NO
#define SET                 1
#define CLEAR               0

Table 0:1 - Defines for Processor Values

#define BIG_ENDIAN_TPM
#define LITTLE_ENDIAN_TPM
#define MOST_SIGNIFICANT_BIT_0
#define LEAST_SIGNIFICANT_BIT_0
#define AUTO_ALIGN

Table 0:4 - Defines for Implemented Curves
```
#ifndef ECC_NIST_P192
#define ECC_NIST_P192 NO
#endif

#ifndef ECC_NIST_P224
#define ECC_NIST_P224 NO
#endif

#ifndef ECC_NIST_P256
#define ECC_NIST_P256 YES
#endif

#ifndef ECC_NIST_P384
#define ECC_NIST_P384 YES
#endif

#ifndef ECC_BN_P256
#define ECC_BN_P256 YES
#endif

#ifndef ECC_BN_P638
#define ECC_BN_P638 NO
#endif

#ifndef ECC_SM2_P256
#define ECC_SM2_P256 NO
#endif

#ifndef FIELD_UPGRADE_IMPLEMENTED
#define FIELD_UPGRADE_IMPLEMENTED NO
#endif

#ifndef RADIX_BITS
#define RADIX_BITS 32
#endif

#ifndef HASH_ALIGNMENT
#define HASH_ALIGNMENT 4
#endif

#ifndef SYMMETRIC_ALIGNMENT
#define SYMMETRIC_ALIGNMENT 4
#endif

#ifndef HASH_LIB
#define HASH_LIB Ossl
#endif

#ifndef SYM_LIB
#define SYM_LIB Ossl
#endif

#ifndef MATH_LIB
#define MATH_LIB Ossl
#endif

#ifndef BSIZE
#define BSIZE UINT16
#endif

#ifndef IMPLEMENTATION_PCR
#define IMPLEMENTATION_PCR 24
#endif

#ifndef PCR_SELECT_MAX
#define PCR_SELECT_MAX ((IMPLEMENTATION_PCR+7)/8)
#endif

#ifndef PLATFORM_PCR
#define PLATFORM_PCR 24
#endif

#ifndef PCR_SELECT_MIN
#define PCR_SELECT_MIN ((PLATFORM_PCR+7)/8)
#endif

#ifndef DRTM_PCR
#define DRTM_PCR 17
#endif

Table 0.7 - Defines for Implementation Values

#ifndef FIELD_UPGRADE_IMPLEMENTED
#define FIELD_UPGRADE_IMPLEMENTED NO
#endif

#ifndef RADIX_BITS
#define RADIX_BITS 32
#endif

#ifndef HASH_ALIGNMENT
#define HASH_ALIGNMENT 4
#endif

#ifndef SYMMETRIC_ALIGNMENT
#define SYMMETRIC_ALIGNMENT 4
#endif

#ifndef HASH_LIB
#define HASH_LIB Ossl
#endif

#ifndef SYM_LIB
#define SYM_LIB Ossl
#endif

#ifndef MATH_LIB
#define MATH_LIB Ossl
#endif

#ifndef BSIZE
#define BSIZE UINT16
#endif

#ifndef IMPLEMENTATION_PCR
#define IMPLEMENTATION_PCR 24
#endif

#ifndef PCR_SELECT_MAX
#define PCR_SELECT_MAX ((IMPLEMENTATION_PCR+7)/8)
#endif

#ifndef PLATFORM_PCR
#define PLATFORM_PCR 24
#endif

#ifndef PCR_SELECT_MIN
#define PCR_SELECT_MIN ((PLATFORM_PCR+7)/8)
#endif

#ifndef DRTM_PCR
#define DRTM_PCR 17
#endif
#endif

#ifndef HCRTM_PCR
#define HCRTM_PCR                       0
#endif

#ifndef NUM_LOCALITIES
#define NUM_LOCALITIES                  5
#endif

#ifndef MAX_HANDLE_NUM
#define MAX_HANDLE_NUM                  3
#endif

#ifndef MAX_ACTIVE_SESSIONS
#define MAX_ACTIVE_SESSIONS             64
#endif

#ifndef CONTEXT_SLOT
#define CONTEXT_SLOT                    UINT16
#endif

#ifndef CONTEXT_COUNTER
#define CONTEXT_COUNTER                 UINT64
#endif

#ifndef MAX_LOADED_SESSIONS
#define MAX_LOADED_SESSIONS             3
#endif

#ifndef MAX_SESSION_NUM
#define MAX_SESSION_NUM                 3
#endif

#ifndef MAX_LOADED_OBJECTS
#define MAX_LOADED_OBJECTS              3
#endif

#ifndef MIN_EVICT_OBJECTS
#define MIN_EVICT_OBJECTS               2
#endif

#ifndef NUM_POLICY_PCR_GROUP
#define NUM_POLICY_PCR_GROUP            1
#endif

#ifndef NUM_AUTHVALUE_PCR_GROUP
#define NUM_AUTHVALUE_PCR_GROUP         1
#endif

#ifndef MAX_CONTEXT_SIZE
#define MAX_CONTEXT_SIZE                2474
#endif

#ifndef MAX_DIGEST_BUFFER
#define MAX_DIGEST_BUFFER               1024
#endif

#ifndef MAX_NV_INDEX_SIZE
#define MAX_NV_INDEX_SIZE               2048
#endif

#ifndef MAX_NV_BUFFER_SIZE
#define MAX_NV_BUFFER_SIZE              1024
#endif

#ifndef MAX_CAP_BUFFER
#define MAX_CAP_BUFFER                  1024
#endif

#ifndef NV_MEMORY_SIZE
#define NV_MEMORY_SIZE                  16384
#endif

#ifndef MIN_COUNTER_INDICES
#define MIN_COUNTER_INDICES             8
#endif

#ifndef MAX_STATIC_PCR
#define NUM_STATIC_PCR                  16
#endif

#ifndef MAX_ALG_LIST_SIZE
#define MAX_ALG_LIST_SIZE               64
#endif

#ifndef PRIMARY_SEED_SIZE
#define PRIMARY_SEED_SIZE               32
#endif
Table 0:2 - Defines for Implemented Algorithms

216 #ifndef ALG_AES
217 #define ALG_AES ALG_YES
218 #endif
219 #ifndef ALG_CAMELLIA
#define ALG_CAMELLIA ALG_NO /* Not specified by vendor */
#endif

#ifndef ALG_CBC
#define ALG_CBC ALG_YES
#endif

#ifndef ALG_CFB
#define ALG_CFB ALG_YES
#endif

#ifndef ALG_CMAC
#define ALG_CMAC ALG_YES
#endif

#ifndef ALG_CTR
#define ALG_CTR ALG_YES
#endif

#ifndef ALG_ECB
#define ALG_ECB ALG_YES
#endif

#ifndef ALG_ECC
#define ALG_ECC ALG_YES
#endif

#ifndef ALG_ECDAA
#define ALG_ECDAA (ALG_YES && ALG_ECC)
#endif

#ifndef ALG_ECDH
#define ALG_ECDH (ALG_YES && ALG_ECC)
#endif

#ifndef ALG_ECDSA
#define ALG_ECDSA (ALG_YES && ALG_ECC)
#endif

#ifndef ALG_ECMQV
#define ALG_ECMQV (ALG_NO && ALG_ECC)
#endif

#ifndef ALG_ECSCHNORR
#define ALG_ECSCHNORR (ALG_YES && ALG_ECC)
#endif

#ifndef ALG_HMAC
#define ALG_HMAC ALG_YES
#endif

#ifndef ALG_KDF1_SP800_108
#define ALG_KDF1_SP800_108 ALG_YES
#endif

#ifndef ALG_KDF1_SP800_56A
#define ALG_KDF1_SP800_56A (ALG_YES && ALG_ECC)
#endif

#ifndef ALG_KDF2
#define ALG_KDF2 ALG_NO
#endif

#ifndef ALG_KEYEDHASH
#define ALG_KEYEDHASH ALG_YES
#endif

#ifndef ALG_MGF1
#define ALG_MGF1 ALG_YES
#endif

#ifndef ALG_OAEP
#define ALG_OAEP (ALG_YES && ALG_RSA)
#endif

#ifndef ALG_OAEP
#define ALG_OAEP (ALG_YES && ALG_RSA)
#endif

#ifndef ALG_RSA
#define ALG_RSA ALG_YES
#endif

#ifndef ALG_RSAES
#define ALG_RSAES (ALG_YES && ALG_RSA)
#endif

#ifndef ALG_RSAES
#define ALG_RSAES (ALG_YES && ALG_RSA)
#endif

#ifndef ALG_RSAPSS
#define ALG_RSAPSS (ALG_YES && ALG_RSA)
#endif

#ifdef ALG_RSASSA
#endif

#define ALG_SHA ALG_NO
/* Not specified by vendor */
#endif

#define ALG_SHA1 ALG_YES
#endif

#define ALG_SHA256 ALG_YES
#endif

#define ALG_SHA384 ALG_YES
#endif

#define ALG_SHA3_256 ALG_NO
/* Not specified by vendor */
#endif

#define ALG_SHA3_384 ALG_NO
/* Not specified by vendor */
#endif

#define ALG_SHA3_512 ALG_NO
/* Not specified by vendor */
#endif

#define ALG_SHA512 ALG_NO
#endif

#define ALG_SM2 (ALG_NO && ALG_ECC)
#endif

#define ALG_SM3_256 ALG_NO
#endif

#define ALG_SM4 ALG_NO
#endif

#define ALG_SYMCIPHER ALG_YES
#endif

#define ALG_TDES ALG_NO
#endif

#define ALG_XOR ALG_YES
#endif

Table 1:00 - Defines for RSA Asymmetric Cipher Algorithm Constants

#ifndef RSA_1024
#endif

#define RSA_1024 (ALG_RSA & YES)
#endif

#define RSA_2048 (ALG_RSA & YES)
#endif

#define RSA_3072 (ALG_RSA & NO)
#endif

#define RSA_4096 (ALG_RSA & NO)
#endif

Table 1:17 - Defines for AES Symmetric Cipher Algorithm Constants
```c
#ifndef AES_128
#define AES_128 (ALG_AES & YES)
#endif

#ifndef AES_192
#define AES_192 (ALG_AES & NO)
#endif

#ifndef AES_256
#define AES_256 (ALG_AES & YES)
#endif

Table 1:18 - Defines for SM4 Symmetric Cipher Algorithm Constants

#ifndef SM4_128
#define SM4_128 (ALG_SM4 & YES)
#endif

#ifndef CAMELLIA_128
#define CAMELLIA_128 (ALG_CAMELLIA & YES)
#endif

#ifndef CAMELLIA_192
#define CAMELLIA_192 (ALG_CAMELLIA & NO)
#endif

#ifndef CAMELLIA_256
#define CAMELLIA_256 (ALG_CAMELLIA & NO)
#endif

Table 1:19 - Defines for CAMELLIA Symmetric Cipher Algorithm Constants

#ifndef TDES_128
#define TDES_128 (ALG_TDES & YES)
#endif

#ifndef TDES_192
#define TDES_192 (ALG_TDES & YES)
#endif

Table 0:5 - Defines for Implemented Commands

#ifndef CC_AC_GetCapability
#define CC_AC_GetCapability CC_YES
#endif

#ifndef CC_AC_Send
#define CC_AC_Send CC_YES
#endif

#ifndef CC_ActivateCredential
#define CC_ActivateCredential CC_YES
#endif

#ifndef CC_Certify
#define CC_Certify CC_YES
#endif

#ifndef CC_CertifyCreation
#define CC_CertifyCreation CC_YES
#endif

#ifndef CC_ChangeEPS
#define CC_ChangeEPS CC_YES
#endif

#ifndef CC_ChangePPS
#define CC_ChangePPS CC_YES
#endif

#ifndef CC_Clear
#define CC_Clear
#endif
```

Table 1:17 - Defines for TDES Symmetric Cipher Algorithm Constants
#define CC_Clear
endif
 ifndef CC_ClearControl
#define CC_ClearControl
endif
 ifndef CC_ClockRateAdjust
#define CC_ClockRateAdjust
endif
 ifndef CC_ClockSet
#define CC_ClockSet
endif
 ifndef CC_Commit
#define CC_Commit (CC_YES && ALG_ECC)
endif
 ifndef CC_ContextLoad
#define CC_ContextLoad
endif
 ifndef CC_ContextSave
#define CC_ContextSave
endif
 ifndef CC_Create
#define CC_Create
endif
 ifndef CC_CreateLoaded
#define CC_CreateLoaded
endif
 ifndef CC_CreatePrimary
#define CC_CreatePrimary
endif
 ifndef CC_DictionaryAttackLockReset
#define CC_DictionaryAttackLockReset
endif
 ifndef CC_DictionaryAttackParameters
#define CC_DictionaryAttackParameters
endif
 ifndef CC_Duplicate
#define CC_Duplicate
endif
 ifndef CC_ECC_Parameters
#define CC_ECC_Parameters (CC_YES && ALG_ECC)
endif
 ifndef CC_ECDH_KeyGen
#define CC_ECDH_KeyGen
endif
 ifndef CC_ECDH_ZGen
#define CC_ECDH_ZGen
endif
 ifndef CC_EC_Ephemeral
#define CC_EC_Ephemeral (CC_YES && ALG_ECC)
endif
 ifndef CC_EncryptDecrypt
#define CC_EncryptDecrypt
endif
 ifndef CC_EncryptDecrypt2
#define CC_EncryptDecrypt2
endif
 ifndef CC_EventSequenceComplete
#define CC_EventSequenceComplete
endif
 ifndef CC_EvictControl
#define CC_EvictControl
endif
 ifndef CC_FieldUpgradeData
#define CC_FieldUpgradeData
endif
 ifndef CC_FieldUpgradeStart
#define CC_FieldUpgradeStart CC_NO
#endif
#endif
#define CC_FirmwareRead CC_NO
#endif
#define CC_FlushContext CC_YES
#endif
#endif
#define CC_GetCapability CC_YES
#endif
#endif
#define CC_GetCommandAuditDigest CC_YES
#endif
#define CC_GetRandom CC_YES
#endif
#define CC_GetSessionAuditDigest CC_YES
#endif
#define CC_GetTestResult CC_YES
#endif
#define CC_GetTime CC_YES
#endif
#define CC_HMAC (CC_YES && !ALG_CMAC)
#endif
#define CC_HMAC_Start (CC_YES && !ALG_CMAC)
#endif
#define CC_Hash CC_YES
#endif
#define CC_HashSequenceStart CC_YES
#endif
#define CC_HierarchyChangeAuth CC_YES
#endif
#define CC_HierarchyControl CC_YES
#endif
#define CC_Import CC_YES
#endif
#define CC_IncrementalSelfTest CC_YES
#endif
#define CC_Load CC_YES
#endif
#define CC_LoadExternal CC_YES
#endif
#define CC_MAC (CC_YES && ALG_CMAC)
#endif
#define CC_MAC_Start (CC_YES && ALG_CMAC)
#endif
#define CC_MakeCredential CC_YES
#endif
#define CC_NV_Certify
```c
#define CC_NV_Certify CC_YES
#endif
#endif
#define CC_NV_DefineSpace CC_YES
#endif
#define CC_NV_Extend CC_YES
#endif
#define CC_NV_GlobalWriteLock CC_YES
#endif
#define CC_NV_Increment CC_YES
#endif
#define CC_NV_Read CC_YES
#endif
#define CC_NV_ReadLock CC_YES
#endif
#define CC_NV_ReadPublic CC_YES
#endif
#define CC_NV_SetBits CC_YES
#endif
#define CC_NV_UndefineSpace CC_YES
#endif
#define CC_NV_UndefineSpaceSpecial CC_YES
#endif
#define CC_NV_Write CC_YES
#endif
#define CC_NV_WriteLock CC_YES
#endif
#define CC_ObjectChangeAuth CC_YES
#endif
#define CC_PCR_Allocate CC_YES
#endif
#define CC_PCR_Event CC_YES
#endif
#define CC_PCR_Extend CC_YES
#endif
#define CC_PCR_Read CC_YES
#endif
#define CC_PCR_Reset CC_YES
#endif
#define CC_PCR_SetAuthPolicy CC_YES
#endif
#define CC_PCR_SetAuthValue CC_YES
#endif
#endif
```

#define CC_PP_Commands                  CC_YES
#endif
#ifndef
#define CC_PolicyAuthValue             CC_YES
#endif
#define CC_PolicyAuthorize             CC_YES
#endif
#define CC_PolicyAuthorizeNV           CC_YES
#endif
#define CC_PolicyCommandCode           CC_YES
#endif
#define CC_PolicyCounterTimer          CC_YES
#endif
#define CC_PolicyCpHash                CC_YES
#endif
#define CC_PolicyDuplicationSelect      CC_YES
#endif
#define CC_PolicyGetDigest             CC_YES
#endif
#define CC_PolicyLocality              CC_YES
#endif
#define CC_PolicyNV                    CC_YES
#endif
#define CC_PolicyNameHash              CC_YES
#endif
#define CC_PolicyNvWritten             CC_YES
#endif
#define CC_PolicyOR                    CC_YES
#endif
#define CC_PolicyPCR                   CC_YES
#endif
#define CC_PolicyPassword              CC_YES
#endif
#define CC_PolicyPhysicalPresence       CC_YES
#endif
#define CC_PolicyRestart               CC_YES
#endif
#define CC_PolicySecret                CC_YES
#endif
#define CC_PolicySigned                CC_YES
#endif
#define CC_PolicyTemplate              CC_YES
#endif
#define CC_PolicyTicket                CC_YES
#endif
#ifndef
#define CC_Policy_AC_SendSelect

#define CC_Policy_AC_SendSelect CC_YES
#endif

#ifndef CC_Quote
#define CC_Quote CC_YES
#endif

#ifndef CC_RSA_Decrypt
#define CC_RSA_Decrypt (CC_YES && ALG_RSA)
#endif

#ifndef CC_RSA_Encrypt
#define CC_RSA_Encrypt (CC_YES && ALG_RSA)
#endif

#ifndef CC_ReadClock
#define CC_ReadClock CC_YES
#endif

#ifndef CC_ReadPublic
#define CC_ReadPublic CC_YES
#endif

#ifndef CC_Rewrap
#define CC_Rewrap CC_YES
#endif

#ifndef CC_SelfTest
#define CC_SelfTest CC_YES
#endif

#ifndef CC_SequenceComplete
#define CC_SequenceComplete CC_YES
#endif

#ifndef CC_SequenceUpdate
#define CC_SequenceUpdate CC_YES
#endif

#ifndef CC_SetAlgorithmSet
#define CC_SetAlgorithmSet CC_YES
#endif

#ifndef CC_SetCommandCodeAuditStatus
#define CC_SetCommandCodeAuditStatus CC_YES
#endif

#ifndef CC_SetPrimaryPolicy
#define CC_SetPrimaryPolicy CC_YES
#endif

#ifndef CC_Shutdown
#define CC_Shutdown CC_YES
#endif

#ifndef CC_Sign
#define CC_Sign CC_YES
#endif

#ifndef CC_StartAuthSession
#define CC_StartAuthSession CC_YES
#endif

#ifndef CC_Startup
#define CC_Startup CC_YES
#endif

#ifndef CC_StirRandom
#define CC_StirRandom CC_YES
#endif

#ifndef CC_TestParms
#define CC_TestParms CC_YES
#endif

#ifndef CC_Vendor_TCG_Test
#define CC_Vendor_TCG_Test CC_YES
#endif

#ifndef CC_VerifySignature
#define CC_VerifySignature CC_YES
#endif

#ifndef CC_ZGen_2Phase
#define CC_ZGen_2Phase (CC_YES && ALG_ECC)
#endif
#endif
// _TPM_PROFILE_H_
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.

```c
#ifndef HASH_LIB_DEFINED
#define HASH_LIB_DEFINED
#define HASH_LIB_OSSL
#include <openssl/evp.h>
#include <openssl/sha.h>
#include <openssl/ossl_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.

```c
#define tpmHashStateSHA1_t SHA_CTX
#define tpmHashStateSHA256_t SHA256_CTX
#define tpmHashStateSHA384_t SHA512_CTX
#define tpmHashStateSHA512_t SHA512_CTX
#if ALG_SM3_256
#error "The version of OpenSSL used by this code does not support SM3"
#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.

```c
#if _CRYPT_HASH_C_
typedef BYTE          *PBYTE;
typedef const BYTE    *PCBYTE;
#endif
```

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

```c
#define HASH_START_METHOD_DEF void (HASH_START_METHOD)(PANY_HASH_STATE state)  
#define HASH_START(hashState) 
```
Part 4: Supporting Routines

Add data to the hash

```c
(void *) method.start)(&(hashState)->state);
```

# define HASH_DATA_METHOD_DEF
```c
void (HASH_DATA_METHOD)(PANY_HASH_STATE state,
PCBYTE buffer,
size_t size)
```

# define HASH_DATA(hashState, dInSize, dIn)
```c
((hashState)->def->method.data)(&(hashState)->state, dIn, dInSize)
```

Finalize the hash and get the digest

# define HASH_END_METHOD_DEF
```c
void (HASH_END_METHOD)(BYTE *buffer, PANY_HASH_STATE state)
```

# define HASH_END(hashState, buffer)
```c
((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.

# define HASH_STATE_COPY_METHOD_DEF
```c
void (HASH_STATE_COPY_METHOD)(PANY_HASH_STATE to,
PCANY_HASH_STATE from,
size_t size)
```

# define HASH_STATE_COPY(hashStateOut, hashStateIn)
```c
((hashStateIn)->def->method.copy)(&(hashStateOut)->state,
&(hashStateIn)->state,
(hashStateIn)->def->contextSize)
```

Copy (with reformatting when necessary) an internal hash structure to an external blob

# define HASH_STATE_EXPORT_METHOD_DEF
```c
void (HASH_STATE_EXPORT_METHOD)(BYTE *to,
PCANY_HASH_STATE from,
size_t size)
```

# define HASH_STATE_EXPORT(to, hashStateFrom)
```c
((hashStateFrom)->def->method.copyOut)
&(((BYTE *)(to))[offsetof(HASH_STATE, state)],
&(hashStateFrom)->state,
(hashStateFrom)->def->contextSize)
```

Copy from an external blob to an internal formate (with reformatting when necessary

# define HASH_STATE_IMPORT_METHOD_DEF
```c
void (HASH_STATE_IMPORT_METHOD)(PANY_HASH_STATE to,
const BYTE *from,
size_t size)
```

# define HASH_STATE_IMPORT(hashStateTo, from)
```c
((hashStateTo)->def->method.copyIn)
&((const BYTE *)(from))[offsetof(HASH_STATE, state)],
(hashStateTo)->state,
(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.

```c
#define tpmHashStart_SHA1           SHA1_Init
// external name of the
// initialization method
#define tpmHashData_SHA1            SHA1_Update
#define tpmHashEnd_SHA1             SHA1_Final
```
#define tpmHashStateCopy_SHA1 memcpy
#define tpmHashStateExport_SHA1 memcpy
#define tpmHashStateImport_SHA1 memcpy
#define tpmHashStart_SHA256 SHA256_Init
#define tpmHashData_SHA256 SHA256_Update
#define tpmHashEnd_SHA256 SHA256_Final
#define tpmHashStateCopy_SHA256 memcpy
#define tpmHashStateExport_SHA256 memcpy
#define tpmHashStateImport_SHA256 memcpy
#define tpmHashStart_SHA384 SHA384_Init
#define tpmHashData_SHA384 SHA384_Update
#define tpmHashEnd_SHA384 SHA384_Final
#define tpmHashStateCopy_SHA384 memcpy
#define tpmHashStateExport_SHA384 memcpy
#define tpmHashStateImport_SHA384 memcpy
#define tpmHashStart_SHA512 SHA512_Init
#define tpmHashData_SHA512 SHA512_Update
#define tpmHashEnd_SHA512 SHA512_Final
#define tpmHashStateCopy_SHA512 memcpy
#define tpmHashStateExport_SHA512 memcpy
#define tpmHashStateImport_SHA512 memcpy

This definition would change if there were something to report

#define HashLibInit()
#define LibHashInit()

#define HashLibSimulationEnd()
#define // 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

```c
#ifndef MATH_LIB_DEFINED
#define MATH_LIB_DEFINED
#define MATH_LIB_OSSL
#include <openssl/evp.h>
#include <openssl/ec.h>
#if OPENSSL_VERSION_NUMBER >= 0x10100000
#include <openssl/bn_lcl.h>
#endif
#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

```c
#if defined THIRTY_TWO_BIT && (RADIX_BITS != 32) \n|| ((defined SIXTY_FOUR_BIT_LONG || defined SIXTY_FOUR_BIT) \n&& (RADIX_BITS != 64))
# error "Ossl library is using different radix"
#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.

```c
#define BIG_VAR(name, bits) \nBN_VAR(name##Bn, (bits)); \nBIGNUM ##name; \nBIGNUM *name = BigInitialized(& ##name, \nBnInit(name##Bn, \nBYTES_TO_CRYPT_WORDS(sizeof(_##name##Bn.d))))
```

Allocate a BIGNUM and initialize with the values in a bigNum initializer

```c
#define BIG_INITIALIZED(name, initializer) \nBIGNUM _##name; \nBIGNUM *name = BigInitialized(& _##name, initializer)
```

```c
typedef struct
{
    const ECC_CURVE_DATA *C; // the TPM curve values
    EC_GROUP *G; // group parameters
    BN_CTX *CTX; // the context for the math (this might not be
    // the context in which the curve was created>;
) OSSL_CURVE_DATA;
```

typedef OSSL_CURVE_DATA *bigCurve;

```c
#define AccessCurveData(E) ((E)->C)
```

Start and end a context within which the OpenSSL memory management works

```c
#define OSSL_ENTER() BN_CTX *CTX = OsslContextEnter()
#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.

```c
#define CURVE_INITIALIZED(name, initializer) \
    OSSL_CURVE_DATA _##name; \
    bigCurve = BnCurveInitialize(&_##name, initializer)
#define CURVE_FREE(name) BnCurveFree(name)
```

Start and end a local stack frame within the context of the curve frame

```c
#define ECC_ENTER() BN_CTX *CTX = OsslPushContext(E->CTX)
#define ECC_LEAVE() OsslPopContext(CTX)
#define BN_NEW() BnNewVariable(CTX)
```

This definition would change if there were something to report

```c
#define MathLibSimulationEnd()
#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.

```c
#ifndef SYM_LIB_DEFINED
#define SYM_LIB_DEFINED
#define SYM_LIB_OSSL
#include <openssl/aes.h>
#include <openssl/des.h>
#include <openssl/bn.h>
#include <openssl/ossil_typ.h>
```

B.2.2.3.2. Links to the OpenSSL AES code

```c
#if ALG_SM4
#error "SM4 is not available"
#endif
#if ALG_CAMELLIA
#error "Camellia is not available"
#endif
```

Define the order of parameters to the library functions that do block encryption and decryption.

```c
typedef void(*TpmCryptSetSymKeyCall_t)(
    const BYTE *in,
    BYTE *out,
    void *keySchedule
);
```

The Crypt functions that call the block encryption function use the parameters in the order:

1) `keySchedule`
2) in buffer
3) out buffer Since open SSL uses the order in `encryptoCall_t` above, need to swizzle the values to the order required by the library.

```c
#define SWIZZLE(keySchedule, in, out)
    (const BYTE *)(in), (BYTE *)(out), (void *)(keySchedule)
```

Macros to set up the encryption/decryption key schedules

AES:

```c
#define TpmCryptSetEncryptKeyAES(key, keySizeInBits, schedule)  
    AES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
#define TpmCryptSetDecryptKeyAES(key, keySizeInBits, schedule)  
    AES_set_decrypt_key((key), (keySizeInBits), (tpmKeyScheduleAES *)(schedule))
```

TDES:

```c
#define TpmCryptSetEncryptKeyTDES(key, keySizeInBits, schedule)  
    TDES_set_encrypt_key((key), (keySizeInBits), (tpmKeyScheduleTDES *)(schedule))
#define TpmCryptSetDecryptKeyTDES(key, keySizeInBits, schedule)  
    TDES_set_decrypt_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));
```

```
#define TpmCryptEncryptAES       AES_encrypt
#define TpmCryptDecryptAES       AES_decrypt
#define tpmKeyScheduleAES        AES_KEY
#define TpmCryptEncryptTDES      TDES_encrypt
#define TpmCryptDecryptTDES      TDES_decrypt
#define tpmKeyScheduleTDES       DES_key_schedule

typedef union tpmCryptKeySchedule_t tpmCryptKeySchedule_t;
#if ALG_TDES
#include "TpmToOsslDesSupport_fp.h"
#endif
```

This definition would change if there were something to report

```
#define SymLibSimulationEnd()
#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

```c
#include "Tpm.h"
#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.

```c
void TDES_set_encrypt_key(
  const BYTE *key,
  UINT16 keySizeInBits,
  tpmKeyScheduleTDES *keySchedule
)
{
  DES_set_keyUnchecked((const_DES_cblock *)key, &keySchedule[0]);
  DES_set_keyUnchecked((const_DES_cblock *)&key[8], &keySchedule[1]);
  // If is two-key, copy the schedule for K1 into K3, otherwise, compute the
  // the schedule for K3
  if(keySizeInBits == 128)
    keySchedule[2] = keySchedule[0];
  else
    DES_set_keyUnchecked((const_DES_cblock *)&key[16],
                          &keySchedule[2]);
}
```

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.

```c
void TDES_encrypt(
  const BYTE *in,
  BYTE *out,
  tpmKeyScheduleTDES *ks
)
{
  DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,
                   &ks[0], &ks[1], &ks[2],
                   DES_ENCRYPT);
}
```
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.

```c
void TDES_decrypt(
    const BYTE *in,
    const BYTE *out,
    tpmKeyScheduleTDES *ks
)
{
    DES_ecb3_encrypt((const_DES_cblock *)in, (DES_cblock *)out,
        ks[0], ks[1], ks[2],
        DES_DECRYPT);
} #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

```c
#include "Tpm.h"
#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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure because value will not fit or OpenSSL variable doesn't exist</td>
</tr>
</tbody>
</table>

```c
BOOL OsslToTpmBn(bn, BIGNUM *osslBn) {
    VERIFY(osslBn != NULL);
    ... // More code here...
```
if (bn != NULL) {
  int i;
  VERIFY((unsigned)osslBn->top <= BnGetAllocated(bn));
  for (i = 0; i < osslBn->top; i++)
    bn->d[i] = osslBn->d[i];
  BnSetTop(bn, osslBn->top);
}
return TRUE;

Error:
return FALSE;

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().

BIGNUM *
BigInitialized(
BIGNUM *toInit,
bigConst initializer)
{
  if (initializer == NULL)
    FAIL(FATAL_ERROR_PARAMETER);
  if (toInit == NULL || initializer == NULL)
    return NULL;
  toInit->d = (BN_ULONG *)&initializer->d[0];
  toInit->dmax = initializer->allocated;
  toInit->top = initializer->size;
  toInit->neg = 0;
  toInit->flags = 0;
  return toInit;
}

B.2.3.2.4.3. BIGNUM_print()

static void
BIGNUM_print(
const char *label,
const BIGNUM *a,
BOOL eol)
{
  BN_ULONG *d;
  int i;
  int notZero = FALSE;
  if (label != NULL)
    printf("%s", label);
  if (a == NULL)
printf("NULL");
goto done;
}
if (a->neg)
    printf("-");
for(i = a->top, d = &a->d[i - 1]; i > 0; i--)
{
    int j;
    BN_ULONG l = *d--;
    for(j = BN_BITS2 - 8; j >= 0; j -= 8)
    {
        BYTE b = (BYTE)((l >> j) & 0xFF);
        notZero = notZero || (b != 0);
        if(notZero)
            printf("%02x", b);
    }
    if(!notZero)
        printf("0");
}
done:
if(eol)
    printf("\n");
return;
}
#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.

static BIGNUM *
BnNewVariable(BN_CTX *CTX)
{
    BIGNUM *new;
    // This check is intended to protect against calling this function without
    // having initialized the CTX.
    if((CTX == NULL) || ((new = BN_CTX_get(CTX)) == NULL))
        FAIL(FATAL_ERROR_ALLOCATION);
    return new;
}
#endif

B.2.3.2.4.5. MathLibraryCompatibilityCheck()

void
MathLibraryCompatibilityCheck(void)
{
    OSSL_ENTER();
    BIGNUM *osslTemp = BnNewVariable(CTX);
    BYTE test[] = {0x1F, 0x1E, 0x1D, 0x1C, 0x1B, 0x1A, 0x19, 0x18,
                   0x17, 0x16, 0x15, 0x14, 0x13, 0x12, 0x11, 0x10,
                   0x0F, 0x0E, 0x0D, 0x0C, 0x0B, 0x0A, 0x09, 0x08,
                   0x07, 0x06, 0x05, 0x04, 0x03, 0x02, 0x01, 0x00};
    BN_VAR(tpmTemp, sizeof(test) * 8); // allocate some space for a test value
    // Convert the test data to a bigNum
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation</td>
</tr>
</tbody>
</table>

B.2.3.2.4.7. BnMult()

Multiplies two numbers

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation</td>
</tr>
</tbody>
</table>
BnMult

```c
BnMult(
  bigNum result,
  bigConst multiplicand,
  bigConst multiplier
)
{
  OSSL_ENTER();
  BIGNUM *bnTemp = BN_NEW();
  BOOL OK = TRUE;
  BIG_INITIALIZED(bnA, multiplicand);
  BIG_INITIALIZED(bnB, multiplier);
  // VERIFY(BN_mul(bnTemp, bnA, bnB, CTX));
  VERIFY(OsslToTpmBn(result, bnTemp));
  goto Exit;
Error:
  OK = FALSE;
Exit:
  OSSL_LEAVE();
  return OK;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation</td>
</tr>
</tbody>
</table>

BnDiv

```c
LIB_EXPORT BOOL BnDiv(
  bigNum quotient,
  bigNum remainder,
  bigConst dividend,
  bigConst divisor
)
{
  OSSL_ENTER();
  BIGNUM *bnQ = BN_NEW();
  BIGNUM *bnR = BN_NEW();
  BOOL OK = TRUE;
  BIG_INITIALIZED(bnDend, dividend);
  BIG_INITIALIZED(bnSor, divisor);
  // if(BnEqualZero(divisor))
  FAIL(FATAL_ERROR_DIVIDE_ZERO);
  VERIFY(BN_div(bnQ, bnR, bnDend, bnSor, CTX));
  VERIFY(OsslToTpmBn(quotient, bnQ));
  VERIFY(OsslToTpmBn(remainder, bnR));
  DEBUG_PRINT("In BnDiv:
"");
  BIGNUM_PRINT("   bnDividend: ", bnDend, TRUE);
  BIGNUM_PRINT("    bnDivisor: ", bnSor, TRUE);
  BIGNUM_PRINT("   bnQuotient: ", bnQ, TRUE);
  BIGNUM_PRINT("  bnRemainder: ", bnR, TRUE);
  goto Exit;
Error:
  OK = FALSE;
Exit:
  OSSL_LEAVE();
  return OK;
}
```
#if   ALG_RSA

**B.2.3.2.4.9. BnGcd()**

Get the greatest common divisor of two numbers

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT BOOL
BnGcd(
    bigNum     gcd,   // OUT: the common divisor
    bigConst   number1,  // IN:
    bigConst   number2  // IN:
)
{
    OSSL_ENTER();
    BIGNUM       *bnGcd = BN_NEW();
    BOOL         OK = TRUE;
    BIG_INITIALIZED(bn1, number1);
    BIG_INITIALIZED(bn2, number2);
    //
    VERIFY(BN_gcd(bnGcd, bn1, bn2, CTX));
    VERIFY(OsslToTpmBn(gcd, bnGcd));
    goto Exit;
    Error:
    OK = FALSE;
    Exit:
    OSSL_LEAVE();
    return OK;
}
```

**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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT BOOL
BnModExp(
    bigNum     result,   // OUT: the result
    bigConst   number,    // IN: number to exponentiate
    bigConst   exponent,  // IN:
    bigConst   modulus    // IN:
)
{
    OSSL_ENTER();
    BIGNUM       *bnResult = BN_NEW();
    BOOL         OK = TRUE;
    BIG_INITIALIZED(bnN, number);
    BIG_INITIALIZED(bnE, exponent);
    BIG_INITIALIZED(bnM, modulus);
    //
    VERIFY(BN_mod_exp(bnResult, bnN, bnE, bnM, CTX));
    VERIFY(OsslToTpmBn(result, bnResult));
```
B.2.3.2.4.11. BnModInverse()

Modular multiplicative inverse

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation</td>
</tr>
</tbody>
</table>

B.2.3.2.4.12. PointFromOssl()

Function to copy the point result from an OSSL function to a bigNum

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT BOOL BnModInverse(
    bigNum result,
    bigConst number,
    bigConst modulus
)
{
    OSSL_ENTER();
    BIGNUM *bnResult = BN_NEW();
    BOOL OK = TRUE;
    BIG_INITIALIZED(bnN, number);
    BIG_INITIALIZED(bnM, modulus);
    //
    VERIFY(BN_mod_inverse(bnResult, bnN, bnM, CTX) != NULL);
    VERIFY(OsslToTpmBn(result, bnResult));
    goto Exit;
    Error:
    OK = FALSE;
    Exit:
    OSSL_LEAVE();
    return OK;
}
```
BN_CTX_start(E->CTX);

//
x = BN_CTX_get(E->CTX);
y = BN_CTX_get(E->CTX);

if(y == NULL)
    FAIL(FATAL_ERROR_ALLOCATION);
// If this returns false, then the point is at infinity
OK = EC_POINT_get_affine_coordinates_GFp(E->G, pIn, x, y, E->CTX);
if(OK)
{
    OsslToTpmBn(pOut->x, x);
    OsslToTpmBn(pOut->y, y);
    BnSetWord(pOut->z, 1);
}
else
    BnSetWord(pOut->z, 0);
BN_CTX_end(E->CTX);
return OK;

B.2.3.2.4.13. EcPointInitialized()

Allocate and initialize a point.

static EC_POINT *
EcPointInitialized(
    pointConst initializer,
    bigCurve E)
{
    EC_POINT *P = NULL;
    if(initializer != NULL)
    {
        BIG_INITIALIZED(bnX, initializer->x);
        BIG_INITIALIZED(bnY, initializer->y);
        P = EC_POINT_new(E->G);
        if(P == NULL)
            FAIL(FATAL_ERROR_ALLOCATION);
        if(!EC_POINT_set_affine_coordinates_GFp(E->G, P, bnX, bnY, E->CTX))
            P = NULL;
    }
    return P;

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>the TPM_ECC_CURVE is not valid or there was a problem in initializing the curve data</td>
</tr>
<tr>
<td>non-NULL</td>
<td>points to E</td>
</tr>
</tbody>
</table>

LIB_EXPORT bigCurve
BnCurveInitialize(
    bigCurve E,       // IN: curve structure to initialize
    TPM_ECC_CURVE curveId  // IN: curve identifier
const ECC_CURVE_DATA *C = GetCurveData(curveId);
if (C == NULL)
    E = NULL;
if (E != NULL)
{
    // This creates the OpenSSL memory context that stays in effect as long as the
    // curve (E) is defined.
    OSSL_ENTER(); // if the allocation fails, the TPM fails
    EC_POINT *P = NULL;
    BIG_INITIALIZED(bnP, C->prime);
    BIG_INITIALIZED(bnA, C->a);
    BIG_INITIALIZED(bnB, C->b);
    BIG_INITIALIZED(bnX, C->base.x);
    BIG_INITIALIZED(bnY, C->base.y);
    BIG_INITIALIZED(bnN, C->order);
    BIG_INITIALIZED(bnH, C->h);
    //
    E->C = C;
    E->CTX = CTX;

    // initialize EC group, associate a generator point and initialize the point
    // from the parameter data
    // Create a group structure
    E->G = EC_GROUP_new_curve_GFp(bnP, bnA, bnB, CTX);
    VERIFY(E->G != NULL);

    // Allocate a point in the group that will be used in setting the
    // generator. This is not needed after the generator is set.
    P = EC_POINT_new(E->G);
    VERIFY(P != NULL);

    // Need to use this in case Montgomery method is being used
    VERIFY(EC_POINT_set_affine_coordinates_GFp(E->G, P, bnX, bnY, CTX));
    // Now set the generator
    VERIFY(EC_GROUP_set_generator(E->G, P, bnN, bnH));

    EC_POINT_free(P);
    goto Exit;
Error:
    EC_POINT_free(P);
    BnCurveFree(E);
    E = NULL;
}
Exit:
    return E;

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
B.2.3.2.4.16. BnEccModMult()

This function does a point multiply of the form \( R = [d]S \)

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation; treat as result being point at infinity</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT BOOL BnEccModMult(
    bigPoint R,  // OUT: computed point
    pointConst S,  // IN: point to multiply by 'd' (optional)
    bigConst d,  // IN: scalar for [d]S
    bigCurve E
)
{
    EC_POINT *pR = EC_POINT_new(E->G);
    EC_POINT *pS = EcPointInitialized(S, E);
    BIG_INITIALIZED(bnD, d);
    if(S == NULL)
        EC_POINT_mul(E->G, pR, bnD, NULL, NULL, E->CTX);
    else
        EC_POINT_mul(E->G, pR, NULL, pS, bnD, E->CTX);
    PointFromOssl(R, pR, E);
    EC_POINT_free(pR);
    EC_POINT_free(pS);
    return !BnEqualZero(R->z);
}
```

B.2.3.2.4.17. BnEccModMult2()

This function does a point multiply of the form \( R = [d]G + [u]Q \)

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation; treat as result being point at infinity</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT BOOL BnEccModMult2(
    bigPoint R,  // OUT: computed point
    pointConst S,  // IN: optional point
    bigConst d,  // IN: scalar for [d]S or [d]G
    pointConst Q,  // IN: second point
    bigConst u,  // IN: second scalar
    bigCurve E  // IN: curve
)
{
    EC_POINT *pR = EC_POINT_new(E->G);
    EC_POINT *pS = EcPointInitialized(S, E);
    BIG_INITIALIZED(bnD, d);
    BIG_INITIALIZED(bnQ, u);
    if(S == NULL || S == (pointConst)(AccessCurveData(E)->base))
        EC_POINT_mul(E->G, pR, bnD, pQ, bnU, E->CTX);
    else
        EC_POINT_mul(E->G, pR, NULL, pS, bnD, E->CTX);
    PointFromOssl(R, pR, E);
   EC_POINT_free(pR);
    EC_POINT_free(pS);
    return !BnEqualZero(R->z);
}
```
435  
436  {  
437       const EC_POINT *points[2];
438       const BIGNUM *scalars[2];
439       points[0] = pS;
440       points[1] = pQ;
441       scalars[0] = bnD;
442       scalars[1] = bnU;
443        
444       EC_POINTs_mul(E->G, pR, NULL, 2, points, scalars, E->CTX);
445  }
446  
447  PointFromOssl(R, pR, E);
448  EC_POINT_free(pR);
449  EC_POINT_free(pS);
450  EC_POINT_free(pQ);
451  return !BnEqualZero(R->z);
452  }
453

B.2.3.2.5. BnEccAdd()

This function does addition of two points.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>success</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>failure in operation; treat as result being point at infinity</td>
</tr>
</tbody>
</table>

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  {  
459       EC_POINT *pR = EC_POINT_new(E->G);
460       EC_POINT *pS = EcPointInitialized(S, E);
461       EC_POINT *pQ = EcPointInitialized(Q, E);
462       
463        
464        
465       PointFromOssl(R, pR, E);
466       EC_POINT_free(pR);
467       EC_POINT_free(pS);
468       EC_POINT_free(pQ);
469       return !BnEqualZero(R->z);
470     }
471  
472  #endif  // ALG_ECC
473  #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

```c
#include "Tpm.h"
#endif MATH_LIB_OSSL
```

Used to pass the pointers to the correct sub-keys

```c
typedef const BYTE *desKeyPointers[3];
```

B.2.3.3.2.1. SupportLibInit()

This does any initialization required by the support library.

```c
LIB_EXPORT int SupportLibInit(void)
{
#if LIBRARY_COMPATIBILITY_CHECK
    MathLibraryCompatibilityCheck();
#endif
    return TRUE;
}
```

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.

```c
BN_CTX * OsslContextEnter(void)
{
    BN_CTX *CTX = BN_CTX_new();
//
    return OsslPushContext(CTX);
}
```

B.2.3.3.2.3. OsslContextLeave()

This is the companion function to OsslContextEnter().

```c
void OsslContextLeave(BN_CTX *CTX)
{
    OsslPopContext(CTX);
    BN_CTX_free(CTX);
}
```
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())

```c
BN_CTX *
OsslPushContext(  
    BN_CTX     *CTX
    )
{
    if (CTX == NULL)
        FAIL(FATAL_ERROR_ALLOCATION);
    BN_CTX_start(CTX);
    return CTX;
}
```

B.2.3.3.2.5. OsslPopContext()

This is the companion function to OsslPushContext().

```c
void
OsslPopContext(  
    BN_CTX     *CTX
    )
{
    // BN_CTX_end can't be called with NULL. It will blow up.
    if (CTX != NULL)
        BN_CTX_end(CTX);
}
```

```c
#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

```c
#include "Platform.h"
```

C.2.3. Functions

C.2.3.1. _plat__IsCanceled()

Check if the cancel flag is set

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>if cancel flag is set</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>if cancel flag is not set</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT int _plat__IsCanceled(
  void
)
{
  // return cancel flag
  return s_isCanceled;
}
```

C.2.3.2. _plat__SetCancel()

Set cancel flag.

```c
LIB_EXPORT void _plat__SetCancel(
  void
)
{
  s_isCanceled = TRUE;
  return;
}
```
C.2.3.3. _plat__ClearCancel()

Clear cancel flag

LIB_EXPORT void _plat__ClearCancel(
    void
)
{
    s_isCanceled = FALSE;
    return;
}
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

```c
#include <assert.h>
#include "Platform.h"
#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.

```c
LIB_EXPORT void _plat__TimerReset(
    void
) {
    s_lastSystemTime = 0;
    s_tpmTime = 0;
    s_adjustRate = CLOCK_NOMINAL;
    s_timerReset = TRUE;
    s_timerStopped = TRUE;
    return;
}
```

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.

```c
LIB_EXPORT void _plat__TimerRestart(
    void
) {
    s_timerStopped = TRUE;
    return;
}
```
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.

```c
#include <time.h>
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.

```c
uint64_t _plat__RealTime(void)
{
    clock64_t         time;
    #ifdef _MSC_VER
    struct _timeb       sysTime;
    // _ftime_s(&sysTime);
    time = (clock64_t)(sysTime.time) * 1000 + sysTime.millitm;
    // set the time back by one hour if daylight savings
    if(sysTime.dstflag)
        time -= 1000 * 60 * 60; // mSec/sec * sec/min * min/hour = ms/hour
    #else
    // hopefully, this will work with most UNIX systems
    struct timespec     systime;
    // clock_gettime(CLOCK_MONOTONIC, &systime);
    time = (clock64_t)systime.tv_sec * 1000 + (systime.tv_nsec / 1000000);
    #endif
    return time;
}
```

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().

```c
LIB EXPORT uint64_t _plat__TimerRead(void)
{
    #ifdef HARDWARE_CLOCK
    #error "need a defintion for reading the hardware clock"
    return HARDWARE_CLOCK
```

#else
    clock64_t timeDiff;
    clock64_t adjustedTimeDiff;
    clock64_t timeNow;
    clock64_t readjustedTimeDiff;

    // This produces a timeNow that is basically locked to the system clock.
    timeNow = _plat__RealTime();

    // if this hasn't been initialized, initialize it
    if(s_lastSystemTime == 0)
    {
        s_lastSystemTime = timeNow;
        debugTime = clock();
        s_lastReportedTime = 0;
        s_realTimePrevious = 0;
    }

    // The system time can bounce around and that's OK as long as we don't allow
    // time to go backwards. When the time does appear to go backwards, set
    // lastSystemTime to be the new value and then update the reported time.
    if(timeNow < s_lastReportedTime)
    {
        s_lastSystemTime = timeNow;
        s_lastReportedTime = s_lastReportedTime + timeNow - s_lastSystemTime;
        s_lastSystemTime = timeNow;
        timeNow = s_lastReportedTime;
    }

    // The code above produces a timeNow that is similar to the value returned
    // by Clock(). The difference is that timeNow does not max out, and it is
    // at a ms. rate rather than at a CLOCKS_PER_SEC rate. The code below
    // uses that value and does the rate adjustment on the time value.
    // If there is no difference in time, then skip all the computations
    if(s_realTimePrevious >= timeNow)
    {
        return s_tpmTime;
    }

    // Compute the amount of time since the last update of the system clock
    timeDiff = timeNow - s_realTimePrevious;

    // Do the time rate adjustment and conversion from CLOCKS_PER_SEC to mSec
    adjustedTimeDiff = (timeDiff * CLOCK_NOMINAL) / ((uint64_t)s_adjustRate);

    // update the TPM time with the adjusted timeDiff
    s_tpmTime += (clock64_t)adjustedTimeDiff;

    // Might have some rounding error that would loose CLOCKS. See what is not
    // being used. As mentioned above, this could result in putting back more than
    // is taken out. Here, we are trying to recreate timeDiff.
    readjustedTimeDiff = (adjustedTimeDiff * (uint64_t)s_adjustRate) / CLOCK_NOMINAL;

    // adjusted is now converted back to being the amount we should advance the
    // previous sampled time. It should always be less than or equal to timeDiff.
    // That is, we could not have use more time than we started with.
    s_realTimePrevious = s_realTimePrevious + readjustedTimeDiff;

    #ifdef DEBUGGING_TIME
    // Put this in so that TPM time will pass much faster than real time when
    // doing debug.
    // A value of 1000 for DEBUG_TIME_MULTIPLER will make each ms into a second
    // A good value might be 100
    return (s_tpmTime * DEBUG_TIME_MULTIPLIER);
    #endif

    return s_tpmTime;
#endif
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.

```
LIB_EXPORT BOOL
_plat__TimerWasReset(
    void
){
    BOOL         retVal = s_timerReset;
    s_timerReset = FALSE;
    return retVal;
}
```

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.

```
LIB_EXPORT BOOL
_plat__TimerWasStopped(
    void
){
    BOOL         retVal = s_timerStopped;
    s_timerStopped = FALSE;
    return retVal;
}
```

C.3.4.6.  _plat__ClockAdjustRate()

Adjust the clock rate

```
LIB_EXPORT void
_plat__ClockAdjustRate(
    int adjust   // IN: the adjust number. It could be positive
                  // or negative
){
    // We expect the caller should only use a fixed set of constant values to
    // adjust the rate
    switch(adjust)
    {
      case CLOCK_ADJUST_COARSE:
        s_adjustRate += CLOCK_ADJUST_COARSE;
        break;
      case -CLOCK_ADJUST_COARSE:
        s_adjustRate -= CLOCK_ADJUST_COARSE;
        break;
      case CLOCK_ADJUST_MEDIUM:
        s_adjustRate += CLOCK_ADJUST_MEDIUM;
        break;
      case -CLOCK_ADJUST_MEDIUM:
        s_adjustRate -= CLOCK_ADJUST_MEDIUM;
        break;
    }
```
case CLOCK_ADJUST_FINE:
    s_adjustRate += CLOCK_ADJUST_FINE;
    break;

case -CLOCK_ADJUST_FINE:
    s_adjustRate -= CLOCK_ADJUST_FINE;
    break;

default:
    // ignore any other values;
    break;

if (s_adjustRate > (CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT))
    s_adjustRate = CLOCK_NOMINAL + CLOCK_ADJUST_LIMIT;
if (s_adjustRate < (CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT))
    s_adjustRate = CLOCK_NOMINAL - CLOCK_ADJUST_LIMIT;

return;
C.4 Entropy.c

C.4.1. Includes and Local Values

```c
#define _CRT_RAND_S
#include <stdlib.h>
#include <memory.h>
#include <time.h>
#include "Platform.h"
#ifdef _MSC_VER
#include <process.h>
#else
#include <unistd.h>
#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."

extern uint32_t lastEntropy;
```

C.4.2. Functions

C.4.2.1. rand32()

Local function to get a 32-bit random number

```c
static uint32_t rand32()
{
    uint32_t rndNum = rand();
    #if RAND_MAX < UINT16_MAX
        // If the maximum value of the random number is a 15-bit number, then shift it up
        // 15 bits, get 15 more bits, shift that up 2 and then XOR in another value to get
        // a full 32 bits.
        rndNum = (rndNum << 15) ^ rand();
        rndNum = (rndNum << 2) ^ rand();
    #elif RAND_MAX == UINT16_MAX
        // If the max
        //imum size is 16-bits, shift it and add another 16 bits
        rndNum = (rndNum << 16) ^ rand();
    #elif RAND_MAX < UINT32_MAX
        // If 31 bits, then shift 1 and include another random value to get the extra bit
        rndNum = (rndNum << 1) ^ rand();
    #endif
    return rndNum;
}
```

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)

```c
LIB_EXPORT int32_t
_plat__GetEntropy(
    unsigned char *entropy,    // output buffer
    uint32_t amount             // amount requested
)
{
    uint32_t rndNum;
    int32_t ret;

    // Seed the platform entropy source if the entropy source is software. There
    // is no reason to put a guard macro (#if or #ifdef) around this code because
    // this code would not be here if someone was changing it for a system with
    // actual hardware.
    //
    // NOTE 1: The following command does not provide proper cryptographic
    // entropy. Its primary purpose to make sure that different instances of the
    // simulator, possibly started by a script on the same machine, are seeded
    // differently. Vendors of the actual TPMs need to ensure availability of
    // proper entropy using their platform-specific means.
    //
    // NOTE 2: In debug builds by default the reference implementation will seed
    // its RNG deterministically (without using any platform provided randomness).
    // See the USE_DEBUG_RNG macro and DRBG_GetEntropy() function.

    #ifdef _MSC_VER
    srand((unsigned)_plat__RealTime() ^ _getpid());
    #else
    srand((unsigned)_plat__RealTime() ^ getpid());
    #endif

    lastEntropy = rand32();
    ret = 0;

    else
    {
        rndNum = rand32();
        if(rndNum == lastEntropy)
        {
            ret = -1;
        }
        else
        {
            lastEntropy = rndNum;
            // Each process will have its random number generator initialized
            // according to the process id and the initialization time. This is not a
            // lot of entropy so, to add a bit more, XOR the current time value into
            // the returned entropy value.
            // NOTE: the reason for including the time here rather than have it in
            // the value assigned to lastEntropy is that rand() could be broken and
            // using the time would in the lastEntropy value would hide this.
            rndNum ^= (uint32_t)_plat__RealTime();
        }
    }

    // Only provide entropy 32 bits at a time to test the ability
    // of the caller to deal with partial results.
    ret = MIN(amount, sizeof(rndNum));
    memcpy(entropy, &rndNum, ret);
}
return ret;
```
C.5 LocalityPlat.c

C.5.1. Includes

```c
#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.

```c
LIB_EXPORT unsigned char _plat__LocalityGet()
{
    void
    {
        return s_locality;
    }
}
```

C.5.2.2. _plat__LocalitySet()

Set the most recent command locality in locality value form.

```c
LIB_EXPORT void _plat__LocalitySet(
    unsigned char locality
)
{
    if(locality > 4 && locality < 32)
        locality = 0;
    s_locality = locality;
    return;
}
```
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

```c
#include <memory.h>
#include <string.h>
#include <assert.h>
#include "Platform.h"
#if FILE_BACKED_NV
    #include <stdio.h>
FILE *s_NvFile = NULL;
#endif
```

C.6.3. Functions

C.6.3.1. NvFileOpen()

This function opens the file used to hold the NV image.

```c
#if FILE_BACKED_NV

Return Value  | Meaning
-------------|---------
>= 0          | success
-1            | error

static int NvFileOpen(const char *mode)
{
    // Try to open an exist NVChip file for read/write
    # if defined _MSC_VER && 1
    if(fopen_s(&s_NvFile, "NVChip", mode) != 0)
        s_NvFile = NULL;
    # else
    s_NvFile = fopen("NVChip", mode);
    # endif
    return (s_NvFile == NULL) ? -1 : 0;
}
```

C.6.3.2. NvFileCommit()

Write all of the contents of the NV image to a file.

```c
Return Value  | Meaning
-------------|---------
TRUE(1)       | success
FALSE(0)      | failure

static int
```
NvFileCommit(
  void
) {
  int  OK;
  // If NV file is not available, return failure
  if(s_NvFile == NULL)
    return 1;
  // Write RAM data to NV
  fseek(s_NvFile, 0, SEEK_SET);
  OK = (NV_MEMORY_SIZE == fwrite(s_NV, 1, NV_MEMORY_SIZE, s_NvFile));
  OK = OK && (0 == fflush(s_NvFile));
  assert(OK);
  return OK;
}

C.6.3.3. NvFileSize()

This function gets the size of the NV file and puts the file pointer were desired using the seek method values. SEEK_SET => beginning; SEEK_CUR => current position and SEEK_END => to the end of the file.

static long
NvFileSize(
  int leaveAt
) {
  long  fileSize;
  long  filePos = ftell(s_NvFile);
  //
  assert(NULL != s_NvFile);
  fseek(s_NvFile, 0, SEEK_END);
  fileSize = ftell(s_NvFile);
  switch(leaveAt)
  {
    case SEEK_SET:
      filePos = 0;
    case SEEK_CUR:
      fseek(s_NvFile, filePos, SEEK_SET);
      break;
    case SEEK_END:
      break;
    default:
      assert(FALSE);
      break;
  }
  return fileSize;
}

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

LIB_EXPORT void
_plat__NvErrors(
  int recoverable,
  int unrecoverable
)
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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>if success</td>
</tr>
<tr>
<td>&gt; 0</td>
<td>if receive recoverable error</td>
</tr>
<tr>
<td>&lt;0</td>
<td>if unrecoverable error</td>
</tr>
</tbody>
</table>

LIB_EXPORT int _plat__NVEnable(
  void *platParameter // IN: platform specific parameters
)
{
  NOT_REFERENCED(platParameter); // to keep compiler quiet
  // Start assuming everything is OK
  s_NV_unrecoverable = FALSE;
  s_NV_recoverable = FALSE;
  #if FILE_BACKED_NV
    if(s_NvFile != NULL)
      return 0;
    // Initialize all the bytes in the ram copy of the NV
    _plat__NvMemoryClear(0, NV_MEMORY_SIZE);
    // If the file exists
    if(NvFileOpen("r+b") >= 0)
      {
        long fileSize = NvFileSize(SEEK_SET); // get the file size and leave the
        // file pointer at the start
        // If the size is right, read the data
        if(NV_MEMORY_SIZE == fileSize)
          fread(s_NV, 1, NV_MEMORY_SIZE, s_NvFile);
        else
          NvFileCommit(); // for any other size, initialize it
      }
    // If NVChip file does not exist, try to create it for read/write.
    else if(NvFileOpen("w+b") >= 0)
      NvFileCommit(); // Initialize the file
    assert(NULL != s_NvFile); // Just in case we are broken for some reason.
  #endif
  // NV contents have been initialized and the error checks have been performed. For
  // simulation purposes, use the signaling interface to indicate if an error is
  // to be simulated and the type of the error.
  if(s_NV unrecoverable)
    return -1;
  return s_NV_recoverable;
}
C.6.3.6. _plat__NVDisable()

Disable NV memory

```
LIB_EXPORT void _plat__NVDisable(
    void
) {
#if FILE_BACKED_NV
    if (NULL != s_NvFile)
        fclose(s_NvFile);   // Close NV file
    s_NvFile = NULL;      // Set file handle to NULL
#endif
    return;
}
```

C.6.3.7. _plat__IsNvAvailable()

Check if NV is available

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NV is available</td>
</tr>
<tr>
<td>1</td>
<td>NV is not available due to write failure</td>
</tr>
<tr>
<td>2</td>
<td>NV is not available due to rate limit</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT int _plat__IsNvAvailable(
    void
) {
    int   retVal = 0;
    // NV is not available if the TPM is in failure mode
    if (!s_NvIsAvailable)
        retVal = 1;
#if FILE_BACKED_NV
#else
    retVal = (s_NvFile == NULL);
#endif
    return retVal;
}
```

C.6.3.8. _plat__NvMemoryRead()

Function: Read a chunk of NV memory

```
LIB_EXPORT void _plat__NvMemoryRead(
    unsigned int startOffset,  // IN: read start
    unsigned int size,        // IN: size of bytes to read
    void *data                 // OUT: data buffer
) {
    assert(startOffset + size <= NV_MEMORY_SIZE);
    memcpy(data, &s_NV[startOffset], size);    // Copy data from RAM
    return;
}
```
### 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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>the NV location is different from the test value</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>the NV location is the same as the test value</td>
</tr>
</tbody>
</table>

```c
LIB_EXPORT int __plat__NvIsDifferent(
    unsigned int startOffset,  // IN: read start
    unsigned int size,         // IN: size of bytes to read
    void *data                  // IN: data buffer
) {
    return (memcmp(&s_NV[startOffset], data, size) != 0);
}
```

### 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.

```c
LIB_EXPORT BOOL _plat__NvMemoryWrite(
    unsigned int startOffset,  // IN: write start
    unsigned int size,         // IN: size of bytes to write
    void *data                  // OUT: data buffer
) {
    if (startOffset + size <= NV_MEMORY_SIZE)
        memcpy(&s_NV[startOffset], data, size);  // Copy the data to the NV image
        return TRUE;
    return FALSE;
}
```

### 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.

```c
LIB_EXPORT void _plat__NvMemoryClear(
    unsigned int start,        // IN: clear start
    unsigned int size          // IN: number of bytes to clear
) {
    assert(start + size <= NV_MEMORY_SIZE);
    // In this implementation, assume that the erase value for NV is all 1s
    memset(&s_NV[start], 0xff, size);
}
```
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

```
LIB_EXPORT void _plat__NvMemoryMove(
    unsigned int sourceOffset, // IN: source offset
    unsigned int destOffset,  // IN: destination offset
    unsigned int size          // IN: size of data being moved
)
{
    assert(sourceOffset + size <= NV_MEMORY_SIZE);
    assert(destOffset + size <= NV_MEMORY_SIZE);
    memmove(&s_NV[destOffset], &s_NV[sourceOffset], size);  // Move data in RAM
    return;
}
```

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.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NV write success</td>
</tr>
<tr>
<td>non-0</td>
<td>NV write fail</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT int _plat__NvCommit(
    void
)
{
    #if FILE_BACKED_NV
        return (NvFileCommit() ? 0 : 1);
    #else
        return 0;
    #endif
}
```

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

```
LIB_EXPORT void _plat__SetNvAvail(
    void
)
{
    s_NvIsAvailable = TRUE;
    return;
}
```

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

```
LIB_EXPORT void _plat__ClearNvAvail(
    void
)
{
    }
```
LIB_EXPORT void _plat_ClearNvAvail(
    void
)
{
    s_NvIsAvailable = FALSE;
    return;
}
C.7  PowerPlat.c

C.7.1.  Includes and Function Prototypes

```c
#include "Platform.h"
#include "_TPM_Init_fp.h"
```

C.7.2.  Functions

C.7.2.1.  _plat__Signal_PowerOn()

Signal platform power on

```c
LIBEXPORT int _plat__Signal_PowerOn()
{
  void

  // Reset the timer
  _plat__TimerReset();

  // Need to indicate that we lost power
  s_powerLost = TRUE;

  return 0;
}
```

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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>power was lost</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>power was not lost</td>
</tr>
</tbody>
</table>

```c
LIBEXPORT int _plat__WasPowerLost()
{
  void

  { 
    BOOL        retVal = s_powerLost;
    s_powerLost = FALSE;
    return retVal;
  }
```

C.7.2.3.  _plat__Signal_Reset()

This a TPM reset without a power loss.

```c
LIBEXPORT int _plat__Signal_Reset()
```
C.7.2.4. _plat__Signal_PowerOff()

Signal platform power off

LIB_EXPORT void _plat__Signal_PowerOff(
    void
) {
    // Prepare NV memory for power off
    _plat__NVDisable();
    return;
}
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.

```c
#ifndef _PLATFORM_DATA_H_
#define _PLATFORM_DATA_H_

From Cancel.c Cancel flag. It is initialized as FALSE, which indicate the command is not being canceled

extern int s_isCanceled;
#endif

#include <sys/types.h>
#include <sys/timeb.h>

#define HARDWARE_CLOCK

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.

extern clock64_t s_realTimePrevious;

These values are used to try to synthesize a long lived version of clock().

extern clock64_t s_lastSystemTime;
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.

extern clock64_t s_tpmTime;
#endif // HARDWARE_CLOCK

This value indicates that the timer was reset

extern BOOL s_timerReset;

This value indicates that the timer was stopped. It causes a clock discontinuity.

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.

#define CLOCK_NOMINAL 30000

A 1% change in rate is 300 counts

#define CLOCK_ADJUST_COARSE 300

A 0.1% change in rate is 30 counts
#define CLOCK_ADJUST_MEDIUM 30

A minimum change in rate is 1 count.

#define CLOCK_ADJUST_FINE 1

The clock tolerance is +/-15% (4500 counts) Allow some guard band (16.7%)

#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.

extern uint64_t s_initClock;

This variable records the timer adjustment factor.

extern unsigned int s_adjustRate;

For LocalityPlat.c Locality of current command.

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.

#if (!defined VTPM) || ((VTPM != NO) && (VTPM != YES))

#   undef VTPM

#   define VTPM YES // Default: Either YES or NO

#endif

For a simulation, use a file to back up the NV.

#if (!defined FILE_BACKED_NV) || ((FILE_BACKED_NV != NO) && (FILE_BACKED_NV != YES))

#   undef FILE_BACKED_NV

#   define FILE_BACKED_NV (VTPM && YES) // Default: Either YES or NO

#endif

#if SIMULATION

#   undef FILE_BACKED_NV

#   define FILE_BACKED_NV YES

#endif // _PLATFORM_DATA_H_

extern unsigned char s_NV[NV_MEMORY_SIZE];

extern BOOL s_NvIsAvailable;

extern BOOL s_NV_unrecoverable;

extern BOOL s_NV_recoverable;

For PPPlat.c Physical presence. It is initialized to FALSE.

extern BOOL s_physicalPresence;

From Power.

extern BOOL s_powerLost;

For Entropy.c

extern uint32_t lastEntropy;

#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

```c
#include "Platform.h"
```

From Cancel.c

```c
BOOL s_isCanceled;
```

From Clock.c

```c
unsigned int s_adjustRate;
BOOL s_timerReset;
BOOL s_timerStopped;
#ifndef HARDWARE_CLOCK
clock64_t s_realTimePrevious;
clock64_t s_tpmTime;
clock64_t s_lastSystemTime;
clock64_t s_lastReportedTime;
#endif
```

From LocalityPlat.c

```c
unsigned char s_locality;
```

From Power.c

```c
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.

```c
uint32_t lastEntropy;
```

For NVMem.c

```c
unsigned char s_NV[NV_MEMORY_SIZE];
BOOL s_NvIsAvailable;
BOOL s_NV_unrecoverable;
BOOL s_NV_recoverable;
```

From PPPlat.c

```c
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

```
#include "Platform.h"
```

C.10.3. Functions

C.10.3.1. _plat__PhysicalPresenceAsserted()

Check if physical presence is signaled

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE(1)</td>
<td>if physical presence is signaled</td>
</tr>
<tr>
<td>FALSE(0)</td>
<td>if physical presence is not signaled</td>
</tr>
</tbody>
</table>

```
LIB_EXPORT int _plat__PhysicalPresenceAsserted(
    void
)
{
    // Do not know how to check physical presence without real hardware.
    // so always return TRUE;
    return s_physicalPresence;
}
```

C.10.3.2. _plat__Signal_PhysicalPresenceOn()

Signal physical presence on

```
LIB_EXPORT void _plat__Signal_PhysicalPresenceOn(
    void
)
{
    s_physicalPresence = TRUE;
    return;
}
```

C.10.3.3. _plat__Signal_PhysicalPresenceOff()

Signal physical presence off

```
LIB_EXPORT void _plat__Signal_PhysicalPresenceOff(
    void
)
{
    s_physicalPresence = FALSE;
    return;
}
```
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

```c
#include "Platform.h"
#include <setjmp.h>
#include "ExecCommand_fp.h"
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 longjump 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.

```c
LIB_EXPORT void _plat__RunCommand(
    uint32_t requestSize, // IN: command buffer size
    unsigned char *request, // IN: command buffer
    uint32_t responseSize, // IN/OUT: response buffer size
    unsigned char **response // IN/OUT: response buffer
)
{
    setjmp(s_jumpBuffer);
    ExecuteCommand(requestSize, request, responseSize, response);
}
```

C.11.3.2. _plat__Fail()

This is the platform depended failure exit for the TPM.

```c
LIB_EXPORT NORETURN void _plat__Fail(

    void
)
{
    longjmp(&s_jumpBuffer[0], 1);
}
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

```c
#include "Platform.h"

const char notReallyUnique[] = "This is not really a unique value. A real unique value should 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.

```c
LIB_EXPORT uint32_t _plat__GetUnique(
    uint32_t which,       // authorities (0) or details
    uint32_t bSize,       // size of the buffer
    unsigned char *b     // output buffer
)
{
    const char *from = notReallyUnique;
    uint32_t *b = &b[($(bSize < sizeof(notReallyUnique)) ? bSize : sizeof(notReallyUnique)) - 1];
    for(retVal = 0;
        *from != 0 && retVal < bSize;
        retVal++)
    {
        *b++ = *from++;
    }
    return retVal;
}
```
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

```c
#include <stdio.h>
#include <time.h>

FILE *fDebug = NULL;
unsigned char *fn = "DebugFile.txt";
static FILE *
fileOpen(unsigned char *fn,
const char *mode
)
{
    FILE *f;
    #if defined _MSC_VER
    if (fopen_s(&f, fn, mode) != 0)
        f = NULL;
    #else
    f = fopen(fn, "w");
    #endif
    return f;
}
```

C.13.2.1. DebugFileOpen()

This function opens the file used to hold the debug data.

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>success</td>
</tr>
<tr>
<td>!= 0</td>
<td>error</td>
</tr>
</tbody>
</table>

```c
int DebugFileOpen()
{
    void
    {
        unsigned char timeString[100];
        time_t t = time(NULL);
        // Get current date and time.
        ctime_s(timeString, sizeof(timeString), &t);
        // Try to open the debug file
        fDebug = fileOpen(fn, "w");
        if(fDebug)
            {
                fprintf(fDebug, "%s\n", timeString);
                fclose(fDebug);
                return 0;
            }
        return -1;
    }
}
void DebugFileClose(
```
void
{
if (fDebug)
    fclose(fDebug);
}

void DebugDumpBuffer(
    int size,
    unsigned char *buf,
    unsigned char *identifier
)
{
    int i;
    // FILE *f = fileOpen(fn, "a");
    if (!f)
        return;
    if (identifier)
        fprintf(fDebug, "%s
", identifier);
    if (buf)
    {
        for (i = 0; i < size; i++)
        {
            if (((i % 16) == 0) && (i))
                fprintf(fDebug, "\n");
            fprintf(fDebug, " %02X", buf[i]);
        }
        if ((size % 16) != 0)
            fprintf(fDebug, "\n");
    }
    fclose(f);
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__
Annex D
(informative)
Remote Procedure Interface

D.1 Introduction

These files provide an RPC interface for a TPM simulation. The simulation uses two ports: a command port and a hardware simulation port. Only TPM commands defined in TPM 2.0 Part 3 are sent to the TPM on the command port. The hardware simulation port is used to simulate hardware events such as power on/off and locality; and indications such as _TPM_HashStart.
D.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 UINT32=0 to the TPM response for regularity.

D.2.2. Typedefs and Defines

```c
#ifndef TCP_TPM_PROTOCOL_H
#define TCP_TPM_PROTOCOL_H

D.2.3. TPM Commands

All commands acknowledge processing by returning a UINT32 == 0 except where noted

```c
#define TPM_SIGNAL_POWER_ON 1
#define TPM_SIGNAL_POWER_OFF 2
#define TPM_SIGNAL_PHYS_PRES_ON 3
#define TPM_SIGNAL_PHYS_PRES_OFF 4
#define TPM_SIGNAL_HASH_START 5
#define TPM_SIGNAL_HASH_DATA 6
#define TPM_SIGNAL_CANCEL_ON 9
#define TPM_SIGNAL_CANCEL_OFF 10
#define TPM_SIGNAL_NV_ON 11
#define TPM_SIGNAL_NV_OFF 12
#define TPM_SIGNAL_KEY_CACHE_ON 13
#define TPM_SIGNAL_KEY_CACHE_OFF 14
#define TPM_REMOTE_HANDSHAKE 15
#define TPM_SET_ALTERNATIVE_RESULT 16
#define TPM_SIGNAL_RESET 17
#define TPM_SIGNAL_RESTART 18
#define TPM_SESSION_END 20
#define TPM_STOP 21
#define TPM_GET_COMMAND_RESPONSE_SIZES 25
#define TPM_TEST_FAILURE_MODE 30
```

D.2.4. Enumerations and Structures

```c
enum TpmEndPointInfo
{
    tpmPlatformAvailable = 0x01,
tpmUsesTbs = 0x02,
tpmInRawMode = 0x04,
tpmSupportsPP = 0x08
};
```
// Existing RPC interface type definitions retained so that the implementation
can be re-used
typedef struct in_buffer
{
    unsigned long BufferSize;
    unsigned char *Buffer;
} _IN_BUFFER;
typedef unsigned char * _OUTPUT_BUFFER;
typedef struct out_buffer
{
    uint32_t BufferSize;
    _OUTPUT_BUFFER Buffer;
} _OUT_BUFFER;

#ifndef WIN32
typedef unsigned long DWORD;
typedef void *LPVOID;
#endif

#undef WINAPI
#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

```c
#include "TpmBuildSwitches.h"
#include <stdio.h>
#ifdef _MSC_VER
#include <windows.h>
#include <winsock.h>
#else
typedef int SOCKET;
#endif
#include <string.h>
#include <stdlib.h>
#include <stdint.h>
#include "TpmTcpProtocol.h"
#include "Manufacture_fp.h"
#include "Simulator_fp.h"

To access key cache control in TPM

```c
void RsaKeyCacheControl(int state);
#ifdef __IGNORE_STATE__
static uint32_t ServerVersion = 1;
#define MAX_BUFFER 1048576
char InputBuffer[MAX_BUFFER];  //The input data buffer for the simulator.
char OutputBuffer[MAX_BUFFER];  //The output data buffer for the simulator.
struct {
    uint32_t largestCommandSize;
    uint32_t largestCommand;
    uint32_t largestResponseSize;
    uint32_t largestResponse;
} CommandResponseSizes = {0};
#endif  // __IGNORE_STATE__

D.3.3.  Functions

D.3.3.1.  CreateSocket()

This function creates a socket listening on PortNumber.

```c
static int
CreateSocket(int PortNumber, SOCKET *listenSocket)
{
    WSADATA wsaData;
    struct sockaddr_in MyAddress;
    int res;

    // Initialize Winsock
    res = WSAStartup(MAKEWORD(2, 2), &wsaData);
    if(res != 0)
```
{  printf("WSAStartup failed with error: %d\n", res);
   return -1;
}

// create listening socket
*listenSocket = socket(PF_INET, SOCK_STREAM, 0);
if(INVALID_SOCKET == *listenSocket)
{
   printf("Cannot create server listen socket. Error is 0x%x\n",
      WSAGetLastError());
   return -1;
}

// bind the listening socket to the specified port
ZeroMemory(&MyAddress, sizeof(MyAddress));
MyAddress.sin_port = htons((short)PortNumber);
MyAddress.sin_family = AF_INET;
res = bind(*listenSocket, (struct sockaddr*) &MyAddress, sizeof(MyAddress));
if(res == SOCKET_ERROR)
{
   printf("Bind error. Error is 0x%x\n", WSAGetLastError());
   return -1;
}

// listen/wait for server connections
res = listen(*listenSocket, 3);
if(res == SOCKET_ERROR)
{
   printf("Listen error. Error is 0x%x\n", WSAGetLastError());
   return -1;
}

return 0;

D.3.3.2. PlatformServer()

This function processes incoming platform requests.

BOOL PlatformServer(SOCKET s)
{
   BOOL OK = TRUE;
   uint32_t Command;
   //
   for(;;)
   {
      OK = ReadBytes(s, (char*)&Command, 4);
      // client disconnected (or other error). We stop processing this client
      // and return to our caller who can stop the server or listen for another
      // connection.
      if(!OK) return TRUE;
      Command = ntohl(Command);
      switch(Command)
      {
      case TPM_SIGNAL_POWER_ON:
         _rpc__Signal_PowerOn(FALSE);
         break;
      case TPM_SIGNAL_POWER_OFF:
         _rpc__Signal_PowerOff();
         break;
      case TPM_SIGNAL_RESET:
         _rpc__Signal_PowerOn(TRUE);
         break;
      }
case TPM_SIGNAL_RESTART:
    _rpc__Signal_Restart();
    break;

case TPM_SIGNAL_PHYS_PRES_ON:
    _rpc__Signal_PhysicalPresenceOn();
    break;

case TPM_SIGNAL_PHYS_PRES_OFF:
    _rpc__Signal_PhysicalPresenceOff();
    break;

case TPM_SIGNAL_CANCEL_ON:
    _rpc__Signal_CancelOn();
    break;

case TPM_SIGNAL_CANCEL_OFF:
    _rpc__Signal_CancelOff();
    break;

case TPM_SIGNAL_NV_ON:
    _rpc__Signal_NvOn();
    break;

case TPM_SIGNAL_NV_OFF:
    _rpc__Signal_NvOff();
    break;

case TPM_SIGNAL_KEY_CACHE_ON:
    _rpc__RsaKeyCacheControl(TRUE);
    break;

case TPM_SIGNAL_KEY_CACHE_OFF:
    _rpc__RsaKeyCacheControl(FALSE);
    break;

case TPM_SESSION_END:
    // Client signaled end-of-session
    TpmEndSimulation();
    return TRUE;

case TPM_STOP:
    // Client requested the simulator to exit
    return FALSE;

case TPM_TEST_FAILURE_MODE:
    _rpc__ForceFailureMode();
    break;

case TPM_GET_COMMAND_RESPONSE_SIZES:
    OK = WriteVarBytes(s, (char *)&CommandResponseSizes,
    sizeof(CommandResponseSizes));
    memset(&CommandResponseSizes, 0, sizeof(CommandResponseSizes));
    if(!OK)
        return TRUE;
    break;

default:
    printf("Unrecognized platform interface command %d\n",
    (int)Command);
    WriteUINT32(s, 1);
    return TRUE;
}

D.3.3.3. PlatformSvcRoutine()

This function is called to set up the socket interfaces to listen for commands.

DWORD WINAPI PlatformSvcRoutine(
    LPVOID           port
)
{
```c
int PortNumber = (int)(INT_PTR)port;

SOCKET listenSocket, serverSocket;
struct sockaddr_in HerAddress;

int res;
int length;
BOOL continueServing;

// res = CreateSocket(PortNumber, &listenSocket);
if(res != 0)
{
    printf("Create platform service socket fail\n");
    return res;
}

// Loop accepting connections one-by-one until we are killed or asked to stop
// Note the platform service is single-threaded so we don't listen for a new
// connection until the prior connection drops.
do
{
    printf("Platform server listening on port %d\n", PortNumber);

    // blocking accept
    length = sizeof(HerAddress);
    serverSocket = accept(listenSocket,
                         (struct sockaddr*) &HerAddress,
                         &length);
    if(serverSocket == INVALID_SOCKET)
    {
        printf("Accept error. Error is 0x%x\n", WSAGetLastError());
        return -1;
    }
    printf("Client accepted\n");
    if(continueServing = PlatformServer(serverSocket);
        closesocket(serverSocket);)
    while(continueServing);
    return 0;
}
```

### 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.

```c
int PlatformSignalService(int PortNumber)
{
    HANDLE hPlatformSvc;
    int ThreadId;
    int port = PortNumber;

    // Create service thread for platform signals
    hPlatformSvc = CreateThread(NULL, 0,
                                (LPTHREAD_START_ROUTINE)PlatformSvcRoutine,
                                (LPVOID)(INT_PTR)port, 0, (LPDWORD)&ThreadId);
    if(hPlatformSvc == NULL)
    {
        printf("Thread Creation failed\n");
        return -1;
    }
```

---

D.3.3.4. PlatformSignalService()
D.3.3.5. RegularCommandService()

This function services regular commands.

```c
int RegularCommandService(int PortNumber)
{
    SOCKET listenSocket;
    SOCKET serverSocket;
    struct sockaddr_in HerAddress;
    int res, length;
    BOOL continueServing;

    // res = CreateSocket(PortNumber, &listenSocket);
    if(res != 0)
    {
        printf("Create platform service socket fail\n");
        return res;
    }

    // Loop accepting connections one-by-one until we are killed or asked to stop
    // Note the TPM command service is single-threaded so we don't listen for
    // a new connection until the prior connection drops.
    do
    {
        // normal behavior on client disconnection is to wait for a new client
        // to connect
        continueServing = TpmServer(serverSocket);
        closesocket(serverSocket);
    } while(continueServing);

    return 0;
}
```

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.

```c
int StartTcpServer(int PortNumber)
{
    int res;
```
D.3.3.7. ReadBytes()

This function reads the indicated number of bytes (NumBytes) into buffer from the indicated socket.

```c
BOOL ReadBytes(
    SOCKET s,
    char *buffer,
    int NumBytes
)
{
    int res;
    int numGot = 0;

    //
    while(numGot < NumBytes)
    {
        res = recv(s, buffer + numGot, NumBytes - numGot, 0);
        if(res == -1)
        {
            printf("Receive error. Error is 0x%x\n", WSAGetLastError());
            return FALSE;
        }
        if(res == 0)
        {
            return FALSE;
        }
        numGot += res;
    }
    return TRUE;
}
```

D.3.3.8. WriteBytes()

This function will send the indicated number of bytes (NumBytes) to the indicated socket.

```c
BOOL WriteBytes(
    SOCKET s,
    char *buffer,
    int NumBytes
)
{
    int res;
    int numSent = 0;

    //
```
while (numSent < NumBytes)
{
    res = send(s, buffer + numSent, NumBytes - numSent, 0);
    if (res == -1)
    {
        if (WSAGetLastError() == 0x2745)
        {
            printf("Client disconnected\n");
        }
        else
        {
            printf("Send error. Error is 0x%x\n", WSAGetLastError());
            return FALSE;
        }
    }
    numSent += res;
}
return TRUE;

D.3.3.9. WriteUINT32()

Send 4 bytes containing hton(1)

BOOL WriteUINT32(SOCKET s, uint32_t val)
{
    UINT32 netVal = htonl(val);
    // return WriteBytes(s, (char*)&netVal, 4);
}

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).

BOOL ReadVarBytes(SOCKET s, char *buffer, uint32_t *BytesReceived, int MaxLen)
{
    int length;
    BOOL res;
    // res = ReadBytes(s, (char*)&length, 4);
    if (!res) return res;
    length = ntohl(length);
    *BytesReceived = length;
    if (length > MaxLen)
    {
        printf("Buffer too big. Client says \d\n", length);
        return FALSE;
    }
    if (length == 0) return TRUE;
    res = ReadBytes(s, buffer, length);
    if (!res) return res;
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).

```c
BOOL WriteVarBytes(SOCKET s, char *buffer, int BytesToSend)
{
    uint32_t netLength = htonl(BytesToSend);
    BOOL res;
    //
    res = WriteBytes(s, (char*)&netLength, 4);
    if(!res) return res;
    res = WriteBytes(s, buffer, BytesToSend);
    if(!res) return res;
    return TRUE;
}
```

D.3.3.12. TpmServer()

Processing incoming TPM command requests using the protocol / interface defined above.

```c
BOOL TpmServer(SOCKET s)
{
    uint32_t length;
    uint32_t Command;
    BYTE locality;
    BOOL OK;
    int result;
    int clientVersion;
    _IN_BUFFER InBuffer;
    _OUT_BUFFER OutBuffer;
    //
    for(;;)
    {
        OK = ReadBytes(s, (char*)&Command, 4);
        // client disconnected (or other error). We stop processing this client
        // and return to our caller who can stop the server or listen for another
        // connection.
        if(!OK) return TRUE;
        Command = ntohl(Command);
        switch(Command)
        {
            case TPM_SIGNAL_HASH_START:
                _rpc__Signal_Hash_Start();
                break;
            case TPM_SIGNAL_HASH_END:
                _rpc__Signal_HashEnd();
                break;
            case TPM_SIGNAL_HASH_DATA:
                break;
        }
    }
```
OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
if(!OK) return TRUE;
InBuffer.Buffer = (BYTE*)InputBuffer;
InBuffer.BufferSize = length;
_rpc__Signal_Hash_Data(InBuffer);
break;
case TPM_SEND_COMMAND:
    OK = ReadBytes(s, (char*)locality, 1);
    if(!OK)
        return TRUE;
    OK = ReadVarBytes(s, InputBuffer, &length, MAX_BUFFER);
    if(!OK)
        return TRUE;
    InBuffer.Buffer = (BYTE*)InputBuffer;
    InBuffer.BufferSize = length;
    OutBuffer.BufferSize = MAX_BUFFER;
    OutBuffer.Buffer = (_OUTPUT_BUFFER)OutputBuffer;
    // record the number of bytes in the command if it is the largest
    // we have seen so far.
    if(InBuffer.BufferSize > CommandResponseSizes.largestCommandSize)
    {
        CommandResponseSizes.largestCommandSize = InBuffer.BufferSize;
        memcpy(&CommandResponseSizes.largestCommand, &InputBuffer[6], sizeof(UINT32));
    }
    _rpc__Send_Command(locality, InBuffer, &OutBuffer);
    // record the number of bytes in the response if it is the largest
    // we have seen so far.
    if(OutBuffer.BufferSize > CommandResponseSizes.largestResponseSize)
    {
        CommandResponseSizes.largestResponseSize
            = OutBuffer.BufferSize;
        memcpy(&CommandResponseSizes.largestResponse, &OutputBuffer[6], sizeof(UINT32));
    }
    OK = WriteVarBytes(s, (char*)OutBuffer.Buffer, OutBuffer.BufferSize);
    if(!OK)
        return TRUE;
    break;
case TPM_REMOTE_HANDSHAKE:
    OK = ReadBytes(s, (char*)clientVersion, 4);
    if(!OK)
        return TRUE;
    if(clientVersion == 0)
    {
        printf("Unsupported client version (0)\n");
        return TRUE;
    }
    OK &= WriteUINT32(s, ServerVersion);
    OK &= WriteUINT32(s, tpmInRawMode | tpmPlatformAvailable | tpmSupportsPP);
    break;
case TPM_SET_ALTERNATIVE_RESULT:
    OK = ReadBytes(s, (char*)result, 4);
    if(!OK)
        return TRUE;
    // Alternative result is not applicable to the simulator.
    break;
case TPM_SESSION_END:
    // Client signaled end-of-session
    return TRUE;
case TPM_STOP:
    // Client requested the simulator to exit
    return FALSE;
default:
        printf("Unrecognized TPM interface command %d\n", (int)Command);
        return TRUE;
    }
    OK = WriteUINT32(s, 0);
    if(!OK)
        return TRUE;
}
return FALSE;
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

```c
#include <stdlib.h>
#include <stdio.h>
#include <setjmp.h>
#include "TpmBuildSwitches.h"
#include <windows.h>
#include <winsock.h>
#include "Platform_fp.h"
#include "ExecCommand_fp.h"
#include "Manufacture_fp.h"
#include "TPM_Init_fp.h"
#include "TPM_Hash_Start_fp.h"
#include "TPM_Hash_Data_fp.h"
#include "TPM_Hash_End_fp.h"
#include "TpmFail_fp.h"
#include "TpmTcpProtocol.h"
#include "Simulator_fp.h"
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.

```c
void _rpc__Signal_PowerOn(BOOL isReset)
{
    // if power is on and this is not a call to do TPM reset then return
    if(s_isPowerOn && !isReset)
        return;
    // If this is a reset but power is not on, then return
    if(isReset && !s_isPowerOn)
        return;
    // Unless this is just a reset, pass power on signal to platform
    if(!isReset)
        _plat__Signal_PowerOn();
    // Power on and reset both lead to _TPM_Init()
    _plat__Signal_Reset();
    // Set state as power on
    s_isPowerOn = TRUE;
}
```
D.4.3.2. **Signal_Restart()**

This function processes the clock restart indication. All it does is call the platform function.

```c
void _rpc__Signal_Restart()
{
    _plat__TimerRestart();
}
```

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.

```c
void _rpc__Signal_PowerOff()
{
    if(!s_isPowerOn) return;
    // Pass power off signal to platform
    _plat__Signal_PowerOff();
    s_isPowerOn = FALSE;
    return;
}
```

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.

```c
void _rpc__ForceFailureMode()
{
    SetForceFailureMode();
}
```

D.4.3.5. **_rpc__Signal_PhysicalPresenceOn()**

This function is called to simulate activation of the physical presence pin.

```c
void _rpc__Signal_PhysicalPresenceOn()
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;
    // Pass physical presence on to platform
    _plat__Signal_PhysicalPresenceOn();
```
D.4.3.6. _rpc__Signal_PhysicalPresenceOff()

This function is called to simulate deactivation of the physical presence pin.

```c
void _rpc__Signal_PhysicalPresenceOff()
{
    if (!s_isPowerOn) return;

    _plat__Signal_PhysicalPresenceOff();
    return;
}
```

D.4.3.7. _rpc__Signal_Hash_Start()

This function is called to simulate a _TPM_Hash_Start() event. It will call

```c
void _rpc__Signal_Hash_Start()
{
    if (!s_isPowerOn) return;

    _TPM_Hash_Start();
    return;
}
```

D.4.3.8. _rpc__Signal_Hash_Data()

This function is called to simulate a _TPM_Hash_Data() event.

```c
void _rpc__Signal_Hash_Data(IN_BUFFER input)
{
    if (!s_isPowerOn) return;

    _TPM_Hash_Data(input.BufferSize, input.Buffer);
    return;
}
```

D.4.3.9. _rpc__Signal_HashEnd()

This function is called to simulate a _TPM_Hash_End() event.

```c
void _rpc__Signal_HashEnd()
```
D.4.3.10. _rpc__Send_Command()

This is the interface to the TPM code.

```c
void _rpc__Send_Command(
    unsigned char locality,
    _IN_BUFFER request,
    _OUT_BUFFER *response
)
{
    // If TPM is power off, reject any commands.
    if(!s_isPowerOn)
    {
        response->BufferSize = 0;
        return;
    }

    // Set the locality of the command so that it doesn't change during the command
    _plat__LocalitySet(locality);
    // Do implementation-specific command dispatch
    _plat__RunCommand(request.BufferSize, request.Buffer,
                        &response->BufferSize, &response->Buffer);
    return;
}
```

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_CANCELED.

```c
void _rpc__Signal_CancelOn(
    void
)
{
    // If TPM is power off, reject this signal
    if(!s_isPowerOn) return;

    // Set the platform canceling flag.
    _plat__SetCancel();
    return;
}
```

D.4.3.12. _rpc__Signal_CancelOff()

This function is used to turn off the indication to cancel a command in process.
void _rpc__Signal_CancelOff()
{
    // If TPM is power off, reject this signal
    if (!s_isPowerOn) return;

    // Set the platform canceling flag.
    _plat__ClearCancel();

    return;
}

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.

void _rpc__Signal_NvOn()
{
    // If TPM is power off, reject this signal
    if (!s_isPowerOn) return;

    _plat__SetNvAvail();

    return;
}

D.4.3.14. _rpc__Signal_NvOff()

This function is used to set the indication that NV memory is no longer available.

void _rpc__Signal_NvOff()
{
    // If TPM is power off, reject this signal
    if (!s_isPowerOn) return;

    _plat__ClearNvAvail();

    return;
}

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.

void _rpc__RsaKeyCacheControl(int state)
{
    #if USE_RSA_KEY_CACHE
    RsaKeyCacheControl(state);
    #else
    NOT_REFERENCED(state);
    #endif
D.4.3.16. _rpc__shutdown()

This function is used to stop the TPM simulator.

```c
void _rpc__shutdown(
    void
) {
    RPC_STATUS status;
    // Stop TPM
    TPM_TearDown();
    status = RpcMgmtStopServerListening(NULL);
    if(status != RPC_S_OK)
        {
            printf("RpcMgmtStopServerListening returned: 0x%x\n", status);
            exit(status);
        }
    status = RpcServerUnregisterIf(NULL, NULL, FALSE);
    if(status != RPC_S_OK)
        {
            printf("RpcServerUnregisterIf returned 0x%x\n", status);
            exit(status);
        }
    return;
}
```
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

```c
#include "TpmBuildSwitches.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <ctype.h>
#include <string.h>
#include <windows.h>
#include <winsock.h>
#include "TpmTcpProtocol.h"
#include "Manufacture_fp.h"
#include "Platform_fp.h"
#include "Simulator_fp.h"
#define PURPOSE
#define DEFAULT_TPM_PORT 2321
void* MainPointer;
```

D.5.3. Functions

D.5.3.1. Usage()

This function prints the proper calling sequence for the simulator.

```c
static void Usage(char *pszProgramName)
{
    fprintf(stderr, "%s", PURPOSE);
    fprintf(stderr, "Usage:
    - Starts the TPM server listening on port %d
", pszProgramName, DEFAULT_TPM_PORT);
    fprintf(stderr, " - Starts the TPM server listening on port PortNum\n", pszProgramName);
    fprintf(stderr, " - This message\n", pszProgramName);
    exit(1);
}
```

D.5.3.2. main()

This is the main entry point for the simulator. It registers the interface and starts listening for clients.

```c
int main(int argc, char *argv[])
{
    int portNum = DEFAULT_TPM_PORT;
    if(argc > 2)
```
```c
{  Usage(argv[0]);
}

if(argc == 2)
{
  if(strcmp(argv[1], "?") == 0)
  {
    Usage(argv[0]);
  }
  portNum = atoi(argv[1]);
  if(portNum <= 0 || portNum > 65535)
  {
    Usage(argv[0]);
  }
}
_plat__NVEnable(NULL);

if(TPM_Manufacture(1) != 0)
{
  exit(1);
}
// Coverage test - repeated manufacturing attempt
if(TPM_Manufacture(0) != 1)
{
  exit(2);
}
// Coverage test - re-manufacturing
TPM_TearDown();
if(TPM_Manufacture(1) != 0)
{
  exit(3);
}
// Disable NV memory
_plat__NVDisable();
StartTcpServer(portNum);
return EXIT_SUCCESS;
}````